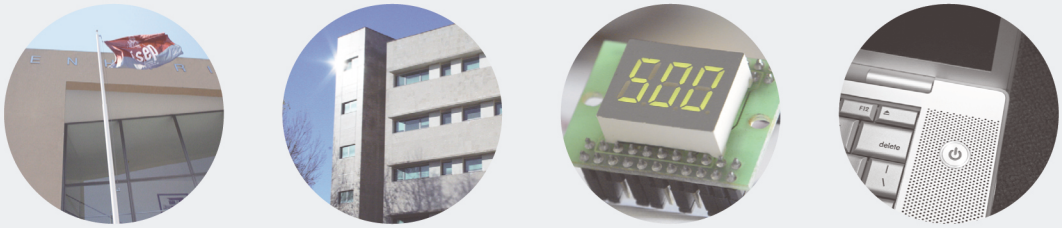
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Smart Alerting for Smart Cities

DIOGO FILIPE FERREIRA VINHAS DA COSTA
outubro de 2019

POLITÉCNICO
DO PORTO

Smart Alerting for Smart Cities

Diogo Filipe Ferreira Vinhas da Costa

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Automação e Sistemas
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
2019

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Disciplina de Tese/Dissertação, do 2.º ano, do Mestrado em Engenharia Eletrotécnica e de Computadores

Candidato: Diogo Filipe Ferreira Vinhas da Costa, Nº 1130508, 1130508@isep.ipp.pt
Orientação científica: Prof. Doutor Jorge Botelho Mamede, jbm@isep.ipp.pt

Empresa:
Supervisão: Eng. André Duarte, aduarte@ubiwhere.com

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Automação e Sistemas
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
2019

Agradecimentos

Com uma etapa tão trabalhosa e importante do meu percurso académico terminada, sinto-me na obrigação de agradecer a várias pessoas que de alguma forma e cada uma diferente da outra, me ajudaram.

Para começar quero agradecer a orientação do Prof. Doutor Jorge Mamede, docente do Instituto Superior de Engenharia do Porto, que fez sempre com que eu me sentisse apoiado, colocando à minha disposição todas as suas vastas competências, promovendo ainda o meu crescimento como aluno.

O meu muito obrigado também, ao meu supervisor em contexto empresarial, André Duarte, que pelo seu profissionalismo e paixão entregue a esta temática, me mostrou como ninguém, o que é ter brio e capacidade de inovação no trabalho. A sua orientação ajudou-me tanto no desenvolvimento como na escrita desta dissertação.

Por último, mas não menos importante (muito pelo contrário), agradeço à minha família e à minha namorada por terem sido incansáveis, tanto no apoio como na paciência que tiveram comigo nesta etapa. Vocês têm o meu eterno agradecimento.

Resumo

O crescimento exponencial do fenómeno da urbanização faz com que a população que vive em áreas urbanas aumente de dia para dia e, consequentemente, os recursos urbanos existentes começam a ser escassos para tanta procura. De facto, a gestão de uma cidade enfrenta, nos dias que correm, desafios como congestionamento do trânsito, segurança pública ou poluição ambiental. Neste cenário surge o conceito de *Smart City* que pode resolver os problemas do desenvolvimento urbano, melhorar a qualidade de vida dos cidadãos e otimizar processos em várias áreas de uma cidade como o sistema de transporte público. As cidades beneficiam também do forte desenvolvimento dos dispositivos de sensorização e atuação bem como das tecnologias de comunicação. Estas últimas permitem que estes equipamentos partilhem informação entre si e com o cidadão. Os dados recolhidos por estes sensores precisam de ser analisados e processados, a fim de potencializar a sua utilidade.

Este trabalho tem como objetivo o desenvolvimento de um sistema de alarmística inteligente, capaz de processar, em tempo real, um grande volume de dados proveniente de diferentes fontes, produzindo como resultados alertas quando for detetada alguma anomalia na informação recebida. Este processamento consiste na comparação dos dados recebidos com padrões/regras definidas previamente, sendo que, quando algum desses padrões é correspondido, um alerta é enviado para os utilizadores interessados.

A arquitetura da solução desenvolvida contempla, na sua fase inicial, um módulo que se encontra constantemente a receber dados de diversas fontes, organizando os mesmos através de operações de filtragem por tipo de informação (tráfego, ambiente, meteorologia, eventos de entretenimento, entre outros). A etapa seguinte é a de processamento da informação, que é o foco principal do presente projeto e onde são utilizadas ferramentas consistentes e capazes de processar um grande volume de dados, responsáveis também pela aplicação das regras nos eventos recebidos. Finalmente, a última fase consiste nas componentes que permitem que os utilizadores e partes interessadas consultem ou sejam notificados com os alertas produzidos no motor de processamento. Foi ainda construída uma *Application Programming Interface* (API) onde as entidades competentes podem

ter acesso ao histórico dos resultados gerados de forma a prever comportamentos futuros.

A elaboração deste projeto levou a um crescimento elevado a nível técnico, uma vez que foram estudadas e implementadas técnicas e ferramentas novas. Foram ainda consolidados alguns conceitos nomeadamente em relação às linguagens de programação utilizadas.

Palavras-Chave: *Smart City*, Analítica, *Complex Event Processing* (CEP), Processamento, Tempo real, Alertas, Padrões

Abstract

The exponential growth of the urbanization phenomenon leads to an ever-increasing number of people living in urban areas day by day and hence, urban resources are becoming scarce for such demand. In fact, nowadays city management faces challenges such as traffic congestion, public safety, or environmental pollution. In this scenario, the Smart City concept emerges and it can solve urban development problems, improve citizens quality of life and optimize processes in various areas of a city such as the public transport system, for example. Cities also benefit from the strong development of sensing and actuation devices as well as communication technologies, which allows information sharing between these equipments and with the citizens. The data collected by these sensors need to be analyzed and processed in order to enhance their usefulness.

This work aims to develop an intelligent alerting system capable of processing, in real time, large volumes of data from different sources, producing as a result, alerts when some anomaly is detected in received information. This processing mechanism consists on comparing received data with patterns/rules previously defined, so when some of these patterns are matched, an alert is sent to interested users.

The developed solution architecture contemplates, on its early phase, a module that is continuously receiving data from several sources. This module organizes these data through filtering operations by type of information, which can be related to traffic, environment, weather, entertainment events, among others. The next step is information processing, which is the project main focus and where consistent tools capable of processing a large amount of data are used, which are responsible for applying the rules to incoming events. Finally, the last phase consists of components that allow users and interested parties to query or be notified with the triggered alerts. An API was also built where entities can have access to the history of generated results in order to predict future behaviors.

The completion of this project led to a high level of technical growth, as new techniques and tools were studied and implemented. Some concepts have also been consolidated, in particular the programming languages used.

Keywords: Smart City, Analytics, CEP, Processing, Real time, Alerts, Patterns

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

AMQP *Advanced Message Queuing Protocol*

API *Application Programming Interface*

CEP *Complex Event Processing*

CPAP *Continuous Positive Airway Pressure*

HDFS *Hadoop Distributed File System*

ICT *Information and Communication Technology*

JAR *Java ARchive*

JSON *JavaScript Object Notation*

MES *Monitoring Event Source*

NoSQL *Not Only SQL*

PM *Project Manager*

ProCiv *Autoridade Nacional de Emergência e Proteção Civil*

PTP *Point to Point*

REST *Representational State Transfer*

RTA *Real Time Analytics*

SQL *Structured Query Language*

TSDB *Time Series Database*

URI *Uniform Resource Identifier*

URL *Uniform Resource Locator*

Capítulo 1

Introdução

1.1 Contextualização

O presente projeto surge da necessidade de se adicionar um módulo importante a um produto desenvolvido pela Ubiwhere, a *Urban Plataform* [1]. Esta funcionalidade a desenvolver consiste no uso da analítica de dados com vista à produção de alertas, em tempo real, quando ocorrer algum incidente ou situação anómala. A nível de investigação também se revela um tema interessante na medida em que impõe que se faça um levantamento do estado da arte das técnicas de analítica em tempo real e sua aplicabilidade no âmbito das *Smart Cities*. A investigação é preponderante para facilitar o acesso ao conhecimento e encontrar resposta para os problemas complexos que possam surgir.

A Ubiwhere é uma empresa de Inovação e Desenvolvimento de software que desenvolve soluções focadas nos setores de Cidades Inteligentes, Telecomunicações e Internet do Futuro. A *Urban Plataform* é um produto inovador que permite uma visão integrada e global dos vários domínios de uma cidade, possibilitando a monitorização da qualidade do ar, gestão do consumo energético, controlo e acompanhamento do tráfego rodoviário, consulta de indicadores dos serviços e qualidade de vida. Este trabalho insere-se no campo da resposta a incidentes, também contemplado pela plataforma.

Para além da vertente empresarial, este projeto possui também um enquadramento académico onde se encontra no âmbito da unidade curricular de Tese /Dissertação (TEDI) do ramo de Automação e Sistemas do mestrado em Engenharia Eletrotécnica e de Computadores. As principais contribuições desta tese são as seguintes:

- Um estudo comparativo das principais técnicas de análise, tendo sido demonstradas as diferenças mais relevantes entre analítica em tempo real e analítica por *batches*;

- Um estudo comparativo das *frameworks* capazes de implementar um motor CEP, tendo sido apresentadas as vantagens e desvantagens do uso de cada uma delas;
- Uma análise comparativa do desempenho de três base de dados não relacionais, onde o foco esteve no estudo do seu desempenho no processo de introdução e consulta de dados, em termos de eficiência;
- Um sistema capaz de receber, processar informação e gerar alertas em tempo real;
- Uma proposta de arquitetura para processamento de *Big Data* em *Smart Cities* que qualquer pessoa pode usar e melhorar;
- Uma *Representational State Transfer* REST API que permite que utilizadores e entidades competentes tenham acesso ao histórico dos alertas produzidos;

1.2 Objetivos

A principal meta deste trabalho é o desenvolvimento de um mecanismo de analítica em tempo real a ser aplicado numa cidade, visando eventos que possam alterar o normal funcionamento desta. Devido à complexidade do que se pretende implementar, decidiu-se que se iria dividir a dissertação em quatro grandes módulos: Investigação, Caracterização do Problema, Arquitetura Proposta e Implementação. No primeiro, o objetivo é fazer a comparação das características de várias metodologias de analítica em tempo real, tendo como propósito a escolha da metodologia mais adequada, sendo essa que é implementada.

Na etapa da descrição do problema que se pretende resolver, são apontados os desafios que este trabalho procura superar, sendo também realizadas comparações de desempenho entre tecnologias relativas às diferentes fases da implementação. Já quando se revela a solução proposta pretende-se desenhar o sistema a ser implementado na fase seguinte. Aqui é desejado que o leitor já se encontre familiarizado com as tecnologias que vão fazer parte do trabalho. Na Implementação, uma das etapas mais extensas desta dissertação, pretende-se desenvolver um sistema inteligente de alarmística em tempo real, onde apesar do foco estar na componente de processamento, compromete-se a cumprir os seguintes requisitos:

- Suportar pelo menos duas fontes de dados distintas;
- Aplicar regras de processamento predefinidas, nos dados recebidos, gerando alertas em tempo real;
- Realizar a atualização das regras ativas de forma automática;

- Criar um histórico de todos os alertas gerados, ao qual se possa aceder através de uma API;
- Gerar alertas para, pelo menos, 2 canais distintos (e-mail, mensagem para o telemóvel, aplicação *Web*, entre outros);
- Criar um protótipo que possa ser instalado numa máquina diferente, sem que sejam necessárias quaisquer configurações manuais.

1.3 Calendarização

Na Figura 1.1 pode ser visto o plano para o desenvolvimento do presente projeto. A cor cinzenta corresponde à duração, em semanas, das tarefas gerais sendo que de seguida foi realizada uma distinção por tipo de sub-tarefa: com a cor verde estão representadas as sub-tarefas onde o trabalho efetuado consistiu em estudos e investigação dos tópicos relacionados com o projeto; a cor azul corresponde às tarefas relativas mais concretamente ao desenvolvimento, que engloba a definição de arquiteturas e implementação das mesmas; por fim a cor laranja identifica a duração da escrita desta dissertação.

	Tarefa	Duração
#	Smart Alerting for Smart Cities	#
1	Estado da Arte	11 semanas
1.1	Cidades Inteligentes e iniciativas relacionadas	3 semanas
1.2	IoT e desafios que dele surgem	3 semanas
1.3	Estudo dos métodos de analítica de Big Data	8 semanas
2	Arquitetura do Sistema	14 semanas
2.1	Caracterização do Problema	2 semanas
2.2	Estudo de ferramentas e tecnologias	11 semanas
2.3	Desenho da solução proposta	8 semanas
3	Implementação	13 semanas
3.1	Processo de distribuição de mensagens	4 semanas
3.2	Motor CEP	9 semanas
3.3	Ativação de Alertas	5 semanas
3.4	Integração dos módulos	7 semanas
3	Testes de Performance	7 semanas
4	Escrita da dissertação	36 semanas

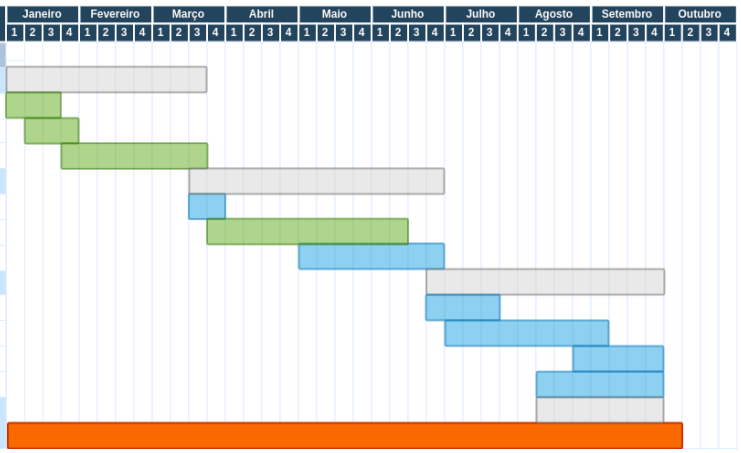


Figura 1.1: Gráfico de Gantt do trabalho desenvolvido

1.4 Organização da dissertação

No capítulo 1 é feita uma introdução ao tema desta dissertação, onde se explica o contexto onde a realização desta se insere, bem como os objetivos que

se propõem alcançar. O capítulo termina sendo apresentado o diagrama temporal, dividido por semanas, da execução das diferentes tarefas que conduziram à implementação da solução final.

O capítulo 2 incide sobre o Estado da Arte das técnicas e conceitos que este trabalho envolve. Aqui, o nível de abstração dos conceitos descritos nas várias secções vai decrescendo, sendo que começa por explicar o termo *Smart City* onde se comparam várias opiniões que estão disponíveis na literatura e relatadas algumas iniciativas que visam proporcionar inteligência à cidade em questão. Na última secção já se descreve um conceito bem especifico que é a análise de dados.

No capítulo 3 é feita a caracterização do problema que se tenta resolver com a implementação deste projeto, sendo descritos, numa primeira fase os desafios do trabalho. Posteriormente e como forma de resposta aos desafios revelados, faz uma comparação de ferramentas e tecnologias capazes de superar os mesmos.

O capítulo 4, por sua vez, consiste na apresentação dos passos tomados para implementar a solução, começando por se mostrar a arquitetura da solução proposta, cujos módulos e suas funções são descritas e dois potenciais casos de uso do sistema implementado. De seguida descreve-se a implementação de cada um dos blocos mostrados na secção anterior. Para cada componente é descrito o seu funcionamento e sempre que se achou relevante, exibido e explicado extratos de código.

No capítulo 5 mostram-se as métricas temporais recolhidas nos testes de desempenho do sistema, indicando-se para cada análise, as conclusões daí obtidas.

Por fim, no capítulo 6, abordam-se as conclusões tiradas no desenvolvimento do trabalho, dando-se destaque ao nível de satisfação de objetivos bem como às futuras melhorias à solução construída. O que resta do documento consiste em anexos onde se colocaram os códigos de vários programas.

Capítulo 2

Estado da Arte

Neste capítulo é feito o estudo do estado da arte no âmbito do trabalho. De forma a fazer um enquadramento correto do tema, este capítulo foi dividido em vários subcapítulos onde se começa pelo conceito de *Smart Cities*, fase a partir da qual a pesquisa se vai afunilando para que no término do capítulo sejam abordados os sistemas de alarmística em tempo real, que é de facto o foco central deste estado da arte.

Resumindo, esta secção divide-se em quatro subcapítulos: um primeiro onde se explica em que consiste o conceito de Cidade Inteligente, assim como as suas diversas dimensões; seguidamente é feita uma ponte para a parte da comunicação entre sensores e uma explicação sobre a importância do IoT (*Internet of Things*) como base da "inteligência" de uma cidade; depois, segue-se um subcapítulo mais específico sobre análise de dados, essencialmente de grandes conjuntos de dados (tradicionalmente denominado de *Big Data*); por fim, efetua-se um estudo sobre analítica em tempo real comparando-se as metodologias existentes para realizar esse processo, com o objetivo de concluir qual será o método a implementar neste projeto.

2.1 Smart Cities

Nos últimos anos houve um forte crescimento das *Information and Communication Technologies* (ICT) que englobam todas as comunicações assim como a integração das telecomunicações com os computadores de forma a permitir que o utilizador manipule informação [2] [3]. O uso crescente das ICT levou a que vários processos citadinos fossem otimizados, consequentemente estas cidades começaram a ser rotuladas como cidade "eletrónica", cidade "digital" ou cidade "inteligente". O termo *Smart City* é o termo mais abstrato com que se pode rotular uma cidade que recorra a tecnologias digitais para melhorar os serviços públicos para os seus habitantes, minimizando os impactos ambientais.

Ao longo da pesquisa foi possível encontrar várias definições e perspetivas sobre o que é ou no que deve consistir uma "Cidade Inteligente". Harrison em 2010 dividiu o termo *Smart* em três componentes: *Instrumented*, *Interconnected* e *Intelligent*. O primeiro refere-se à capacidade que uma Cidade tem de recolher e integrar dados em tempo real provenientes de sensores, dispositivos pessoais, etc. *Interconnected* retrata a integração destes dados em plataformas que permitam a troca de informações entre várias cidades. Finalmente, o termo *Intelligent* diz respeito à existência de uma parte complexa de análise e modelação de modo a otimizar as decisões operacionais [4].

De facto, na literatura existe uma grande diversidade no que toca à definição do conceito de *smart city*, sendo que os autores não se focam apenas no aspeto do uso de novas tecnologias mas também na sustentabilidade e eficiência dos recursos. Exemplo disso é a definição da Gartner em 2011 [5] onde sugere que uma Cidade deve aproveitar todo o fluxo de informações para o analisar, tendo como resultado um ecossistema mais sustentável: "*A smart city is based on intelligent exchanges of information that flow between its many different subsystems. This flow of information is analyzed and translated into citizen and commercial services. The city will act on this information flow to make its wider ecosystem more resource-efficient and sustainable. The information exchange is based on a smart governance operating framework designed to make cities sustainable.*".

Por sua vez, Chen, em 2010, foca-se mais nos serviços oferecidos aos cidadãos: "*Smart cities will take advantage of communications and sensor capabilities sewn into the cities' infrastructures to optimize electrical, transportation, and other logistical operations supporting daily life, thereby improving the quality of life for everyone.*".

Recentemente, a população mundial tem sofrido um aumento exponencial o que faz com que as expectativas sobre os padrões de vida também cresçam ao mesmo ritmo. Várias previsões apontam para que em 2050, cerca 70 % da população mundial viva em áreas urbanas. Uma vez que as cidades já consomem grande parte dos recursos e energia do planeta, com o fenómeno da urbanização pode também existir um significativo impacto ambiental. Portanto, é aqui que começa a emergir a necessidade de criação de "Cidades Inteligentes" pois, apesar dos custos associados à sua implementação, pode resultar na redução do consumo de energia, água ou de emissões de gases poluentes e tornar mais eficiente o uso dos recursos existentes.

Tal como já referido, o termo *Smart City* é muito abstrato pelo que pode envolver várias dimensões como mostra a Figura 2.1. Uma Cidade não precisa de munir todos os setores com "inteligência" para ser considerada uma Cidade Inteligente.

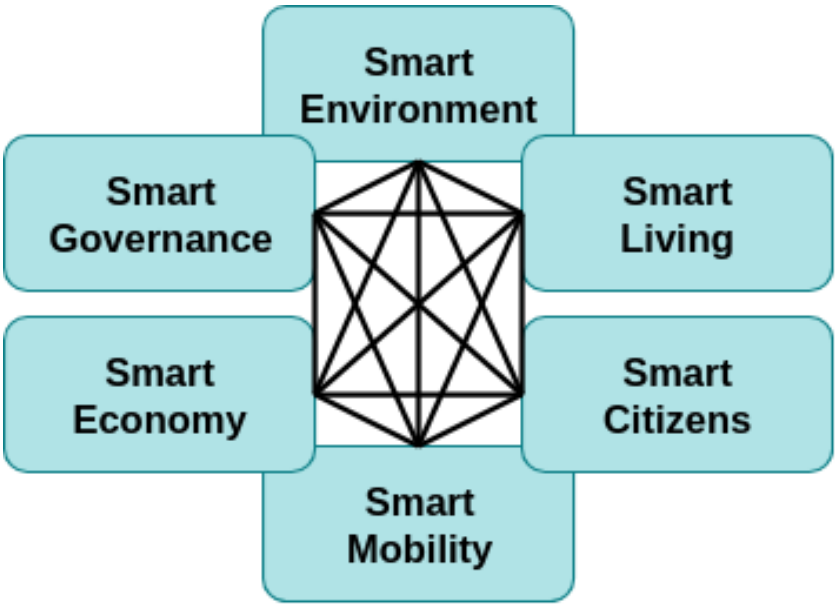


Figura 2.1: Principais dimensões de uma Cidade Inteligente

De facto, uma cidade pode ser vista como um grupo de sistemas, abrangendo inúmeros aspetos onde se pode aplicar "inteligência". Ainda assim, na literatura, a maior parte dos autores procura organizar as dimensões de uma cidade em conjuntos mais genéricos, sendo que são dadas diferentes nomenclaturas para os descrever. Porém as seis componentes que a Figura 2.1 reporta, constituem a base do paradigma de uma Cidade Inteligente:

I Dada a relatividade do termo ***Smart Economy***, não é possível obter uma definição clara do que significa. Assim sendo, procura-se perceber que atributos distinguiriam uma economia "inteligente" de uma perfeitamente convencional. Concluiu-se que uma cidade deve procurar desenvolver ideias inovadoras de forma a melhorar a relação qualidade-preço baseado na otimização dos recursos [6]. Porém, parece existir um consenso em relação a certas características de uma *Smart Economy* como: Inovadora (procura por criatividade e estimula novas ideias); Digital (promove o uso de novas tecnologias); Eficiente (economia equilibrada com crescimento sustentável); Socialmente responsável (medidas em concordância com o bem estar da população) [7].

II O conceito de ***Smart Governance*** prende-se com a capacidade e abertura que as entidades administrativas como o governo possuem de, não só processar a informação, mas também tomar decisões baseando-se nas tecnologias digitais. Um exemplo simples desta transição rumo à alocação de inteligência

num serviço público, é o uso do voto eletrónico numas eleições, o que traz maior transparência e capacidade de resposta, isto é, maior eficiência [8].

III As mudanças ambientais afetam cada vez mais tanto as infraestruturas citadinas, como as condições habitacionais. Isto origina novos desafios no planeamento das cidades assim como o aparecimento do conceito ***Smart Environment***. Metas como melhoria da qualidade do ar e da água ou redução da poluição sonora, podem ser atingidas recorrendo às tecnologias digitais.

IV ***Smart Living*** refere-se à presença das ICT no quotidiano da população, nomeadamente em muitas das suas tarefas. Esta é uma das dimensões que está melhor desenvolvida e implementada, uma vez que hoje em dia, uma série de serviços e tarefas estão alojados em dispositivos como um computador ou um telemóvel, o que confere aos processos mais segurança e rapidez.

V As "Pessoas" são talvez o ponto comum entre todos os componentes de uma *Smart City* pelo que constituem também uma dimensão das mesmas, à qual se deu o nome de ***Smart Citizens***. Preocupação por constante aprendizagem, grande diversidade de etnias, flexibilidade para adaptação às mudanças, natureza democrática e proatividade em ações sociais são alguns dos atributos chave de uma população "inteligente" [7].

VI A última fatia de uma Cidade Inteligente, que é também o foco da presente dissertação, é a ***Smart Mobility***. Esta componente está altamente marcada pelas tecnologias de informação e comunicação, uma vez que muitos dos serviços que tornam possível a sua existência dependem de **informação** em tempo real, à qual se tem acesso, por exemplo, através da **comunicação** com sensores. Esta capacidade de uso da tecnologia gera benefícios para cidadãos (melhoria da experiência de mobilidade nas áreas urbanas), operadores de transporte (maior equilíbrio entre procura e oferta leva a um uso mais eficiente dos recursos), planeamento urbano (planeamento das futuras infraestruturas e serviços de transportes fortemente sustentado por um grande modelo de dados reais, conseguindo-se prever comportamentos), entre outros [9].

Neste momento, o mundo encontra-se numa fase onde o conceito de *Smart City* não é uma ideia de futuro mas sim uma realidade em crescimento muito rápido, permitindo às entidades governamentais o aproveitamento de todo o potencial das tecnologias digitais. Estas tecnologias permitem uma grande colaboração entres cidadãos, comunidades e governo. De seguida, relatam-se alguns exemplos das iniciativas mais gritantes da mudança de paradigma nas cidades.

Em **Singapura**, onde a penetração dos serviços móveis é uma das maiores do mundo, existe a ambição de que a cidade-estado se torne na primeira *Smart*

Nation. De facto, várias medidas de grande impacto têm sido tomadas nesse sentido e, para além de existirem aplicações móveis centradas no cidadão, saúde e transportes, praticamente todos os serviços governamentais de Singapura estão disponíveis *online*. Em 2014 houve também um grande investimento, que até hoje se mantém, na colocação de sensores ao longo de toda a cidade de forma a monitorizar tudo desde a limpeza até ao tráfego, sendo que já é possível detetar quando uma pessoa está a fumar numa zona não autorizada assim como quando alguém atira lixo de um edifício alto. Singapura desenvolveu um modelo 3D dinâmico da cidade-estado chamado *Virtual Singapore* que permite, a quem faz o planeamento, analisar os padrões de tráfego, assim como do fluxo pedestre ou estudar o potencial de determinados edifícios se lhes forem colocados painéis solares. Esta plataforma pode também ser muito importante para executar testes à dispersão da população num determinado pólo urbano e assim estabelecer os melhores procedimentos e percursos em caso de emergência [10].

Outro aspeto diferenciador e importante das cidades que estão a apostar na implementação de inteligência é o nível de abertura ao cidadão normal das plataformas para onde os sensores enviam os dados recolhidos. Em vez de serem apenas as entidades governamentais a tomarem as decisões, o que não seria sensato, uma vez que não são eles quem maioritariamente usufrui dos serviços citadinos, Singapura, assim como Barcelona e Londres tornaram viável que os cidadãos ajudem a determinar quais os próximos passos para a cidade.

A segunda cidade que vale a pena destacar é **Barcelona** [11], que foi nomeada capital europeia da inovação em 2014 e está envolvida em inúmeros projetos com vários focos desde iluminação inteligente, infraestruturas para carregamento de veículos elétricos, rede Wi-Fi em lugares e transportes públicos, isto é, todos relacionados com o conceito de *Smart City*. Barcelona faz um uso de grande escala de sensores para a monitorização de métricas de vários setores da cidade. A capital da Catalunha tem postes de iluminação LED inteligentes que só são ativados quando é detetado movimento, o que leva a uma poupança energética de pelo menos 30 %. Foi também implementado, estando atualmente a ser utilizado, um sistema de eliminação de lixo que consiste na criação de depósitos inteligentes que usam o vácuo e sugam os resíduos, armazenando-os no subsolo. Assim, minimiza-se o cheiro do lixo e a poluição sonora gerada pelos habituais veículos de recolha. Os parques de estacionamento estão também equipados com sensores, cujos dados são recolhidos por um sistema que informa os condutores sobre quais os parques a evitar e aqueles com maior disponibilidade, reduzindo assim as emissões de carbono e possíveis congestionamentos do tráfego.

Em 2008, Barcelona viveu uma grave seca, onde o nível de escassez de água foi tal que foi necessário importar uma grande quantidade de França. Foi talvez este o acontecimento que levou a cidade a ser pioneira na implementação de um sistema de sensores para a irrigação. Este sistema analisa a chuva, usando

paralelamente as previsões do nível de precipitação que esteja prevista ocorrer para alterar o funcionamento dos irrigadores em concordância com essa análise, conservando uma maior quantidade de água.

De facto, esta cidade apresenta várias soluções inovadoras no âmbito das *Smart Cities*, não fosse a ela a anfitriã da *Smart City Expo World Congress* - o maior evento internacional de desenvolvimento urbano.

Também **Londres** está bem posicionada na lista das cidades que têm investido mais na inclusão de inteligência nos seus serviços. Segundo várias estimativas, a população londrina irá aumentar cerca de 1 milhão nos próximos 10 anos, esperando-se em 2030, que o número de habitantes passe a fasquia dos 10 milhões, o que faz com que emerjam preocupações como o aumento da pressão sobre o sistema de saúde, transportes ou gestão da poluição assim como o quase crónico problema do congestionamento do tráfego.

O maior exemplo das mais recentes iniciativas levadas a cabo em Londres, são os *Heathrow pods*, um sistema dinâmico de transporte rápido pessoal cuja função é transportar no máximo quatro passageiros e as suas bagagens ao longo de quase 4 km entre o Terminal 5 do aeroporto e o parque de estacionamento desse terminal. Ao todo são 21 veículos autónomos, alimentados por 4 baterias que são carregadas enquanto não existem passageiros para transportar. Todas as garantias de segurança e acompanhamento dos trajetos são feitos numa central de controlo. Este é um dos melhores sistemas de transporte do mundo pelo que eliminou cerca de 70 000 viagens de autocarro por ano o que equivale a 100 toneladas de dióxido de carbono emitido no mesmo período. Vantagens como o fim do congestionamento, tempos de espera e grande aumento de conforto também são de destacar [12].

Um dos aspetos que caracterizam uma *Smart City* prende-se com o envolvimento da população na procura por melhores serviços e como tal desenvolveu-se a *London Datastore*, um portal totalmente acessível a qualquer cidadão onde mais de 700 conjuntos de dados são partilhados para ajudar qualquer pessoa a perceber a cidade mas também a desenvolver soluções para os seus problemas. Este portal reporta de facto uma grande quantidade de dados e estatísticas acerca das várias vertentes da cidade como Emprego e Economia, Ambiente, Transportes, Saúde, entre outros. A criação desta plataforma levou também ao desenvolvimento de muitas aplicações que usam os dados para gerir operações mais específicas.

Já a cidade norte americana de **Seattle** em parceria com a universidade de Washington tenta abordar temas emergentes como preparação para catástrofes e para a redução da emissão de gases poluentes. Quanto à poluição ambiental, Seattle combate em duas frentes: através da analítica tentam reduzir as emissões de gases dos edifícios em 45% e têm ainda implementado um sistema adaptável

de gestão do tráfego onde os sinais de trânsito luminosos se adaptam às mudanças das condições climatéricas e do piso.

Já no que diz respeito à prevenção das catástrofes, de forma a combater as chuvas fortes e tempestades que tanto atormentam esta cidade, deixando-a completamente bloqueada em termos de circulação, escolas fechadas, estradas alagadas, etc., foi criada a plataforma *RainWatch*. Este projeto é uma ferramenta de gestão de emergência que alerta os departamentos governamentais de maneira a que estes possam ser pro-ativos no combate a um determinado acontecimento. O funcionamento consiste na monitorização da percentagem de chuva acumulada numa dada área assim como um acompanhamento das direções que uma tempestade toma. Em 2017 foram instaladas cerca de 800 câmaras para facilitar o trabalho das forças policiais que as usaram para aprimorar a tecnologia de deteção de tiroteios *ShotSpotter* [13].

Como se pôde constatar, existem várias cidades que estão cada vez mais a apostar na atribuição de "inteligência" aos seus serviços. Os exemplos que se relataram são apenas algumas das iniciativas mais inovadoras que existem pois muitas outras cidades estão empenhadas nesta transformação que consiste em tirar o maior proveito das tecnologias digitais como as ICT assim como do conceito de IoT (Internet of Things).

2.2 IoT

Quando se fala na forte transformação digital que o mundo está a passar, isto é, a forma como as empresas fazem uso da tecnologia para melhorar as suas operações e garantir melhores resultados, fala-se numa mudança desde a maneira como conduzimos até à forma como fazemos compras e até como trazemos energia às nossas casas. Sensores sofisticados ou *chips* embebidos em objetos físicos que nos rodeiam, transmitem dados valiosos que nos permitem perceber como é que esse objeto funciona e como é que nos podemos relacionar. Contudo, é necessário saber tratar essa informação. Se falarmos em melhorar um processo fabril, alertar os residentes de uma cidade em tempo real sobre a percentagem de ocupação de um parque ou monitorizar a nossa saúde pessoal, é o conceito de *Internet of Things* que permite a comunicação entre dispositivos ou objetos. No fundo, este conceito é sinónimo de conectividade entre todos as "coisas" físicas [14].

Todo o processo começa nos próprios objetos, onde os seus sensores partilham os dados com uma plataforma ou nó intermédio responsável por direcionar devidamente a informação, seja para uma *dashboard* ou para uma função de análise dessas informações. Este intermediário é muitas vezes chamado de *gateway*.

A realidade é que a IoT permite, virtualmente, o surgimento de infinitas oportunidades e conexões entre equipamentos, que causam grande impacto. Segundo

o grupo Gartner, devem existir cerca de 20 biliões de objetos conectados entre si e a revista Forbes perspetiva que o mercado IoT irá estar avaliado em 267 biliões de dólares pela mesma altura [15].

Há um conjunto de motivos que levaram ao aparecimento da IoT começando pelo facto de atualmente ser relativamente fácil obter uma ligação à Internet com elevada velocidade, sem atrasos e com grande fiabilidade. Depois, é possível testemunhar a criação de cada vez mais equipamentos com Wi-Fi embutido assim como redes móveis [11]. Dando como exemplo uma aplicação na saúde, um paciente que tenha sido diagnosticado com apneia do sono, deve receber um equipamento denominado de CPAP[1] com o qual deve dormir, que contém (em muitos casos) uma célula de comunicação por rádio que faz o *upload* dos dados da respiração para uma interface que está disponível para o seu médico, podendo este tirar logo as suas conclusões. Voltando às possíveis razões para a explosão da indústria do IoT, pode também haver uma relação com a presença cada vez mais intensa dos *smartphones* no quotidiano da população. Estes equipamentos podem controlar, monitorizar ou interagir com uma série de dispositivos IoT [17].

De facto, a IoT atingiu um patamar tão alto a nível de uso e importância que atualmente está presente em vários domínios dentro dos quais pode ter inúmeras aplicações, desde medicina e cuidados médicos, sistema de transportes, segurança e vigilância, etc [11]. Estes domínios podem ser agrupados em três segmentos que se consideram os principais: na **previsão de problemas** que inclui qualquer tipo de sistema de monitorização de um equipamento mecânico, com o qual se pode enviar alertas para um determinado destino para que se tomem medidas que acabem com o risco de ocorrência de alguma anomalia [14]. Isto é fácil de exemplificar ao imaginar um carro que avisa o condutor quando os pneus estão com pouca pressão ou quando o nível do óleo está baixo, estando assim a contribuir para que a vida útil do veículo não seja prejudicada; outra aplicação em destaque é **auto-otimização da produção**, que consiste numa situação em que para ser gerado um determinado output, existem várias etapas pelas quais o(s) *input(s)* devem passar (por exemplo, uma linha de montagem), sendo que quando ocorre algum problema numa das etapas, pode causar grande impacto na etapa seguinte e consequentemente em todo o processo. Isto não acontece se as etapas forem capazes de comunicarem entre si, permitindo que o input seja reintroduzido na mesma etapa ou redirecionado para uma etapa similar até o que causou o conflito estar resolvido; por fim, o último grande grupo de aplicações do IoT insere-se na **gestão automática de inventário** e aqui entra como exemplo uma iniciativa inovadora implementada pela Heineken que foi desenvolver barris de cerveja que contêm monitores que mostram o nível de cerveja que se encontra

[1] Continuous Positive Airway Pressure - consiste num pequeno aparelho compressor de ar, silencioso e que liga a uma máscara ajustada ao nariz e/ou boca do paciente [16]

lá dentro assim como a sua frescura, o que garante um melhor serviço aos seus clientes.

Assim, como se pode concluir, estes três grupos podem cobrir qualquer aplicação de IoT nos diversos setores. A medicina pode por exemplo encaixar na questão da prevenção de problemas, o sistema de transportes pode usar a gestão automática de inventário na medida em que pode haver uma monitorização da utilização dos diferentes meios de transporte, de forma a otimizar os recursos disponíveis.

Porém, apesar do tamanho do impacto e de todas a vantagens que esta indústria traz, também existem implicações e desafios que a mesma tenta responder [17]. A primeira é vista como a mais crítica que é a **segurança**.

Pode, realmente, ser assustador pensar que temos tantos dados, muitas vezes informação sensível, dos nossos dispositivos a circular pela Internet, sendo vulnerável a ataques informáticos. Serviços como aplicações de gestão da conta bancária, que normalmente contêm em "cache" credenciais de acesso ou códigos de cartões, devem ter mecanismos de segurança complexos. As ameaças aos dados podem ser classificados em dois tipos: externas, como ataques ao sistema por parte de outras entidades ou internas, que representam no uso indevido das próprias informações.

Já que se optou por falar primeiro da segurança, há um assunto com uma importância significativa que está diretamente relacionado com a segurança, que é a **privacidade**. A privacidade pode ser vista como o acesso controlado às informações e deve possuir como características: secretismo, anonimato e isolamento.

O desafio seguinte da *Internet of Things* é a **infraestrutura** [17]. Esta deve ser adequada para suportar tantas aplicações e serviços diferentes e está dividida em três fases:

1 *Dispositivos* - representam toda a recolha de dados ou medições de um objeto. Os atuadores também pertencem a esta parte da infraestrutura na medida em que podem alterar o estado físico de um determinado objeto que está a ser monitorizado. Aqui insere-se a etapa de sensorização/atuação da qual fazem parte dispositivos industriais até sistemas de vigilância, detetores de nível da água ou acelerómetros. A IoT está se a expandir rapidamente, neste aspeto, graças ao uso de tecnologias como o *Power over Ethernet* (POE), que permite que os equipamentos estejam alimentados sem recurso a uma fonte de energia, mas sim através de um simples cabo de Ethernet.

2 *Comunicação* - esta camada está presente ao longo de todo um ecossistema IoT uma vez que podemos ter diferentes tipos de comunicação ao nível da rede de sensores, entre essa rede e uma *gateway* ou camada de aplicação. De facto, uma das características importantes a escolher quando se pretende construir

um sistema IoT é o protocolo de comunicação. É necessário ter em conta que os sensores recolhem os dados em bruto, cada um de forma diferente, logo a camada de comunicação deve estar preparada para uniformizar a informação que circula entre o sistema, de modo a que este se torne mais robusto.

3 *Aplicação* - O *output* que quem usufrui do sistema quer ver, surge nesta etapa ou como resultado da mesma. Esta camada pode também ser subdividida em vários níveis, sendo que o primeiro é quase obrigatório por ser a interface com a camada física. A partir daqui é possível encontrar grande variedade no que toca à construção da camada de aplicação, já que existem sistemas onde só existe monitorização de uma atividade e os dados são mostrados através de uma dashboard ou então, sendo um pouco mais complexo, a informação proveniente dos sensores pode ser processada e analisada antes de existir interação com as pessoas ou alvo final.

Tendo sido introduzido o conceito de IoT, que pode ser melhor aproveitado através de metodologias que tratem os dados enviados pelos sensores de forma adequado, descreve-se de seguida possíveis técnicas que visam precisamente analisar a informação recolhida.

2.3 Análise de Dados

À medida que se começa a passar de conceitos como *Smart Cities* e IoT para etapas mais específicas no que diz respeito ao âmbito do presente trabalho, é fundamental abordar o tema da analítica de dados.

Para se perceber a necessidade do tratamento eficiente dos dados, comecemos por olhar para a progresso tecnológico que o mundo está, continuamente, a sofrer, onde os telefones fixo evoluíram para *smartphones* com sistemas operativos como o *Android* ou o *iOS* que trazem ainda mais inteligência ao quotidiano e ao equipamento em si. O processamento de dados na ordem dos megabytes por computadores volumosos ou o armazenamento muito limitado desses dados em disquetes por exemplo, que já num passado recente deram origem aos discos rígidos, sofreram também uma grande mudança já que atualmente o armazenamento pode ser feito na *cloud* onde os recursos são praticamente ilimitados. Também na condução se verificou uma grande revolução com o desenvolvimento dos carros autónomos. Ou seja, a quantidade de dados gerada por cada uma destas atividades é enorme, por exemplo, veja-se o caso dos carros autónomos, estes com recurso a sensores altamente modernos, recolhem informação acerca do tamanho de um obstáculo, a distância entre o carro e esse obstáculo e muito mais, sendo que depois decide como reagir. Isto é processo contínuo, pelo que se prevê a dimensão de dados gerados a cada quilómetro.

Os carros autónomos, ou os conceitos por trás dos mesmos, não são mais que exemplos de IoT pelo que se pode concluir que a quantidade de dados gerada pelos dispositivos é cada vez maior, uma vez que agora os mesmos estão conectados entre si através da Internet.

Outro dos fatores que mais contribuem para a produção de muitos dados são as redes sociais. Numa época onde existe um desmedido uso de plataformas como o *Facebook*, *Instagram* ou o *Youtube*, são gerados dados em qualquer operação que os utilizadores realizem. A simples partilha de um vídeo já envolve uma quantidade exorbitante de informação que não está, inicialmente, estruturada. A Domo, uma empresa americana de software fundada em 2010, desenvolveu uma plataforma totalmente móvel, com um sistema operativo baseado na *cloud* e aplicando inteligência artificial, que unifica todos os componentes de um negócio/empresa e sintetiza de uma forma visual num ecrã de um telemóvel. Em cada um dos últimos anos, esta empresa reuniu informações relativas à quantidade de dados gerados nos *sites* das principais plataformas sociais e interativas, tendo como fontes, as próprias empresas e *sites* focados em estatísticas como o *statisticbrain.com*. Este processo resultou numa publicação, ao qual deram o nome de *Data Never Sleeps*, que consiste num gráfico que mostra a atividade em várias indústrias, nomeadamente as tecnológicas. O gráfico representado na Figura 2.2 [18] foi construído com base na publicação relativa ao ano de 2018.

Figura 2.2: Dados gerados por minuto em 2018

Esta questão do aumento significativo das fontes de dados levou ao aparecimento do conceito de *Big Data*. Portanto, é importante saber o que significa exatamente ou quando é que se pode considerar um conjunto de dados como *Big Data*.

Dan Ariely, em 2013, disse "*Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it.*", o que permite perceber a ambiguidade da matéria, o que é normal visto que é difícil medir uma variável que aumenta de dia para dia. *Big Data* refere-se a conjuntos de dados com dimensão tal, que não é possível guardá-los ou processá-los com recurso às abordagens tradicionais [19], ainda assim é necessário ter uma noção o mais clara possível de que tipo de dados irão ser difíceis de processar. Para responder à imprecisão deste conceito, *Big Data* pode ser representado pelos "V". A família dos V's começou a ser formada em 2001 quando Doug Laney [20] caracterizou o conceito de *Big Data* em três fatores - Volume, Velocidade e Variedade - e, até hoje, estes são os únicos V em comum na grande maioria dos artigos e publicações. O **Volume**, segundo todos os autores, refere-se ao tamanho do conjunto de dados criado a partir de todas as fontes, o que tem realmente uma dimensão considerável, complexo de gerir e que continua a crescer exponencialmente. Um volume de dados tão grande e não estruturado torna impossível o seu processamento recorrendo por exemplo a abordagens baseadas em SQL. Previsões apontam para que em 2020 o número de objetos conectados entre si passe a fasquia dos 50 mil milhões (ver Figura 2.3 [21]).

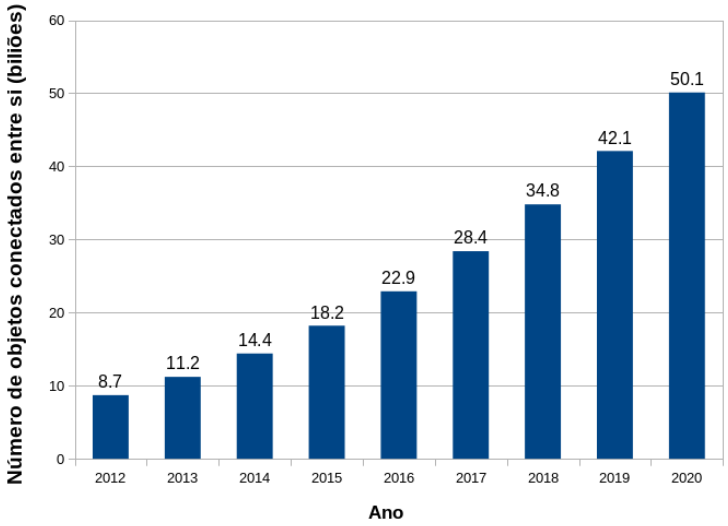


Figura 2.3: Número de objetos conectados entre si no mundo

A **Velocidade** diz respeito à rapidez com que os dados surgem. Como são processos quase instantâneos, a *Velocidade* é uma das razões pelas quais é tão difícil processar estas quantidades de informação. A velocidade com que se faz *stream* dos dados não é a única fase importante, existindo também o *feedback* que muitas vezes é exigido, englobando todo o processo desde a entrada dos dados até ao resultado/decisão [22]. A *Velocidade* em *Big Data* compreende essencialmente dois aspetos: o crescimento e a transmissão. O primeiro está diretamente relacionado com o *Volume* e, como foi dito, é exponencial e é causado por fatores como o aumento do número de utilizadores da Internet, a emergência do *IoT*, o processamento na *cloud*, entre outros [23]. Já a transmissão refere-se ao percurso dos dados de um ponto para outro, sendo que este é um aspeto crítico uma vez que os utilizadores querem cada vez menos latência e mais largura de faixa, o que envolve um maior investimento na infraestrutura e onde o cenário ideal de resposta é o processamento em tempo real.

A **Variedade** indica o nível de diversidade dos dados, isto é, a sua heterogeneidade. Ainda assim, na literatura são feitas diferentes interpretações sobre em que se foca a referida diversidade de dados. Em [23] e [24] por exemplo, avalia-se a *Variedade* dos dados pela forma como eles estão estruturados. Os dados que sejam passíveis de se organizar em tabelas dizem-se estruturados e são normalmente mais fáceis de processar. Já os vídeos, imagens, informações científicas (previsões meteorológicas, análises sísmicas, etc.) são considerados conjuntos não estruturados e constituem cerca de 90 % da *Big Data* [25]. Outras abordagens enfatizam as fontes dos dados, avaliando as suas diferenças assim como a frequência com que estas transmitem ou o tipo de dados. Há ainda autores que se focam na finalidade dos dados, ou seja, defendem que a *Variedade* assenta nas diferentes funções que são necessárias para tratar diferentes tipos de dados [26].

Em [26] o autor realizou um estudo que envolveu 18 artigos científicos onde o objetivo passou por perceber os três V que geram mais concordância entre os autores e aqueles que não são tão globalmente considerados. O resultado desse estudo foi passado para um gráfico que pode ser visto na Figura 2.4. Os valores que estão sobre as fatias correspondem à quantidade de artigos onde o respetivo V é tido em conta, sendo que só se colocou aqueles que são estatisticamente relevantes.

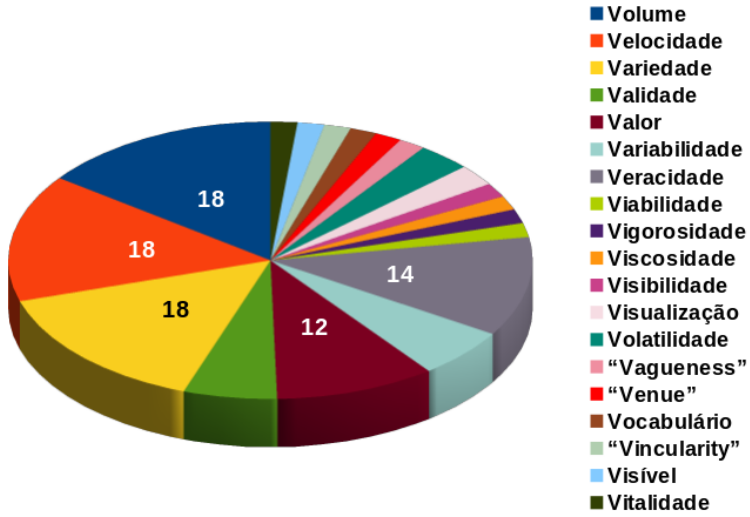


Figura 2.4: Presença dos V em 18 artigos

Analisando o gráfico pode-se concluir que de facto, os três V que se descreveram anteriormente são os que todos os autores têm em conta relativamente à caracterização de *Big Data*. Também é possível perceber algumas razões pelas quais alguns dos V apresentados são muitas vezes descartados que é a notória redundância entre determinadas nomenclaturas como por exemplo *Visibilidade*, *Visualização* e *Visível* ou então a ambiguidade de alguns V como *Vocabulário*.

Porém, existem ainda outros dois V que estão a ganhar importância e sobretudo cada vez mais aceitação por parte de mais autores: o **Valor** e a **Veracidade**. O primeiro pode ser visto como a descrição do grau de importância dos dados baseado nas decisões que serão tomadas com recurso a eles. Por este ponto de vista, este *V* assume uma importância grande pois é ele que vai definir o esforço e o investimento tido para garantir o correto processamento dos dados e sua segurança. Por outro lado [23], há quem entenda o *Valor* como um *V* um pouco diferente dos restantes, isto é, para alguns autores a quantificação do *Valor* de conjunto de dados devia ser feita apenas em função do *output* gerado, defendendo que os dados por si só não têm qualquer importância. Já a *Veracidade* indica a precisão e autenticidade dos dados provenientes das fontes. Este *V* é essencial uma vez que dados falsos ou imprecisos não têm qualquer *Valor*, sendo que a importância da veracidade dos dados varia consoante a sua finalidade. Esta questão pode tornar-se um problema devido ao facto de ser complicado verificar/validar um volume tão grande de informação não estruturada [22].

Até este ponto, descreveu-se em que consiste o conceito de *Big Data* e como podemos caracterizar um determinado conjunto de dados. Contudo a informação

só se torna útil se se conseguir fazer alguma operação com ela, daí surge o subcapítulo de Analítica.

A analítica, segundo Niall Sclater consiste na "análise de dados, tipicamente grandes conjuntos de dados, usando matemática, estatística e software". De facto, o trabalho efetuado na área da analítica é sempre suportado, pelo menos, por um computador que corre um determinado software ou por uma linguagem de programação pois é quase impossível operar com uma dimensão considerável de dados, manualmente. Já Dimitris Bertsimas vai mais longe afirmando que a analítica é a "ciência baseada no uso de dados para construir modelos que conduzam a melhores decisões, que por sua vez adicionam valor aos indivíduos, empresas ou instituições", o que é uma definição mais prática e orientada à aplicação. Na literatura, os autores consideram essencialmente a existência de três tipos de analítica como é mostrado na Figura 2.5 [27]: Descritiva, Preditiva e Prescritiva.

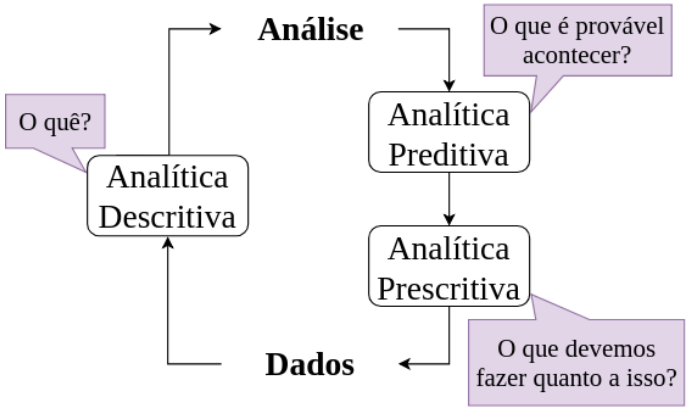


Figura 2.5: Diferentes tipos de analítica

A analítica descritiva decorre normalmente nas fases introdutórias do processamento de dados e caracteriza-se por se focar nos padrões de comportamentos passados, sendo que usa esse histórico para organizar a informação de forma mais eficiente para outro tipo de análise mais avançada. Mecanismos como *clustering* ou categorização, que são próprios deste tipo de analítica, são aplicados a conjuntos de dados de tamanho considerável [28], resultando em algo que seja interpretável por humanos. Esta análise permite-nos aprender a partir de comportamentos anteriores e a perceber como os mesmos podem influenciar os *outputs* futuros. Passando para um caso de aplicação deste tipo de analítica, em [29], os autores desenvolveram uma ferramenta de apoio aos gestores de projeto, a *Project Management Intelligence (PMInt)*. O objetivo deste sistema é retornar, ao *Project Manager* (PM), informação relativa aos sentimentos dos membros constituintes

das equipas do projeto. O princípio de funcionamento da ferramenta consiste em, por um lado, receber informações, nomeadamente em texto, das fontes de dados que podem ser e-mails trocados entre membros das equipas, publicações nas redes sociais, entre outros. Estes dados são pedidos pela ferramenta que usa um sistema de *machine learning* chamado "MonkeyLearn" [30] que a partir de uma frase, retira palavras chave sobre sentimentos e constrói um dicionário com uma *label* com a classificação de "Feliz" ou "Triste" e outra com o nível de confiança (entre 0 e 1), permitindo que o sentimento presente na frase seja corretamente classificado. Finalmente, existe a interface com o PM onde este pode escolher qual a fonte de dados, a partir da qual quer recolher informação. Esta ferramenta, que é um exemplo claro de analítica descritiva, permite ao gestor de projeto tomar melhores decisões na gestão dos membros das suas equipas.

Por outro lado, na analítica preditiva faz-se uso do histórico de dados, padrões comportamentais, estatísticas para prever futuras ocorrências. Normalmente os algoritmos que fazem este tipo de análise não são estáticos, na medida em que aprendem constantemente com novas informações que recebem, sendo que quantos mais dados tiver em seu poder, mais o algoritmo se torna robusto e melhor preparado para antecipar os acontecimentos futuros. Em [31] propôs-se utilizar este tipo de analítica, conjugado com *streaming* de dados em tempo real, para melhorar o sistema de previsão do tempo de chegada dos autocarros públicos em Nashville, Estados Unidos da América. O sistema que os autores descrevem, usa dados históricos sobre os tempos de chegada e partida dos autocarros, horários dos mesmos mas, uma vez por minuto, recebe também informação em tempo real relativamente à posição do autocarro. Esta constante aprendizagem pela qual o algoritmo passa, permite que as previsões sejam muito mais precisas.

Por fim, a analítica prescritiva apresenta-se como a etapa seguinte à preditiva adicionando sugestões de ações, como *output*, que o sistema ou organização deve efetuar para otimizar os resultados, tendo em conta os conjuntos de dados recebidos. Esta é uma analítica mais complexa, uma vez que combina modelos matemáticos (operações, *machine learning*, estatísticas, etc.), algoritmos e regras (boas práticas, limites, restrições, etc.) de forma a poder simular o futuro e retornar a receita que responde da melhor forma a eventos próximos [28]. De facto, este tipo de analítica pode ser muito útil quando bem desenvolvido, por exemplo em [32], introduz-se um sistema que usa analítica descritiva e prescritiva com o objetivo de melhorar a competitividade dos investigadores. Os autores consideram a parte descritiva que consiste em analisar, por ano, o histórico de atividades (trabalhos académicos, industriais e pessoais) de um investigador, assim como a sua capacidade de investigação avaliada com base em vários indicadores. Com isto, um investigador pode perceber o que uma pessoa fez, as duas forças e fraquezas assim como quão bem está classificado em comparação com outros investigadores da mesma área. Já o princípio de funcionamento da parte da analítica prescritiva

traduz-se numa procura, com base nos dados da analítica descritiva, de potenciais grupos de investigadores modelo que estão melhores posicionados em termos de performance de investigação. Quem usa este mecanismo apenas tem que escolher um dos grupos encontrados e o sistema fornece informações sobre esse grupo como forças e fraquezas, assim como sugestões de atividades para o utilizador se aproximar do nível daquele grupo e assim melhorar a sua competitividade.

2.3.1 Batch Analytics

Aqui introduz-se o conceito de *Batch Analytics* ou como muitas vezes é denominado: *Batch Processing*. Este tipo de analítica envolve o armazenamento de uma grande quantidade de informação que é processada em grupos de dados e com uma periodicidade bem definida [33]. A cada um dos processos que são efetuados sobre os conjuntos de dados (*batches*) dá-se o nome de *batch jobs*, que são normalmente estáticos sendo que um sistema pode ter vários *jobs* a serem executados em paralelo, o que garante uma maior eficiência no processamento de grandes quantidades de informação [34]. Relembrando o que foi dito, em relação a *Big Data*, sobre a família dos V em 2.3, esta técnica de analisar conjuntos de dados previamente armazenados foca-se essencialmente no V de Volume descuidando o aspeto da velocidade, já que o objeto sujeito a processamento são dados históricos [35]. Através de uma análise deste tipo consegue-se perceber o que se passou no contexto do problema, o que em alguns casos é o desejado e onde o tempo de resposta não é o mais importante. Por exemplo, processar os salários de todos os colaboradores de uma empresa sendo este um evento que tem, habitualmente, uma periodicidade definida (fim/início de cada mês), não há grande necessidade de uma analítica contínua, bastando que no momento apropriado sejam processados os dados do colaborador, recolhidos ao longo do mês (pontualidade, assiduidade, dias de férias, etc), tendo como resultado o valor a pagar a cada trabalhador.

Para implementar este tipo de analítica, existem várias *frameworks* que suportam processamento por conjuntos de dados, sendo que a mais popular é o ***Apache Hadoop*** [36], que como o próprio nome indica, é um projeto desenvolvido pela Apache, a maior fundação de software *open source*. Esta ferramenta dedicada ao processamento e armazenamento de dados não estruturados em larga escala, consiste fundamentalmente em dois módulos:

- *Hadoop distributed file system* (HDFS): oferece um sistema de ficheiros distribuído para armazenamento de grandes quantidades de dados em várias máquinas distribuídas na mesma rede. Este armazenamento é feito na forma de pequenos blocos de memória (*chunks*), o que aumenta a eficiência de utilização dos dados e apresenta tolerância a falhas [37];

- *MapReduce*: esta *framework* baseia-se num modelo de programação que permite o processamento em paralelo de grande escala em conjuntos de máquinas inseridas num(a) *cluster* ou rede. O *MapReduce* é definido por duas funções: *map* e *reduce*. O *input* é um conjunto de pares chave/valor. A função *map* é responsável por "partir" ou dividir um desses pares num conjunto intermédio de pares, sendo que alguns podem estar vazios. A partir deste momento todos os valores intermédios que tenham a mesma chave são agrupados numa lista. Esta chave intermédia assim como o conjunto intermédio dos seus valores são os *inputs* da função *reduce*. Por fim, a função *reduce* executa uma operação sobre a lista intermédia dos valores (como por exemplo a soma) e retorna um par chave/valor onde o valor é o resultado da operação que efetuou [38]. A Figura 2.6 representa o comportamento deste modelo no processamento de vários pares chave/valor que contêm o nome do documento de texto como chave e o número de expressões inglesas nesse documento. De referir que na mesma figura está apenas representado o que um *worker* (processo/recurso/serviço que executa determinadas tarefas) consegue fazer, sendo que a popularidade deste tipo de *frameworks* reside no facto de este funcionamento ser escalável para a introdução de mais *workers* que podem, inclusive, trocar dados entre si [39].

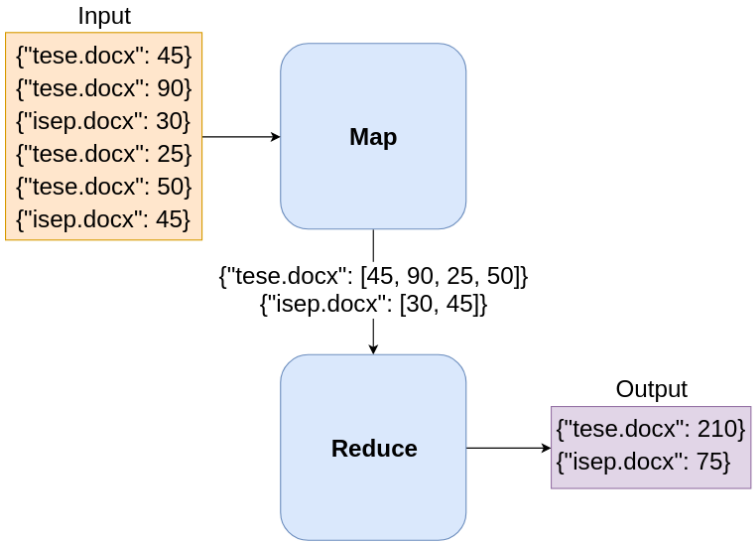


Figura 2.6: Princípio de funcionamento do *MapReduce*

Em [40], os autores propõem a aplicação de um modelo baseado em *MapReduce* nas imagens da análise microscópica dos tecidos celulares no tratamento, diagnóstico e previsão de doenças graves como o cancro. Nomeadamente a nível

microscópico, as imagens produzidas podem atingir dimensões em termos de resolução que tornam o seu processamento muito complexo para um computador. Os autores defendem que um computador portátil comum poderia demorar semanas a concluir o processamento de imagens deste género, através de métodos tradicionais. A abordagem mais comum é dividir a imagem em fragmentos formando um mosaico e uma vez que se procura a eficiência dos recursos computacionais, muitas vezes estes fragmentos não apresentam dimensão suficientemente pequena para um resultado preciso. Esta falta de precisão ocorre quando um objeto pertence simultaneamente a dois fragmentos (ver Figura 2.7) (problema da definição dos limites dos fragmentos), sendo que este objeto é muitas vezes ignorado.

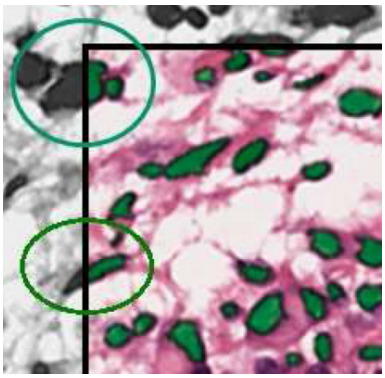


Figura 2.7: Objeto pertencente a dois fragmentos

Então, para responder ao desafio do tempo de processamento destas imagens assim como ao problema dos limites dos fragmentos, os autores propõem uma *framework* de análise eficiente e escalável baseada no modelo *MapReduce*. Assim conseguem responder à questão do tempo de processamento ao paralelizar o mesmo pelas várias máquinas da rede; estão sempre protegidos em caso de alguma máquina falhar devido a problemas de hardware porque a *framework* suporta tolerância a falhas; o facto de poderem escalar o processamento por vários recursos permitiu que fosse implementado um esquema de sobreposição de várias divisões da imagem o que elimina a o problema do cruzamento de objetos entre fragmentos.

Há ainda um tipo de analítica, orientada ao tempo de execução, um pouco diferente das que se falaram anteriormente que se focavam em dados históricos, ou pelo menos partem deles para chegar a um resultado, que é a analítica em tempo real, ou ***Real Time Analytics*** (RTA), onde se insere o tema desta investigação. Este tema será abordado no sub-capítulo seguinte.

2.3.2 Real Time Analytics

Tal como foi referido, a última revolução industrial, à qual se deu o nome de "Indústria 4.0", a expansão da capacidade das infraestruturas de rede, a proliferação do IoT, etc., permitiram que muitas empresas e organizações mudassem a forma de operar, extraindo mais valor dos dados provenientes dos seus equipamentos e pessoas. Este avanço tecnológico levou também a uma maior exigência no processamento da informação, não só de modo a serem gerados resultados o mais precisos possível mas, principalmente, com grande rapidez, sendo que o cenário ideal é que este tempo de processamento seja zero ou muito próximo disso. As RTA consistem exatamente nisso, isto é, quando o tempo decorrido entre a extração de dados e o processamento da resposta é quase nulo. Este conceito de analítica em tempo real traz novos desafios, nomeadamente à forma como se trata os dados no momento em que os recebemos, já que tradicionalmente tem-se a informação guardada em bases de dados, mas essa solução pode promover demasiada latência [41].

Este tipo de analítica é, efetivamente focada no tempo de resposta que vulgarmente se denomina de latência. Em [42] os autores defendem que existem cinco momentos onde os tempos de ação têm impacto nas aplicações em tempo real:

- **Transferência dos eventos/dados**: Este ponto refere-se ao momento em que os dados são enviados para serem processados, sendo que este processo deve acontecer em tempo real. Existem duas alternativas de o fazer: transferir os dados para uma unidade central de processamento ou distribuir os dados por pontos de processamento intermédios. O primeiro é próprio de aplicações que lidem com conjuntos de dados de dimensão pouco significativa. Já a segunda abordagem é mais aconselhável para grandes quantidades de informação, já que graças a mecanismos como filtragem, grande parte do tráfego na rede assim como o tempo de processamento são minimizados;

- **Deteção de anormalidades**: Aqui olha-se para a ocorrência de situações anómalas ou exceções como, por exemplo, um aumento inesperado do tráfego numa determinada zona da cidade. A deteção destas situações é baseada em regras, sendo que estas regras podem ser criadas manualmente por quem desenha a aplicação ou ser criadas dinamicamente como resultado de uma aprendizagem dos valores históricos;

- **Analítica em tempo real**: Os autores consideram que este passo consiste no uso da analítica em tempo real para determinar as causas da ocorrência dos eventos inesperados. Esta analítica pode ser baseada na conjugação de vários serviços, onde uns dependem dos dados recolhidos em tempo real, outros podem depender de informações complementares guardadas em bases de dados;

- **Tomada de decisão em tempo real**: Esta parte surge como resultado da análise anterior e tem como *output* a melhor ação para melhorar a operação em causa ou então pode alertar ou desencadear outros serviços de analítica. Idealmente, as decisões também se baseiam em regras previamente definidas conseguindo assim reduzir o tempo desta etapa;

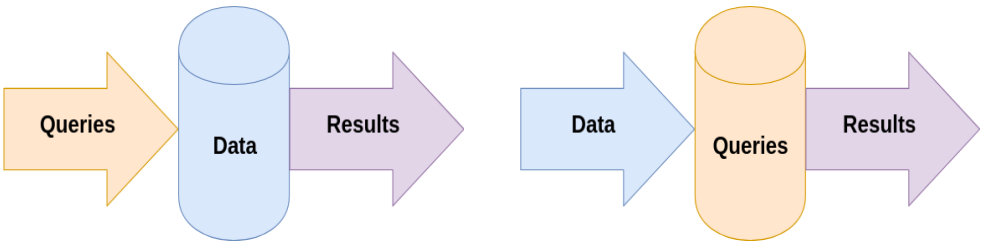
- **Respostas em tempo real**: Este passo engloba todo o ciclo de uma ação gerada no processo de analítica, isto é, execução e monitorização da resposta.

Como já se pode concluir, as RTA apresentam-se como uma forma de análise diferente dos métodos tradicionais. Estas caracterizam-se por oferecer processamento instantâneo de resposta face aos contínuos inputs. De facto, existem aplicações/áreas onde se o tempo de resposta não for muito curto, então as informações são inúteis. A analítica em tempo real proporciona fiabilidade e eficiência quando aplicada corretamente. Por isso é normal que hoje em dia esteja implementado em setores como a economia onde a ideia pode passar por analisar a bolsa ou as cotas de mercado. Através da aplicação de RTA é possível, por um lado ser proativo nas decisões de comprar ou vender cotas no momento certo, mas também, perceber as tendências de evolução do mercado pois quanto mais cedo se executam ações mais lucro se obtém. As RTA aplicam-se também na área militar, onde, numa situação complexa de guerra, é importante fazer uso das informações que se podem recolher, relativas à localização do exército inimigo ou do tipo de equipamentos que estes usam. Quantos mais dados forem recolhidos, mais precisa irá ser a decisão [42].

Podia-se continuar a enumerar as diversas aplicações de RTA noutras áreas sociais, tal é a sua omnipresença atual. Contudo, no presente documento é necessário destacar as aplicações de RTA nas cidades inteligentes nomeadamente no setor da *smart mobility*. Como se referiu em 2.1, considera-se normalmente, que esta área da mobilidade integra serviços como o sistema de transportes públicos, gestão do tráfego e gestão da lotação dos parques de estacionamento. Com o objetivo de atrair a população e dinamizar os seus sistemas de transportes públicos, existem várias iniciativas [43] que visam melhorar a segurança e qualidade do transportes que, através de aplicações de RTA, devem ser capazes de fazer previsões constantes do estado do trânsito e monitorização da localização do transporte em causa, tendo em conta ainda o histórico de procura. Quanto à gestão do tráfego, podem ser gerados alertas ou qualquer tipo de notificação enviado para o condutor de um veículo de forma a sugerir que mude de trajeto quando o atual se apresenta congestionado. Da mesma forma, pode ser enviada para o condutor, informação em tempo real relativa à taxa de ocupação dos parques de estacionamento ou o número de veículos à procura de lugar, minimizando assim prováveis congestionamentos nas redondezas desse parque.

Vistas algumas aplicações de RTA em ambiente real, apresentam-se agora os diferentes métodos a aplicar analítica em tempo real. Geralmente, na literatura [44] [39] consideram-se dois principais ramos de RTA: *Streaming Analytics* e *Micro-Batch Analytics*. Este último apresenta o mesmo princípio de funcionamento que as *Batch Analytics*, descritas em 2.3.1, mas numa tentativa de alcançar valores reduzidos de latência, retira-se complexidade a cada um dos *batch jobs*, podendo assim diminuir a sua periodicidade de execução [45]. A vantagem em usar este tipo de analítica é ser possível, por um lado aceder a dados históricos (porque existe suporta armazenamento de informação), assim como obter tempos de resposta mais ajustados ao significado de *real time*. Contudo, cada *worker* continuará a poder realizar apenas uma tarefa (sobre um conjunto de dados), de cada vez, sendo esta a sua desvantagem.

Já as *Streaming Analytics* caracterizam-se pela ausência de armazenamento dos dados, ou seja, a informação chega e é diretamente processada, eliminando etapas como escrita e leitura de base de dados, o que lhe confere grande velocidade durante o processo de análise, sendo esta a sua principal vantagem. Um conceito importante que surgiu com o aparecimento da análise de *streams* de dados foi o de janela temporal deslizante. O operador que permite define quando é que um conjunto de eventos ou *inputs* é descartado, estabelecer uma condição que, quando satisfeita, desencadeia um processamento sobre os dados dessa janela, ou então gerir o estado dos eventos intermédios [46]. Por outro lado, em *Streaming Analytics* não é possível reutilizar os dados mais tarde, uma vez que não há armazenamento. Este processamento direto de *streams* de dados é também chamado de *Data In Motion* ou *Complex Event Processing* (CEP), que é descrito de seguida. Na Figura 2.8 [34] está representado de forma muito sucinta e direta a diferença no fluxo dos dados entre *Batch* e *Streaming Analytics*.



(a) *Batch Analytics* (b) *Streaming Analytics*

Figura 2.8: Diferença entre os dois tipos de analítica

2.3.2.1 Complex Event Processing

O aumento do número de tecnologias de processamento de informação levou a que houvesse necessidade de ter um motor de análise que consiga suportar débitos elevados de entrada de dados. O CEP é um motor de processamento (monitorização e análise) de informação que suporta entrada de dados (de forma contínua), provenientes de várias fontes, analisando tendências, anomalias e eventos com o objetivo de retornar uma resposta o mais rápido possível, ou seja, tenta responder em tempo real [47] [48]. O CEP deve usar ferramentas capazes de continuamente aplicar/comparar regras, previamente definidas, aos eventos que recebe [48]. Assim, quando uma regra é ativada, gera-se um alerta. Em linguagem "CEP" um evento pode ser considerado uma porção de dados, uma mensagem ou informação que o sistema recebe ou algo que este deteta. Os eventos podem e devem ser organizados pelo seu tipo para que os que pertençam à mesma categoria, sejam processados da mesma forma [49].

John Bates, um cientista de computação, defende que *Complex Event Processing* consiste no tratamento constante de ações na forma de eventos, que descrevem a ação que ocorreu naquele momento, permitindo a realização da sua análise e que através da observação de padrões, pode resultar em oportunidades ou ameaças ao bom funcionamento do sistema que se está a monitorizar [50]. Já em [49], os autores reiteram que CEP é a deteção, em tempo real, de situações específicas a partir da ocorrência de determinados eventos (*inputs* do sistema).

As aplicações deste tipo de sistema são cada vez mais variadas mas têm porém uma característica e uma necessidade em comum, isto é, um motor CEP apresenta-se como a solução mais eficaz em contextos onde há uma elevada e contínua produção de dados (*Big Data* 2.3), sendo que é esperado que haja um processamento em tempo real desses dados, dada a relevância dos mesmos [51]. Como já se viu nos Capítulos 2.1 e em 2.2, o conceito de cidade inteligente está em clara expansão e, consequentemente, o investimento em infraestruturas, que permitem que a cidade esteja preparada para esse paradigma, também aumentou consideravelmente. Grande parte desse investimento traduz-se em instrumentos de comunicação como sensores e atuadores [52]. Ora, estes sensores recolhem um grande volume de dados, desde informações de tráfego até condições ambientais e para tirar o máximo proveito desses dados, é necessário processá-los com a mínima latência possível (em tempo real), caso contrário, as informações perdem utilidade [49]. Conclui-se por isso, que o CEP é a solução adequada para lidar com esta quantidade de dados.

Em [53], os autores desenvolveram uma *framework* baseada na tecnologia CEP que promove a segurança pública dando suporte às equipas de salvamento (bombeiros, primeiros socorros, proteção civil, etc.) através da criação de alarmes em caso de ocorrência de situações que careçam da sua atuação. Um dos pontos

que este artigo destaca prende-se com o uso eficiente dos recursos das equipas de salvamento, isto é, a plataforma desenvolvida não só alerta para a necessidade de atuação mas também especifica a gravidade da ocorrência assim como a sua localização, pois sem esta informação, havia a possibilidade de serem gastos mais recursos do que o necessário. Os autores consideram ambientes fechados (edifícios, escolas, etc) como os seus *use cases*. O trabalho consiste então na monitorização de parâmetros como a temperatura, movimento ou fumo, através de sensores colocados nas divisões e pisos da infraestrutura. Esta informação dos sensores está constantemente a ser enviada para um servidor, onde se encontra a inteligência do protótipo, uma vez que é nesta fase que os dados são processados, sendo-lhes aplicadas regras que contêm *thresholds*. Quando esses *thresholds* são atingidos, é gerado um alerta através de uma chamada de telefone para a equipa de salvamento. Os pormenores que permitem à equipa decidir sobre a alocação de recursos a ser feita, podem ser consultados numa aplicação *Web* onde os alertas são mostrados em tempo real.

Outro ramo relevante onde a tecnologia CEP é aplicada, é o ramo da saúde e cuidados médicos. Em [54], os autores propõem uma solução de monitorização remota de pacientes, com base num motor CEP. Neste sistema, as fontes de dados são sensores e instrumentos colocados no corpo do paciente, que enviam os dados recolhidos para uma aplicação móvel através de uma comunicação *bluetooth* ou Wi-Fi. O *smartphone* do paciente deve estar registado no servidor do hospital e nele o paciente pode escolher o tipo de análise que quer efetuar aos seus indicadores. De seguida, o resultado dessa análise, depois de convertido numa linguagem legível para os componentes seguintes, tem dois destinos: para a tela que o paciente vê no seu *smartphone* como um resumo dos resultados obtidos e para o servidor do hospital onde os resultados podem ser vistos por profissionais competentes nomeadamente os médicos e assim controlar remotamente o estado do paciente.

Por último, descreve-se um caso de aplicação de um motor CEP no contexto da sinistralidade das estradas numa cidade. Em [55], a abordagem passa por aproveitar a crescente sofisticação dos instrumentos de monitorização de veículos para construir um sistema de deteção de movimentos atípicos e potencialmente perigosos por parte dos condutores. O sistema foi desenvolvido para ser executado num *smartphone*. Através de uma comunicação *bluetooth*, informações do veículo como sua velocidade, travagens e mudanças de direção são recolhidos pelos sensores e enviados para o dispositivo móvel do condutor, que por sua vez também recolhe dados sobre a sua orientação, localização, etc. Estas *streams* de dados são depois consumidas por um motor CEP que as agrupa em janelas de tempo para que a análise seja relativa a um período de tempo, o que faz sentido quando se quer por exemplo, analisar a velocidade durante um minuto de viagem. O processamento consiste também numa produção de dados estruturados, isto é,

os eventos chegam de forma não estruturada (*raw data*) e nesta fase são convertidos para um formato mais representativo do comportamento do condutor. De seguida, um módulo de deteção de anormalidades é responsável por aplicar as regras CEP aos dados estruturados, com o objetivo de encontrar padrões de desvio de um comportamento normal de condução. Por fim, como resultado dessa análise, é gerada uma pontuação com base em determinados critérios e pode a partir dessa classificação, ser construído um perfil do condutor. Os autores conseguiram de facto, desenvolver uma solução que caracteriza o comportamento dos condutores com uma precisão a rondar os 90% com recurso apenas aos sensores dos veículos e a um *smartphone* que como não existe armazenamento de uma grande dimensão de dados, consegue apresentar um bom desempenho no processamento.

2.4 Sumário

Este capítulo serviu para apresentar o estado da arte no tema do trabalho, sendo que para isso foi se especificando cada vez mais o foco de investigação até se chegar ao verdadeiro âmbito da presente dissertação. Por isso, na primeira secção deste capítulo introduz-se o conceito de *Smart City*, conceito esse que apesar de não ter uma definição consensual na literatura, entende-se que se traduz no uso das mais recentes tecnologias de informação e comunicação para a otimização dos serviços existentes, se forma a promover a inovação e sustentabilidade nas várias dimensões de uma cidade (dimensão social, ambiental, económica, etc). Ainda nesta secção são dados alguns exemplos de boas iniciativas que preparam da melhor forma as cidades, que investem nestas, para o crescimento exponencial da vertente digital.

Segue-se uma abordagem ao IoT, onde se descreve o conceito que está tão em evidência nos dias que correm. Concluiu-se então que a *Internet of Things* é uma abstração que engloba essencialmente a comunicação e os dados partilhados entre objetos. Visto que estes objetos, nomeadamente os sensores e atuadores numa cidade, se apresentam cada vez mais complexos e completos em termos de características, surge um problema que é processar os dados por eles produzidos e assim tirar o máximo proveito da sua instalação.

Do problema anterior fez-se a ponte para discussão do tema da analítica de dados. Nesta secção, inicialmente descreve-se a noção de *Big Data* e as características de um determinado conjunto de dados pode ser considerado como tal. Este rótulo pode ser atribuído através do método da família dos V: volume, que se refere à dimensão quantitativa dos dados; velocidade, que é relativo à rapidez com os dados entram no sistema; variedade, referente à estrutura dos dados que é cada vez menos homogénea. Posteriormente, entra-se na fase final deste capítulo quando se compara os tipos de analítica e aqui destacam-se dois grandes modelos: analítica por *batches* e analítica em tempo real. A primeira envolve obri-

gatoriamente armazenamento de informação e caracteriza-se pelo processamento de acontecimentos passados, sendo muitas vezes utilizada quando se pretende prever tendências de um determinado processo. Já a analítica em tempo real foca-se na análise do que está a ocorrer naquele momento tendo como principal característica a rapidez com que gera resultados.

Por fim, e ainda dentro da analítica em tempo real, é apresentado o conceito que consiste numa das bases deste trabalho, o CEP. Este motor de processamento foca-se na deteção de padrões e anomalias em *streams* de dados através da aplicação de regras sobre estas.

Tendo sido abordado o estado da arte no âmbito do projeto, apresenta-se de seguida, no Capítulo 3, o desafio ao qual se tenciona responder.

Capítulo 3

Análise do Problema e Especificação de Tecnologias

Neste capítulo, que se apresenta a um nível mais técnico do que os anteriores, descreve-se o problema que a presente solução pretende resolver. Numa primeira instância, o leitor fica contextualizado sendo exibido um esquema elucidativo da globalidade do problema. De seguida, o referido esquema é partido em três componentes que são explicados então de forma mais técnica (*low level*).

3.1 Caracterização do Problema

Como já foi mencionado no capítulo 2, a quantidade de dados estruturados e não estruturados que precisam de ser analisados por métodos mais avançados de forma a detetar padrões para obter melhores resultados no processo de decisão, é cada vez maior e mais diversificada. De facto, várias áreas de negócio enfrentam este desafio do processamento de informação, nomeadamente de dados não estruturados pois é sobre este tipo de dados que os métodos tradicionais apresentam maior dificuldade em operar. A Figura 3.1 mostra uma estatística, realizada em 2011 [56], sobre o tipo de dados que estavam a ser recolhidos até à data, refletindo então uma grande presença de dados estruturados. Também aqui se destaca o facto de cerca metade das respostas consideradas, apontarem já para a existência de informação semi-estruturada e complexa. Já mais recentemente, em 2017, a empresa Ciklum, reconhecida internacionalmente pelos seus trabalhos na área da engenharia de software, nomeadamente nos serviços de analítica, publicou um artigo no seu *blog* [57], onde se realça a relevância dos dados não estruturados, estando previsto que em 2022, 93 % da informação digital seja do tipo não estruturada.

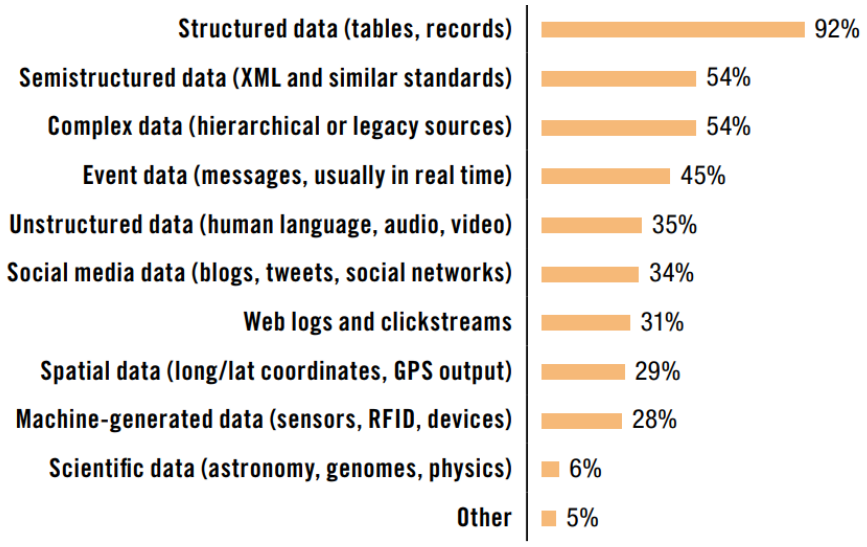


Figura 3.1: 450 respostas de pessoas que trabalham com analítica de *Big Data*

Os avanços na área das telecomunicações e tecnologias de redes permitem então que haja uma maior aproximação (virtual) de todos os objetos e como tal, instrumentos de monitorização como sensores são cada vez mais utilizados. As estradas, por exemplo, estão bastante equipadas com sensores que fazem constantemente leituras às condições do piso com o objetivo de evitar acidentes de viação [58]. Efetivamente, um dos setores onde a monitorização dos processos, com recurso a sensores e atuadores, é mais patente, são as *Smart Cities*. Atualmente, numa cidade consegue-se monitorizar temperatura, pressão atmosférica, humidade, qualidade do ar, tráfego, entre outros, o que ajuda as entidades governamentais a controlar os eventos que ocorrem nessa cidade assim como a melhorar os serviços de quem lá vive. Contudo, todos estes sensores produzem uma quantidade abismal de informação que precisa de ser processada com a maior brevidade possível [59].

Um dos domínios, dentro de uma cidade, onde a sua eficiência depende mais de uma rápida ou até instantânea resposta aos eventos são os sistemas de alarmística, sendo que sem essa característica, os alertas perderiam toda a sua importância. Em [60], é proposta uma solução, focada no tráfego automóvel, que notifica (alerta) os condutores, em tempo real, sobre a ocorrência de algum congestionamento na direção em que estes circulam e calcula alternativas descongestionadas à rota dos mesmos, evitando assim o congestionamento em cenário urbano e reduzindo o tempo de viagem dos condutores. Já em [61], os autores propõe um sistema que através de uma monitorização constante das estradas, gera alertas sobre acontecimentos potencialmente perigosos, como condições do

piso ou acidentes, sendo que os alertas são transmitidos em tempo real através de uma frequência de rádio. Por outro lado, em [62], o foco da solução de alarmística proposta está na poluição do ar. Aqui, os autores introduzem uma solução que monitoriza continuamente a concentração de partículas no ar e calcula assim as áreas com o ar mais poluído, partilhando essa informação com a população através de uma aplicação móvel.

Surge assim um dos grandes objetivos do presente trabalho, construir um sistema de alarmística inteligente - *Smart Alerting*. Como já foi referido, para o desenvolvimento de um sistema inteligente estar concluído, deve possuir determinadas características. As funções chave que uma solução de alarmística inteligente deve apresentar são:

- Recolher informação em tempo real;
- Suportar a entrada de diferentes fontes de dados, como tráfego, ambiente, meteorologia, parques de estacionamento, redes sociais, entre outras;
- Determinar o nível de gravidade do evento com grande precisão;
- Gerar alertas com tempos de reacção na ordem dos milissegundos;
- Encaminhar os alertas para diferentes canais (e-mail, SMS, etc), mediante a sua importância.

A Figura 3.2) apresenta o cenário onde se pretende implementar a solução proposta, isto é, considera-se uma cidade com vários instrumentos de monitorização espalhados pela sua infraestrutura (como por exemplo, nas estradas). O sistema deve conseguir processar, em tempo real, dados de diversas fontes aplicando sobre estes, regras previamente definidas de forma a gerar alertas condizentes com a sua severidade. A solução deve também ser eficiente a nível de recursos de processamento, ou seja, deve haver inteligência suficiente para perceber que se os motores de analítica (onde se deu o nome de "Processamento" na Figura 3.2) não estiverem a receber dados relativos à meteorologia, por exemplo, as funções que contêm as regras sobre esse domínio não precisam de ser executadas.

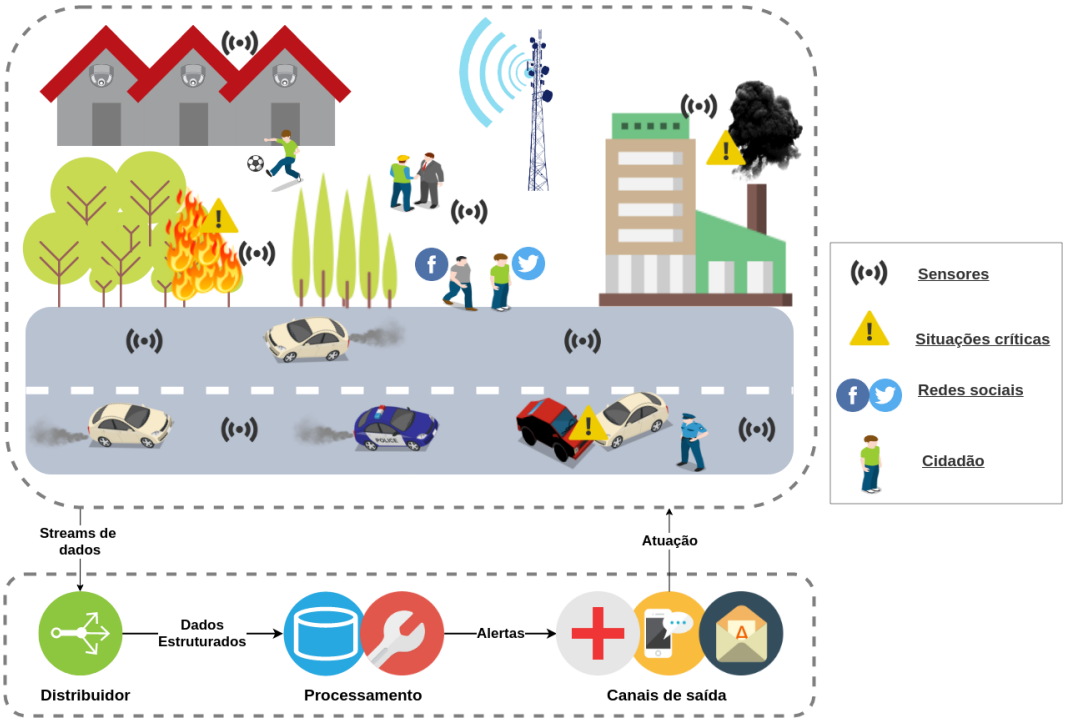


Figura 3.2: Cenário da monitorização e produção de alarmes numa cidade

Na Figura 3.2 estão representadas várias situações críticas a ocorrer: incêndio florestal, acidente rodoviário e alta emissão de gases poluentes. Considerando casos como o da primeira ocorrência (incêndio), o sistema de alarmística ideal pode apresentar grande utilidade tanto na sua prevenção como no seu tratamento (extinção). A nível de prevenção, é possível alertar cidadãos normais e entidades de combate aos incêndios, através da correlação entre dados das previsões meteorológicas (temperaturas elevadas e valores de humidade reduzidos), e dos sensores que estejam a monitorizar a presença de elementos químicos na atmosfera, conseguindo-se assim perceber em que zonas existe uma maior concentração de oxigénio (zonas verdes ou rurais) e determinar quando e onde existe maior risco de incêndio. Depois deste processamento são gerados alertas de diversos tipos e para diferentes canais. A Autoridade Nacional de Emergência e Proteção Civil (ProCiv), por exemplo, já disponibiliza um portal Web onde qualquer pessoa consegue consultar o estado atual das ocorrências em Portugal [63] e até já notificam os cidadãos por intermédio de um SMS como se mostra na Figura 3.3.

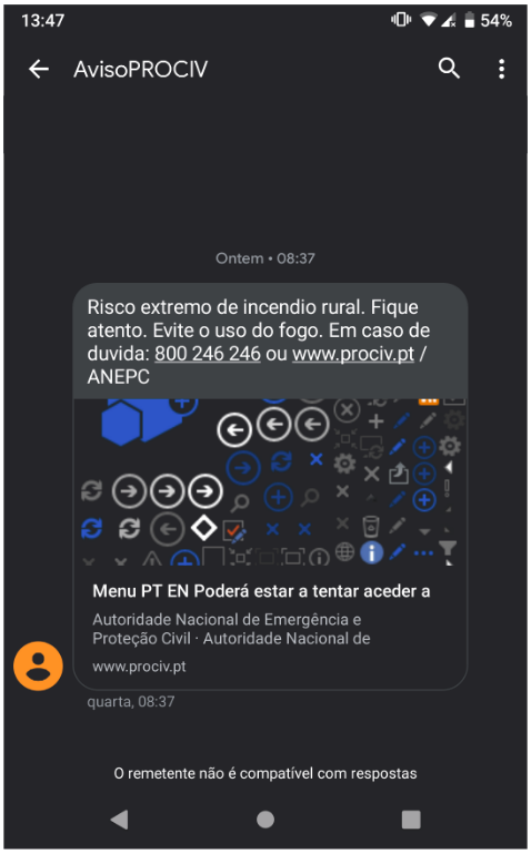


Figura 3.3: Mensagem de texto da ProCiv sobre o risco de incêndio

Uma vez que já foi explicado, de uma perspetiva global, o problema que se pretende solucionar, segue-se agora a apresentação das tecnologias com as quais se consegue construir um sistema como o projetado, isto é, nos sub capítulos que se seguem consistem na divisão do bloco de baixo da Figura 3.2 nos seus principais componentes, denotando uma linguagem mais técnica.

3.2 Processamento de Mensagens

Um dos mecanismos mais importantes no desenvolvimento de uma solução onde haja troca de informação em tempo real, é, sem dúvida, a troca de mensagens/dados entre componentes do sistema. De facto, a transferência de dados é essencial e como tal deve ser realizada de forma eficiente e segura, sendo que no caso dos sistemas de alarmística, torna-se ainda mais importante assegurar que não haja perda de informação. Dentro dos métodos que suportam o processamento de mensagens em tempo real e à escala de *Big Data*, considera-se a

existência de dois tipos de metodologia de *messaging* [64]: *point to point* (PTP) e *publish-subscribe*. Os sistemas PTP são caracterizados, como o próprio nome indica, por terem apenas um recetor para cada mensagem sendo que este, em caso de sucesso, confirma a receção da mensagem [65]. Já o segundo método, *publish-subscribe*, já pode ou não existir mais que um recetor para casa mensagem. Como o sistema que se quer implementar se revela complexo e escalável, conclui-se que o método *publish-subscribe* se apresenta como a melhor opção.

Antes de se caracterizar duas das tecnologias mais populares que se baseiam no método *publish-subscribe*, é importante conhecer algumas terminologias e conceitos que se vai usar ao longo do resto da dissertação:

- **Broker**: Um *broker* de mensagens pode ser considerado um serviço intermédio (*middleware*), que é capaz de distribuir mensagens de três formas: um emissor para vários recetores, vários emissores para um recetor, vários emissores para vários recetores [66];
- **Producer**: Função ou aplicação que envia/produz as mensagens para um *broker*, sendo que em algumas tecnologias pode também ser denominado de *publisher* [67];
- **Consumer** Função ou aplicação que recebe/lê informação do *broker*, sendo denominado de *subscriber* em algumas tecnologias [67];
- **Queue** Uma *queue* pode ser vista como uma fila (*buffer*) onde as mensagens, provenientes dos *producers*, ficam organizadas por ordem de chegada, à espera de serem consumidas por um ou mais *consumers*. O tamanho da *queue* depende da tecnologia em causa [68];
- **Zookeeper**: Serviço responsável pela gestão e coordenação do sistema, guardando o estado dos seus componentes (*brokers*, *consumers*, etc.) de forma a evitar falhas no processamento de mensagens, ou seja, se um *consumer* se desliga por alguma razão, o *Zookeeper* sabe que mensagens já foram consumidas e quais as que não foram e espera que um *consumer* se ligue para continuar o processamento [69].

Agora que se definiu alguns termos importantes, já é possível abordar as tecnologias que se consideram necessárias para sustentar de forma coesa a arquitetura proposta, no que ao processamento de mensagens diz respeito. O **Apache Kafka** [70] e o **RabbitMQ** [71] são dois *brokers* de mensagens que se baseiam no método *publish-subscribe* e são ambos software *open-source* mas, apresentam algumas diferenças na sua arquitetura e, por consequência, no seu funcionamento.

O funcionamento do **RabbitMQ**, tal como em sistemas como o ActiveMQ [72], Qpid [73], entre outros, foca-se no conceito de *queues* de mensagens como

forma de encaminhar a informação desde a fonte até ao destino [74]. Esta ferramenta é conhecida pela sua entrega eficiente de mensagens e implementa o protocolo *Advanced Message Queuing Protocol* (AMQP), cuja arquitetura se apresenta na Figura 3.4 [75]. Aqui, quem produz a mensagem, envia esta acompanhada por uma chave (*routing key*) para um ou mais componentes denominados de *exchanges*, que funcionam como *routers*, pois decidem que caminho percorrem as mensagens. De seguida, cada *exchange* encarrega-se de enviar as mensagens para a *queue* registada com uma chave (*binding key*) igual à *routing key*. Por fim, cada *queue* envia as mensagens para os consumidores que a "subscreveram" [75]. Assim sendo, os *producers* não sabem o estado das mensagens depois de as enviar, nem os *consumers* têm qualquer conhecimento sobre quem enviou os dados, o que resulta uma poupança de processamento para a aplicação.

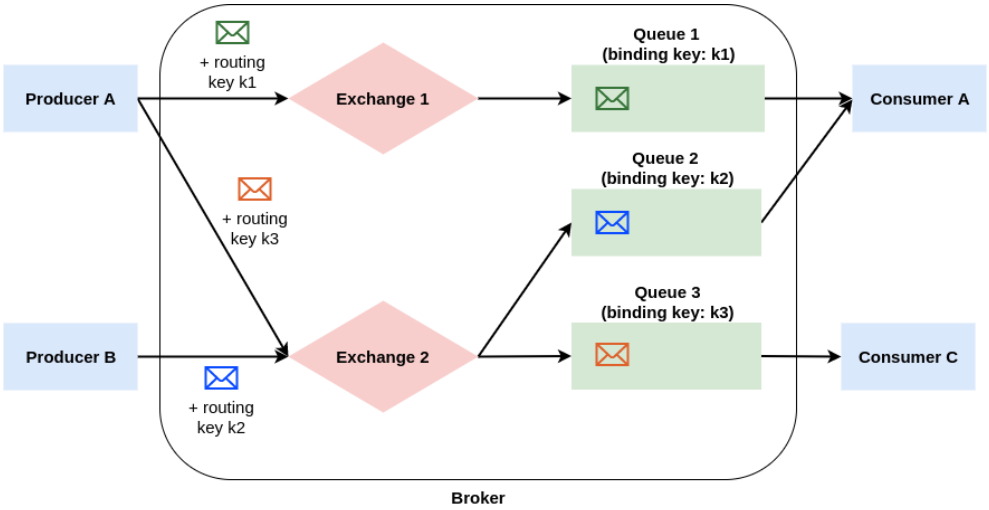


Figura 3.4: Princípio de funcionamento do AMQP

O RabbitMQ oferece escalabilidade às aplicações que o usam, graças à sua capacidade de possuir várias *queues*. Contudo, a lógica destas está otimizada para que o seu conteúdo seja nulo ou quase nulo, pelo que quando isso não acontece, e há necessidade de persistência de mensagens nas *queues*, seja por requisito da aplicação ou por não haver *consumers* disponíveis, o desempenho do sistema desce significativamente [76].

Já o **Kafka**, que foi inicialmente desenvolvido pelo LinkedIn com o intuito de processar ficheiros de registo e outras informações históricas, é considerado por muitos autores um sistema que apresenta latência baixa no processamento de mensagens e que suporta elevadas taxas de produção de dados [75] [77] [78] [79]. A Figura 3.5 [75] [77] mostra a arquitetura do funcionamento do Kafka. Como

o esquema ilustra, o mecanismo começa com o envio de mensagens por parte dos *producers* para um *broker*, que as colocam em partições. Todas as mensagens enviadas são categorizadas por um **tópico**. Um tópico Kafka [80] é uma categoria ou nome para onde os registos/dados são enviados. Continuando o fluxo da arquitetura, têm-se os *consumers*, que podem subscrever um ou mais tópicos, e que consomem todas as mensagens categorizadas com esse nome. Finalmente, o *Zookeeper*, que gere e controla os *clusters* Kafka, os tópicos e as partições, desempenhando por isso um papel importante no funcionamento do sistema. Um *cluster* é um conjunto de *brokers* [81].

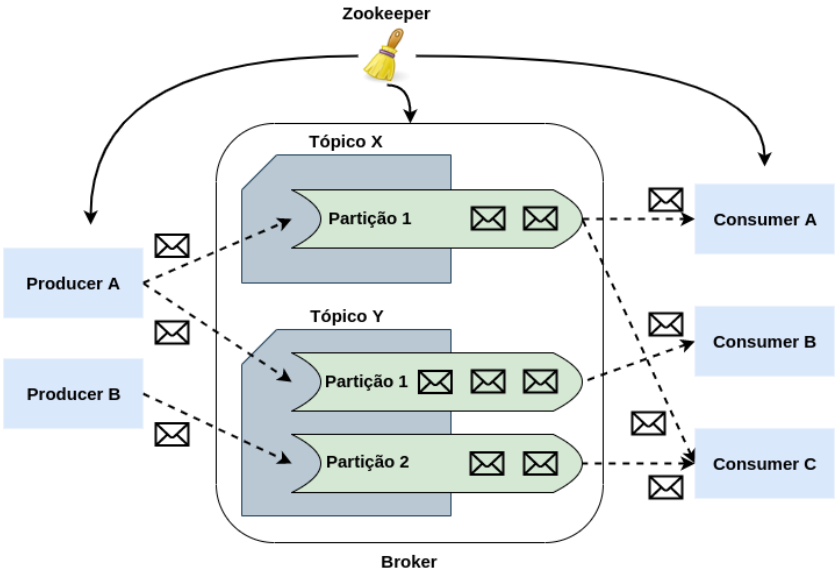


Figura 3.5: Arquitetura do Apache Kafka

Ao contrário do que acontece nos sistemas típicos de processamento de mensagens, no Kafka, quando um registo é colocado numa partição, é-lhe atribuído um número sequencial, denominado de *offset*, que o identifica unicamente. Posto isto, os *consumers* são responsáveis por gerir o *offset*/posição na partição, que a vai incrementando à medida que recebe mensagens [69]. Graças à persistência do Kafka, um *consumer* pode inclusive decidir se quer voltar a processar mensagens passadas ou se quer receber as mais recentes. Isto permite que a possível falha de um ou mais *consumers* não tenha grande impacto [80].

Além da quantidade de dados não ser problema para o Kafka, este sistema também oferece garantias quanto à ordem de entrega das mensagens devido ao mecanismo de atribuição de *offsets* a cada registo.

Quando se projeta um sistema de alarmística em tempo real como o pretendido nesta dissertação, as características mais relevantes da tecnologia responsável pelo processamento de mensagens são a velocidade ou latência com que a informação percorre as aplicações do sistema, a quantidade de dados processados por segundo (*throughput*) e a fiabilidade de entrega desses dados. Neste sub-capítulo abordaram-se duas das tecnologias mais usadas para o processamento de mensagens ou como é denominado na literatura, *data ingestion* [81]. Em [75] fez-se uma análise comparativa do *throughput* e da latência precisamente entre as duas tecnologias referidas. A conclusão obtida neste estudo foi que o Kafka apresenta maior *throughput* de dados mesmo que se trate de um grande volume de dados, enquanto que o RabbitMQ tem uma melhor performance no que diz respeito à latência, a que deve à utilização do modelo *push* para consumir os dados, sendo que no Kafka é aplicado o modelo *pull*, onde os consumidores têm que pedir as mensagens ao *broker*. Contudo, esta latência aumenta quando as *queues* começam a ficar sobrecarregadas de mensagens, pois estão otimizadas para terem sempre consumidores para onde enviar a informação.

De facto, em [77], a questão da diferença da latência entre os dois sistemas é clarificada de forma mensurável, onde os testes efetuados mostram que o Kafka, mesmo com replicação (várias partições para garantir fiabilidade dos resultados), apenas se apresenta cerca de 10 ms mais lento que o RabbitMQ.

Já em [69], os autores realizaram um teste ao *throughput* do Kafka comparando-o ao de um sistema de mensagens por *queuing*, tanto na produção como no consumo de dados e realmente, o Kafka voltou a apresentar, em média e em escalas de dezenas de milhares de mensagens, resultados muito satisfatórios:

- Um *producer* Kafka publica aproximadamente duas vezes mais mensagens que o RabbitMQ;

- Um *consumer* Kafka pode consumir 22 000 mensagens por segundo, sendo quatro vezes superior ao RabbitMQ.

Perante toda a análise e investigação, realizadas com base na literatura existente sobre o tema, desenhou-se a Tabela 3.1 que ilustra as principais diferenças entre Kafka e RabbitMQ, relativamente às características com maior relevância no contexto do problema.

Tabela 3.1: Comparação entre Kafka e RabbitMQ

	Apache Kafka		**RabbitMQ**	
	Volume de dados pequeno	Volume de dados grande	Volume de dados pequeno	Volume de dados grande
Unidade de distribuição	Tópico		Queue	
Throughput	Excelente	Excelente	Excelente	Bom
Latência	Bom	Bom	Excelente	Médio

Chegou-se então à conclusão que o Apache Kafka é a solução mais adequada para o processamento de mensagens ao longo do sistema implementado, pela sua escalabilidade, velocidade e sobretudo pela capacidade de processamento de grandes quantidades de dados. De seguida, transita-se para o que é normalmente a fase seguinte do que se pretende implementar com este trabalho, isto é, o processamento, sendo aqui que o projeto essencialmente se foca.

3.3 CEP Frameworks

Relembrando o que foi dito em 2.3.2.1, CEP é um mecanismo/motor que suporta a análise de dados e o seu contexto, em tempo real, sendo capaz de desencadear eventos ou alertas aquando da deteção de anomalias que se possam traduzir em oportunidades ou ameaças. Contudo, este motor necessita de uma tecnologia ou *framework* que o implemente.

A principal e mais popular *framework* que implementa em linguagem Java o mecanismo CEP é o **Esper** [82] [83] [84], desenvolvido pela EsperTech, sendo um software *open-source*. Esta poderosa *framework* permite o rápido desenvolvimento de aplicações que processam grandes volumes de dados, sejam eles históricos ou em tempo real [84]. O Esper pode ser dividido em três componentes:

- **Linguagem**: Quando se fala na linguagem na *framework* Esper, refere-se a *Event Processing Language* (EPL) e é através desta que se definem as regras CEP. A EPL tem uma sintaxe muito similar à das *queries* SQL [83];

- ***Compiler***: Este componente é responsável por converter linguagem EPL em *byte code* (código intermédio percetível para quem o vai executar) [84];

- ***Runtime*** Por fim, nesta fase é carregado e executado o código produzido pelo *compiler*, sendo gerado então o motor que processa *streams* de dados em tempo real, que ainda consegue analisar eventos históricos [84];

O Esper foi desenvolvido e otimizado para processar eventos complexos (CEP), sendo que para tal possui características como permitir paralelismo de processamento em grande escala, isto é, suporta o processamento da mesma *querie* em múltiplos nós do sistema assim como o processamento de múltiplas *queries* [84]. Esta solução destaca-se também no que diz respeito à latência, que é muito reduzida devido à forma como o *compiler* opera, otimizando as regras CEP para um código que seja rapidamente executado pelo hardware [84]. Contudo, o Esper apenas executa *queries* sobre as *streams* de dados, isto é, necessita que alguma ferramenta organize e envie essas *streams* para poderem ser processadas pelo motor Esper. Surge então a necessidade de olhar para as tecnologias de processamento de *streams* e perceber se existe a possibilidade de as integrar com o Esper.

Na literatura, encontram-se vários estudos de comparação entre as principais *frameworks* de *stream processing* [85] [86] [78] [87], sendo que o Apache Storm [88] e o Apache Flink [89] são os casos mais estudados, pelo que se vai explicar o seu funcionamento. No presente capítulo vai-se também abordar o Twitter Heron [90] que é visto como um pequeno *upgrade* do Storm [91].

Começando pelo **Apache Storm**, esta é uma ferramenta *open-source* de processamento de dados estruturados e não estruturados, em tempo real, sendo provavelmente a mais utilizada [92]. O Storm foi projetado para ser escalável, robusto, tolerante a falhas e fácil de implementar, o que faz dele uma solução muito eficaz no processamento de *Big Data* [86]. Quanto à sua arquitetura, é necessário distinguir dois conceitos: *cluster* e topologia/programa. Um *cluster* Storm (ver Figura 3.6 [93]) possui um nó *master*, denominado de *Nimbus* e vários nós *workers* (rever significado de *worker* em 2.3.1), denominados de *Supervisors*. O *Nimbus* é responsável pela distribuição do código pelo *cluster*, atribuição de tarefas aos *workers* e monitorização de erros. Já cada *Supervisor* executa os processos que lhe foram atribuídos. Toda a coordenação existente entre *Nimbus* e *Supervisors* é feita por um *cluster Zookeeper*, que é um componente importante na medida em que guarda o estado dos restantes nós localmente, ou seja, o sistema não perde estabilidade mesmo quando o *Nimbus* ou os *Supervisors* falham [78].

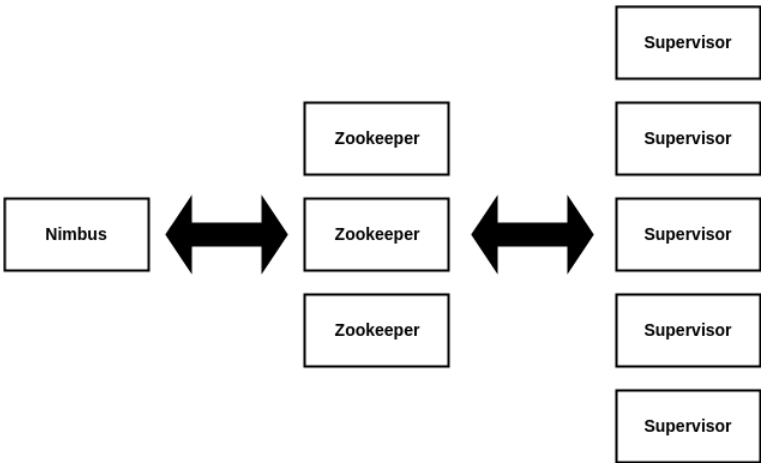


Figura 3.6: Componentes de um *cluster* Storm

Já uma topologia Storm (também denominada de programa) (ver Figura 3.7 [88]), foca-se em dois conceitos: *Spouts* e *Bolts*. Os primeiros funcionam como fontes de streams de dados (no contexto do Storm, *streams* são sequências de *tuples* [93]). Um *spout*, por sua vez, pode ler *streams* a partir de muitas fontes como por exemplo, de um tópico Kafka ou de uma base de dados. Já os *Bolts* recebem as *streams* provenientes dos *Spouts* e processam essa informação, podendo de seguida, emitir o resultado desse processamento para outros *Bolts* ou para fora da topologia [86] [78].

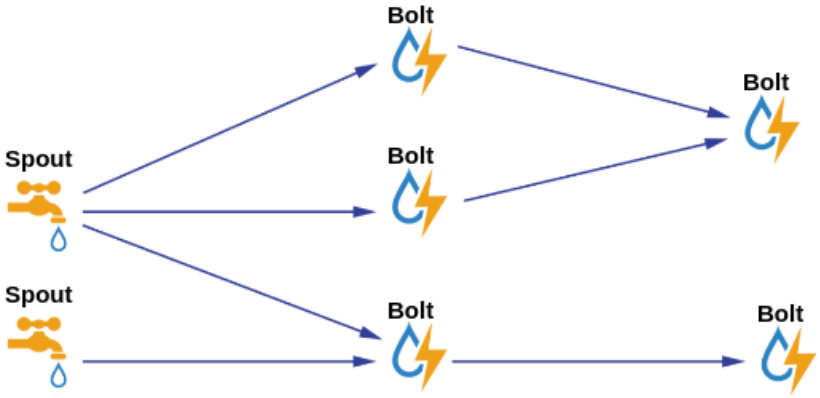


Figura 3.7: Arquitetura de uma topologia Storm

Na criação da topologia pode-se especificar o nível de paralelismo pretendido, ou seja, o Storm é capaz de distribuir vários *Bolts* para que não haja perda

de dados e de forma a aumentar a velocidade de processamento. Mesmo que alguma máquina no sistema falhe, o Storm garante a entrega de informação. Cada topologia é submetida num *cluster*, ficando a cargo do *Nimbus* a distribuição de tarefas pelas várias máquinas.

O Twitter é uma das empresas mais conhecidas que usava o Apache Storm como o seu motor de processamento de *streams*, contudo, com o crescimento do volume de dados (o Twitter processa diariamente, em tempo real, biliões de eventos [94]), começaram-se a revelar algumas dificuldades do Storm em lidar com tamanha quantidade de informação, fruto da sua arquitetura rígida [94]. Então, o Twitter decidiu desenvolver o seu próprio motor de processamento, o Heron [90], que é totalmente compatível com Storm, em termos de código, evitando assim reescrever todos programas.

O **Heron** foi inicialmente projetado para responder especificamente aos requisitos do Twiter mas depressa ganhou popularidade por se apresentar com algumas melhorias face ao Storm [94]. A principal diferença entre Heron e Storm foi a introdução do conceito de *job scheduler*, que é o serviço responsável tanto por alocar os recursos necessários para a execução da topologia como por começar todos os processos destinados àquela máquina [94]. Este motor está então pensado para ser fácil de desenvolvimento, suportar grandes volumes de dados, melhorar a produtividade do sistema e apresentar um desempenho mais eficiente [90].

Em relação ao funcionamento do Heron, este começa com a submissão de uma topologia (ver Figura 3.8 [94] [95]) no *job scheduler*, que por sua vez cria vários *containers*[1]. Um dos *containers* consiste na *Topology Master*, serviço responsável por monitorizar toda a topologia enquanto esta se encontrar ativa, sendo que está sempre em comunicação com o cluster *Zookeeper* de forma a ter conhecimento do estado dos componentes [95]. Os restantes *containers* criados possuem cada um, um *Stream Manager*, um *Metrics Manager* e um conjunto de instâncias Heron, que correspondem a *bolts* e *spouts*. O *Stream Manager* tem a função de definir o caminho (*routing*) que um *tuple* toma entre as instâncias da topologia [94]. Já o *Metrics Manager* recolhe métricas e indicadores sobre os processos a decorrer no seu *container* [94].

[1]Um *container*, no âmbito deste projeto, é como um pacote que pode conter código, configurações, lista de dependências, entre outros.

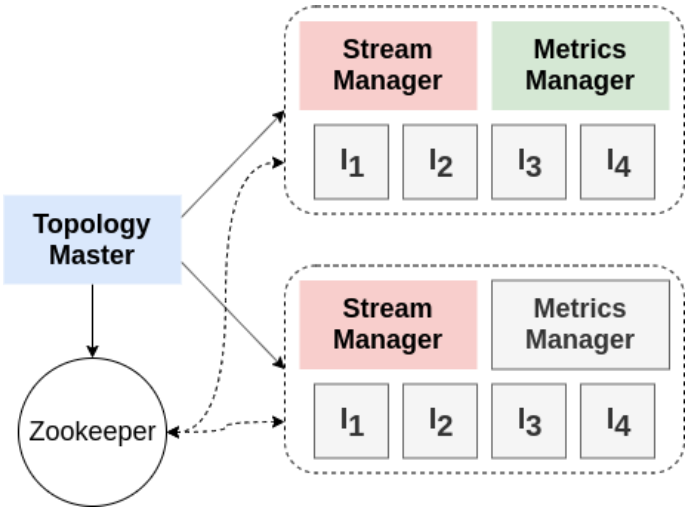


Figura 3.8: Arquitetura de uma topologia Heron

Uma das características mais importantes que, ao contrário do Storm, o Heron possui, é o suporte de *back pressure* [95]. Este termo refere-se à possibilidade de as topologias se ajustarem automaticamente em caso de um *bolt* ou *spout* falhar, isto é, sem que haja perda de informação. O Heron, como apresenta uma arquitetura modular, promove o isolamento entre topologias mas também entre componentes dentro de uma topologia. Com isto, *spouts* e *bots* são executados como instâncias separadas, o que ajuda a isolar problemas e recuperar dos mesmos [94].

Por fim, analisa-se o **Apache Flink** que, a par do Storm, é uma das *frameworks* de processamento de *streams* de dados mais populares. Este motor de processamento combina algumas das melhores características das ferramentas já descritas, como o modelo de programação do *MapReduce* (descrito em 2.3.1) apesar de possuir funções adicionais como filtragem, união ou agregação [86]. O Flink suporta também o processamento, em tempo real, de *streams* de dados recolhidos por sistemas como o Kafka [96]. Esta é portanto uma ferramenta muito poderosa, destacando-se pela rapidez de processamento e a escalabilidade da quantidade de dados que o seu sistema permite [97]. Estas características são comprovadas pela notabilidade de alguns dos seus clientes como a Uber, o Ebay e Huawei, que são empresas que lidam diariamente com grandes volumes de eventos [98].

O Flink possui um sistema de recuperação de falhas próprio, que não só reinicia a aplicação mas também recupera o estado dos componentes no momento da falha. Este sistema baseia-se em *checkpoints* de estado da aplicação. Assim, em caso de falha, a aplicação inicia a partir do último *checkpoint* validado. Esta característica resulta em fiabilidade e consistência na entrega de dados, isto é,

semântica de processamento *exactly-once* - os dados são entregues exatamente uma vez.

O modelo de programação do Flink está dividido em vários níveis de abstração mas para compreender de que forma o processamento de informação é feito, é necessário entender o que acontece no cerne da Application Programming Interface (API) (situado numa camada intermédia), deste modelo [99]. Este *core* do Flink pode ser dividido em duas API fundamentais:

- *DataSet API*, mais direcionada para conjuntos de dados estruturados e com tamanho definido, ou seja, o tipo de informação que se descreveu no processamento por *batches* em 2.3.1 [99] [100];
- *DataStream API*, sendo esta mais focada em *stream processing*, uma vez que contém as funcionalidades necessárias para o tratamento de dados em tempo real, estruturados e não estruturados [100].

Num nível superior às duas API anteriores, considera-se ainda uma *Table API*, onde as funções são similares às *queries* que se encontram na linguagem SQL como selecionar, agrupar, juntar, entre outras. Esta API tem como objetivo permitir que o utilizador execute as operações SQL sobre as *streams* de dados do Flink com menos complexidade que habitualmente essas *queries* impõe [99] [78].

Os blocos principais de um programa Flink (ver Figura 3.9 [99]) são as fontes (*sources*) de dados, um conjunto de transformações (processamento) e os destinos (*sinks*) dos dados já processados. Os operadores de transformação disponíveis, vão depender do tipo de *stream* que se está a processar (*Data Stream* ou *Data Set*). O que se vê na Figura é uma vista condensada de um programa (também denominado de *job*), porque cada um dos quatro blocos pode ser executado mais que uma vez em processos paralelos.

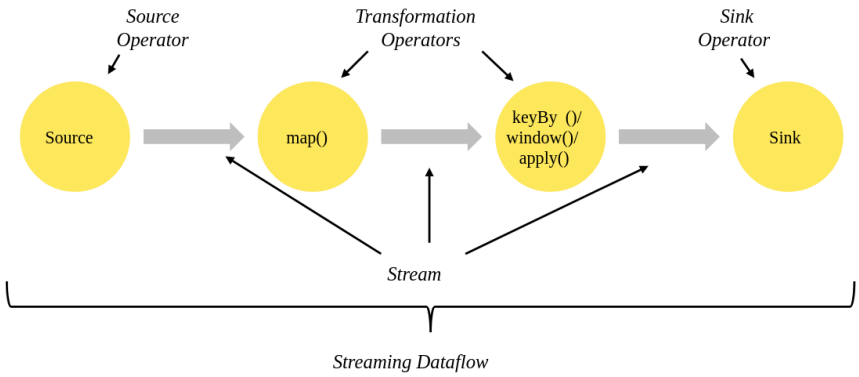


Figura 3.9: Fluxo de dados num motor Flink

Para além das funções constituintes destas API, o Apache Flink possui também bibliotecas com finalidades mais específicas como o *Gelly*, uma API que contém um conjunto de métodos e serviços para simplificar a análise de gráficos de fluxo de dados. Contudo, importa sobretudo realçar a biblioteca CEP que o Flink tem implementada, pois, no contexto do presente trabalho, confere-lhe vantagem em relação às restantes *frameworks* de processamento de *streams* em cima descritas (Heron e Storm). Esta biblioteca permite definir padrões complexos que se pretendem detetar em *streams* de eventos, uma vez que contém uma grande variedade de operações a executar sobre estes como, por exemplo, a condição *where*, onde de seguida o utilizador pode colocar uma simples regra do tipo *"velocidade >= 140"* para corresponder ao padrão, que o evento deve satisfazer [101].

Depois do utilizador submeter o código (ver Figura 3.10 [102]), este passa por um processo denominado de *client* que o compila e otimiza, tendo como resultado um gráfico, ao qual se dá o nome de *Dataflow Graph* e que tem a mesma estrutura do que está na Figura 3.9. De seguida, o *client* envia o programa para o *Job Manager*, visto como *master* quee é o processo responsável por coordenar toda a execução do programa, ou seja, define o tempo de execução de cada tarefa, acompanha o estado dos *workers* do sistema, controlo de falhas, coordenação dos *checkpoints*, entre outros [102]. O processamento de dados propriamente dito é responsabilidade de cada um dos referidos *workers*, aos quais se dá o nome de *Task Manager*. Este serviço executa então um ou mais operadores do programa que o utilizador criou, reportando o estado de cada um deles ao *Job Manager* [102] [100]. Tanto o *Job Manager* como o(s) *Task Manager(s)* fazem parte daquilo que se chama o *Runtime* do Flink.

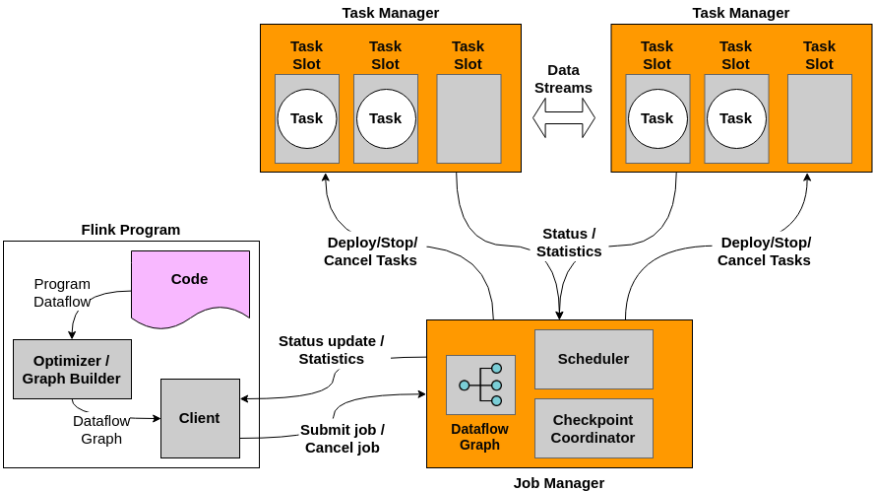


Figura 3.10: *Runtime* do Flink

Tendo já sido descrito o funcionamento de três *frameworks* de processamento de *streams* de dados - **Apache Storm**, **Heron** e **Apache Flink** - e ainda de um motor somente focado na tecnologia CEP - **Esper** - podem-se tirar as primeiras conclusões. O Esper é, de facto, a ferramenta mais completa que implementa um motor CEP com atributos que a destacam entre os demais, como a escalabilidade e grande rapidez de processamento, contando ainda com uma vasta e clara documentação. Contudo, tal é o foco nas *queries* sobre os dados, que esta *framework* precisa de outro serviço que lhe organize a informação. O Apache Storm é um exemplo de uma *framework* robusta e eficaz no que diz respeito a processar grandes volumes de dados, permitindo ainda a integração com o Esper. No entanto, uma topologia Storm é considerada um só processo, o que significa que se algum dos seus componentes falhar (*bolts* e *spouts*), a topologia não se adapta e o processo termina, o que confere a esta ferramenta uma debilidade, que pode não ser crítica uma vez que é possível executar várias topologias em paralelo. Já o Heron veio resolver esta questão da rigidez da arquitetura implementando a funcionalidade de *back pressure*, além de ter otimizado alguns processos em relação ao Storm, resultando numa *framework* mais eficiente em termos de latência e *throughput* de dados. Por último analisou-se o Apache Flink, uma *framework* poderosa capaz de competir com o Storm a nível de escalabilide e com a velocidade do Heron, que oferece uma função embutida que implementa um motor CEP, evitando assim que se tenha que integrar duas *frameworks*.

Como se viu, o Esper, apesar de ser uma poderosa solução projetada especificamente para implementar um motor CEP, onde são despoletadas ações quando os dados recebidos satisfazem alguma das regras predefinidas, o sistema não contempla a parte de organização das *streams* de dados. Uma vez que a solução proposta nesta dissertação suporta a entrada de vários tipos de dados e permite relações entre estes, o Esper só seria considerado como a *framework* a utilizar se fosse integrado com um motor de processamento de streams de dados. Daqui surgiu o interesse restantes ferramentas analisadas. Em [103] os autores exploram a viabilidade de colocar um motor Esper em cada *bolt* de uma topologia Storm.

Tendo então por base vários estudos comparativos das ferramentas de *stream processing* [104] [87] [85] [86] [91] assim como as especificações disponíveis nas respetivas páginas *Web* destas [89] [88] [90], construiu-se a Tabela 3.2 que resume a análise efetuada e que exibe as diferenças mais relevantes entre as mesmas no âmbito do projeto.

Tabela 3.2: Comparação entre Storm, Heron e Flink

	Apache Storm	**Heron**	**Apache Flink**
Formato dos Dados	*Tuple*	*Tuple*	*DataStream*
Fontes de Dados	*Spouts*	*Spouts*	Kafka, RabbitMQ, Twitter, Ficheiros locais, entre outros
Modelo de Programação	Topologia	Topologia	Programa/*Job*
Linguagem	Java, Python, Ruby	Java, Python	Java, Python
User Interface	Sim	Sim	Sim
Latência	Bom	Excelente	Excelente
Tolerância a falhas	*Daemons*	*Back Pressure*	*Checkpoints*
Implementação de motor CEP	Apenas através da integração com Esper	Apenas através da integração com Esper	Embutido

Através da consulta da tabela anterior pode-se inferir que existem algumas diferenças entre as *frameworks* analisadas, sendo que umas têm maior relevância que outras para os objetivos que este trabalho se propôs cumprir. A primeira característica é formato dos dados, que se refere à estrutura da informação que circula entre as funções de cada um dos sistemas. As fontes de dados, no contexto desta tabela, referem-se à etapa inicial dos programas de cada uma das ferramentas. Enquanto que no Storm e no Heron, existe um elemento concreto que desempenha o papel de fonte (*spout*), num programa Flink podem ser integrados diretamente vários sistemas, como por exemplo um tópico Kafka. Já o terceiro e quarto atributos dizem respeito ao modelo e linguagem de programação, respetivamente. Em relação a este último campo, é oportuno mencionar que, a nível de suporte, existe uma predominância da comunidade que usa o Java como linguagem de programação. Na tabela segue-se então uma característica que todas as ferramentas possuem, a *User Interface*. Esta componente consiste numa página onde o utilizador consegue monitorizar o estado da topologia/programa, podendo ser consultados indicadores de performance, registo de dados processados, logs, entre outros. Por fim, as últimas e mais relevantes características que se deve descortinar são a latência, tolerância a falhas e a compatibilidade com o CEP. O primeiro refere-se ao atraso sofrido pela informação, desde que chega ao motor de processamento até sair e nesta matéria tanto o Apache Flink como o

Heron apresentam um bom desempenho, superiorizando-se ao Storm. Quanto aos mecanismos de recuperação em caso de falha, considera-se que o *Back Pressure* e o sistema por *Checkpoints* são mais eficientes que a técnica utilizada pelo Storm. Finalmente, era importante saber como é que se iria implementar um motor CEP com cada *framework* e aqui o Flink suplantou as restantes ferramentas porque possui uma biblioteca já implementa que responde a esse desafio, enquanto que o Storm e Heron precisam sempre de integrar o seu sistema com o Esper.

Daqui se concluiu que a melhor ferramenta a utilizar para o módulo de processamento, onde o tema do projeto se foca, é o Apache Flink.

3.4 Armazenamento

Por fim, antes de encerrar o capítulo "Arquitetura do Problema", resta abordar a componente de armazenamento - base de dados. Apesar de esta parte não estar no âmbito do funcionamento de um motor CEP, é importante que se guarde um histórico dos eventos ocorridos ou dos alertas que se gerou.

A base de dados a utilizar num sistema como o que se pretende implementar neste projeto, deve conseguir suportar uma extensa e crescente quantidade dados mas também se deve ter em conta a variedade da estrutura da informação. Portanto um dos requisitos será a escalabilidade do suporte do dados, sendo que o outro está mais focado ainda no tema do deste trabalho, isto é, a base de dados não deve criar qualquer tipo de entropia nem atraso no fluxo de dados do sistema, devendo então ser ágil nomeadamente no momento da inserção de matéria nas suas tabelas.

Na literatura [105] [106] [107] [108] considera-se essencialmente a existência de dois tipos de base de dados: relacional e *Not Only SQL* (*NoSQL*). O primeiro é o tipo mais comum de base de dados, onde a informação é guardada em várias tabelas de forma estruturada. O acesso aos seus dados é feito através de *queries* em linguagem *Structured Query Language* (SQL) [106]. Já as bases de dados *NoSQL* caracterizam-se pela sua flexibilidade, no que diz respeito ao tipo de dados que suportam e pela rapidez com que é possível consultar o que está armazenado [108].

Com o aumento do volume de dados gerado e com o crescimento da sua complexidade (*Big Data*), a performance das bases de dados relacionais decresce pela rigidez dos seus esquemas/modelos de dados [108]. Por esse motivo, as bases de dados *NoSQL* ganharam relevância, sendo que se conhecem quatro variantes deste tipo de base de dados que diferem entre si em termos de modelo de armazenamento e consulta (*query*) dos seus dados [108] [106]:

- *Graph*, onde os dados são armazenados na forma de grafos que possuem vértices e arestas. Os vértices ou *nodes* são entidades como pessoas, objetos,

entre outros. As arestas *edges* representam a relação entre dois vértices. Estas bases de dados são particularmente úteis quando se pretende perceber as ligações e relações entre os dados em vez dos dados por individualmente [108];

- Chave-Valor (*key-value*), que não tem um esquema de armazenamento definido e a informação está dividida em duas partes, uma chave e um ou mais valores correspondentes à mesma chave. A natureza simples destas bases de dados oferece, a quem as usa, um sistema rápido e eficiente de gestão de dados [105]. O Redis [109] é um exemplo de uma base de dados que implementa este modelo;

- *Column-oriented*, onde os dados, como o próprio nome indica, se encontram organizados por colunas. Este modelo junta conceitos das bases de dados relacionais e do modelo *key-value*, isto é, numa tabela é possível ver que uma linha tem uma coluna chave e uma ou mais colunas com os valores correspondentes. Este modelo caracteriza-se pela sua notória escalabilidade [108]. Um dos exemplos de base de dados que usa este modelo é o Apache Cassandra [110];

- *Document-based*, que suporta todos os tipos de dados, estando estes armazenados em coleções de documentos. Os documentos podem conter os dados em estruturas complexas como o formato *JavaScript Object Notation*[2] (JSON) [106].

Em [105], os autores fizeram uma análise comparativa do desempenho da MariaDB, que é uma base de dados relacional e o Redis, uma base de dados *NoSQL* que implementa o modelo chave-valor. Nesta análise foram realizados testes ao desempenho destas bases de dados na execução de quatro tipos de operação: inserção, remoção, atualização e seleção. Todas as operações foram feitas para diferentes quantidades de dados. Concluiu-se que, apesar de o Redis não ter superado a MariaDB em todas as operações, é possível perceber que a flexibilidade das base de dados como Redis proporciona uma melhor performance nas operações de inserção, remoção e seleção. A base de dados relacional que foi objeto de estudo, apresentou-se mais rápida na operação de *update* dos dados quando a quantidade é maior.

Existe ainda um outro tipo de base de dados não relacional que é denominado de *Time Series Databases* (TSDB). Estas são otimizadas para permitirem o armazenamento de dados organizados em pares *timestamp/valor* [112]. Uma

[2]JSON é um formato de dados que é fácil de interpretar tanto por pessoas, como por máquinas. Um objeto JSON é uma coleção de pares chave-valor, onde o o valor pode ser uma lista ou sequência de vários valores [111]

vez que, habitualmente, estas base de dados são utilizadas para armazenar informação de sensores e outros instrumentos de medida, a sua popularidade cresceu rapidamente nos últimos anos com o surgimento e expansão do conceito de IoT. De facto, *time-series data* pode ser definido como um conjunto de dados que no seu todo representa a forma como um processo/sistema/comportamento muda ao longo do tempo [113]. Um exemplo do uso deste tipo de base de dados é uma aplicação *Web*, onde sempre que um utilizador entra na sua conta, normalmente o que é feito na base de dados é o *update* do campo *último_login* colocando o instante de tempo da última vez que o *login* daquele utilizador foi realizado. No sistema por *time-series*, a monitorização das alterações de um determinado objeto são feitas através de comandos *insert* em vez de *update*, tornando possível a deteção de tendências, padrões de comportamento, entre outros [112]. Esta característica pode ser muito útil em soluções orientadas a eventos, isto é, onde a arquitetura promove a previsão, produção, deteção de eventos.

Uma das mais recentes bases de dados projetada especificamente para lidar com *time-series data* é o **InfluxDB**. Aqui, os dados são então guardados por ordem de chegada, ou seja, organizados pelo seu *timestamp*. Esta base de dados apresenta duas características importantes: a possibilidade de gerir dados antigos através daquilo que a documentação denomina de *retention policies* [114], onde pode ser definido o nível de persistência da informação nas tabelas, sendo que se pode optar por apagar os dados que já estejam armazenados há um determinado período de tempo ou então pode-se substituir estes por um conjunto mais pequeno que os represente, através de operações de agregação. Esta propriedade é especialmente relevante em TSDB uma vez que, devido à sua forma de guardar sequencialmente os dados não havendo substituição destes, a informação armazenada escala facilmente para grandes quantidades [113]. A outra característica que confere ao InfluxDB um ganho em relação às restantes bases de dados, é o suporte de *queries* contínuas, ou seja, permite que se monitorize a entrada de uma determinada *stream de dados* [113].

Em relação à sua estrutura de dados, em todas as tabelas do InfluxDB existe uma coluna com o nome de *time* que guarda o *timestamp* do momento que o registo foi efetuado, por isso, todos as informações armazenadas podem ser identificadas pelo instante de tempo que entraram no sistema. Cada registo numa tabela, que pode conter um ou mais elementos, é denominado de *point* [112]. Cada *point* pode ser identificado pelos quatro componentes seguintes [115]:

- *time*, que como já foi referido, é uma coluna obrigatória em todas as tabelas e representa a hora e data da criação do registo;
- *fields*, que são pares chave/valor onde as chaves são *strings* que nomeiam as colunas das tabelas e os valores respetivos consistem nos dados que normalmente queremos armazenar, pelo que podem ser *strings*, números inteiros

ou decimais ou um parâmetro booleano (1 ou 0). Importa destacar que numa tabela InfluxDB existe sempre pelo menos um campo do tipo *field* e que como os seus valores podem variar muito entre si, não é aconselhável filtrar os resultados por este componente;

- *tags*, que são opcionais e consistem também em pares chave/valor, onde ambos são armazenados como *strings*. Normalmente é útil definir *tags* porque são categorizadas, o que significa que podem ser usadas como filtro nas *queries* efetuadas;
- *measurement*, cujo conceito é similar ao de uma tabela nas base de dados relacionais, sendo que, neste caso englobas as potenciais *tags*, os *fields* e a coluna *time*.

Em [113] os autores realizaram um teste de performance a três bases de dados: Apache Cassandra, MongoDB e InfluxDB, comparando os seus desempenhos a vários níveis. O Cassandra é uma tecnologia de base de dados *NoSQL* que segue o modelo *Column-oriented* anteriormente descrito conhecida pela sua robustez e capacidade para lidar com grandes quantidades de dados apresentando um sistema tolerante a falhas. Esta base de dados suporta ainda o registo de *timestamps* por cada entrada de informação. Já o MongoDB pertence também à família das bases de de dados *NoSQL*, apresentando porém uma estrutura *Document-based*. Os testes consistiram na avaliação de cada base de dados em três aspetos: *ingestion time*, ou seja, o tempo (em segundos) que a base de dados demora a efetuar o armazenamento de 300 milhões de registos; *retrieval time* consistindo na medição do tempo, em segundos, que a base de dados retorna resultados, sendo que foram realizadas *queries* com filtros diferentes; *disk usage*, em *Gigabytes*, de forma a avaliar o impacto do armazenamento de grande volumes de dados. No teste ao tempo gasto no registo de dados, o InfluxDB superou claramente as outras duas bases de dados devido ao facto de possuir processos otimizados nesses aspetos. O Cassandra consome 300 milhões de registos em aproximadamente 4 horas e meia, sendo 10 vezes mais lento que o InfluxDB. No processo de obtenção de dados, no que diz respeito à filtragem por *timestamp*, como seria de esperar graças à natureza do sistema, o InfluxDB voltou a superar as outras duas tecnologias de forma significativa. Por outro lado, quando se filtraram os resultados por uma das colunas, os resultados já foram mais equilibrados sendo que o MongoDB apresentou o melhor desempenho global. Neste aspeto, o InfluxDB, conforme expectável não reage bem às *queries* com filtros por *fields*. Por fim, em termos de espaço ocupado no disco, o InfluxDB devido às suas técnicas de compressão dos dados, revela-se um sistema muito mais eficiente no uso dos recursos de armazenamento da máquina onde encontra alojado.

Concluiu-se então que o InfluxDB, é a solução mais equilibrada e adequada para o sistema que se quer implementar no presente trabalho, já que este tipo de

soluções geram processam grandes quantidades de informação. Talvez o aspeto mais importante na escolha da tecnologia para a base de dados de um sistema de alarmística inteligente é o *ingestion time* da mesma porque o mecanismo de registo de informação não pode causar impacto na latência da solução.

3.5 Sumário

Este capítulo foi criado com o objetivo de, antes de apresentar uma proposta de solução, dar a conhecer o problema que se quer solucionar. Aproveitou-se também esta etapa para fazer um estudo comparativo das ferramentas e tecnologias que se poderão utilizar para responder aos diferentes desafios. Na primeira secção deste capítulo é exibido e descrito o cenário onde se aplicará a solução proposta, isto é, uma cidade, onde existem inúmeros sensores a funcionar como fontes de dados (cada uma com a sua estrutura e frequência de envio de informação), e o objetivo é criar um sistema de alarmística inteligente capaz de processar, em tempo real, todos os dados e gerar com o mínimo de latência possível, alertas para os destinos adequados.

No que resta do capítulo, percorre-se as funções basilares de um sistema típico de processamento em tempo real sem esquecer a particularidade da necessidade de implementar um motor CEP na solução. Por isso, começa-se por se discutir a componente de processamento de mensagens, onde são abordadas duas tecnologias diferentes, sendo que ambas seguem a metodologia *publish-subscribe*. Analisou-se então o Apache Kafka e o RabbitMQ, onde o primeiro se foca mais no conceito de tópico como unidade de software que mantém as mensagens até estas serem recebidas por um *consumer*. Já o RabbitMQ caracteriza-se por implementar uma arquitetura que respeita os fundamentos do AMQP, onde a unidade de distribuição são as *queues*. Concluiu-se que apesar do RabbitMQ apresentar uma velocidade de entrega de mensagens muito satisfatória, o Kafka também não fica atrás e tem um melhor desempenho em termos de escalabilidade, sendo por isso a tecnologia escolhida para implementar a troca de mensagens entre os módulos do sistema.

A Secção 3.3 centra-se na componente mais importante do trabalho, ou seja, no processamento dos dados e integração de um motor CEP na mesma. Descrevem-se aqui um total de quatro *frameworks*, sendo que uma delas, o Esper, difere das restantes na medida em que é focada somente no motor CEP. Esta, apesar de implementar um sistema muito robusto, o mesmo não contempla a organização da informação onde vai aplicar as regras/*queries*. Daqui surgiu a necessidade de explorar ferramentas de processamento de dados como o Apache Storm, Heron e Apache Flink. Analisando cada uma delas concluiu-se que qualquer uma conseguiria cumprir com o pretendido no que diz respeito a este projeto. Contudo, numa perspetiva de escalabilidade, existem diferenças de performance a ter conta

como por exemplo a forma como a *framework* reage à falha de um dos componentes. Numa situação dessas, é importante perceber a dimensão das perdas ou quanto tempo demora a recuperação. O Flink possui uma particularidade muito interessante no âmbito desta dissertação, que é o suporte do motor CEP através de uma biblioteca própria, ou seja, não há necessidade de integração com o Esper ou outra ferramenta. Tendo em conta todas estas premissas, a *framework* escolhida para implementar o motor de processamento é o Flink.

Por fim, aborda-se a questão do armazenamento de informação, que, não sendo a característica mais relevante para o cumprimento dos objetivos propostos, considera-se que é importante realizar o registo de um histórico. Por isso, na secção 3.4 comparam-se dois tipos de base de dados: relacional e *NoSQL*. A primeira consiste no tipo de armazenamento tradicional onde as *queries* efetuadas obedecem à linguagem SQL. Já o segundo tipo caracteriza-se pela rapidez de resposta assim como a flexibilidade denotada no tipo de dados que suporta. À semelhança do que acontece nas secções 3.2 e 3.3, onde se apresenta uma conlcusão final sobre que ferramenta se vai utilizar para a respetiva componente, aqui na secção de Armazenamento é evidenciado também que a base de dados escolhida é o InfluxDB, sustentando esta decisão com a comparação desta, com outras duas tecnologias *NoSQL* muito populares e consequentemente muito capazes.

Capítulo 4

Implementação

4.1 Solução Proposta

Como já foi referido, reunindo toda a informação necessária para preencher tecnologicamente cada etapa do sistema, foi possível sustentar as decisões efetuadas assim como desenhar uma arquitetura que cumprisse todos os seus requisitos. O sistema suporta a entrada de dados de diferentes fontes, processando-os em tempo real de forma a gerar alertas para as parte interessadas, numa questão de milissegundos. Na Figura 4.1 estão representados os módulos da arquitetura proposta, assim como as ligações entre si.

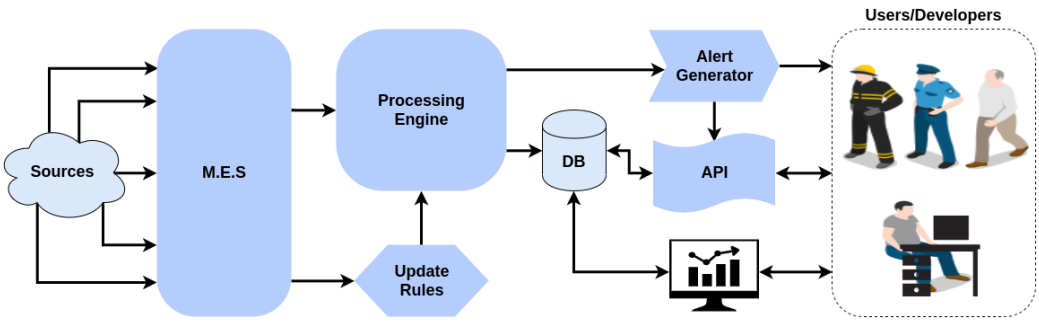


Figura 4.1: Arquitetura da solução proposta

Acredita-se que esta arquitetura apresenta a robustez necessária para lidar com a quantidade de dados que pode surgir no sistema, assim como a eficiência que garante fiabilidade ao produto. O sistema proposto recebe dados de várias fontes como tráfego, ambiente, meteorologia, redes sociais, parques de estacionamento, etc. sendo estes organizados de seguida pelo módulo ao qual se deu o nome de *Monitoring Event Source* (MES) que organiza as *streams* de dados pelo tipo,

enviando-as para o bloco de processamento CEP e para um módulo que controla as regras executadas, uma vez que, na eventualidade de apenas se estarem a receber dados sobre tráfego e ambiente, as regras relativas à meteorologia não precisam de estar a criar mais complexidade no sistema. O módulo do motor CEP compara as regras, previamente definidas, ao eventos que recebe, gerando alertas quando uma dessas regras é correspondida. Depois, os resultados/alertas do processamento têm duas finalidades: o armazenamento na base de dados InfluxDB e o módulo *Alert Generator*. Este último consiste num serviço que, mediante a severidade do alerta (característica definida no *tuple* que o motor de processamento produz), envia-o para diferentes destinos. Quanto ao armazenamento na base de dados, este deve ser feito com a maior celeridade possível de forma a não causar qualquer atraso ou entropia no sistema. Foi desenvolvida também, uma REST API que se comporta como interface entre utilizadores ou *developers* e a base de dados.

REST, acrónimo de *Representational State Transfer*, é um estilo arquitetónico que estimula a comunicação e a escalabilidade dos serviços *Web*, guiando-se por vários princípios: separação do que é a interface do cliente e o que são os recursos do servidor, garantindo assim portabilidade do cliente por diferentes plataformas e escalabilidade do sistema (*client-server*); cada pedido do cliente para o servidor deve conter toda a informação necessária para o mesmo ser corretamente interpretado, uma vez que o lado do servidor não guarda o estado da conexão (*stateless*); introdução de algum elemento na resposta a um pedido que indique se aqueles dados da mesma são passíveis de ser guardados em cache (*cacheable*); interface uniforme, na medida em que possui identificação dos recursos por um *Uniform Resource Identifier* (URI), representação do estado dos recursos através dos formatos JSON ou XML e uso de verbos HTTP (*GET*, *POST*, *DELETE*, entre outros) para aceder aos recursos; divisão do sistema em níveis, isto é, entre o cliente, que faz o pedido sobre o estado de um recurso, e o servidor, que atende esse pedido, devem existir camadas de segurança, cache, entre outras [116] [117].

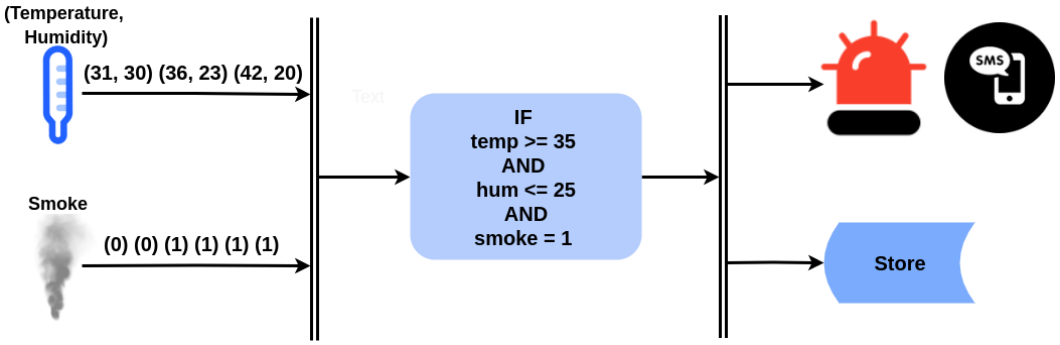
Por fim, foi feita uma integração da base de dados com uma ferramenta de visualização gráfica dos dados que estão armazenados. Esta, com uma interface muito intuitiva, permite-nos fazer *queries* à base de dados e visualizar os resultados através de gráficos com formatos variados.

No sub-capítulo seguinte vão ser apresentados dois possíveis casos de uso deste sistema, de forma a perceber qual é, na prática, o papel dos blocos principais da arquitetura representada pela Figura anterior.

4.1.1 Conceção de um Alerta - Use Case

A arquitetura que aqui se propõe pode ser aplicada a muitos contextos diferentes, uma vez que a entrada do sistema suporta dados das mais variadas fontes,

tendo apenas que se adaptar as regras ao contexto. Como o próprio título do trabalho indica, o *use case* escolhido é uma cidade, ou melhor dizendo, uma cidade inteligente. Este caso de uso inicia-se com a receção de dados, seguindo-se o seu processamento. O resultado desse processamento é armazenado e, no caso de ser um alerta assinalado com uma severidade alta, também é comunicado em tempo real aos utilizadores interessados. De seguida, é ilustrado o fluxo de acontecimentos que ocorrem no sistema até ser gerado um resultado, que pode ser o que neste exemplo se chama *Aviso* - acontecimento com determinado nível de severidade que é provável acontecer - ou um *Alerta* - acontecimento que já está a decorrer. Ao todo, considera-se cinco tipos de fonte de dados: quatro sensores - temperatura/humidade, fumo, velocidade nas estradas e lotação dos parques de estacionamento - e ainda a rede social *Twitter*, o que já demonstra a diversidade de fontes de dados que o sistema consegue suportar. A Figura 4.2 retrata o exemplo de um alerta a ser gerado a partir do processamento de dados que estão a ser enviados, de forma contínua, por sensores de temperatura/humidade e de fumo, onde o objetivo é perceber se um incêndio está a ocorrer. Para o conseguir, devem ser definidas regras contendo *thresholds* que quando atingidos ou ultrapassados, é disparado um alerta.



z

Figura 4.2: Exemplo da produção de um evento *Alerta*

Uma das premissas do sistema idealizado é relacionar diferentes sensores (fontes), para melhor sustentar a decisão por parte do processamento, uma vez que podem ocorrer diferentes combinações de eventos e o sistema precisa de garantir alguma filtragem daqueles que, efetivamente não são graves. No exemplo apresentado, um alerta de incêndio é enviado para as partes interessadas se os valores de temperatura, humidade e fumo ultrapassarem os limites (*thresholds*) definidos. Já a Figura 4.3 demonstra um exemplo de um alerta diferente do anterior na medida em que, a partir dos dados provenientes destas fontes (*twitter*, tráfego

e parques de estacionamento), podem ser gerados eventos de múltiplos tipos. Neste exemplo estabelecem-se relações, para uma determinada localização, entre os dados da rede social Twitter, nomeadamente, os *hashtags*, dados dos sensores que monitorizam a lotação dos parques de estacionamento e os sensores do tráfego rodoviário. Uma regra que relacione estes três tipos de dados pode ser necessária num contexto onde exista algum concerto ou evento desportivo, que faça com que haja uma perturbação do normal funcionamento de uma cidade.

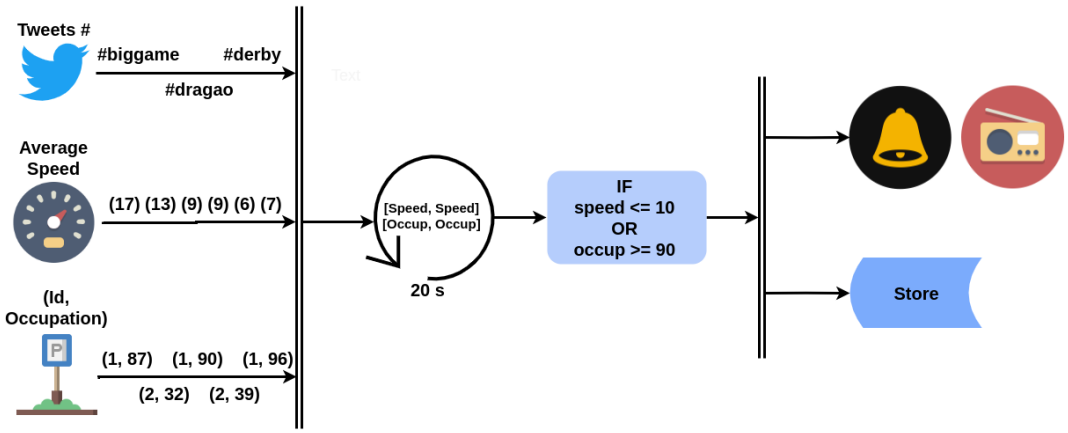


Figura 4.3: Exemplo da produção de um evento *Aviso*

Também neste exemplo, há uma diferença no processamento da informação relativa à lotação dos parques e da velocidade nas estradas, uma vez que estes dados não são analisados elemento a elemento, mas sim em janelas de tempo de vinte segundos. Por outras palavras, considerando o exemplo da velocidade, durante vinte segundos testa-se, para a mesma rua, se a velocidade for sempre inferior ao limite definido. Tal como mencionado, pode-se processar estes dados com finalidades diferentes:

- Alertar os condutores sobre o congestionamento do tráfego em determinadas vias, de forma a que estes possam alterar as suas rotas. Esta solução traz importantes benefícios ambientais já que contribui para o aumento da fluidez do trânsito, logo, menos gases poluentes são produzidos pelos veículos;
- Informar os condutores sobre os parques de estacionamento com mais lugares livres que estejam localizados perto do local do evento desportivo.

Dependendo da quantidade de fontes de dados diferentes, o sistema pode ser otimizado com o objetivo de produzir alertas cada vez mais precisos assim como

para ter a oportunidade de criar um histórico mais detalhado sobre os alertas gerados. De facto, a abordagem proposta permite lidar com streams de dados que estão sempre a mudar, tanto na quantidade como na sua estrutura, pelo que possui a capacidade de se adaptar e de integrar vários eventos de uma vez. Como resultado, a solução desenvolvida está forçosamente preparada para futuras integrações com outras fontes.

4.2 Entrada de Dados

No contexto do presente projeto, a entrada de dados consiste nos mecanismos que garantem a entrega da informação, proveniente dos sensores ou outras fontes, ao módulo de processamento. Percebe-se portanto, a importância deste componente no funcionamento do sistema, tendo sido por isso que se dedicou a Secção 3.2 ao tema, onde se compararam duas tecnologias de processamento de mensagens - Apache Kafka e RabbitMQ. Quando se projeta um sistema de alarmística inteligente, é fundamental que o mecanismo que troca informação entre serviços seja fiável e que apresente uma latência muito reduzida pelo que a decisão sobre que tecnologia usar deve estar bem sustentada. Em relação à solução implementada, como já foi referido, desenvolveu-se um sistema desde raiz por isso, tendo a primeira etapa consistido em escolher pelo menos duas fontes de dados relacionáveis.

A escolha acabou por recair em dados de tráfego rodoviário e dados de previsão meteorológica. Para o primeiro recorreu-se à plataforma **Here**, que possui um espaço para programadores [118], onde é possível aceder a várias APIs que oferecem informações sobre os mais variados contextos como mapas 2D, mapas de calor, localizações de parques, empresas e atrações, informações de trânsito e acidentes, etc. Todas estas interfaces seguem uma interface REST, logo, para obter as informações desejadas é necessário construir um pedido GET cujo *Uniform Resource Locator* (URL) deve especificar o recurso que se quer aceder. Por exemplo, quando se quer aceder à página principal do Google, através de um *browser*, faz-se um pedido GET com o URL *https://www.google.com*.

Na Figura 4.4 mostra-se o URL do pedido que retorna, em formato JSON, os dados do tráfego do centro da cidade do Porto, em tempo real. A resposta do pedido contém vários indicadores sobre cada rua da localização escolhida como a velocidade média, índices de qualidade de condução, nível de confiança nos valores retornados, sentido do trânsito, entre outros.

Figura 4.4: URL do pedido para obter dados sobre o tráfego na cidade do Porto

A API de tráfego da plataforma Here - *Traffic API* - o URL é constituído por 6 elementos:

- *Base* - **https://traffic.api.here.com** - Este elemento é fixo porque identifica a API do tráfego;
- *Path* - identifica o caminho para o recurso;
- *Resource* - identifica o recurso ao qual se quer aceder, sendo que esta API possui quatro recursos: *tiles*, *incidents*, *flow* e *flowavailability*;
- *Addressing Scheme* - filtra os resultados por área geográfica, podendo ser definido de diferentes formas (coordenadas geográficas, coordenadas cartesianas, entre outras);
- *Application Code* - código único que cada programador recebe quando se regista no portal;
- *Application Id* - outra identificação única que cada programador recebe no ato do registo.

Já a informação sobre a previsão meteorológica foi obtida através da plataforma OpenWeather que, à semelhança do Here, possui uma API onde é possível retirar dados atuais, históricos e previsões sobre a meteorologia. Para o presente trabalho interessa obter apenas as previsões. Para o conseguir, deve-se aceder ao recurso *forecast* que retorna a previsão meteorológica de três em três horas a partir do momento que se efetua o pedido até aos próximos cinco dias. O URL do pedido efetuado para esta API é apresentado na Figura 4.5. Este contém o domínio base da API, seguindo-se o caminho para o recurso e o próprio recurso. Em termos de parâmetros, foram colocados três: o filtro de localização onde se definiu que se queriam obter os resultados da previsão para a cidade do Porto; o código de aplicação que é único, sendo gerado quando o utilizador se regista na plataforma; e por último o filtro das unidades que define a unidade de medida dos valores de temperatura.

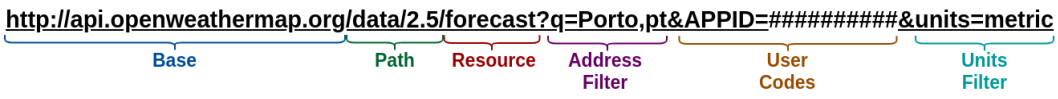


Figura 4.5: URL do pedido para obter dados sobre as previsões meteorológicas na cidade do Porto

A resposta deste pedido contém, para cada intervalo de três horas durante os próximos cinco dias, informação sobre a direção e velocidade do vento, temperatura em graus Celsius, pressão, humidade relativa e nebulosidade, apresentando os dados em formato JSON.

Por fim, tendo já duas fontes de dados (tráfego e meteorologia), fez-se o tratamento desta informação retirando apenas os parâmetros necessários. Para cada uma das fontes de dados foram então construídas mensagens cujos formatos são facilmente interpretados pelo módulo para onde são enviadas. A mensagem relativa aos dados de tráfego consiste em três elementos separados por dois pontos: o primeiro é aquilo a que se deu o nome de contexto, ou seja, tráfego; o segundo é o nome da rua e o terceiro é a velocidade média nessa rua. Já a mensagem da meteorologia contém cinco elementos: o primeiro é o contexto, que neste caso será meteorologia; o segundo é a cidade à qual os dados se referem; o terceiro é data e hora daquela previsão; o quarto é o nível de humidade; por fim, o último elemento é a velocidade do vento.

Este serviço, que foi escrito em Python, está continuamente a ser executado, isto é, quando todas as mensagens sobre o tráfego das ruas do centro do Porto e a previsão meteorológica para os próximos cinco dias, forem enviadas, o sistema volta a fazer os mesmos pedidos às respetivas API e volta a enviar as mensagens. Como não se teve acesso a sensores reais, que seriam as fontes de dados idealizadas, este foi o método encontrado para os substituir, trabalhando igualmente com valores verdadeiros. Estas mensagens são enviadas para módulo, ao qual se chamou de *Monitoring Event Source* (MES) (ver arquitetura proposta em 4.1), através de um tópico Kafka para onde as mensagens são produzidas. Como se vai ver na secção seguinte, no MES foi criado um consumidor para esse tópico.

4.2.1 Arquitetura do Módulo do Monitorização de Entrada de Dados

Tal como foi dito, este é o módulo que funciona como interface entre as fontes de dados e o processamento. Para além desta ponte, este módulo também comunica com a componente de controlo de regras/topologias. A Figura 4.6 demonstra o princípio de funcionamento desta componente, que consiste, numa primeira fase, na criação de um *consumer* Kafka que subscreve o tópico *main-source*, tópico para o qual as mensagens, provenientes das fontes, são enviadas. Tendo sido criado o consumidor que subscreve o tópico mencionado, este serviço entra num ciclo onde, cada mensagem recebida por esse *consumer*, é dividida em elementos individuais, sendo que o principal objetivo é perceber o contexto de cada mensagem (primeiro elemento), isto é, se se trata de um evento de tráfego, meteorologia ou outro. De seguida e considerando que o primeiro elemento é igual a "contexto", através de um *producer* Kafka, os elementos são enviados pela mesma ordem que chegaram, para dois tópicos distintos:

- *jar_update*: Este é o tópico que vai ser consumido pelo módulo de controlo de regras;

- *contexto_processed*: O nome deste tópico varia portanto, com o contexto da mensagem. Por exemplo, se o primeiro elemento da mensagem recebida é *traffic* (tráfego) então os elementos são enviados para o tópico *traffic_processed*.

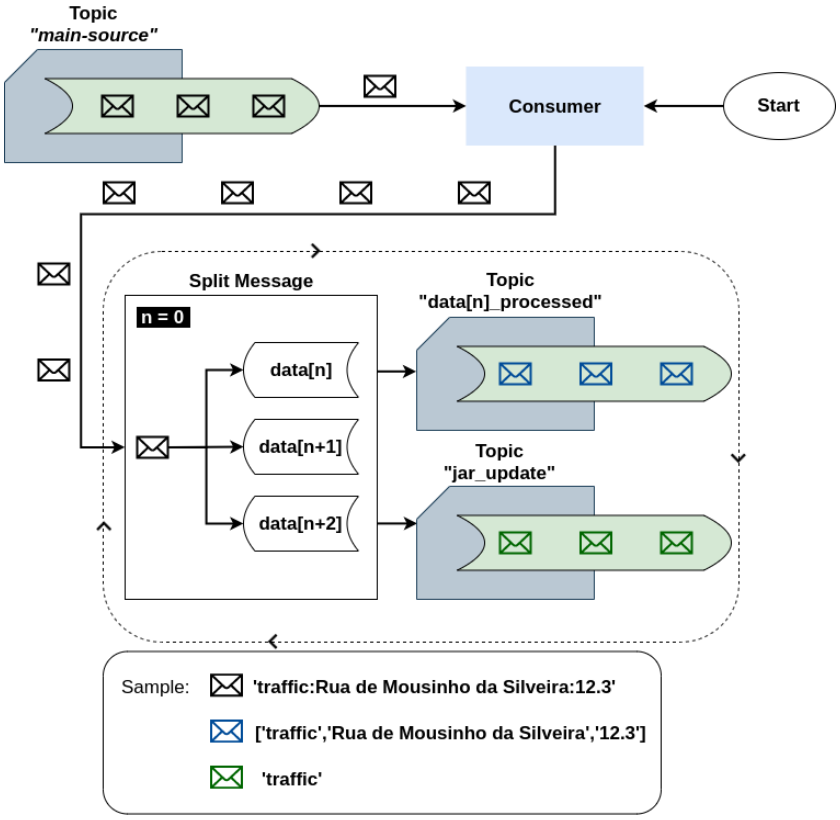


Figura 4.6: Funcionamento do módulo MES

O módulo MES é então responsável por esta filtragem da informação oriunda das fontes de dados. Mais tarde, pela descrição do módulo do motor CEP, onde o processamento é executado, irá ser percetível de que forma esta metodologia contribui para a fluidez e eficiência do sistema. De seguida, é explicada a componente que desempenha o papel de processador de dados, sendo considerada a parte mais importante do sistema: implementar um motor CEP, que é o foco deste trabalho.

4.3 Módulo de Processamento de Dados

Na secção 3.3 foram descritas as soluções estudadas para responder a um dos desafios do presente projeto, isto é, implementar um motor CEP que suportasse diferentes fontes de dados, gerando alertas em tempo real. Nesta descrição, foram então comparadas as principais características, vantagens e desvantagens de quatro *frameworks* - Esper, Apache Storm, Heron e Apache Flink - chegando-se à conclusão que a ferramenta mais eficiente e adequada para o sistema que aqui se implementa é o Flink.

Nesta secção, é explicado o funcionamento dos programas Flink desenvolvidos, sendo também ilustradas graficamente as etapas que os dados passam até ser(em) gerado(s) o(s) alerta(s). Antes do término desta secção, ainda se destaca a biblioteca CEP do Flink, apresentando-se o princípio do seu funcionamento.

Apesar de terem sido desenvolvidos mais do que um programa Flink, aqui vão relatados os passos para a implementação de um deles, uma vez que todos têm uma estrutura idêntica, sendo que esta questão vai ser totalmente clarificada ao longo desta secção e da próxima. Como se viu em 3.3, é possível submeter vários *jobs* para o *Job Manager* executar.

Os programas Flink foram desenvolvidos em Java, principalmente porque, a nível de suporte na página Web da *framework*, a maior parte do conteúdo está escrito nessa linguagem. O primeiro passo foi criar o projeto e para isso utilizou-se a ferramenta **Maven** [119] que permite construir e gerir qualquer projeto baseado em Java. Tal como já foi dito, nesta sub-secção são as explicadas as etapas para a criação de uma das topologias desenvolvidas e, para servir como exemplo, foi escolhido o programa que processa os dados de tráfego rodoviário.

Começou-se então por construir a estrutura do projeto através do comando: *mvn -B archetype:generate* seguido de três argumentos:

- *-DarchetypeGroupId=dcosta*, para identificar unicamente o grupo ao qual o pacote que vai ser gerado, pertence;
- *-DgroupId=dcosta*, para identificar unicamente o grupo que está a criar o projeto;
- *-DartifactId=traffic-topology*, para identificar unicamente o arquivo que vai ser gerado.

Uma vez executado este comando, é criado um diretório com o nome de *traffic-topology* que contém a estrutura padrão de um arquivo gerado pelo Maven, como se mostra na Figura 4.7. No seu conteúdo está incluída a árvore do projeto, constituída pelo ficheiro *pom.xml* e por um diretório - *src* - onde se coloca o

código fonte na pasta *main/java/dcosta* e, se necessário, o código de testes em *tests/java/dcosta*.

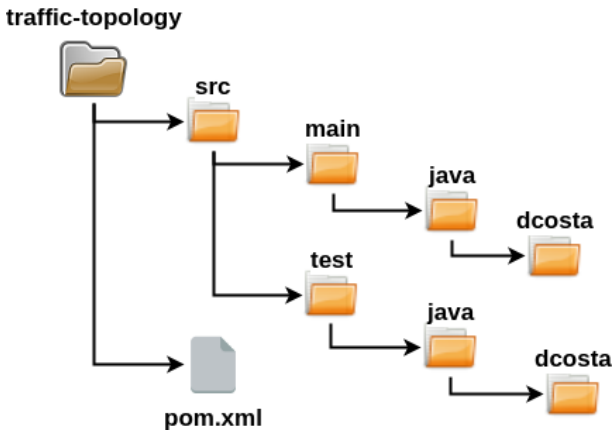


Figura 4.7: Árvore do projeto construído com o Maven

O ficheiro *pom.xml* é muito importante pois é nele que se coloca toda a informação necessária para que o Maven, quando o código estiver completo, gere o ficheiro *Java ARchive* (JAR) que é submetido no *Job Manager* do Flink. Neste ficheiro escreveram-se todas as dependências que o código precisa para ser compilado e executado.

Tendo a estrutura adequada para desenvolver um projeto em Java, começou-se então a escrever o código do programa Flink responsável pelo processamento dos dados relativos ao tráfego rodoviário. Este programa reproduz um fluxo de dados padrão do Apache Flink, que foi descrito na Secção 3.3, isto é, foram desenvolvidas as funções de receção de informação, tratamento desta e envio dos resultados (eventos gerados) para os módulos adequados. A arquitetura deste *job*, é exibida na Figura 4.8. O programa - *TrafficAnalysis.java* - contém uma *stream* de dados, que será a sua fonte de informação, através da subscrição do tópico Kafka *traffic-processed*, onde se encontram as mensagens relativas ao tráfego do centro da cidade do Porto. De relembrar que no caso dos dados do tráfego, cada mensagem possui três campos: contexto (tráfego), nome da rua e velocidade naquela rua. Foi também criado também o padrão que se quer detetar nos eventos que estão a ocorrer, ou seja, uma regra que é ativada sempre que o valor da velocidade é inferior a 10,0 km/h.

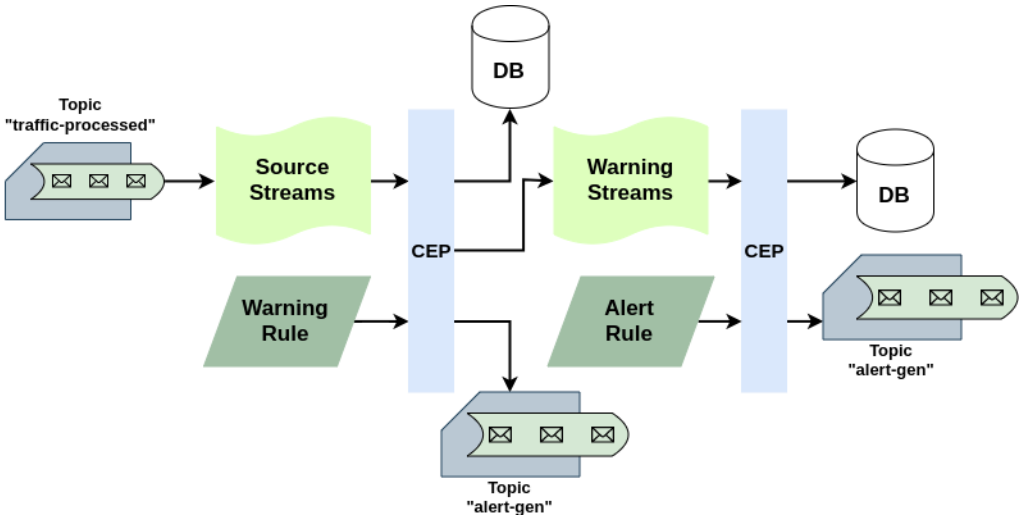


Figura 4.8: Arquitetura do programa Flink de processamento dos dados de tráfego

De seguida, a biblioteca CEP do Flink é utilizada precisamente para comparar os dados que se recebe do tópico Kafka com o padrão definido. Quando são produzidos resultados no processamento neste primeiro motor CEP, considera-se que estes são do tipo *Warning* e têm severidade média. Estes resultados e a sua severidade são armazenados na base de dados e enviados para o tópico Kafka *alert-gen*. Segue-se a declaração de uma nova regra, que tenta detetar, num intervalo de vinte segundos, a produção de dois eventos do tipo *Warning*. Logo, para usar esta regra sobre os dados gerados pelo processamento anterior, é implementado um segundo motor CEP que aplica este padrão filtrando os dados pelo nome da rua. Por fim, se o padrão for detetado, é gerado um evento do tipo *Alert* (severidade alta), sendo este enviado para o tópico *alert-gen* e guardado na base de dados.

Em termos de código, destaca-se a classe onde se executa a função principal (*main*), cujo princípio de funcionamento foi já explicado neste secção. Para melhor compreender a forma como o Flink desempenha as funções presentes na Figura 4.8, é necessário introduzir-se alguns conceitos próprios desta *framework*. Essencialmente importa perceber a classe *DataStream*, que consiste numa coleção de dados que pode e é, normalmente, ilimitada. Esta coleção é inicialmente criada quando se lhe adiciona uma fonte através da operação *addSource*, sendo que depois, novas classes *DataStream* podem ser geradas a partir da primeira aplicando transformações nesta. O blocos, sombreados a azul, do esquema da Figura 4.9 representam as etapas básicas de um programa Flink e foram incluídas na classe *TrafficAnalysis.java* (ver Anexo A). A base de todos os programas é o ambiente

de execução por isso este deve ser criado através do método *getExecutionEnvironment()*. De seguida constrói-se a primeira coleção de dados adicinando uma que, no caso deste *job*, é um tópico Kafka. Tendo a primeira coleção de dados, pode-se transformá-la aplicando funções como por exemplo o *map* - recebe uma classe do tipo *DataStream*, converte os dados num formato diferente e retorna uma *DataStream* com esses dados convertidos. Quando se obtém o estado final pretendido da coleção de dados, deve ser escolhido um sitío para onde enviar a mesma. Esta operação é realizada através da associação da função *addSink* à classe *DataStream*. Por fim, o ambiente de execução criado no início do programa é executado.

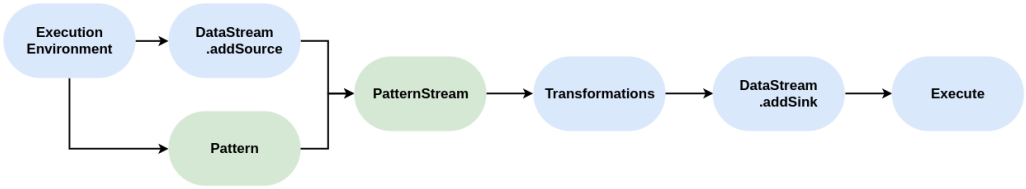


Figura 4.9: Blocos de código constituintes da classe *TrafficAnalysis.java*

Já os blocos sombreados com a cor verde, no esquema anterior, não fazem parte de um programa Flink típico mas no contexto da componente de processamento deste projeto, são muito importantes como vai ser explicado na sub-secção seguinte.

Antes de se passar à explicação da função CEP implementada, importa referir que, concluído todo o código necessário à criação da topologia/*job* de análise ao tráfego, compilou-se o mesmo através da execução do comando Maven *mvn clean install* no diretório onde se encontram o ficheiro *pom.xml* e a pasta *src*. Este comando, se não existirem erros no código, irá gerar um diretório denominado de *target* que contém um ficheiro JAR com todos os módulos, funções e dependências do projeto. Este é o ficheiro que consegue ser interpretado pelo *Job Manager*, responsável por implementar a topologia.

4.3.1 Função CEP

O Flink oferece uma API [120] que permite definir padrões complexos que se querem detetar em coleções de dados - *Pattern API*. Pode-se definir um padrão para detetar eventos que ocorram determinado número de vezes ou podem ser definidas condições, que os eventos devem satisfazer para esse padrão ser detetado. Estes padrões são definidos no Flink através da classe *Pattern*.

Tendo sido especificado o padrão pretendido, aplica-se aos dados onde se procura detetar potenciais correspondências. Esta função de associação da classe *Pattern* com a classe *DataStream* é realizada através da função CEP que cria uma classe denominada de *PatternStream*. Na Figura 4.10 é mostrado o extrato de código onde se implementa um motor CEP no *job* de análise do tráfego. O que se vê na linha 116 desta porção de código é a classe *PatternStream* a ser definida, dando-lhe o nome de *tempPatternStream*, onde a função CEP é chamada. Como argumentos desta função colocaram-se então a fonte de dados utilizada - *inputEventStream* (linha 117) - e o padrão que se quer detetar - *lowSpeedRule* (linha 118) - isto é, sempre que a velocidade for inferior a 10,0 km/h.

```
116         PatternStream<MonitoringEvent> tempPatternStream = CEP.pattern(
117                 inputEventStream,
118                 lowSpeedRule);
```

Figura 4.10: Extrato de código onde se implementa a função CEP do Flink

Como foi demonstrado em 4.3, para além de serem armazenados e enviados para um tópico Kafka, os resultados deste motor CEP são ainda utilizados como um dos argumentos do motor CEP seguinte. Sintetizando, é através da utilização desta função que se consegue cumprir um dos grandes objetivos do presente trabalho, ou seja, a implementação de um motor CEP no processamento de dados em tempo real. De seguida, vai ser descrito o bloco *Update Rules* da arquitetura geral do sistema (ver Figura 4.1), onde se explica a forma como o sistema reage à entrada de fontes de dados diferentes, contribuindo assim para o cumprimento de mais um objetivo proposto - suportar diferentes tipos de dados.

4.4 Submissão e Atualização de Topologias

Nesta secção é explicado de que forma é que o Flink permite a submissão de uma topologia para que possa estar em *runtime* (ver Figura 3.10 em 3.3). Tendo o programa e as duas dependências num ficheiro (arquivo JAR), é possível submetê-lo através de dois modos distintos: manualmente, cujo processo é feito pela interface gráfica do Flink ou ainda com recurso à API de monitorização que a *framework* possui. Depois desta introdução ao modo como se submete um *job*, demonstra-se o funcionamento do módulo de controlo de topologias/regras, que é responsável por atualizar continuamente as topologias que estão ativas, garantindo assim uma maior eficiência no uso dos recursos de processamento.

Relativamente à interface gráfica, esta apresenta-se como uma ferramenta muito útil para realizar um controlo rigoroso sobre todos os parâmetros do sistema

de processamento, como por exemplo características tanto do *Job Manager* como do(s) *Task Manager(s)*. A *framework* nomeia esta interface de *Apache Flink Dashboard* e a sua página inicial pode ser visualizada na Figura 4.11. Do lado esquerdo da referida Figura é possível seis separadores:

- *Overview*: nesta página, como se é capaz de comprovar pela Figura seguinte, são apresentadas as informações chave sobre o atual estado do sistema Flink: número de *Task Managers*; número total de *Task Slots* e quantas estão disponíveis; dados sobre os programas que estão a ser executados assim como daqueles que já terminaram. Na Figura 4.11 é percetível que duas topologias foram submetidas no sistema, sendo que uma falhou e outra foi cancelada. O número total de *Task Slots* é igual às que estão disponíveis precisamente por não estarem topologias no estado *running*;

- *Running Jobs*: neste separador, como o próprio nome indica, apresenta-se uma lista das topologias ativas naquele momento, cujos parâmetros são o instante de tempo que a topologia entrou no sistema, o instante que esta foi terminada, o tempo decorrido desde a entrada da topologia, a identificação do *job*, o estado de cada tarefa da topologia e o estado global da topologia. Se pretendido, pode-se aceder a mais detalhes de cada programa fazendo um clique sobre a topologia desejada. Esta possibilidade pode ser útil no caso de uma topologia falhar pois consegue-se perceber o erro que causou esse término. Ainda neste menu mais específico de cada *job*, é possível cancelar a sua execução;

- *Completed Jobs*: aqui são apresentados todos os programas que terminaram através de uma lista com uma estrutura similar à do separador anterior. Podem então ser consultados mais detalhes sobre cada topologia falhada.

- *Task Managers*: neste separador podem ser consultadas as características do(s) *Task Manager(s)* presentes no sistema. São mostrados parâmetros como o número de processadores, utilizados pelo Flink, da máquina onde o sistema está a ser executado, memória física disponível assim como que quantidade da mesma está a ser consumida. Estas podem ser informações importantes quando se tenciona estudar o gasto a nível de recursos que sistemas como este pode causar;

- *Job Manager*: já nesta página são apresentados parâmetros relativos ao *Job Manager* como já acontece no separador anterior, com a diferença que aqui, é possível aceder a um separador de *Logs*, onde são registadas todas operações efetuadas por este componente, sendo importante para passar de uma abstração do *modus operandi* do mesmo, como o esquema da Figura 3.10 em 3.3, para uma situação real;

- *Submit new Job*: finalmente, aqui está disponível a opção de submeter programas/*jobs* no sistema. Este processo é feito em duas etapas: primeiro seleciona-se, entre os ficheiros locais da máquina, o arquivo JAR pretendido, sendo este enviado para uma lista criada nesta página. Segue-se por fim, a submissão do programa a partir da seleção do mesmo na referida lista.

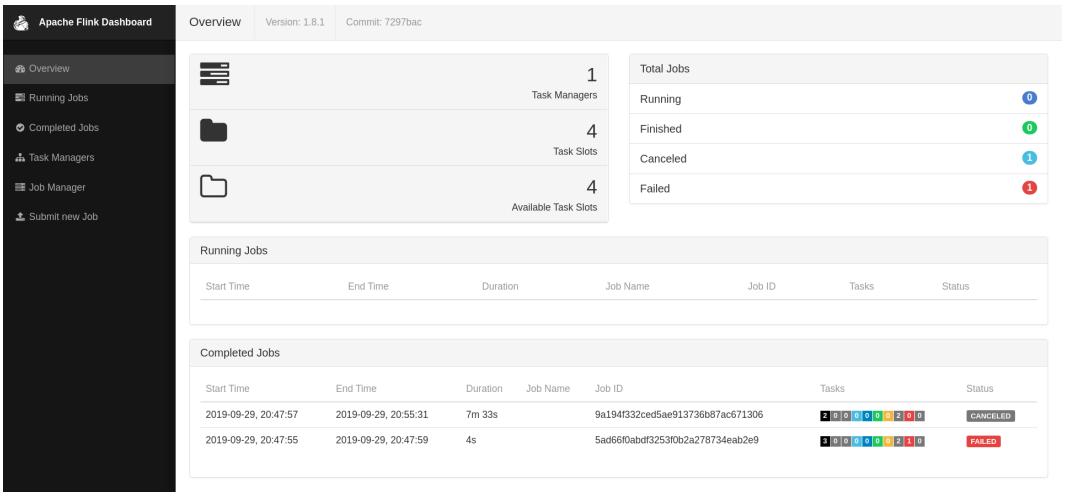


Figura 4.11: *Screenshot* da página inicial da interface gráfica do Flink

Apesar desta interface ter sido útil nas etapas iniciais do desenvolvimento, um dos objetivos desta dissertação é construir uma solução o mais automatizada possível, isto é, todos os processos que sejam passíveis de automatização devem ter essa característica. Então, um dos mecanismos que não necessita de ser realizado manualmente é a submissão de topologias, que englobaria sempre a interação do utilizador com esta interface. Por esta razão, optou-se pela segunda hipótese revelada no parágrafo introdutório desta secção, ou seja, a utilização da *Monitoring REST API* [121] do Flink. Esta API permite que o utilizador consulte o estado do sistema, desde estatísticas das topologias, parâmetros do *Job Manager* bem como dos potenciais *Task Managers*, estado de cada tarefa, submissão ou cancelamento de programas, etc., sendo basicamente possível, realizar todas as operações que a *dashboard* suporta mas, neste caso, em vez de os resultados serem mostrados graficamente, as respostas são dados em formato JSON. Como já se verificou, esta API apresenta uma interface REST, por isso, todas as ações atrás enumeradas devem ser feitas com recurso a pedidos HTTP. A Figura 4.12 representa um exemplo de um pedido HTTP que pode ser realizado à API de monitorização. Aqui o cliente, que é a máquina onde o pedido é feito, executa um pedido **GET** com o URL *http://(IP):8081/jobs* - onde no *IP* se coloca o

endereço onde o sistema Flink está alojado, 8081 é a porta utilizada por definição e o *jobs* é o *endpoint* acedido. Na resposta deste pedido é retornado um resumo de todas as topologias assim como o seu estado atual. Esta resposta vem, como referido, no formato JSON.

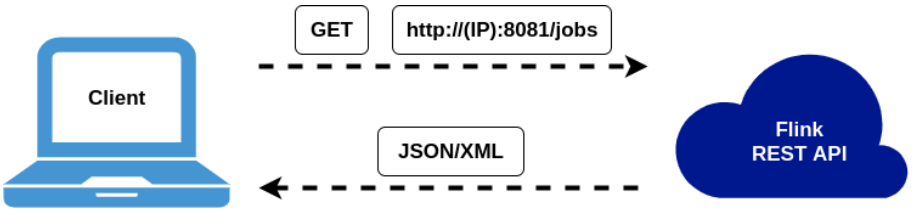


Figura 4.12: Exemplo de um pedido HTTP à API de monitorização do Flink

Se for considerado que uma topologia contém um determinado número de regras relativas a uma ou mais fontes de dados, esta API é então usada para a remoção e submissão de regras no sistema desenvolvido. O módulo responsável por este mecanismo é explicado de seguida, sendo apresentado o princípio do seu funcionamento.

4.4.1 Funcionamento do Módulo de Controlo de Regras

O sistema deve estar preparado para atualizar as regras sempre que detete a existência de uma nova fonte de dados. Um método que solucionaria este problema seria a submissão inicial de inúmeras topologias que englobassem um grande número de tipos de dados, conseguindo-se assim um sistema preparado para qualquer fonte. Porém, com o aumento do número de *jobs* ativos, a quantidade de recursos necessários para os executar também aumentaria e podia surgir daqui um risco de falha do sistema no caso de se esgotar os recursos disponíveis. Logo, considera-se esta abordagem eficaz mas muito pouco eficiente, na medida em que se torna redundante possuir uma topologia que processe dados cujas fontes nem sequer estejam a enviar informação. Daí surgiu a necessidade de desenvolver este módulo de controlo de regras que, para além de automaticamente submeter as topologias adequadas quando começam a entrar dados de um tipo de diferente dos existentes, também pára e remove os programas que não estão a processar qualquer informação.

Na Figura 4.13 está representado o princípio de funcionamento do módulo ao qual se deu o nome de *Update Rules* na arquitetura da solução proposta (ver Figura 4.1). Inicialmente, começa-se por criar um consumidor Kafka que subscreve o tópico *jar_update*. Note-se que este tópico contém mensagens apenas com o tipo de dados que está a entrar no sistema, tal como foi descrito na secção 4.2

de Entrada de Dados. Por cada mensagem que o *consumer* receber, guarda-se o conteúdo da mesma numa variável. Ao fim de cada 20 mensagens recebidas, o *core* do programa arranca. Primeiro, testa-se se é necessário parar e remover algum *job* ativo, sendo que este processo é feito através da consulta de uma variável que contém as topologias ativas. Se alguma das topologias que estiver presente nesta variável for relativa a um tipo de dados que não foi recebido pelo consumidor Kafka, então a condição é satisfeita e o serviço fica encarregue de remover essa topologia. Quando não existir nenhuma topologia que possa ser cancelada, o programa prossegue até à próxima condição, que testa se é necessário submeter alguma topologia, comparando os tipos de dados recebidos com as topologias ativas naquele momento. Quando esta condição não for satisfeita, significa que o programa já atualizou corretamente os *jobs* ativos na componente de processamento. Por isso, faz-se um *reset* à variável que contém as mensagens recebidas pelo *consumer*, fazendo assim com que este serviço espere pelos próximos 20 eventos para recomeçar a atualização de regras.

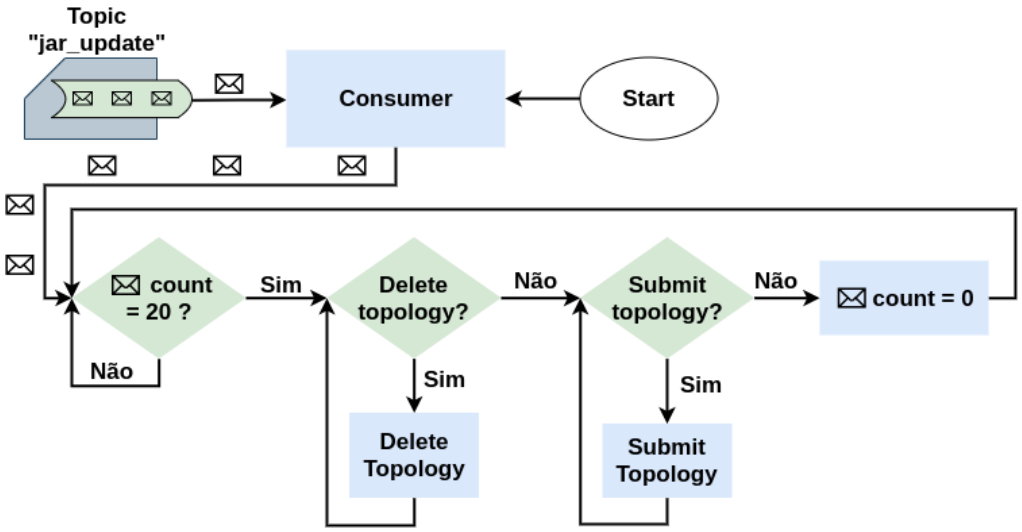


Figura 4.13: Princípio de funcionamento do serviço que atualiza as regras ativas no sistema

A nível de código desenvolvido para a implementação deste módulo, foi escrito em linguagem Python e pode ser consultado de forma integral no Anexo B. Deste, importa destacar as etapas onde se fazem os pedidos HTTP à API de monitorização do Flink, tanto para a remoção como para a submissão de topologias. A biblioteca Python utilizado para efetuar pedidos HTTP foi o *Requests*, que permite escolher o tipo de pedido (GET, POST, PATCH, DELETE, etc.),

adicionar parâmetros ao URL do pedido, entre outros. Para a remoção completa de uma topologia são efetuados dois pedidos distintos:

- PATCH para o endpoint */jobs/id_job*, onde em *id_job* se coloca a identificação da topologia, que está guardada na variável de topologias ativas. Este pedido pára a execução da topologia;
- DELETE para o endpoint */jars/id_jar*, onde se substitui *id_jar* pela identificação do arquivo JAR correspondente. Este pedido remove o ficheiro JAR da lista daqueles que podem ser submetidos.

Quando se pretende submeter uma topologia, também é necessária a realização de dois pedidos:

- POST para o endpoint */jars/upload* adicionando ao corpo do pedido, o arquivo JAR adequado. Este pedido envia o ficheiro JAR para o sistema de processamento;
- POST para o endpoint */jars/jar_id/run*, onde se substitui *jar_id* pela identificação do arquivo JAR que se quer submeter.

É assim conseguido o cumprimento de mais um dos objetivos propostos, uma vez que este mecanismo, a cada vinte eventos recebidos, faz a gestão das topologias/regras ativas no bloco de processamento, garantindo assim um sistema automatizado neste aspeto. Segue-se agora a fase final da descrição da implementação da presente solução, abordando-se o envio e leitura dos alertas por parte dos utilizadores do sistema, processo que pode ser feito de várias formas.

4.5 Ativação de Alertas

Como já foi referido, o foco principal do presente trabalho é, sem dúvida, o processamento de dados em tempo real integrando um motor CEP. Porém, decidiu-se construir um sistema que contemple a etapa anterior à do processamento - entrada de dados - bem como a fase posterior que consiste normalmente no aproveitamento dos resultados por aparte dos utilizadores, conseguindo-se assim fechar o fluxo de dados completo.

Nesta secção descreve-se então o modo como os alertas assim como todos os resultados gerados pela componente de processamento, são enviados para as partes interessadas. Tal como foi explicado na Secção 4.1, onde se mostrou a arquitetura proposta para o sistema de alarmística, os alertas gerados são entregues aos destinatários adequados através de várias formas: pela interação direta com o utilizador, consistindo no envio de notificações em casos que se justifique

fazê-lo; com recurso a uma ferramenta de visualização gráfica que acede aos dados que estão armazenados; uma REST API que foi desenvolvida mais a pensar na necessidade de consultar o histórico mas, também para disponibilizar um servidor *WebSockets*, conceito que se vai introduzir ainda nesta secção.

Para o primeiro caso, isto é, o bloco denominado de *Alert Generator* na Figura 4.1 da arquitetura da solução proposta, que produz notificações para as partes interessadas nos alertas, desenvolveu-se um serviço que mediante a severidade do alerta recebido, decide o canal que usa para o fazer chegar ao utilizador. Este serviço, cujo código se encontra no Anexo C, foi desenvolvido em linguagem Python e o seu funcionamento está representado na Figura 4.14. A execução do programa começa pela construção de um *consumer* que recebe continuamente as mensagens que estiverem no tópico Kafka *alert-gen*. Estas mensagens são os resultados produzidos pela componente de processamento e contêm elementos fixos como o contexto (se é sobre o tráfego, meteorologia, etc.), o tipo *Warning* ou *Alert* e a severidade do alerta, sendo que os restantes elementos variam com o contexto. Posteriormente, cada mensagem entra numa função responsável pela sua modelação, construindo-se um dicionário com quatro pares chave/valor. Os valores são os elementos da mensagem e as chaves indicam o que cada valor significa. Este dicionário é passado de seguida para uma etapa de decisão sobre o canal escolhido para fazer chegar o alerta até ao utilizador. Aqui, o programa acede ao valor do dicionário, que está associado à chave *severity* baseando a decisão neste elemento. Se a severidade estiver definida como *medium* então o programa envia um e-mail, cujo corpo contém todas as informações sobre o alerta, para as partes interessadas. Por outro lado, se a severidade do alerta estiver assinalada como *critical* então o programa executa o envio por *WebSockets*, onde faz o papel de cliente nesta comunicação.

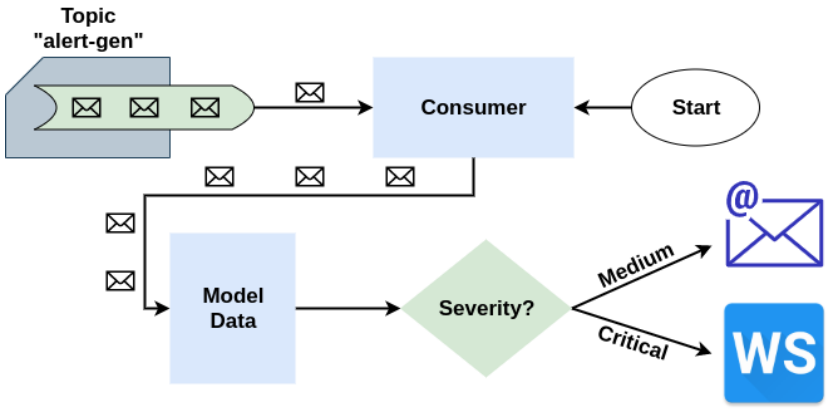


Figura 4.14: Princípio de funcionamento do módulo de distribuição de alertas

O protocolo WebSockets proporciona uma comunicação bi-direcional e persistente entre cliente e servidor, que permite uma distribuição de mensagens quase instantânea, resultando assim numa conexão com latência muito reduzida [122]. Tanto o cliente como o servidor podem enviar dados a qualquer altura desde que a conexão esteja iniciada. O cliente estabelece uma conexão *WebSocket* através de um processo denominado de *handshake*. Este processo consiste no envio, por parte do cliente, de um pedido HTTP com determinados parâmetros que fazem com que o servidor saiba que o cliente pretende estabelecer uma conexão *WebSocket*. Se o servidor suportar este protocolo então envia uma resposta com os mesmos parâmetros para o cliente [122]. Aqui, deixa de existir uma comunicação por pedidos HTTP para passar a existir uma conexão *WebSockets*, Uma das diferenças fáceis de identificar entre os dois tipos de comunicação está no início do URL, já que em vez do típico esquema *http://*, o protocolo *WebSockets* impõe o uso de *ws://* nos seus pedidos.

4.5.1 Endpoints da API

Tal como referido, o cliente da comunicação é então o serviço de distribuição de alertas descrito. Já o servidor é um *endpoint* da API desenvolvida. Esta API, que apresenta uma arquitetura REST, foi desenvolvida com dois objetivos: oferecer a possibilidade ao utilizador de consultar um histórico dos alertas gerados; implementar um servidor *WebSockets* ao qual entidades como bombeiros ou polícia se pudessem conectar e acompanhar, em tempo real, o aparecimento de alertas. Para completar estas tarefas, a API implementada suporta dois pedidos para os seguintes *endpoints*:

- GET para */warning/data_type* - A resposta deste pedido retorna, em formato JSON, uma completa descrição dos alertas armazenados, seguindo um modelo de dados da Fiware [123]. Neste *endpoint*, *data_type* deve ser substituído pelo tipo de dados (tráfego, meteorologia, etc.), sobre os quais se pretende obter o histórico dos alertas. Este pedido permite ainda que se filtrem os resultados temporalmente, colocando no URL os parâmetros *start* e *end*, cujos valores definem o intervalo de tempo que se quer obter os resultados. Se o utilizador pretender, pode ainda adicionar o parâmetro *last* onde define quantos resultados quer visualizar por ordem do alerta mais recente para o mais antigo;

- (WebSocket) */ws* - Neste *endpoint* encontra-se alojado o servidor *WebSocket* ao qual o serviço de distribuição de alertas se deve conectar sempre que disparar um alerta com severidade crítica. Para qualquer cliente se ligar a este servidor deve iniciar uma comunicação para o URL ***ws://IP:porta/ws***.

Esta REST API foi desenvolvida através da *framework* ***Fast API*** [124], que permite um rápido desenvolvimento, em linguagem Python, com menor probabilidade de aparecimento de erros que as restantes ferramentas. A documentação existente também se perfila como útil e bastante percetível, sendo que uma das razões que mais contribuiu para escolha desta ferramenta em detrimento de outras, foi o facto de suportar o protocolo *WebSockets*.

E assim chega ao seu término, a descrição da implementação de cada uma das componentes do sistema desenvolvido, pelo que se considera que, até esta secção do capítulo, se cobriu todos os módulos do esquema da arquitetura proposta. De seguida, e para concluir o capítulo de Implementação, é feito um sumário do que foi dito, apresentando um esquema que coloca as tecnologias utilizadas no interior dos módulos da Figura 4.1.

4.6 Integração Final dos Módulos

O objetivo deste capítulo foi cumprido, uma vez que foram descritas todas as componentes e serviços que se propôs desenvolver, quando se apresentou a arquitetura do sistema na secção 4.1. Para cada um destes módulos foi feita uma explicação do seu funcionamento, sendo que sempre que relevante, se mostraram também extratos do código abordando as funções desenvolvidas que permitem que as respetivas componentes desempenhem o seu papel.

Tal como já foi referido anteriormente, apesar do foco desta dissertação estar na aplicação de um motor CEP no processamento e análise de dados, em tempo real, decidiu-se construir um sistema desde raiz, ou seja, uma solução que engloba todas as etapas do fluxo de informação, desde a sua entrada, passando pelo processamento e terminando com o envio e disponibilização dos resultados para os utilizadores interessados. No decorrer da apresentação dos passos executados para a implementação do sistema, percebe-se que várias tecnologias foram utilizadas por isso, desenhou-se o esquema da Figura 4.15, onde se associa cada serviço e tecnologia descrita neste capítulo aos respetivos blocos indicados na arquitetura proposta (ver secção 4.1). Sempre que adequado foram colocados, na parte superior dos blocos, os logótipos das linguagens de programação utilizadas para cada um deles. Em determinados blocos achou-se que se justificaria colocar também uma representação da tecnologia usada.

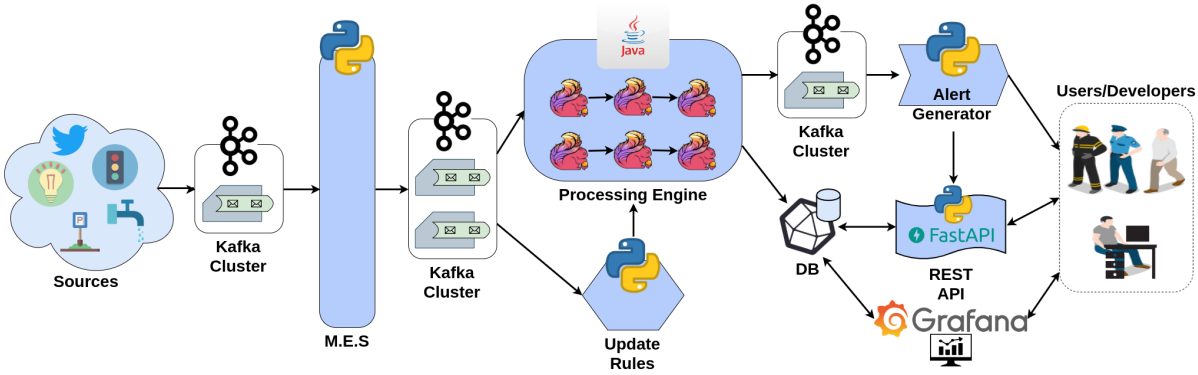


Figura 4.15: Integração das tecnologias usadas com os blocos mais gerais da solução

Percorrendo o esquema percebe-se que fluxo de informação tem início nas fontes de dados, que podem ser de vários tipos: redes sociais, energia, tráfego rodoviário, ambiente, parques de estacionamento, etc. Todos os dados são enviados para um tópico **Kafka**, que como se pode ver, foi a ferramenta escolhida para a distribuição de mensagens entre os módulos. Este tópico Kafka está por sua vez a ser consumido pelo serviço de monitorização da entrada de dados, ao qual se deu o nome de M.E.S (Monitoring Event Source). O M.E.S, desenvolvido em linguagem Python, é responsável pela filtragem dos eventos recebidos pelo seu tipo, que é o primeiro elemento do *tuple* recebido. Depois de aceder a este elemento, publica informação para dois tópicos: um que é sempre fixo, onde a mensagem enviada apenas contém o tipo de dados e para um tópico cujo nome depende do contexto da informação, sendo que para este já são enviados todos os elementos do *tuple* recebido. Segue-se uma camada da arquitetura constituída por dois módulos: motor de processamento e o serviço de atualização de regras. O primeiro consiste em uma ou mais topologias **Flink** responsáveis pelo processamento dos dados, sendo que cada uma delas, é relativa a um tipo de dados ou à relação entre determinados vários, pelo que cada uma apenas subscreve o tópico com mensagens referentes aos tipos de dados que processam. Estes programas Flink implementam um motor CEP e foram escritos em Java. Já o bloco que controla as regras que estão ativas no sistema de processamento, ou seja, no fundo controla as topologias que estão ativas. Este serviço subscreve o tópico cujas mensagens apenas contém o contexto dos eventos recebidos e, a cada 20 eventos, vai verificar se alguma topologia precisa de ser removida bem como se alguma precisa de ser submetida no Flink. De facto, este módulo garante que não estão a ser utilizados recursos de processamento para além do necessário, aumentando a eficiência do sistema. Os resultados do bloco de processamento têm dois destinos: um tópico Kafka e a base de dados **InfluxDB**. O tópico é

consumido por um serviço, desenvolvido em Python e nomeado de *Alert Generator*, que está encarregue de distribuir os alertas mediante a sua severidade. O programa notifica os utilizadores e partes interessadas através de um e-mail com todas as informações relativas ao evento, quando se considerar que este denota severidade média. Por outro lado os alertas que devem ser vistos em tempo real, tamanha é a sua importância e gravidade, são enviados para um servidor ***WebSockets***. Este protocolo proporciona uma troca de mensagens quase instantânea entre cliente e servidor. Foi também desenvolvida uma REST API, com recurso à *framework* **Fast API** onde os utilizadores podem consultar o histórico de todos os resultados produzidos assim como aceder ao servidor *WebSocket*, onde aparecerão, em tempo real, os alertas com maior severidade. Por fim importa referir que foi também utilizada uma ferramenta de visualização gráfica, o **Grafana** que consegue realizar uma grande variedade de *queries* à base de dados.

De forma a aproximar a solução desenvolvida a um verdadeiro produto, era importante garantir que esta conseguisse ser instalada em qualquer máquina, sem a necessidade de se estar a instalar todos os pacotes e bibliotecas necessárias ao funcionamento do sistema. À semelhança do mecanismo de atualização de topologias, este processo de instalação da solução foi também automatizado através da tecnologia **Docker**, cujos fundamentos são abordados de seguida.

4.6.1 Sistema Assente em Docker

O Docker [125] é uma ferramenta que facilita a criação e desenvolvimento de aplicações, permitindo a sua execução em qualquer máquina, através de *containers* [126]. Um *container* é uma unidade de software capaz de comprimir o código e todas as suas dependências num pacote, fazendo com que a aplicação seja executada rapidamente e de forma segura a partir de qualquer distribuição Linux. Um *container* pode ser comparado a uma máquina virtual mas enquanto esta cria um sistema operativo completo de forma a ser compatível com as aplicações, o Docker e os seus *containers* são executados na camada de aplicação do sistema operativo nativo, sendo assim uma solução mais eficiente e móvel [127].

Na Figura 4.16 estão representados os passos necessários para a formação de um *container*. O ponto de partida é a criação de um ficheiro de texto chamado *Dockerfile*, onde se prepara o ambiente do *container*, sendo aqui definido através de instruções e passos que normalmente se efetuariam manualmente [128]. Por exemplo, se o pretendido for executar um serviço Python, então neste ficheiro devem ser importadas as dependências que o serviço precisa para funcionar, definir o diretório de trabalho, copiar para esse diretório os potenciais recursos que esse serviço necessita, colocar o comando para o executar, entre outros. A etapa seguinte é construir a imagem Docker, através da execução do comando - *docker build –tag=nome da imagem .* - no diretório onde estiver o *Dockerfile*. Este

comando vai gerar um ficheiro executável que contém todas as bibliotecas, configurações, código, etc. do serviço que se quer implementar. Finalmente, para construir o *container*, executa-se o comando - *docker run nome da imagem* - onde podem ser adicionados vários argumentos como a porta onde o container vai estar a ser executado no caso de se tratar de um serviço Web.

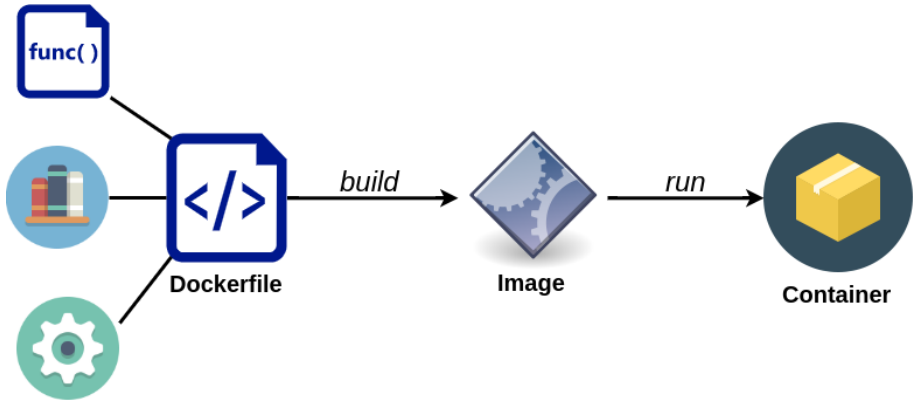


Figura 4.16: Etapas da produção de um *container*

Outro conceito que é importante introduzir é o de serviço no contexto do *Docker*. No fundo, serviços são *containers* em produção, sendo que cada serviço apenas executa uma imagem, mas é capaz de escolher as condições de execução, podendo definir quantas réplicas do *container* vão ser executadas, em que portas vão estar alojadas, entre outras. A plataforma Docker permite então escalar facilmente uma aplicação sem se importar com a quantidade de serviços que esta engloba. Para isso basta criar o ficheiro *docker-compose.yml* [129].

De facto, o *Compose* é uma ótima ferramenta para a instalação de uma aplicação em diferentes máquinas pois contém todas as propriedades dos seus serviços da mesma, contribuindo assim para a portabilidade de uma solução. No presente projeto construiu-se um ficheiro deste tipo, de modo a executar todos os serviços descritos neste capítulo apenas com recurso a um comando. A típica estrutura de um ficheiro *docker-compose.yml* engloba a definição de quatro parâmetros: *version* - que influencia o formato como se colocam as propriedades no ficheiro *Compose* ((atualmente está na versão 3.7); *services* - que se querem executar bem como as condições com que são executados como já foi explicado; *networks* - (opcional), que são usadas quando se pretende que dois serviços partilhem a mesma rede por haver algum tipo de comunicação entre si; *volumes* - (opcional), que são usados no caso de se pretender criar um volume no interior do *container* [130].

A Figura 4.17 mostra o extrato do código do ficheiro *docker-compose.yml* que se encontra disponível para consulta no Anexo D, onde se vê como foi definido o serviço responsável por executar a REST API desenvolvida. Na linha 130 colocou-se o nome do serviço - *end_api* - que possui os seguintes argumentos: *build*, onde se coloca o caminho para o ficheiro *Dockerfile* que gera a imagem para executar a API; *environment*, onde se colocam as variáveis de ambiente do serviço em causa, sendo que neste caso apenas se define que não se quer armazenar no disco os *outputs* do serviço; *ports*, onde se coloca a porta onde a API irá estar alojada; *networks*, onde se coloca a rede onde a API estará, sendo que se deve colocar este argumento com o mesmo nome da rede nos serviços que precisarem de comunicar com a API, como por exemplo o módulo *Alert Generator* que como se viu na Figura 4.15 desta secção.

```
130     end_api:
131         build: ../api/.
132         environment:
133           - PYTHONUNBUFFERED=1
134         ports:
135           - 6000:6000
136         networks:
137           - kafka
```

Figura 4.17: Definição do serviço

Em relação aos restantes serviços que se colocaram no ficheiro *Compose*, não há diferenças relevantes a destacar para com o serviço da REST API, com exceção à presença do argumento *image* em vez do *build*. Usou-se o argumento *image* quando o serviço consiste numa tecnologia ou ferramenta que não se desenvolveu de raiz como por exemplo o Kafka. A imagem Docker do Kafka foi desenvolvida e está disponível para os utilizadores usarem as suas funções e capacidades se assim o pretenderem, da mesma forma que uma pessoa que pretenda usar uma determinada ferramenta, deve primeiro, fazer o *download* desta. No total, o *docker-compose* criado (ver Anexo D) instala treze serviços com os seguintes nomes:

- *zookeeper* - serviço responsável por guardar os estados do *cluster* Kafka e dos operadores do Flink (*Job Manager* e *Task Manager(s)*);
- *kafka1* - serviço que instala o Kafka e cria um *cluster*, definindo as suas propriedades;

- *manager* - serviço que implementa uma interface gráfica para monitorizar o *cluster* Kafka, onde é possível consultar, entre outros parâmetros, a quantidade de mensagens a passar em cada tópico;
- *grafana* - serviço responsável por alojar uma instância *Web* do Grafana que comunique com a base de dados;
- *influx* - serviço que cria base de dados utilizada;
- *jobmanager* - serviço encarregue de criar um *Job Manager* para o sistema Flink;
- *taskmanager* - serviço encarregue de criar um *Task Manager* para o sistema Flink;
- *alert-generator* - este serviço implementa o módulo descrito na secção 4.5;
- *producer-python* - serviço que funciona como fonte de dados para o sistema, substituindo os habituais sensores;
- *monitoring-event* - serviço que implementa o módulo descrito na secção 4.2.1;
- *request-python* - serviço desenvolvido para testar a reação da REST API aos eventuais pedidos;
- *upload-python* - serviço encarregue de atualizar as topologias no Flink;
- *end_api* - como já foi dito, este serviço implementa a REST API desenvolvida.

Tendo construído este ficheiro, basta executar o comando - *docker-compose up* - para que todos os serviços descritos sejam executados.

4.7 Sumário

Neste capítulo apresenta-se, numa primeira fase, a arquitetura da solução proposta para responder aos desafios apontados no capítulo anterior assim como potenciais casos de uso para este sistema. Na primeira secção dá-se então a conhecer os módulos que se propõe desenvolver e o modo como estes se encontram ligados entre si. O foco desta dissertação está na componente de processamento de dados e na produção de alertas, em tempo real, através da deteção de padrões nesses dados (CEP). Contudo, propôs-se desenvolver um sistema desde a sua raiz, que é a entrada de dados e respetiva filtragem destes (se necessário), passando pela etapa obrigatória de processamento, até os utilizador e partes interessadas

serem notificados sobre os alertas produzidos. Nesta secção descreve-se a função de cada bloco da arquitetura sem se revelar que tecnologia e ferramentas se usam.

Mostra-se ainda aqui, dois possíveis casos de uso do sistema que se pretende implementar. Aqui faz-se uma distinção entre alerta e aviso que são os resultados possíveis da componente de análise e processamento. Considera-se que um alerta é uma informação relativa a um evento que já ocorreu, tendo normalmente um nível de severidade alto. Por outro lado, um aviso é definido como algo onde não é preciso obrigatoriamente uma atuação sobre o evento mas que pretende evitar situações perigosas ou desagradáveis para os utilizadores. Para terminar, convém destacar que os casos de uso deste sistema podem ser quase ilimitados uma vez que, por um lado, as fontes de dados podem ser muito diferentes umas das outras e depois, dependendo do tipo de alerta que se pretende gerar, entre estas fontes podem ser definidas várias relações. Num dos exemplos mostrados relaciona-se os dados provenientes dos sensores de temperatura com os dos sensores de fumo. No outro exemplo já se associam três fontes: uma rede social, sensores de tráfego rodoviário e sensores dos parques de estacionamento.

Na secção seguinte é feita a descrição da implementação dos blocos identificados na Figura 4.1 da secção anterior, onde se apresentou a arquitetura proposta. Importa destacar que neste projeto, apesar do foco estar na componente de processamento de informação em tempo real, foi construído o sistema completo, isto é, desde as fontes de dados até ao *output* dos resultados.

Para cada um dos blocos é feita uma explicação do funcionamento do mesmo, sendo essa explicação apoiada pelo fluxograma das funções criadas e, quando necessário, por extratos de código constituintes destas. A explicação inicia-se pela entrada dos dados, que assume um papel importante no sistema pois deve garantir que toda a informação é processada assim como enviar os dados para o destino adequado. De seguida, descreve-se o funcionamento do motor CEP utilizado, sendo esta parte considerada como a mais relevante da solução proposta. Segue-se a abordagem ao módulo que faz o controlo das regras "ativas"no sistema. Termina-se a descrição dos blocos da arquitetura, explicando o modo como os utilizadores podem interagir com o sistema, ou seja, a forma como estes recebem os alertas. Por fim, o capítulo encerra sendo feita a integração dos módulos descritos com as tecnologias correspondentes.

Capítulo 5

Testes e Avaliação de Desempenho

De forma a demonstrar o funcionamento da solução desenvolvida, avaliou-se qualitativamente vários processos da mesma, através de testes de performance. Neste capítulo, é por isso feita uma discussão dos resultados dos testes efetuados, suportada por métricas temporais que caracterizam a reação do sistema em diferentes cenários. Dado que, o grande objetivo deste projeto é construir uma plataforma que produza alertas em **tempo real**, os testes baseiam-se nas medições dos *timestamps* parciais e finais nos principais componentes do sistema. Numa primeira fase fez-se uma análise temporal desde que um evento entra no sistema até que o utilizador possa ver o alerta gerado, isto é, uma lógica de *end-to-end*. Tal como se relatou na Secção 4.6, os alertas são enviados para a base de dados e para um módulo denominado de *Alert Generator*, que por sua vez os distribui por e-mail ou para um servidor *WebSockets*. Assim sendo, foi testado o tempo de resposta do sistema para estes três *outputs*.

De seguida, a avaliação centra-se na etapa mais crítica da solução, ou seja, na componente de processamento da informação. Aqui, foram recolhidos os *timestamps* à entrada e saída das funções do Flink que fazem o tratamento dos eventos recebidos. Tanto nesta análise como na da lógica *end-to-end*, foi testado o desempenho do sistema em três cenários distintos: uma fonte de dados e envio de um evento por segundo; duas fontes de dados e envio de um evento por segundo; duas fontes de dados e envio contínuo de eventos.

Por fim, analisa-se o comportamento do módulo *Update Rules* (rever Figura 4.15) de forma a quantificar e determinar se a abordagem escolhida, isto é, atualizar as topologias a cada 20 eventos, faz com que haja perda de informação.

5.1 End-to-End

Tal como referido, começou-se por avaliar o comportamento do sistema na sua globalidade, ou seja, desde o momento da receção do evento até à leitura do alerta pelas partes interessadas. Estes testes consistiram no registo dos *timestamps* na entrada e na saída de todas etapas percorridas pelos eventos. De modo a estimar com precisão o desempenho do protótipo, definiu-se que a amostra utilizada em todos os testes teria como tamanho, 10. Nesta secção, todas as tabelas apresentam uma estrutura idêntica onde se destacam as seguintes características:

- Dez linhas onde se colocaram os resultados das 10 experiências realizadas. Estes consistem em tempos de resposta, registados em milissegundos, de cada etapa do sistema;
- Uma última linha que contém a média dos valores da amostra;
- O conteúdo sombreado a azul, que se refere aos nomes dos processos pelos quais um evento passa até ser transformado num alerta: a coluna *MES* diz respeito ao bloco *Monitoring Event Source* (ver Figura 4.15); a coluna *Flink* é relativa ao módulo de processamento de dados; a coluna *Generator* é referente ao bloco *Alert Generator*. Finalmente, na última coluna foi colocado o nome do destino do alerta, pelo que varia ao longo desta secção.

A cada tabela encontra-se sempre associado um gráfico que representa os dados refletidos na mesma. Cada gráfico espelha os resultados da respetiva tabela de duas formas: por barras verticais, que estão legendadas com o nome de "Duração", onde os valores correspondem à média do tempo consumido por cada etapa; e por uma linha da soma acumulada dos valores anteriores, permitindo assim avaliar onde existe maior latência, pelo declive de cada reta dessa linha. Importa ainda salientar que as tabelas apresentadas nas primeiras duas secções deste capítulo, são uma versão resumida das tabelas que se encontram no Anexo E, sendo nestas exibidos os valores dos *timestamps* que serviram de base para o cálculo do tempo de resposta de cada etapa. Por exemplo, os valores colocados na coluna *MES* correspondem ao resultado da equação: $OutMES - InMES$, que exprime a diferença entre o *timestamp* recolhido à saída do módulo (Out) e à entrada do mesmo (In).

Começou-se então por medir, para os três cenários previamente mencionados, o tempo que o sistema desenvolvido demora a produzir um alerta através do envio de um e-mail, sendo que os dados estão expostos nas Tabelas 5.1, 5.2 e 5.3. As versões completas destes dados estão, respetivamente, nas tabelas E.4, E.5, E.6.

Tabela 5.1: Tempo gasto, em milissegundos, em cada etapa do sistema até ser enviado um alerta por e-mail, num cenário com 1 fonte de dados e envio de 1 Mensagem/segundo

#	MES	Flink	Generator	Email
#1	**1**	**160**	**3**	**926**
#2	**0**	**159**	**0**	**1035**
#3	**1**	**247**	**0**	**1709**
#4	**1**	**241**	**0**	**1615**
#5	**0**	**243**	**0**	**1630**
#6	**0**	**43**	**0**	**1670**
#7	**1**	**125**	**0**	**1727**
#8	**0**	**108**	**0**	**1674**
#9	**0**	**122**	**0**	**1599**
#10	**1**	**123**	**0**	**1800**
Média	**0,5**	**157,1**	**0,3**	**1538,5**

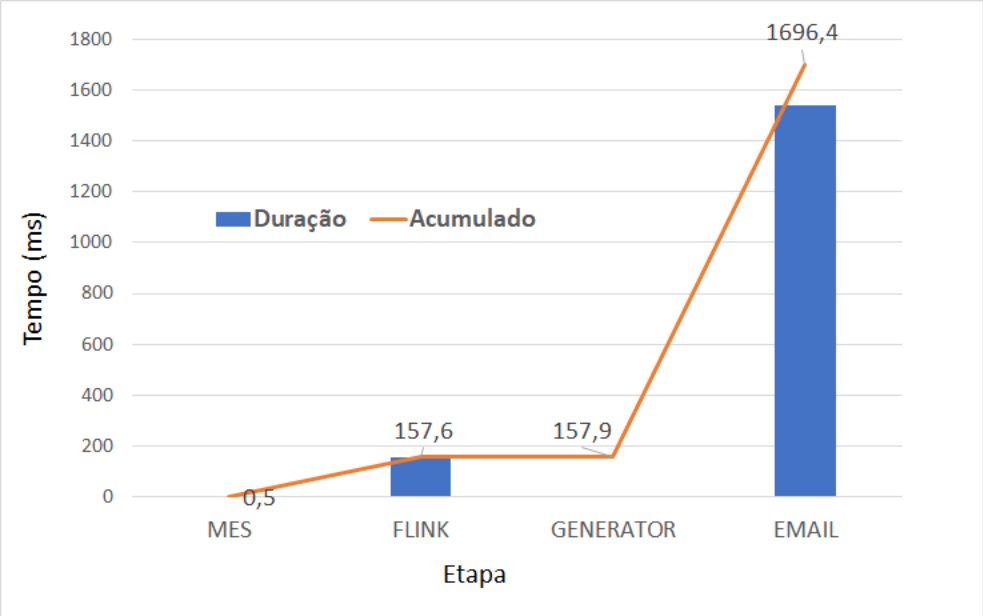


Figura 5.1: Resposta temporal da produção de um e-mail, num cenário com 1 fonte de dados e envio de 1 Mensagem/segundo

Tabela 5.2: Tempo gasto, em milissegundos, em cada etapa do sistema até ser enviado um alerta por e-mail, num cenário com 2 fontes de dados e envio de 1 Mensagem/segundo

#	MES	Flink	Generator	Email
#1	**1**	**229**	**0**	**1609**
#2	**0**	**223**	**0**	**1005**
#3	**1**	**224**	**0**	**979**
#4	**1**	**123**	**0**	**928**
#5	**0**	**116**	**1**	**983**
#6	**0**	**214**	**0**	**932**
#7	**1**	**205**	**1**	**924**
#8	**0**	**193**	**0**	**1665**
#9	**1**	**189**	**0**	**1606**
#10	**1**	**82**	**1**	**1936**
Média	**0,6**	**179,8**	**0,3**	**1256,7**

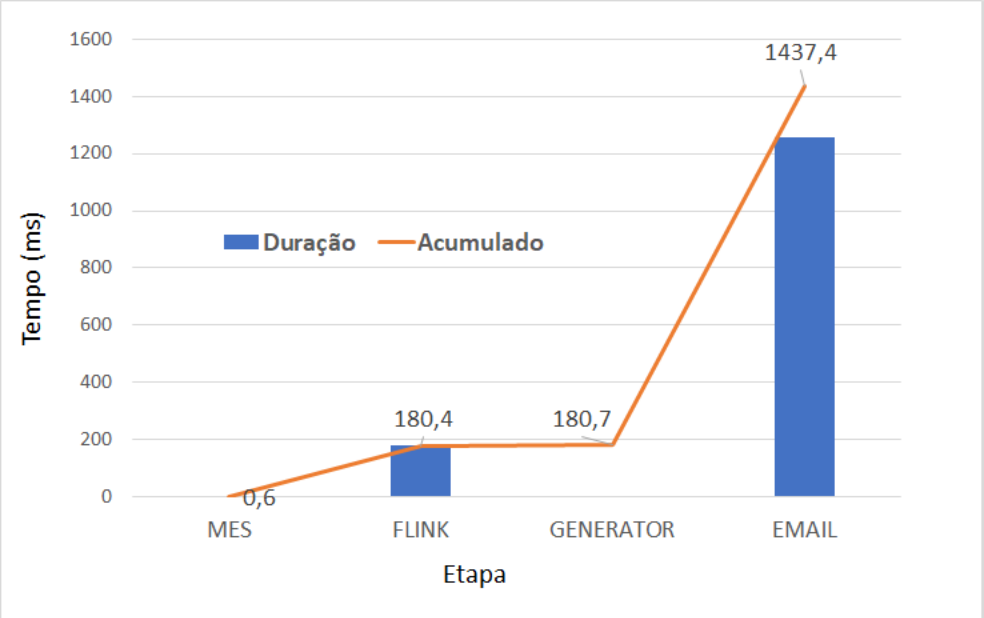


Figura 5.2: Resposta temporal da produção de um e-mail, num cenário com 2 fontes de dados e envio de 1 Mensagem/segundo

Tabela 5.3: Tempo gasto, em milissegundos, em cada etapa do sistema até ser enviado um alerta por e-mail, num cenário com 2 fontes de dados e envio contínuo

#	MES	Flink	Generator	Email
#1	**0**	**172**	**1**	**941**
#2	**0**	**169**	**0**	**1871**
#3	**1**	**247**	**1**	**1614**
#4	**1**	**182**	**0**	**1623**
#5	**2**	**65**	**0**	**1618**
#6	**1**	**156**	**0**	**1809**
#7	**0**	**255**	**0**	**1723**
#8	**1**	**225**	**0**	**1755**
#9	**1**	**224**	**4**	**1731**
#10	**1**	**214**	**0**	**1696**
Média	**0,8**	**190,9**	**0,6**	**1638,1**

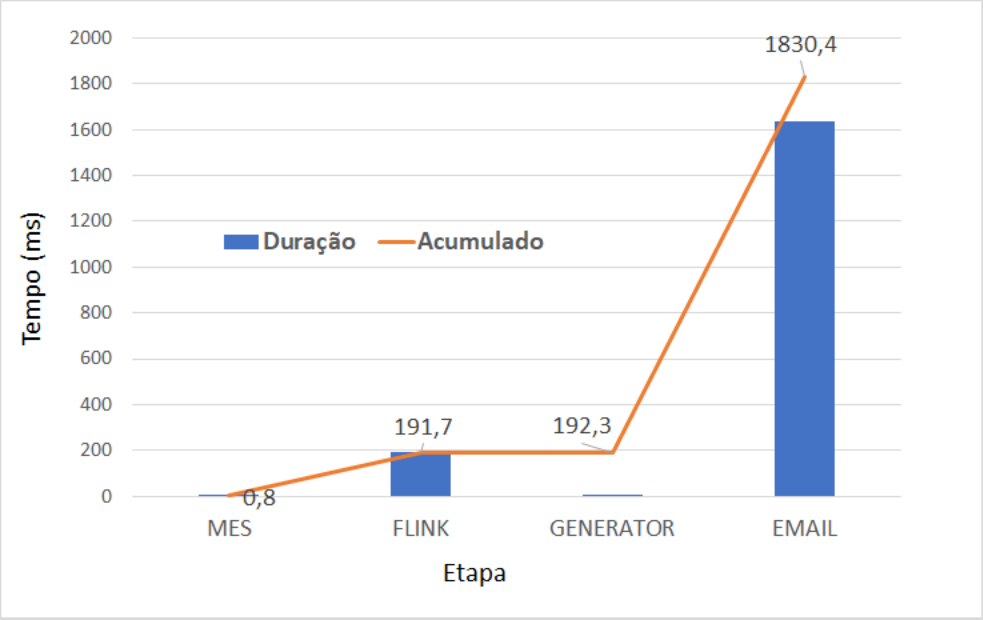
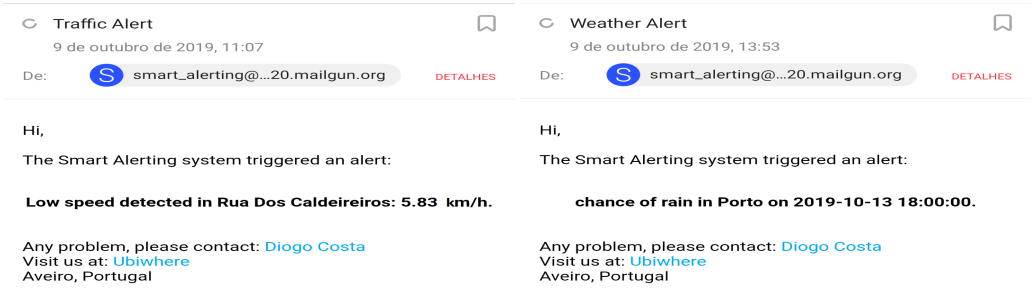


Figura 5.3: Resposta temporal da produção de um e-mail, num cenário com 2 fontes de dados e envio contínuo

Os gráficos representados nas Figuras 5.1, 5.2 e 5.3 apresentam um formato muito idêntico, sendo que, no que diz respeito às notificações por e-mail do sistema, é possível concluir que:

- O alerta chega ao utilizador em aproximadamente 1,5 segundos, não sendo um resultado preocupante porque os alertas enviados por este canal não têm severidade tão alta;
- Com o aumento do número de fontes de dados, o tempo de resposta do motor Flink variou na ordem dos 20 ms, o que é um sinal muito satisfatório. Desta forma, para o atraso ser efetivamente significativo, isto é, na casa dos segundos, seria necessário existirem mais de 50 fontes de dados. Estes resultados foram obtidos com a execução de uma só topologia, por isso tendo em conta que o Flink permite paralelizar a execução das mesmas, estes primeiros testes já deixam antever que a escalabilidade da solução é uma realidade;
- A etapa de maior duração é de facto, o envio do e-mail. Este facto pode ser justificado pelo uso da biblioteca *Simple Mail Transfer Protocol* (SMTP) do Python, cujos processos podem ter influência neste tempo de resposta;
- O módulo MES assim como o *Alert Generator*, causam um atraso insignificante no tempo de reposta global do sistema, cumprindo assim com o pretendido, uma vez que estes módulos foram projetados para funcionar como distribuidores, não contemplando funções de processamento.

Na Figura 5.4 são mostrados dois exemplos de e-mails que são enviados pelo protótipo desenvolvido. O corpo do e-mail segue uma estrutura definida, pelo que a única diferença notada nos exemplos ilustrados, é a mensagem que contém essencialmente, as informações sobre o alerta.



(a) *Tráfego* (b) *Meteorologia*

Figura 5.4: Exemplos do corpo do e-mail enviado pelo sistema desenvolvido

Segue-se agora a demonstração dos resultados obtidos, para os mesmos três cenários que os testes anteriores, na análise temporal da produção de um alerta

para um servidor *WebSockets*. O sistema decide usar este canal para enviar o alerta, quando este apresenta uma severidade alta, uma vez que o previsto é que este tipo de comunicação seja muito mais rápida que a notificação por e-mail. Os resultados são então mostrados nas Tabelas 5.4, 5.5 e 5.6, sendo cada uma delas acompanhada com um gráfico que reflete o seu conteúdo. Tal como já foi referido, as tabelas que contêm os *timestamps*, com os quais se construíram as que se seguem, estão no Anexo E (ver Tabelas E.1, E.2 e E.3).

Tabela 5.4: Tempo gasto, em milissegundos, em cada etapa do sistema até ser enviado um alerta para o servidor WebSockets, num cenário com 1 fonte de dados e envio de 1 mensagem/segundo

#	MES	Flink	Generator	Web Socket
#1	**1**	**198**	**1**	**9**
#2	**0**	**87**	**2**	**22**
#3	**1**	**89**	**3**	**17**
#4	**0**	**94**	**1**	**64**
#5	**0**	**82**	**1**	**7**
#6	**0**	**82**	**1**	**6**
#7	**1**	**181**	**0**	**5**
#8	**1**	**176**	**1**	**17**
#9	**1**	**175**	**2**	**17**
#10	**1**	**171**	**0**	**6**
Média	**0,6**	**133,5**	**1,2**	**17**

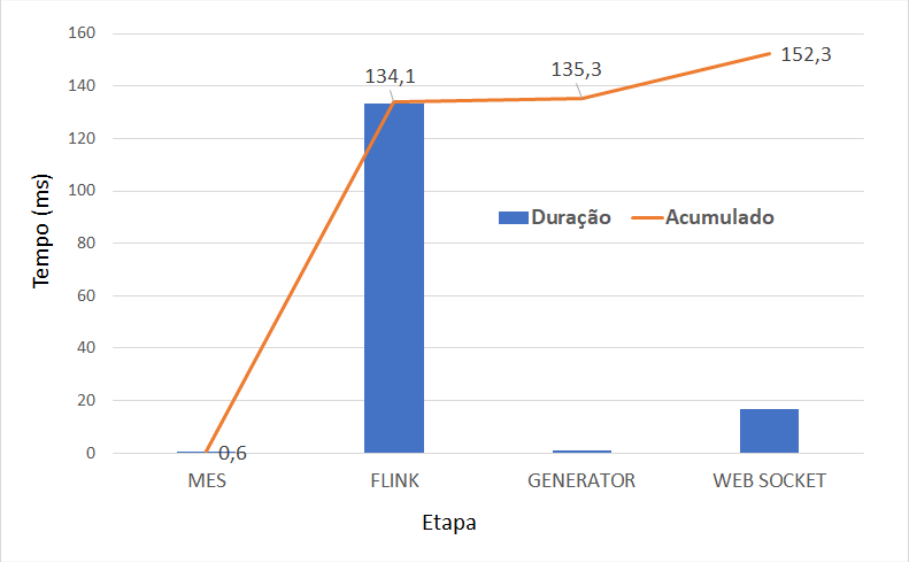


Figura 5.5: Resposta temporal da produção de um alerta no servidor *WebSockets*, num cenário com 1 fonte de dados e envio de 1 Mensagem/segundo

Tabela 5.5: Tempo gasto, em milissegundos, em cada etapa do sistema até ser enviado um alerta para o servidor WebSockets, num cenário com 2 fontes de dados e envio de 1 mensagem/segundo

#	MES	Flink	Generator	Web Socket
#1	**1**	**152**	**1**	**15**
#2	**0**	**156**	**0**	**7**
#3	**0**	**156**	**1**	**12**
#4	**0**	**153**	**1**	**7**
#5	**0**	**90**	**1**	**5**
#6	**0**	**80**	**1**	**14**
#7	**1**	**133**	**1**	**6**
#8	**1**	**128**	**1**	**6**
#9	**0**	**127**	**2**	**17**
#10	**1**	**221**	**1**	**13**
Média	**0,4**	**139,6**	**1**	**10,2**

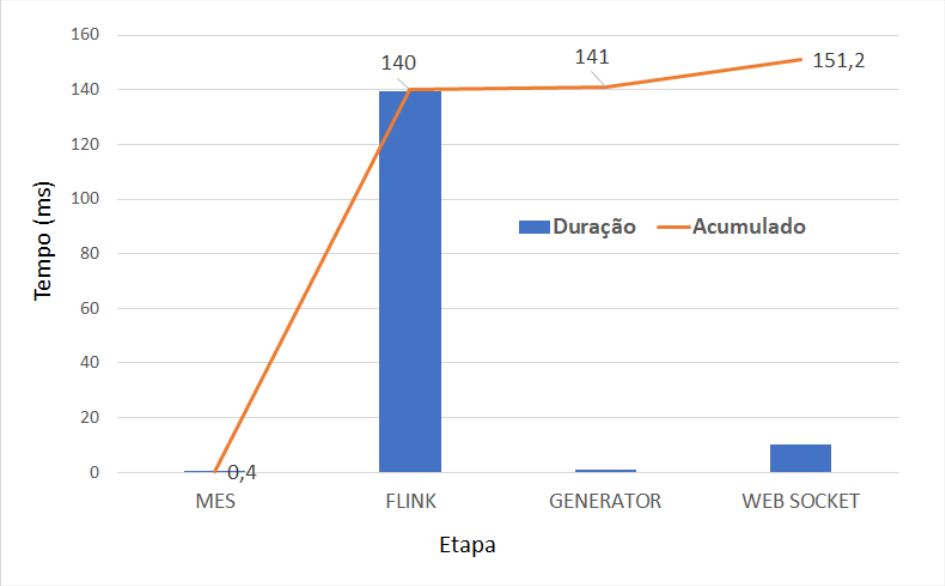


Figura 5.6: Resposta temporal da produção de um alerta no servidor *WebSockets*, num cenário com 2 fontes de dados e envio de 1 Mensagem/segundo

Tabela 5.6: Tempo gasto, em milissegundos, em cada etapa do sistema até ser enviado um alerta para o servidor WebSockets, num cenário com 2 fontes de dados e envio contínuo

#	MES	Flink	Generator	Web Socket
#1	**0**	**117**	**0**	**3**
#2	**1**	**210**	**0**	**3**
#3	**0**	**207**	**1**	**6**
#4	**1**	**203**	**2**	**15**
#5	**0**	**118**	**1**	**10**
#6	**1**	**107**	**1**	**12**
#7	**1**	**193**	**3**	**10**
#8	**0**	**77**	**1**	**10**
#9	**1**	**76**	**1**	**5**
#10	**0**	**159**	**0**	**4**
Média	**0,5**	**146,7**	**1**	**7,8**

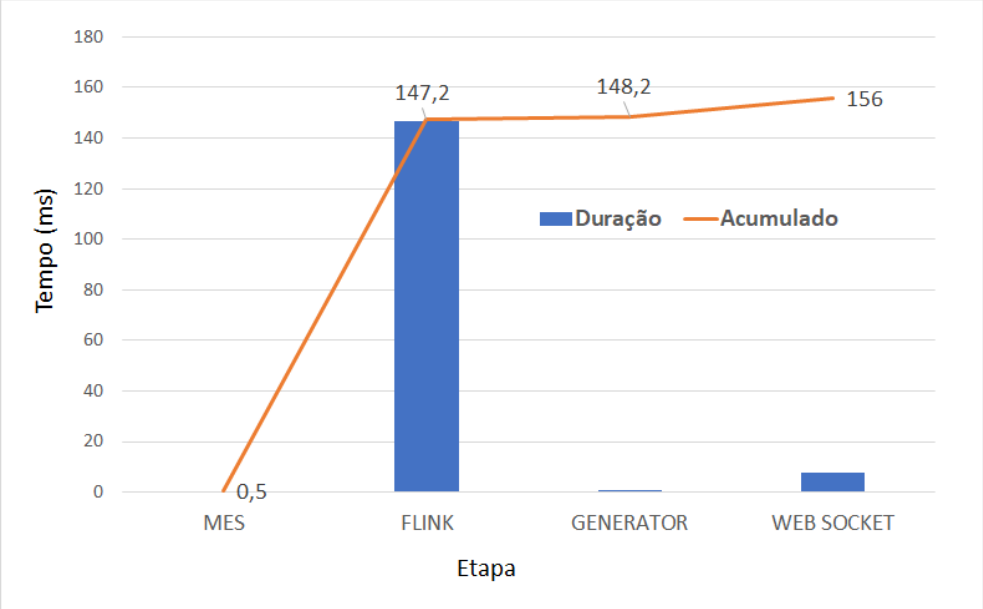


Figura 5.7: Resposta temporal da produção de um alerta no servidor *WebSockets*, num cenário com 2 fontes de dados e envio contínuo

Através da observação dos três gráficos anteriores, que correspondem aos três cenários avaliados, percebe-se que, apesar das condições relativas à entrada dos

dados terem mudado, o comportamento do protótipo não variou de forma evidente. As conclusões que se tiraram dos resultados destes testes, relativos à performance do sistema no envio de alertas para um servidor de *WebSockets*, são as seguintes:

- O utilizador consegue receber o alerta em sensivelmente, 150 ms, o que para se ter uma ideia, equivale a um décima de segundo. Este resultado confirma que se cumpriu um dos grandes objetivos do presente projeto, isto é, a produção de alertas em tempo real;

- Tal como se verificou no desempenho da solução, no envio de alertas por e-mail, o motor de processamento não apresenta diferenças relevantes no seu tempo de resposta quando tanto o número de fontes de dados, como a periodicidade da entrada dos mesmos, varia;

- A diferença do tempo de resposta global do sistema entre o envio dos alertas para um servidor *WebSockets* e por e-mail, sustenta a decisão de se enviar através do primeiro método, a informação com severidade mais alta;

- Como já se tinha observado, nem o módulo MES nem o serviço que distribui os alertas para os canais adequados (*Alert Generator*), geram latência no fluxo de dados.

Por último, fez-se uma avaliação ao atraso que o registo dos alertas na base de dados podia eventualmente causar. O InfluxDB foi escolhido como tecnologia da base de dados utilizada, precisamente com o objetivo de evitar que esse atraso pudesse acontecer. Por isso, é esperado que o tempo de resposta do sistema não seja afetado por este processo de registo, uma vez que todos os alertas produzidos, são guardados. Por esta razão, realizaram-se testes idênticos aos que se mostraram anteriormente nesta secção, ou seja, fez-se a recolha dos *timestamps* à entrada e saída dos módulos pelos quais o alerta passa, até ser registado na base de dados. As Tabelas E.7, E.8, E.9 providenciam todas as métricas recolhidas neste âmbito, sendo que as Tabelas 5.7, 5.8 e 5.9, apresentam, respetivamente, a mesma informação mas de forma resumida. A única diferença que se verifica em relação aos testes anteriores, é a ausência da coluna *Generator*, o que se deve ao facto de os alertas não passarem por esse módulo antes de serem registados como foi descrito na arquitetura da solução (ver Figura 4.15).

Tabela 5.7: Tempo gasto, em milissegundos, em cada etapa do sistema até o alerta ser registado na base de dados, num cenário com 1 fonte de dados e envio de 1 Mensagem/segundo

#	MES	Flink	Database
#1	**0**	**123**	**1**
#2	**1**	**222**	**1**
#3	**0**	**220**	**0**
#4	**1**	**21**	**1**
#5	**0**	**118**	**1**
#6	**0**	**225**	**0**
#7	**0**	**230**	**0**
#8	**0**	**230**	**2**
#9	**0**	**230**	**1**
#10	**0**	**111**	**1**
Média	**0,2**	**173**	**0,8**

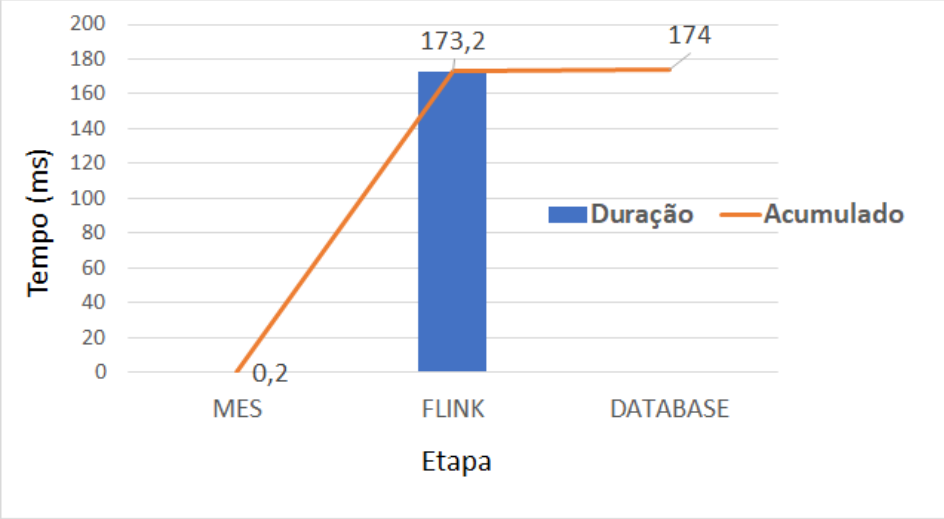


Figura 5.8: Resposta temporal do registo de um alerta na base de dados, num cenário com 1 fonte de dados e envio de 1 Mensagem/segundo

Tabela 5.8: Tempo gasto, em milissegundos, em cada etapa do sistema até o alerta ser registado na base de dados, num cenário com 2 fontes de dados e envio de 1 Mensagem/segundo

#	MES	Flink	Database
#1	**0**	**117**	**1**
#2	**0**	**110**	**1**
#3	**1**	**108**	**1**
#4	**0**	**197**	**0**
#5	**0**	**127**	**1**
#6	**0**	**124**	**0**
#7	**0**	**215**	**1**
#8	**0**	**222**	**1**
#9	**0**	**112**	**0**
#10	**1**	**219**	**1**
Média	**0,2**	**155,1**	**0,7**

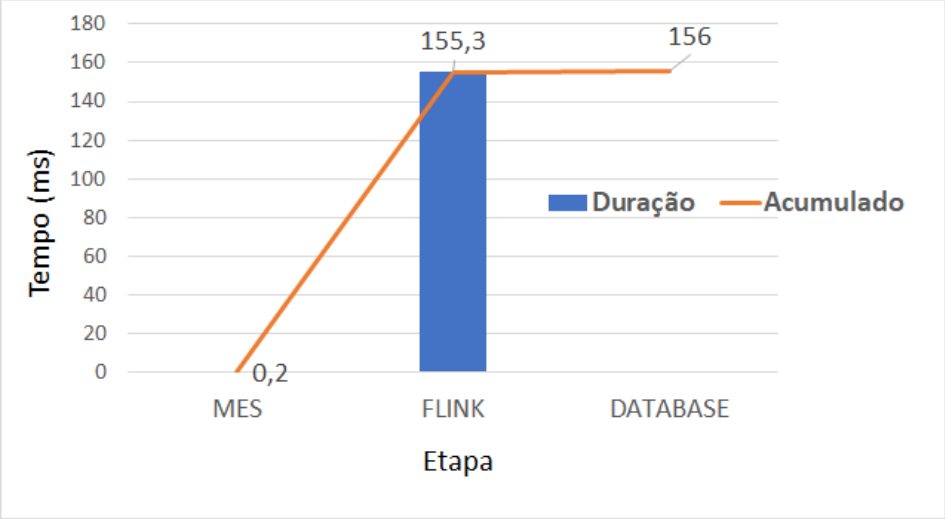


Figura 5.9: Resposta temporal do registo de um alerta na base de dados, num cenário com 2 fontes de dados e envio de 1 Mensagem/segundo

Tabela 5.9: Tempo gasto, em milissegundos, em cada etapa do sistema até o alerta ser registado na base de dados, num cenário com 2 fontes de dados e envio contínuo

#	MES	Flink	Database
#1	**1**	**248**	**5**
#2	**0**	**46**	**1**
#3	**0**	**149**	**1**
#4	**0**	**35**	**2**
#5	**0**	**135**	**1**
#6	**0**	**114**	**1**
#7	**0**	**100**	**1**
#8	**0**	**113**	**2**
#9	**0**	**222**	**1**
#10	**0**	**37**	**9**
Média	**0,1**	**119,9**	**2,4**

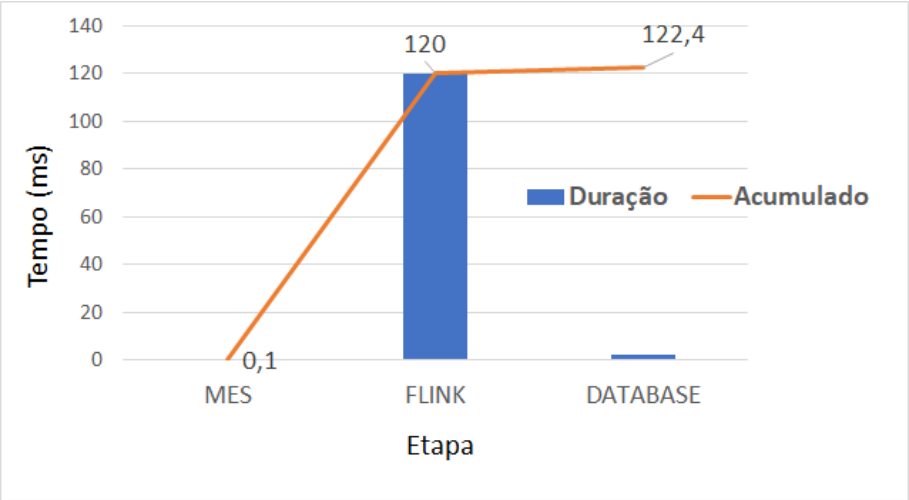


Figura 5.10: Resposta temporal do registo de um alerta na base de dados, num cenário com 2 fontes de dados e envio contínuo

Esta análise ao desempenho do sistema, no processo de registo de alertas, revelou-se importante para comprovar que a tecnologia usada cumpre com os requisitos propostos. Os resultados obtidos, para os três cenários diferentes, mostram que o registo na base de dados demora escassos milissegundos, daí se poder

afirmar que este processo é quase instantâneo. O histórico criado pelo sistema desenvolvido, pode ser acedido através da API descrita na Secção 4.5.1 do capítulo de Implementação, assim como pela ferramenta de visualização Grafana. Esta permite executar *queries* à base de dados e construir uma *dashboard* com vários painéis, onde cada um deles contém um gráfico com o resultado da *query* realizada. A Figura 5.11 representa o gráfico obtido num pedido à base de dados. No caso retratado, a *query* foi feita à tabela *traffic_porto* e é mostrada a evolução, ao minuto, da velocidade na rua Mouzinho da Silveira no Porto. Na parte inferior da Figura, é possível ver como é construída a *query*.

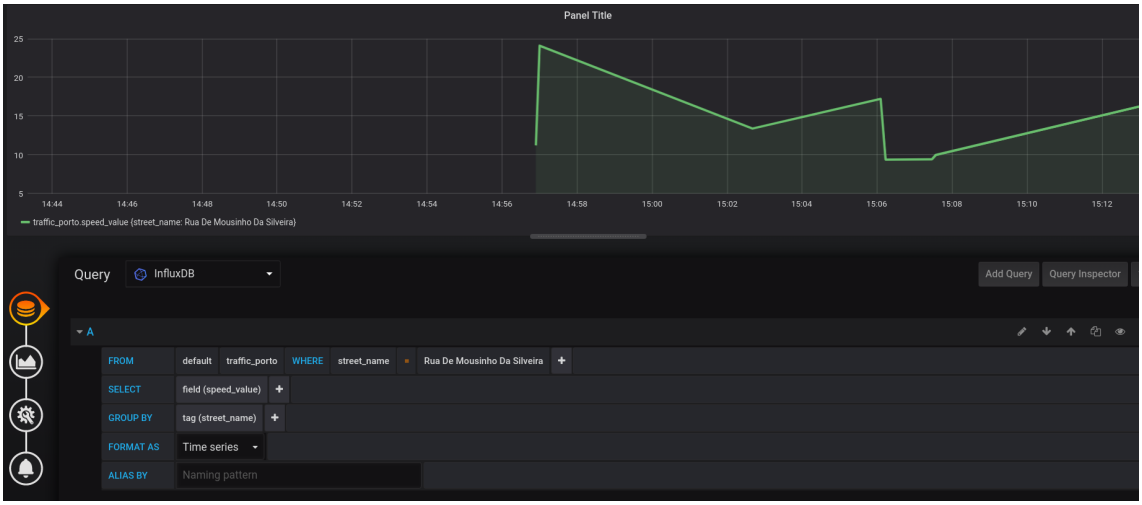


Figura 5.11: Resultado de uma *query* realizada na plataforma Grafana

Depois de se ter avaliado o desempenho da solução desenvolvida, desde a sua fase inicial (entrada de dados), até à produção de alertas para três destinos, em três cenários diferentes, concluiu-se que o sistema teve o comportamento esperado. Contudo, em todos os testes revelados nesta secção, constatou-se que a etapa de processamento apresenta um tempo de resposta relativamente constante, entre 100 ms a 200 ms. Não sendo estes, valores preocupantes, é interessante para a discussão final dos resultados, perceber qual a origem deste tempo de reação. Por isso, de seguida, diminui-se o nível de abstração desta etapa, pelo que se recolhem métricas temporais das funções do motor Flink.

5.2 Motor de Processamento

Para além de ser o módulo encarregue da função mais exigente, o foco do presente trabalho está na componente de processamento, por isso torna-se fun-

damental aprofundar a avaliação de performance nesta etapa. À semelhança do que aconteceu na análise global do sistema (revelada na secção anterior), onde se decompôs o funcionamento em várias etapas, aqui, nos testes ao comportamento do motor Flink, também se fez um separação por fases. O objetivo é averiguar e, posteriormente discutir, qual a função deste módulo que requer maior tempo para ser executada. Decidiu-se então dividir o funcionamento do motor Flink em três funções: *Apply Rule*, *Trigger Alert* e *Send Alert*. A primeira consiste na aplicação da regra predefinida em cada evento recebido. Quando a regra é correspondida, a função *Trigger Alert* é executada, sendo gerado um alerta. Por último, a etapa *Send Alert* traduz-se na saída do alerta do motor de processamento. Estes foram então os três pontos de recolha das métricas temporais. Tal como tinha sido feito nos testes globais, foram recolhidos os *timestamps* de cada função para os três cenários já revelados: uma fonte de dados e envio de um evento por segundo; duas fontes de dados e envio de um evento por segundo; duas fontes de dados e envio contínuo de eventos. Os resultados obtidos podem ser consultados no Anexo E, nas Tabelas E.10, E.11 e E.12, sendo que nesta secção é apresentada a mesma informação de forma sintetizada nas Tabelas 5.10, 5.11 e 5.12, respetivamente. Depois de serem mostrados os resultados dos testes realizados, os mesmos são representados graficamente na Figura 5.12, permitindo assim uma interpretação mais clara do comportamento do módulo.

Tabela 5.10: Tempo gasto, em milissegundos, em cada função do motor Flink até o alerta ser enviado, num cenário com 1 fonte de dados e envio de 1 Mensagem/segundo

#	**Apply Rule**	**Trigger Alert**	**Send Alert**
#1	**193**	**1**	**0**
#2	**173**	**1**	**1**
#3	**178**	**0**	**0**
#4	**181**	**1**	**4**
#5	**153**	**1**	**0**
#6	**51**	**0**	**0**
#7	**52**	**1**	**0**
#8	**130**	**1**	**0**
#9	**127**	**1**	**1**
#10	**221**	**0**	**0**
Média	**145,9**	**0,7**	**0,6**

Tabela 5.11: Tempo gasto, em milissegundos, em cada função do motor Flink até o alerta ser enviado, num cenário com 2 fontes de dados e envio de 1 Mensagem/segundo

#	**Apply Rule**	**Trigger Alert**	**Send Alert**
#1	**226**	**1**	**1**
#2	**201**	**0**	**0**
#3	**108**	**4**	**1**
#4	**194**	**1**	**2**
#5	**191**	**1**	**0**
#6	**196**	**1**	**1**
#7	**189**	**1**	**1**
#8	**187**	**0**	**1**
#9	**186**	**0**	**1**
#10	**94**	**1**	**0**
Média	**177,2**	**1**	**0,8**

Tabela 5.12: Tempo gasto, em milissegundos, em cada função do motor Flink até o alerta ser enviado, num cenário com 2 fontes de dados e envio contínuo

#	**Apply Rule**	**Trigger Alert**	**Send Alert**
#1	**48**	**8**	**3**
#2	**164**	**1**	**1**
#3	**153**	**0**	**0**
#4	**275**	**0**	**0**
#5	**75**	**0**	**1**
#6	**147**	**1**	**2**
#7	**148**	**0**	**0**
#8	**268**	**0**	**0**
#9	**74**	**1**	**0**
#10	**52**	**1**	**1**
Média	**140,4**	**1,2**	**0,8**

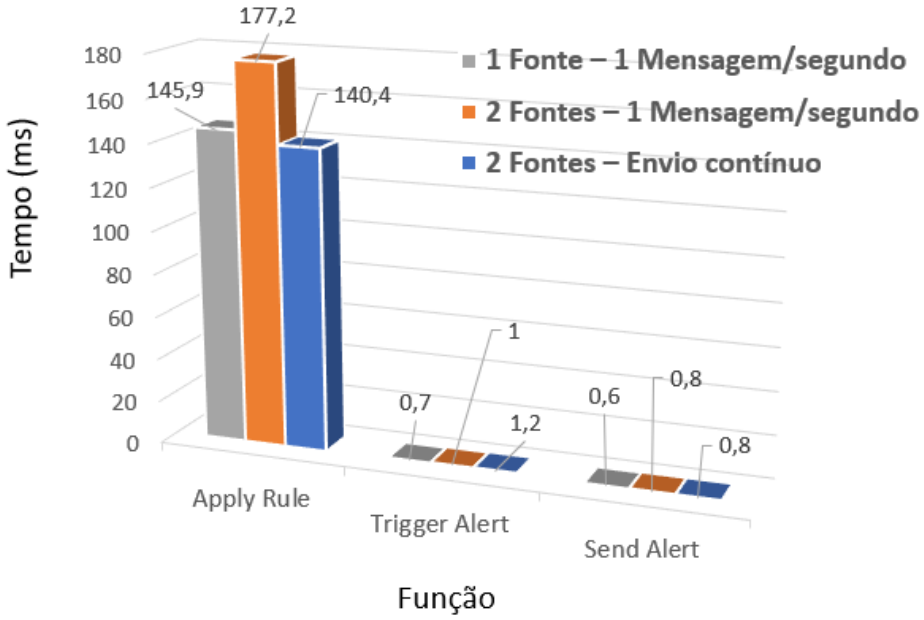


Figura 5.12: Resposta temporal do desempenho das diferentes funções do motor de processamento, para três cenários distintos

Através do gráfico da Figura 5.12, percebe-se rapidamente que a fase que exige mais tempo na componente de processamento é a função que aplica as regras definidas previamente, nas *streams* de dados recebidas (*Apply Rule*). Com recurso aos testes efetuados, não se consegue explicar a razão exata para um maior tempo de resposta nesta função em particular. Porém, conseguem-se descartar duas hipóteses de parâmetros dos quais este tempo não depende: número de fontes de dados e periodicidade da entrada dos mesmos, pois os cenários utilizados diferem entre si nestes aspetos. Seria interessante, numa perspetiva futura, avaliar a influência que a complexidade das regras aplicadas tem no tempo de reação desta função. A regra definida nestes testes consiste no uso do símbolo lógico de comparação $<$ (menor), para detetar, no caso dos dados de tráfego, valores de velocidade inferiores a 10 km/h.

De seguida, para terminar o presente capítulo, avalia-se o desempenho do serviço de atualização de topologias, cuja função permite que todos os tipos de dados sejam tratados pelo sistema. Contudo, no intervalo entre entrada de dados de novos sensores e ativação das regras que os processam, existe a possibilidade de perda de alguns eventos como se discute na próxima secção.

5.3 Atualização de Topologias

Conforme foi dito na fase introdutória deste capítulo, é importante que se faça uma análise crítica ao desempenho do sistema no que diz respeito à atualização de topologias ou, por outras palavras, do controlo das regras ativas no módulo de processamento. De facto, este processo é responsável por fazer com que o sistema de alarmística esteja a processar todos os tipos de dados que recebe, assim como evitar que estejam ativas, as topologias referentes a tipos de dados que não estão a circular no sistema. A remoção das regras que não são necessárias naquele momento, está mais relacionada com a eficiência da solução do que propriamente com a sua eficácia, ou seja, não é um aspeto crítico para o correto funcionamento da solução. Por outro lado, a submissão de topologias é crucial para o cumprimento do propósito do sistema desenvolvimento. Foram por isso efetuados testes à função que submete regras no motor Flink, que consistem no registo dos *timestamps* à entrada e à saída da mesma. Esta análise temporal foi realizada para os casos da submissão de uma, duas e três topologias. Os resultados obtidos são exibidos nas Tabelas 5.13, 5.14 e 5.15, onde a última coluna retrata a diferença entre as métricas temporais das duas colunas que a precede.

Tabela 5.13: Tempos de resposta, em milissegundos, do serviço de atualização de regras, na submissão de 1 topologia

#	**To submit**	**Submitted**	**Submission**
#1	**1570709714400**	**1570709717289**	**2889**
#2	**1570710455266**	**1570710458887**	**3621**
#3	**1570710588950**	**1570710591055**	**2105**
#4	**1570710699569**	**1570710702044**	**2475**
#5	**1570710821847**	**1570710824449**	**2602**
#6	**1570710930607**	**1570710933244**	**2637**
#7	**1570711051710**	**1570711054181**	**2471**
#8	**1570711167707**	**1570711169760**	**2053**
#9	**1570711278473**	**1570711281077**	**2604**
#10	**1570711398062**	**1570711400600**	**2538**
Média			**2599,5**

Tabela 5.14: Tempos de resposta, em milissegundos, do serviço de atualização de regras, na submissão de 2 topologias

#	To submit	Submitted	Submission
#1	**1570711562422**	**1570711567483**	**5061**
#2	**1570711742681**	**1570711747561**	**4880**
#3	**1570713479358**	**1570713484658**	**5300**
#4	**1570713610261**	**1570713616579**	**6318**
#5	**1570713739783**	**1570713744834**	**5051**
#6	**1570713865896**	**1570713871434**	**5538**
#7	**1570713995465**	**1570714002391**	**6926**
#8	**1570714125553**	**1570714132269**	**6716**
#9	**1570714239512**	**1570714245779**	**6267**
#10	**1570714363511**	**1570714368841**	**5330**
Média			**5738,7**

Tabela 5.15: Tempos de resposta, em milissegundos, do serviço de atualização de regras, na submissão de 3 topologias

#	To submit	Submitted	Submission
#1	**1570714804251**	**1570714813390**	**9139**
#2	**1570715152712**	**1570715162358**	**9646**
#3	**1570715281785**	**1570715290680**	**8895**
#4	**1570715415916**	**1570715425899**	**9983**
#5	**1570715564152**	**1570715572101**	**7949**
#6	**1570715695878**	**1570715704910**	**9032**
#7	**1570715836032**	**1570715843822**	**7790**
#8	**1570715960449**	**1570715968618**	**8169**
#9	**1570716096385**	**1570716104926**	**8541**
#10	**1570716297600**	**1570716304459**	**6859**
Média			**8600,3**

Com base nas três tabelas anteriores, podem ser tiradas conclusões relativas ao tempo gasto na submissão de novas topologias. Consequentemente, é possível estimar o número de eventos perdidos sempre que há um ou mais tipos de dados a entrar no sistema. Para isso, obtém-se a média do tempo de resposta do serviço de atualização de regras por cada fonte de dados nova, a partir dos valores médios

das tabelas referidas:

$$\frac{2599,5 + \frac{5738,7}{2} + \frac{8600,3}{3}}{3} \approx 2778,5ms \approx 2,8s$$

Desta operação conclui-se que o sistema demora aproximadamente 2,8 s a submeter uma topologia. Tendo em conta que este serviço é executado ao fim de cada 20 eventos recebidos e que a frequência com que eles são enviados, varia, pode-se começar a desenvolver uma equação que exprime uma estimativa do número de eventos não processados (perdidos), sempre que há entrada de novas fontes ou sensores:

$$y = f(x, u, v) = \frac{2,8x}{u} + v, v = \{z \in \mathbb{N} | 1 \leq z \leq 20\}$$

O que a equação sugere é que o número de eventos perdidos - y - depende então de 3 variáveis: x, que é o número de topologias que são necessárias submeter; u, a periodicidade, em segundos, com que os eventos da nova fonte de dados são enviados; v, que consiste no número de mensagens relativas a esse novo tipo de dados, que já tenham sido enviadas antes do processo de submissão, pelo que este valor nunca será maior que 20. Este número definido como a taxa de atualização de topologias, deve-se ao facto de se ter concluído que, uma taxa menor poderia não ser muito eficiente, na medida em que um sensor pode apresentar falhas iniciais no envio de informação, pelo que se espera que a solução só ative regras sobre dados que já estão a entrar de forma consistente no sistema.

5.4 Sumário

Neste capítulo foram demonstrados os testes que se fizeram ao sistema desenvolvido. Estes testes incidiram sobre três vertentes: a globalidade do sistema, onde se percorreu todo o fluxo de dados, desde a sua entrada até à sua saída; o motor de processamento, que foi dividido em três fases com o objetivo de analisar a nível temporal, o comportamento das funções que o constituem; o serviço de controlo de regras de forma a quantificar uma possível perda de informação durante o seu processo.

Pelos resultados obtidos nos testes globais ao sistema, concluiu-se que, nos três cenários implementados, onde se variou a frequência com que a informação entra no sistema assim como o número de fontes de dados, as diferenças de desempenho da solução denotadas, deveram-se ao tipo de canal por onde seguem os *outputs*. O envio dos alertas por e-mail revelou-se como o método mais demoroso, porém, o utilizador é notificado em menos de dois segundos. Por outro lado, a produção dos resultados do processamento para um servidor *WebSockets* confirmou a rapidez

de comunicação que esta tecnologia oferece. O desempenho do sistema, quando se mediu o seu tempo de resposta no registo de informação na base de dados, foi o expectável uma vez que a ferramenta utilizada, o InfluxDB, foi projetada para permitir o armazenamento de dados sem latência significativa.

Em relação à análise mais profunda da componente de processamento, isto é, o motor Flink, a partir da recolha dos *timestamps* à entrada e saída de cada função, percebe-se que a maior parte do tempo gasto nesta etapa concentra-se na função que deteta os padrões nos dados recebidos.

Por fim, os testes efetuados ao processo de atualização de topologias, permitiram concluir que o número de eventos não processados ou perdidos durante a sua execução depende de três fatores: número de topologias a submeter; periodicidade de entrada dos dados; quantidade de dados do novo tipo, enviada antes deste serviço ser executado.

Em suma, comprovou-se a capacidade do sistema desenvolvido em produzir alertas em tempo real, isto é, na ordem dos milissegundos. As pessoas responsáveis por supervisionar o desenvolvimento deste projeto, por parte da Ubiwhere, olham com agrado para as métricas aqui exibidas, nomeadamente para os resultados obtidos no envio dos alertas para um servidor *WebSockets*. Esta avaliação mostrou muito bons indicadores para uma futura integração na sua plataforma, uma vez que do lado da empresa, apenas é necessário criar uma página/interface que espelhe para o utilizador os alertas recebidos por aquele servidor.

Capítulo 6

Conclusões e Trabalho Futuro

Este trabalho propôs-se a desenvolver um sistema de analítica, capaz de disparar alertas em tempo real no caso da ocorrência de algum evento que perturbe o bom funcionamento de uma cidade, em todas as suas dimensões. Apesar do foco principal ser o módulo de processamento, a solução desenvolvida contempla todas as etapas do fluxo de dados num sistema típico de processamento, isto é, desde a receção da informação até à entrega dos resultados às partes interessadas. Na componente de processamento, o pretendido foi gerar alertas através da comparação dos dados recebidos com regras que são ativadas e removidas automaticamente, mediante o tipo de informação a circular no sistema.

Primeiro, foi feita uma análise comparativa das principais técnicas de analítica que existem, sendo que, de modo a contextualizar o leitor, foram antes introduzidos dois conceitos: *Smart City* e *IoT*. Esta abordagem fez com que se tivesse uma perfeita noção da necessidade de explorar a componente de análise. Aqui se concluiu que, perante os requisitos do sistema que se desenvolveu, o CEP é o motor de processamento que melhor se adequa aos mesmos. Este método consiste na deteção de padrões em *streams* de dados. Por isso, decidiu-se que estes padrões seriam as nossas regras.

Seguiu-se a apresentação do problema em questão, onde inicialmente se faz uma abstração muito *high level* do mesmo. Posteriormente, dividiram-se esses desafios e fez-se um trabalho de investigação sobre as ferramentas e tecnologias com as quais se poderia montar o sistema, resultando numa fase mais técnica do conteúdo da dissertação. Esta etapa permitiu tirar conclusões sobre as vantagens e desvantagens do uso de cada ferramenta analisada: para a troca de mensagens ao longo das componentes do sistema optou-se por usar o Apache Kafka, muito por culpa da sua escalabilidade assim como do seu desempenho no tratamento de dados produzidos continuamente; para o processamento de dados escolheu-se o Apache Flink, baseado na comparação desta com outras duas ferramentas. A única conclusão final obtida nesta fase diz respeito à base de dados que se

utilizaria na solução desenvolvida e essa escolha acabou ser o InfluxDB. Uma das premissas deste trabalho é que todos os processos devem ser otimizados de forma a não causar nenhum atraso na produção de resultados e como tal, o registo na base de dados não podia ser exceção, sendo que o InfluxDB denota um grande desempenho neste aspeto.

Tendo o problema bem definido e conhecimento sobre o que as ferramentas existentes são capazes de oferecer, ficou-se apto a desenhar a arquitetura da solução desenvolvida. Numa fase inicial entendeu-se por bem dar a conhecer as funções dos módulos do funcionamento do sistema sem se revelar as tecnologias com que cada um deles foi implementado. Posteriormente foram exibidos dois possíveis casos de uso para o sistema desenvolvido onde se concluiu que soluções deste calibre podem ter inúmeras aplicações tamanha é a variedade de relações que se podem estabelecer entre fontes de dados diferentes.

Finalmente, em termos de implementação do protótipo decidiu-se dividir o trabalho em várias camadas. Aqui começou-se por desenvolver o sistema responsável por distribuir as mensagens, onde se chegou à conclusão que a melhor ferramenta para o fazer é o Kafka. O segundo passo foi o mais importante, isto é, o módulo do processamento. Depois de se ter escolhido o Flink como *framework* de processamento, as dificuldades mais relevantes sentidas no desenvolvimento deste projeto concentraram-se nesta etapa. De facto, a curva de aprendizagem do Flink revelou-se relativamente acentuada, tendo sido investido mais tempo nesta parte. Contudo, com alguma insistência e com a ajuda da documentação existente, conseguiu-se construir um motor CEP suficientemente robusto e preciso para gerar alertas, em tempo real, a partir de regras previamente definidas. Os resultados do processamento são sempre gerados associando-lhes um nível de severidade, que é o parâmetro que vai definir como é que ele é entregue aos utilizadores. Devido à rapidez da comunicação cliente/servidor que o protocolo *WebSockets* proporciona, este foi o método escolhido para notificar as partes interessadas sobre os alertas com severidade mais elevada. Terminou-se a implementação quando se colocaram todos os serviços necessários ao funcionamento do sistema a serem executados de forma automática através do uso do *Docker*.

A elaboração desta dissertação permitiu adquirir novas competências na área da analítica de Big Data com especial aprofundamento do domínio das *Real Time Analytics*. Pode-se aqui afirmar que as decisões tomadas ao longo do desenvolvimento do projeto foram fortemente sustentadas pela união do estudo de inúmeros artigos, presentes na literatura disponibilizada maioritariamente pelo IEEE, vista como a maior e mais popular organização de apoio ao desenvolvimento tecnológico, com os conhecimentos já obtidos por experiência académica e profissional.

6.1 Satisfação dos Objetivos

Na Tabela 6.1 é mostrado de forma pragmática em que etapa do desenvolvimento se cumpriram os objetivos propostos no Capítulo 1. Na Secção 4.2.1, onde foi descrito o módulo M.E.S, percebe-se a capacidade do sistema em suportar qualquer tipo de dados, sendo deve ser possível identificá-lo pelo conteúdo das mensagens. O trabalho detalhado na Secção 4.3 demonstra o cumprimento do objetivo de aplicar regras nos dados recebidos, o que é feito através da biblioteca CEP do Flink, sendo que os alertas gerados nesse processo são entregues ao utilizador em escassos milissegundos (tempo real), afirmação comprovada pelos testes revelados na Secção 5.1. Quanto ao mecanismo de controlo das regras, este é realizado de forma automático como se comprometeu a fazer, sendo que a explicação do modo como foi conseguido encontra-se na Secção 4.4.1. O histórico dos alertas produzidos pode ser acedido a partir de uma REST API, cujo *endpoint* foi descrito na Secção 4.5.1, que foi desenvolvida em Python na *framework* Fast API. Nas Secções 4.5 e 5.1 foi mostrado que se criaram dois canais por onde os resultados podem chegar aos utilizadores - e-mail e um servidor *WebSocket*. Por fim, com recurso à tecnologia Docker, foi possuir construir um protótipo que pode ser instalado em qualquer máquina, sendo este processo explicado na Secção 4.6.1.

Tabela 6.1: Secções onde se atingiram os objetivos propostos

Objetivo	**Secção**
Suporte de pelo menos duas fontes de dados distintas	**4.2.1**
Aplicação de regras predefinidas nos dados recebidos, gerando alertas em tempo real;	**4.3; 5.1**
Atualização automática das regras ativas	**4.4.1**
Construção de um histórico com todos os alertas gerados, ao qual se possa aceder através de uma API;	**4.5.1**
Gerar alertas para, pelo menos, dois canais distintos	**4.5; 5.1**
Portabilidade do protótipo	**4.6.1**

6.2 Futuras Melhorias

Apesar de terem sido cumpridos os objetivos propostos, um projeto de software nunca se pode considerar como totalmente acabado, sendo que existe sempre algo que pode ser adicionado seja pela evolução das tecnologias ou pela adição de funcionalidades. A solução desenvolvida não é exceção e como trabalho futuro, são aqui apresentadas três possibilidades que se considera que poderiam melhorar a performance e robustez da mesma.

A primeira foca-se mais na escalabilidade do sistema, isto é, a adição de mais filtros à entrada dos dados provenientes dos sensores. O que se tinha pensado seria a inclusão de um filtro por localização para além do existente filtro por tipo de dados. Uma vez que já são usadas ferramentas poderosas como o Kafka e o Flink que suportam facilmente o paralelismo de processos, esta adição permitiria criar um sistema mais central onde, os dados de uma determinada cidade fossem enviados para os módulos de processamento que contêm as regras definidas especialmente para aquela cidade.

Agora mais numa ótica de acrescentar *features* que complementem significativamente o sistema desenvolvido, seria pertinente adicionar uma componente de analítica por *batches*, isto é, *Batch Analytics*. No atual formato da solução, os alertas gerados pelo módulo de processamento já estão a ser armazenados numa base de dados, sendo que o que aqui se pretendia era guardar também os dados antes de serem processados. O que aqui se propõe é desenvolver um sistema baseado na arquitetura *lambda*, onde são consideradas três camadas: *batch*, *speed* e *serving* [131]. A primeira consiste no processamento de informação já armazenada, sendo um módulo mais lento. O *speed layer* diz respeito ao que se desenvolveu neste trabalho, ou seja, processamento em tempo real. Já a camada de *serving* é responsável por combinar os *outputs* das camadas anteriores, através da realização de *queries* às mesmas. Isto permitiria uma maior precisão nos alertas produzidos, uma vez que seriam sustentados também por uma aprendizagem das tendências passadas.

Por fim, outra adição interessante embora mais complexa, seria a colocação de algum tipo de inteligência artificial à entrada do sistema de forma a que fosse possível perceber, sem qualquer indicação nesse sentido, qual o tipo de dados que entra no mesmo. Na versão atual da solução, cada mensagem contém um elemento identificador do contexto da mesma. Contudo seria interessante não haver essa necessidade e o sistema ser capaz de, apenas a partir dos restantes elementos, decifrar de que tipo de dados se trata (tráfego, qualidade do ar, redes sociais, etc.).

Bibliografia

[1] "Urban Platform." https://urbanplatform.ubiwhere.com/#urban-platform. [cited on p. 1]

[2] J. Murray, "Cloud network architecture and ICT - Modern Network Architecture." https://itknowledgeexchange.techtarget.com/modern-network-architecture/cloud-network-architecture-and-ict/. [cited on p. 5]

[3] "What is ICT? | Business | tutor2u." https://www.tutor2u.net/business/reference/what-is-ict. [cited on p. 5]

[4] V. Albino, U. Berardi, and R. M. Dangelico, "Smart Cities: Definitions, Dimensions, Performance, and Initiatives," *Journal of Urban Technology*, vol. 22, pp. 3–21, jan 2015. [cited on p. 6]

[5] H. Fernando, A. Velosa, L. Anavitarte, and B. Tratz-Ryan, "Market Trends: Smart Cities Are the New Revenue Frontier for Technology Providers, 2011," in *Gartner Research*, Gartner, 2011. [cited on p. 6]

[6] A. Arroub, B. Zahi, E. Sabir, and M. Sadik, "A literature review on Smart Cities: Paradigms, opportunities and open problems," in *2016 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pp. 180–186, IEEE, oct 2016. [cited on p. 7]

[7] T. M. Vinod Kumar and B. Dahiya, "Smart Economy in Smart Cities," *Advances in 21st Century Human Settlements*, pp. 3–76, 2017. [cited on p. 7, 8]

[8] G. V. Pereira, P. Parycek, E. Falco, and R. Kleinhans, "Smart governance in the context of smart cities: A literature review," *Information Polity*, vol. 23, pp. 143–162, jun 2018. [cited on p. 8]

[9] Schneider, "Smart Cities cornerstone series URBAN MOBILITY IN THE SMART CITY AGE," tech. rep., Schneider Eletric, 2014. [cited on p. 8]

[10] "Virtual Singapore." https://www.smartnation.sg/what-is-smart-nation/initiatives/Urban-Living/virtual-singapore. [cited on p. 9]

[11] C. A. Medina, M. R. Perez, and L. C. Trujillo, "IoT Paradigm into the Smart City Vision: A Survey," in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 695–704, IEEE, jun 2017. [cited on p. 9, 12]

[12] "What Makes London One of the Smartest Cities in the World." https://www.smartcity.press/londons-smart-city-initiatives/. [cited on p. 10]

[13] "6 of the smartest smart cities in the world - Richard van Hooijdonk." https://www.richardvanhooijdonk.com/en/blog/6-smartest-smart-cities-world/. [cited on p. 11]

[14] Y. Sun, Y. Xia, H. Song, and R. Bie, "Internet of Things Services for Small Towns," in *2014 International Conference on Identification, Information and Knowledge in the Internet of Things*, pp. 92–95, IEEE, oct 2014. [cited on p. 11, 12]

[15] "2017 Roundup Of Internet Of Things Forecasts." https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts/#123ec3741480. [cited on p. 12]

[16] "O que é CPAP e BIPAP? | Blog do cpap." http://www.cpapfit.com.br/blog/o-que-e-cpap-e-bipap/. [cited on p. 12]

[17] Z. Hassan, H. Ali, and M. Badawy, "Internet of things (iot): Definitions, challenges, and recent research directions," *International Journal of Computer Applications*, vol. 128, pp. 975–8887, 10 2015. [cited on p. 12, 13]

[18] "Data Never Sleeps 6 | Domo." https://www.domo.com/learn/data-never-sleeps-6#/. [cited on p. 15]

[19] P. Radhika, P. P. Kumar, S. L. Sailaja, and V. Gayatri, "Confrontation and oppurtunities of big data — A survey," in *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, pp. 153–157, IEEE, mar 2017. [cited on p. 16]

[20] D. Laney, "Application Delivery Strategies," tech. rep., Meta Group, 2001. [cited on p. 16]

[21] "The internet of things and big data: Unlocking the power | ZDNet." https://www.zdnet.com/article/the-internet-of-things-and-big-data-unlocking-the-power/. [cited on p. 16]

[22] M. A.-u.-d. Khan, M. F. Uddin, and N. Gupta, "Seven V's of Big Data understanding Big Data to extract value," in *Proceedings of the 2014 Zone 1 Conference of the American Society for Engineering Education*, pp. 1–5, IEEE, apr 2014. [cited on p. 17, 18]

[23] R. Patgiri and A. Ahmed, "Big Data: The V's of the Game Changer Paradigm," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 17–24, IEEE, dec 2016. [cited on p. 17, 18]

[24] R. Dautov and S. Distefano, "Quantifying volume, velocity, and variety to support (Big) data-intensive application development," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2843–2852, IEEE, dec 2017. [cited on p. 17]

[25] M. van Rijmenam, "Why The 3V's Are Not Sufficient To Describe Big Data." https://datafloq.com/read/3vs-sufficient-describe-big-data/166. [cited on p. 17]

[26] A. Alsaig, V. Alagar, and O. Ormandjieva, "A Critical Analysis of the V-Model of Big Data," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 1809–1813, IEEE, aug 2018. [cited on p. 17]

[27] A. Shi-Nash and D. R. Hardoon, "DATA ANALYTICS AND PREDICTIVE ANALYTICS IN THE ERA OF BIG DATA," in *Internet of Things and Data Analytics Handbook*, pp. 329–345, Hoboken, NJ, USA: John Wiley & Sons, Inc., dec 2016. [cited on p. 19]

[28] H. Kaur and A. Phutela, "Commentary upon descriptive data analytics," in *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, pp. 678–683, IEEE, jan 2018. [cited on p. 19, 20]

[29] R. T. Hans and E. Mnkandla, "Work in progress — Design and development of a project management intelligence (PMInt) tool," in *2016 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pp. 308–313, IEEE, nov 2016. [cited on p. 19]

[30] "MonkeyLearn - Text Analysis." https://monkeylearn.com/. [cited on p. 20]

[31] F. Sun, Y. Pan, J. White, and A. Dubey, "Real-Time and Predictive Analytics for Smart Public Transportation Decision Support System," in *2016*

IEEE International Conference on Smart Computing (SMARTCOMP), pp. 1–8, IEEE, may 2016. [cited on p. 20]

[32] S.-K. Song, D. J. Kim, M. Hwang, J. Kim, D.-H. Jeong, S. Lee, H. Jung, and W. Sung, "Prescriptive Analytics System for Improving Research Power," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, pp. 1144–1145, IEEE, dec 2013. [cited on p. 20]

[33] N. Martin, M. Swennen, B. Depaire, M. Jans, A. Caris, and K. Vanhoof, "Batch Processing: Definition and Event Log Identification," tech. rep., Hasselt University, Belgium, 2015. [cited on p. 21]

[34] F. Abuqabita, R. Al-Omoush, and J. Alwidian, "A Comparative Study on Big Data Analytics Frameworks, Data Resources and Challenges," *Modern Applied Science*, vol. 13, p. 1, jun 2019. [cited on p. 21, 26]

[35] T. Chardonnens, P. Cudre-Mauroux, M. Grund, and B. Perroud, "Big data analytics on high Velocity streams: A case study," in *2013 IEEE International Conference on Big Data*, pp. 784–787, IEEE, oct 2013. [cited on p. 21]

[36] "Apache Hadoop." https://hadoop.apache.org/. [cited on p. 21]

[37] W. Tom, *Hadoop: The Definitive Guide. The Definitive Guide FOURTH EDITION*. O'Reilly, 2015. [cited on p. 21]

[38] M. Goudarzi, "Heterogeneous Architectures for Big Data Batch Processing in MapReduce Paradigm," *IEEE Transactions on Big Data*, vol. 5, pp. 18–33, mar 2019. [cited on p. 22]

[39] S. Shahrivari, "Beyond Batch Processing: Towards Real-Time and Streaming Big Data," *Computers*, vol. 3, pp. 117–129, oct 2014. [cited on p. 22, 26]

[40] H. Vo, J. Kong, D. Teng, Y. Liang, A. Aji, G. Teodoro, and F. Wang, "MaReIA: a cloud MapReduce based high performance whole slide image analysis framework," *Distributed and Parallel Databases*, vol. 37, pp. 251–272, jun 2019. [cited on p. 22]

[41] B. Yadranjiaghdam, N. Pool, and N. Tabrizi, "A Survey on Real-Time Big Data Analytics: Applications and Tools," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 404–409, IEEE, dec 2016. [cited on p. 24]

[42] N. Mohamed and J. Al-Jaroodi, "Real-time big data analytics: Applications and challenges," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 305–310, IEEE, jul 2014. [cited on p. 24, 25]

[43] Manjunatha and B. Annappa, "Real Time Big Data Analytics in Smart City Applications," in *2018 International Conference on Communication, Computing and Internet of Things (IC3IoT)*, pp. 279–284, IEEE, feb 2018. [cited on p. 25]

[44] S. Trinks and C. Felden, "Real time analytics — State of the art: Potentials and limitations in the smart factory," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 4843–4845, IEEE, dec 2017. [cited on p. 26]

[45] M. Ghesmoune, M. Lebbah, and H. Azzag, "Micro-Batching Growing Neural Gas for Clustering Data Streams Using Spark Streaming," *Procedia Computer Science*, vol. 53, pp. 158–166, 2015. [cited on p. 26]

[46] R. Tudoran, B. Nicolae, and G. Brasche, "Data Multiverse: The Uncertainty Challenge of Future Big Data Analytics," *Lecture Notes in Computer Science*, pp. 17–22, 2017. [cited on p. 26]

[47] V. Akila, V. Govindasamy, and S. Sandosh, "Complex event processing over uncertain events: Techniques, challenges, and future directions," in *2016 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)*, pp. 204–221, IEEE, apr 2016. [cited on p. 27]

[48] G. Liu, W. Zhu, C. Saunders, F. Gao, and Y. Yu, "Real-time Complex Event Processing and Analytics for Smart Grid," *Procedia Computer Science*, vol. 61, pp. 113–119, 2015. [cited on p. 27]

[49] F. F. Scattone and K. R. Braghetto, "A Microservices Architecture for Distributed Complex Event Processing in Smart Cities," tech. rep., University of Sao Paulo, 2018. [cited on p. 27]

[50] J. Bates, "Secrets Revealed: Trading Tools Uncover Hidden Opportunities | GlobalTrading," *Global Trading*, 2011. [cited on p. 27]

[51] F. Xiao, C. Zhan, H. Lai, and L. Tao, "Parallel processing data streams in complex event processing systems," in *2017 29th Chinese Control And Decision Conference (CCDC)*, pp. 6157–6160, IEEE, may 2017. [cited on p. 27]

[52] R. Kitchin, "The Real-Time City? Big Data and Smart Urbanism," *SSRN Electronic Journal*, jul 2013. [cited on p. 27]

[53] M. Mongiello, L. Patrono, T. Di Noia, F. Nocera, A. Parchitelli, I. Sergi, and P. Rametta, "A Complex Event Processing based smart aid system for fire and danger management," in *2017 7th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI)*, pp. 44–49, IEEE, jun 2017. [cited on p. 27]

[54] A. Dhillon, S. Majumdar, M. St-Hilaire, and A. El-Haraki, "MCEP: A Mobile Device Based Complex Event Processing System for Remote Healthcare," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 203–210, IEEE, jul 2018. [cited on p. 28]

[55] I. Vasconcelos, R. O. Vasconcelos, B. Olivieri, M. Roriz, M. Endler, and M. C. Junior, "Smartphone-based outlier detection: a complex event processing approach for driving behavior detection," *Journal of Internet Services and Applications*, vol. 8, p. 13, dec 2017. [cited on p. 28]

[56] P. Russom, "Introduction to Big Data Analytics," tech. rep., TDWI, 2011. [cited on p. 31]

[57] "Big Data and the Challenge of Unstructured Data." https://www.ciklum.com/blog/big-data-and-the-challenge-of-unstructured-data/. [cited on p. 31]

[58] N. T. Tariq RS, "Big Data Challenges," *Computer Engineering & Information Technology*, vol. 04, no. 03, 2015. [cited on p. 32]

[59] A. Meslin, N. Rodriguez, and M. Endler, "A Scalable Multilayer Middleware for Distributed Monitoring and Complex Event Processing for Smart Cities," in *2018 IEEE International Smart Cities Conference (ISC2)*, pp. 1–8, IEEE, sep 2018. [cited on p. 32]

[60] A. M. de Souza, A. Boukerche, G. Maia, E. Cerqueira, A. A. F. Loureiro, and L. A. Villas, "SPARTAN: A Solution to Prevent Traffic Jam with Real-Time Alert and Re-Routing for Smart City," in *2016 IEEE 84th Vehicular Technology Conference (VTC-Fall)*, pp. 1–5, IEEE, sep 2016. [cited on p. 32]

[61] A. Ksiksi, S. Al Shehhi, and R. Ramzan, "Intelligent Traffic Alert System for Smart Cities," in *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pp. 165–169, IEEE, dec 2015. [cited on p. 32]

[62] S. R. Garzon, S. Walther, S. Pang, B. Deva, and A. Küpper, "Urban air pollution alert service for smart cities," in *ACM International Conference Proceeding Series*, (New York, New York, USA), pp. 1–8, ACM Press, 2018. [cited on p. 33]

[63] "ProCiv." http://www.prociv.pt/pt-pt/SITUACAOOPERACIONAL/Paginas/default.aspx. [cited on p. 34]

[64] B. R. Hiraman, C. V. M, and C. Karve Abhijeet, "A Study of Apache Kafka in Big Data Stream Processing," in *2018 International Conference on Information , Communication, Engineering and Technology (ICICET)*, pp. 1–3, IEEE, aug 2018. [cited on p. 36]

[65] "Point-to-Point Messaging Domain (The Java EE 6 Tutorial)." https://docs.oracle.com/cd/E19798-01/821-1841/bnceb/index.html. [cited on p. 36]

[66] N. Aizenbud-Reshef, "Coverage analysis for message flows," in *Proceedings 12th International Symposium on Software Reliability Engineering*, pp. 276–286, IEEE Comput. Soc, 2001. [cited on p. 36]

[67] "RabbitMQ tutorial - Publish/Subscribe — RabbitMQ." https://www.rabbitmq.com/tutorials/tutorial-three-python.html. [cited on p. 36]

[68] "RabbitMQ tutorial - "Hello world!"— RabbitMQ." https://www.rabbitmq.com/tutorials/tutorial-one-python.html. [cited on p. 36]

[69] J. Kreps, L. Corp, N. Narkhede, and J. Rao, *Kafka: a Distributed Messaging System for Log Processing*. Semanticscholar, 2011. [cited on p. 36, 38, 39]

[70] "Apache Kafka." https://kafka.apache.org/. [cited on p. 36]

[71] "Messaging that just works — RabbitMQ." https://www.rabbitmq.com/. [cited on p. 36]

[72] "ActiveMQ." https://activemq.apache.org/. [cited on p. 36]

[73] "Home - Apache Qpid™." https://qpid.apache.org/index.html. [cited on p. 36]

[74] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," in *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, pp. 132–137, IEEE, sep 2015. [cited on p. 37]

[75] V. John and X. Liu, "A Survey of Distributed Message Broker Queues," tech. rep., University of Waterloo, 2017. [cited on p. 37, 39]

[76] "RabbitMQ » Blog Archive » Sizing your Rabbits - Messaging that just works." https://www.rabbitmq.com/blog/2011/09/24/sizing-your-rabbits/#more-97. [cited on p. 37]

[77] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17*, (New York, New York, USA), pp. 227–238, ACM Press, 2017. [cited on p. 37, 39]

[78] E. Shahverdi, "Comparative Evaluation for the Performance of Big Stream Processing Systems," tech. rep., University of Tartu, 2018. [cited on p. 37, 41, 42, 45]

[79] R. Shree, T. Choudhury, S. C. Gupta, and P. Kumar, "KAFKA: The modern platform for data management and analysis in big data domain," in *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, pp. 1–5, IEEE, aug 2017. [cited on p. 37]

[80] "Apache kafka." https://kafka.apache.org/documentation/#producerapi. [cited on p. 38]

[81] P. Le Noac'h, A. Costan, and L. Bouge, "A performance evaluation of Apache Kafka in support of big data streaming applications," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 4803–4806, IEEE, dec 2017. [cited on p. 38, 39]

[82] S. Kuboi, K. Baba, S. Takano, and K. Murakami, "An Evaluation of a Complex Event Processing Engine," in *2014 IIAI 3rd International Conference on Advanced Applied Informatics*, pp. 190–193, IEEE, aug 2014. [cited on p. 40]

[83] K. Jayan and A. K. Rajan, "Preprocessor for Complex Event Processing System in Network Security," in *2014 Fourth International Conference on Advances in Computing and Communications*, pp. 187–189, IEEE, aug 2014. [cited on p. 40]

[84] "Esper - EsperTech." http://www.espertech.com/esper/. [cited on p. 40, 41]

[85] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo, "A Comparative Study on Streaming Frameworks for Big Data," tech. rep., Latin America Data Science Workshop, 2018. [cited on p. 41, 47]

[86] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo, "An experimental survey on big data frameworks," *Future Generation Computer Systems*, 2018. [cited on p. 41, 42, 44, 47]

[87] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking Distributed Stream Data Processing Systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1507–1518, IEEE, apr 2018. [cited on p. 41, 47]

[88] "Apache Storm." https://storm.apache.org/. [cited on p. 41, 42, 47]

[89] "Apache Flink: Stateful Computations over Data Streams." https://flink.apache.org/. [cited on p. 41, 47]

[90] "Heron." https://apache.github.io/incubator-heron/. [cited on p. 41, 43, 47]

[91] S. Chatterjee and C. Morin, "Experimental Study on the Performance and Resource Utilization of Data Streaming Frameworks," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 143–152, IEEE, may 2018. [cited on p. 41, 47]

[92] D.-H. Im, C.-H. Cho, and I. Jung, "Detecting a large number of objects in real-time using apache storm," in *2014 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 836–838, IEEE, oct 2014. [cited on p. 41]

[93] "Tutorial." https://storm.apache.org/releases/2.0.0/Tutorial.html. [cited on p. 41, 42]

[94] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang, "Twitter Heron: Towards Extensible Streaming Engines," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 1165–1172, IEEE, apr 2017. [cited on p. 43, 44]

[95] "Heron: Real-time Stream Data Processing at Twitter - YouTube." https://www.youtube.com/watch?v=pUaFOuGgmco. [cited on p. 43, 44]

[96] "Apache Flink 1.9 Documentation: Streaming Connectors." https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/connectors/. [cited on p. 44]

[97] "Apache Flink: What is Apache Flink? — Architecture." https://flink.apache.org/flink-architecture.html. [cited on p. 44]

[98] "Apache Flink: Powered by Flink." https://flink.apache.org/poweredby.html. [cited on p. 44]

[99] "Apache Flink 1.9 Documentation: Dataflow Programming Model." https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html. [cited on p. 45]

[100] P. Carbone, A. Katsifodimos, Kth, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink^{TM}: Stream and Batch Processing in a Single Engine," tech. rep., IC2EW, 2015. [cited on p. 45, 46]

[101] "Apache Flink 1.9 Documentation: FlinkCEP - Complex event processing for Flink." https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/libs/cep.html#flinkcep-complex-event-processing-for-flink. [cited on p. 46]

[102] "Apache Flink 1.9 Documentation: Distributed Runtime Environment." https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/runtime.html. [cited on p. 46]

[103] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic complex event processing exploiting prediction," in *2015 IEEE International Conference on Big Data (Big Data)*, pp. 213–222, IEEE, oct 2015. [cited on p. 47]

[104] J. I. Rodríguez-Molano, L. E. Contreras-Bravo, and E. R. López-Santana, "Big Data Tools for Smart Cities," in *Lecture Notes in Computer Science*, pp. 649–658, Springer Link, 2018. [cited on p. 47]

[105] W. Puangsaijai and S. Puntheeranurak, "A comparative study of relational database and key-value database for big data applications," in *2017 International Electrical Engineering Congress (iEECON)*, pp. 1–4, IEEE, mar 2017. [cited on p. 49, 50]

[106] A. Gupta, S. Tyagi, N. Panwar, S. Sachdeva, and U. Saxena, "NoSQL databases: Critical analysis and comparison," in *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*, pp. 293–299, IEEE, oct 2017. [cited on p. 49, 50]

[107] D. Alves Florencio, D. Ricardo Freitas de Oliveira, E. Laisa Soares Xavier Freitas, and F. da Fonseca de Souza, "Which Fits Better? A Comparative Analysis about NoSQL Key-Value Databases," *IEEE Latin America Transactions*, vol. 15, pp. 2251–2256, nov 2017. [cited on p. 49]

[108] K. Sahatqija, J. Ajdari, X. Zenuni, B. Raufi, and F. Ismaili, "Comparison between relational and NOSQL databases," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 0216–0221, IEEE, may 2018. [cited on p. 49, 50]

[109] "Redis." https://redis.io/. [cited on p. 50]

[110] "Apache Cassandra." http://cassandra.apache.org/. [cited on p. 50]

[111] "JSON." https://www.json.org/. [cited on p. 50]

[112] S. Rinaldi, F. Bonafini, P. Ferrari, A. Flammini, E. Sisinni, and D. Bianchini, "Impact of Data Model on Performance of Time Series Database for Internet of Things Applications," in *2019 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, pp. 1–6, IEEE, may 2019. [cited on p. 50, 51]

[113] S. Di Martino, L. Fiadone, A. Peron, A. Riccabone, and V. N. Vitale, "Industrial Internet of Things: Persistence for Time Series with NoSQL Databases," in *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pp. 340–345, IEEE, jun 2019. [cited on p. 51, 52]

[114] "Database management using InfluxQL | InfluxData Documentation." https://docs.influxdata.com/influxdb/v1.7/query_language/database_management/. [cited on p. 51]

[115] "InfluxDB key concepts | InfluxData Documentation." https://docs.influxdata.com/influxdb/v1.7/concepts/key_concepts/. [cited on p. 51]

[116] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," in *Web Engineering: 16th International Conference, ICWE 2016*, pp. 21–39, SpringerLink, 2016. [cited on p. 56]

[117] "What is REST — A Simple Explanation for Beginners, Part 2: REST Constraints." https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-2-rest-constraints-129a4b69a582. [cited on p. 56]

[118] "Build apps with HERE Maps API and SDK Platform Access | HERE Developer." https://developer.here.com/?cid=www.here.com-main_menu. [cited on p. 59]

[119] "Maven – Welcome to Apache Maven." https://maven.apache.org/index.html. [cited on p. 63]

[120] "Apache Flink 1.9 Documentation: FlinkCEP - Complex event processing for Flink." https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html#the-pattern-api. [cited on p. 66]

[121] "Apache Flink 1.9 Documentation: Monitoring REST API." https://ci.apache.org/projects/flink/flink-docs-release-1.9/monitoring/rest_api.html. [cited on p. 69]

[122] Lijing Zhang and Xiaoxiao Shen, "Research and development of real-time monitoring system based on WebSocket technology," in *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*, pp. 1955–1958, IEEE, dec 2013. [cited on p. 74]

[123] "Alert - Fiware-DataModels." https://fiware-datamodels.readthedocs.io/en/latest/Alert/doc/spec/index.html. [cited on p. 74]

[124] "FastAPI." https://fastapi.tiangolo.com/. [cited on p. 75]

[125] "Enterprise Container Platform | Docker." https://www.docker.com/. [cited on p. 77]

[126] A. Azab, "Enabling Docker Containers for High-Performance and Many-Task Computing," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 279–285, IEEE, apr 2017. [cited on p. 77]

[127] S. Kumar Pentyala, "Emergency communication system with Docker containers, OSM and Rsync," in *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, pp. 1064–1069, IEEE, aug 2017. [cited on p. 77]

[128] "Get Started, Part 2: Containers | Docker Documentation." https://docs.docker.com/get-started/part2/. [cited on p. 77]

[129] "Get Started, Part 3: Services | Docker Documentation." https://docs.docker.com/get-started/part3/. [cited on p. 78]

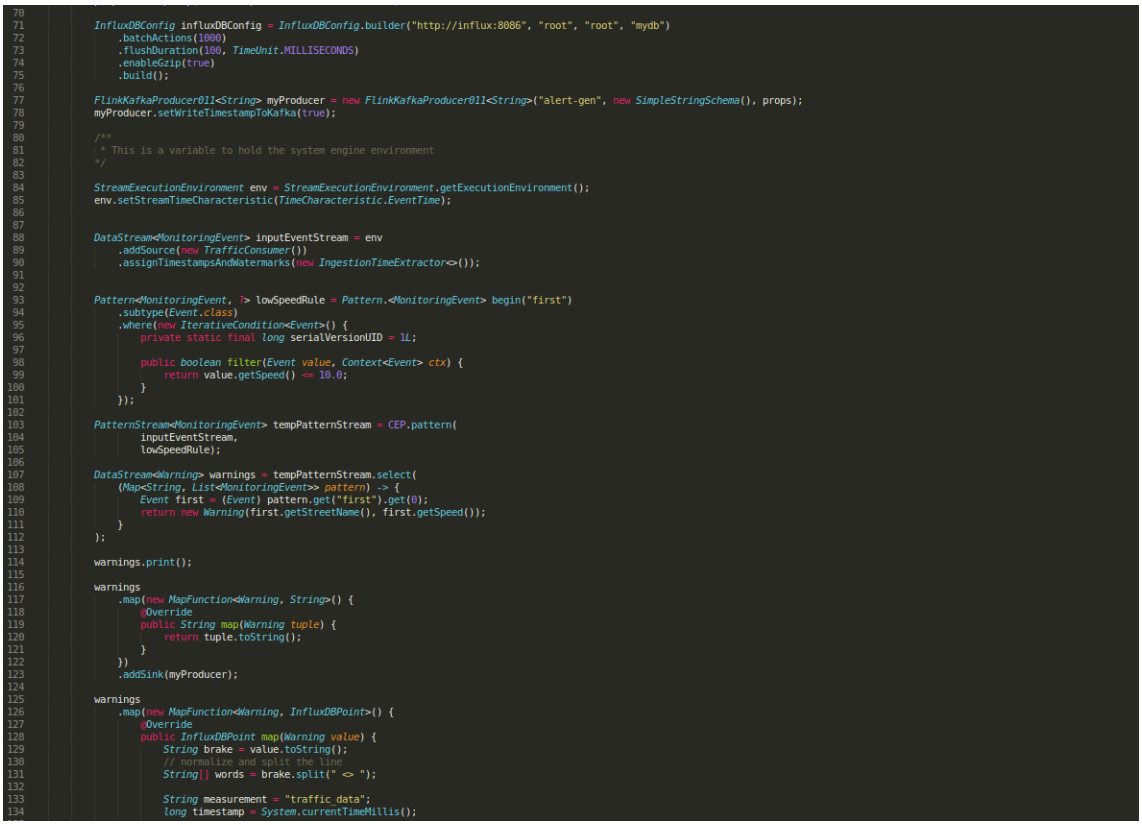
[130] "Compose file version 3 reference | Docker Documentation." https://docs.docker.com/compose/compose-file/. [cited on p. 78]

[131] A. Sanla and T. Numnonda, "A Comparative Performance of Real-time Big Data Analytic Architectures," in *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, pp. 1–5, IEEE, jul 2019. [cited on p. 108]

Anexo A

TrafficAnalysis.java

```java
package dcosta;

import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.tuple.Tuple5;
import org.apache.flink.api.java.tuple.Tuple;
import org.apache.flink.api.java.DataSet;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.SplitStream;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.functions.IngestionTimeExtractor;
import org.apache.flink.streaming.api.collector.selector.OutputSelector;

import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer011;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer011;

import org.apache.flink.streaming.connectors.influxdb.InfluxDBConfig;
import org.apache.flink.streaming.connectors.influxdb.InfluxDBPoint;
import org.apache.flink.streaming.connectors.influxdb.InfluxDBSink;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.FoldFunction;
import org.apache.flink.api.common.functions.FilterFunction;

import org.apache.flink.cep.CEP;
import org.apache.flink.cep.PatternSelectFunction;
import org.apache.flink.cep.pattern.Pattern;
import org.apache.flink.cep.PatternStream;
import org.apache.flink.cep.pattern.conditions.SimpleCondition;
import org.apache.flink.cep.pattern.conditions.IterativeCondition;

import org.apache.flink.streaming.connectors.cassandra.CassandraSink;

import org.apache.flink.util.Collector;

import java.util.Properties;
import java.util.Map;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.concurrent.TimeUnit;

import org.apache.commons.lang3.StringEscapeUtils;

import dcosta.Events.Event;
import dcosta.Events.MonitoringEvent;
import dcosta.Events.Warning;
import dcosta.Events.Alert;
import dcosta.Sources.TrafficConsumer;

/**
 * This is a model class to analyze traffic events
 *
 * @author Diogo Costa
 */

public class TrafficAnalysis {

    public static void main(String[] args) throws Exception {

        Properties props = new Properties();
        props.setProperty("bootstrap.servers", "kafka1:9092");
```



```
70
71          InfluxDBConfig influxDBConfig = InfluxDBConfig.builder("http://influx:8086", "root", "root", "mydb")
72              .batchActions(1000)
73              .flushDuration(100, TimeUnit.MILLISECONDS)
74              .enableGzip(true)
75              .build();
76
77          FlinkKafkaProducer011<String> myProducer = new FlinkKafkaProducer011<String>("alert-gen", new SimpleStringSchema(), props);
78          myProducer.setWriteTimestampToKafka(true);
79
80          /**
81           * This is a variable to hold the system engine environment
82           */
83
84          StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
85          env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
86
87
88          DataStream<MonitoringEvent> inputEventStream = env
89              .addSource(new TrafficConsumer())
90              .assignTimestampsAndWatermarks(new IngestionTimeExtractor<>());
91
92
93          Pattern<MonitoringEvent, ?> lowSpeedRule = Pattern.<MonitoringEvent> begin("first")
94              .subtype(Event.class)
95              .where(new IterativeCondition<Event>() {
96                  private static final long serialVersionUID = 1L;
97
98                  public boolean filter(Event value, Context<Event> ctx) {
99                      return value.getSpeed() <= 10.0;
100                 }
101             });
102
103         PatternStream<MonitoringEvent> tempPatternStream = CEP.pattern(
104                 inputEventStream,
105                 lowSpeedRule);
106
107         DataStream<Warning> warnings = tempPatternStream.select(
108             (Map<String, List<MonitoringEvent>> pattern) -> {
109                 Event first = (Event) pattern.get("first").get(0);
110                 return new Warning(first.getStreetName(), first.getSpeed());
111             }
112         );
113
114         warnings.print();
115
116         warnings
117             .map(new MapFunction<Warning, String>() {
118                 @Override
119                 public String map(Warning tuple) {
120                     return tuple.toString();
121                 }
122             })
123             .addSink(myProducer);
124
125         warnings
126             .map(new MapFunction<Warning, InfluxDBPoint>() {
127                 @Override
128                 public InfluxDBPoint map(Warning value) {
129                     String brake = value.toString();
130                     // normalize and split the line
131                     String[] words = brake.split(" <> ");
132
133                     String measurement = "traffic_data";
134                     long timestamp = System.currentTimeMillis();
```

```
                    HashMap<String, String> tags = new HashMap<>();
                    tags.put("street_name", StringEscapeUtils.unescapeJava(words[1]));
                    tags.put("type", words[0]);
                    tags.put("severity", "medium");

                    HashMap<String, Object> fields = new HashMap<>();
                    fields.put("speed_value", Double.parseDouble(words[2]));

                    return new InfluxDBPoint(measurement, timestamp, tags, fields);
                }
            })
            .addSink(new InfluxDBSink(influxDBConfig));

        // Alert pattern: Two consecutive traffic warnings appearing within a time interval of 20 seconds
        Pattern<Warning, ?> alertRule = Pattern.<Warning>begin("first")
                .next("second")
                .within(Time.seconds(20));

        // Create a pattern stream from our alert pattern
        PatternStream<Warning> alertPatternStream = CEP.pattern(
                warnings.keyBy("streetName"),
                alertRule);

        // Generate a traffic alert only if the first traffic warning's average speed is higher than
        // second warning's speed
        DataStream<Alert> alerts = alertPatternStream
            .flatSelect(
                (Map<String, List<Warning>> pattern, Collector<Alert> out) -> {
                    Warning first = pattern.get("first").get(0);
                    Warning second = pattern.get("second").get(0);

                    if (first.getAverageSpeed() > second.getAverageSpeed()) {
                        out.collect(new Alert(first.getStreetName(), first.getAverageSpeed(), second.getAverageSpeed()));
                    }
                },
                TypeInformation.of(Alert.class));

        alerts
            .map(new MapFunction<Alert, String>() {
                @Override
                public String map(Alert tuple) {
                    return tuple.toString();
                }
            })
            .addSink(myProducer);

        alerts
            .map(new MapFunction<Alert, InfluxDBPoint>() {
                @Override
                public InfluxDBPoint map(Alert value) {
                    String brake = value.toString();

                    // normalize and split the line
                    String[] words = brake.split(" <> ");

                    String measurement = "traffic_data";
                    long timestamp = System.currentTimeMillis();

                    HashMap<String, String> tags = new HashMap<>();
                    tags.put("street_name", StringEscapeUtils.unescapeJava(words[2]));
                    tags.put("type", words[0]);
                    tags.put("severity", words[1]);

                    HashMap<String, Object> fields = new HashMap<>();
                    fields.put("speed_value", Double.parseDouble(words[3].replace("%","")));

                    return new InfluxDBPoint(measurement, timestamp, tags, fields);
                }
```

```
            })
            .addSink(new InfluxDBSink(influxDBConfig));

        alerts.print();
        env.execute();
    }
}
```

Anexo B

jar_upload.py

```python
import os
import kafka
from kafka import KafkaConsumer

import datetime
import time
import json
import requests

KAFKA_VERSION = (2, 2, 1)
upload_url = 'http://jobmanager:8081'
data_types = []
active_topologies = {}


# Creation of kafka consumer #
consumer = KafkaConsumer(
    'jar_update',
    bootstrap_servers=['kafka1:9092'],
    auto_offset_reset='latest',
    api_version=KAFKA_VERSION,
    enable_auto_commit=True)

for message in consumer:
    result = (message.value).decode('unicode-escape').strip('"')
    data_types.append(result)

    # Every 20 events, check if it's necessary add or remove topologies #
    if (len(data_types) >= 20):
        diff_types = list(set(data_types))

        to_delete = [i for i in active_topologies if i not in diff_types]
        if len(to_delete) != 0:
            for topol in to_delete:

                # Stop job #
                id_job = active_topologies[f'{topol}'][1]
                stop_job = requests.patch(f'{upload_url}/jobs/{id_job}')

                # Remove jar #
                id_jar = active_topologies[f'{topol}'][0]
                delete_jar = requests.delete(f'{upload_url}/jars/{id_jar}')

                # Update active topologies #
                active_topologies.pop(f'{topol}')

        for elem in diff_types:
            if elem not in active_topologies:
                # Upload jar #
                path = f'/code/{elem}-topology-0.1.jar'
                upload_request = requests.post(upload_url + "/jars/upload",
                    files={"jarfile": (os.path.basename(path), open(path, "rb"), "application/x-java-archive")})

                # Submit job #
                upload_response = json.loads(upload_request.text)
                jar_id = (upload_response['filename'].split("/"))[-1]
                submit_request = requests.post(f'{upload_url}/jars/{jar_id}/run')


                # Update active topologies #
                ids = [jar_id, (json.loads(submit_request.text))['jobid']]
                active_topologies.update({f'{elem}': ids})
                ids = []
        # Update data types #
        data_types = []
```

Anexo C

alertGenerator.py

```python
from kafka import KafkaConsumer
#from twilio.rest import Client
import logging
import datetime
import time
import json
import smtplib
import asyncio
from websocket import create_connection

import mail_structures

from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

KAFKA_VERSION = (2, 2, 1)

consumer = KafkaConsumer(
    'alert-gen',
    bootstrap_servers=['kafka1:9092'],
    auto_offset_reset='latest',
    api_version=KAFKA_VERSION,
    enable_auto_commit=True)

sender = "smart_alerting@sandboxef2a63864860484ca4d478ed89b86f20.mailgun.org"
receiver = "diogo.costa@proef.com"

def alert_modeling(input_str):
    data = input_str.split(" <> ")

    if data[0] == "traffic":
        model = {
        "context": data[0],
        "type": "warning",
        "severity": "medium",
        "message": data[1].replace("\x00", "") + " <-> " + data[2]
        }
    else:
        model = {
        "context": data[0],
        "type": data[1],
        "severity": data[2],
        "message": data[3].replace("\x00", "")
        }

    return model

def send_email(context, type, severity, message):

    mail = mail_structures.get_mail_body(context, message)

    s = smtplib.SMTP('smtp.mailgun.org', 587)

    s.login('postmaster@sandboxef2a63864860484ca4d478ed89b86f20.mailgun.org',
        '60f53553f131d129ad74b72b061f2d92-73ae490d-e0662808')

    s.sendmail(sender, receiver, mail)

    s.quit()

```

```
61
62  async def send_socket(context, type, severity, message):
63      print("BEFORE SEND " , message , int(time.time()*1000.0))
64      brake_message = message.split(" <-> ")
65      if context == 'weather':
66          socket_msg = "Watch out, " + brake_message[0] + " in " + brake_message[1] + " on " + brake_message[2] + "!"
67      elif context == 'traffic':
68          socket_msg = "Be careful, low speed detected in " + brake_message[0] + ": " + brake_message[1] + "!"
69
70      ws = create_connection("ws://end_api:6000/ws")
71      ws.send(socket_msg)
72      print (ws.recv())
73
74  def main():
75      for message in consumer:
76          result = (message.value).decode('utf-8')
77          alert = alert_modeling(result)
78          if (alert['severity'] == 'critical'):
79              send_email(**alert)
80          elif (alert['severity'] == 'medium'):
81              asyncio.run(send_socket(**alert))
82
83  if __name__ == "__main__":
84      main()
```

Anexo D

docker-compose.yml

```yaml
version: '3.7'
services:
    zookeeper:
        image: wurstmeister/zookeeper
        environment:
          - ZOOKEEPER_ID=1
          - ZOOKEEPER_SERVER_1=zookeeper
          - ZOOKEEPER_SERVER_2=zookeeper
          - ZOOKEEPER_CLIENT_PORT=2181
        ports:
          - 2181:2181
        networks:
          - kafka
    kafka1:
        image: wurstmeister/kafka
        depends_on:
          - zookeeper
        environment:
          - KAFKA_ADVERTISED_PORT=9092
          - KAFKA_ADVERTISED_HOST_NAME=kafka1
          - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
          - KAFKA_LOG_RETENTION_HOURS=4
          - KAFKA_LOG_RETENTION_BYTES=1000000000
          - JMX_PORT=9997
          - KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Djava.rmi.server.hostname=kafka1 -Dcom.sun.management.jmxremote.rmi.port=9997
        expose:
          - 9997
        ports:
          - 9092:9092
        networks:
          - kafka

    manager:
        image: hlebalbau/kafka-manager:stable
        ports:
          - "9000:9000"
        environment:
          ZK_HOSTS: "zookeeper:2181"
          APPLICATION_SECRET: "random-secret"
        command: -Dpidfile.path=/dev/null
        networks:
          - kafka

    grafana:
        image: grafana/grafana
        ports:
          - 3000:3000
        volumes:
          - "grafana-storage:/var/lib/grafana"
        networks:
          - kafka
```

```
      - kafka

  influx:
    image: influxdb
    ports:
      - "8086:8086"
      - "8083:8083"
      - "2003:2003"
    environment:
      - INFLUXDB_GRAPHITE_ENABLED=true
    volumes:
      - "influx-storage:/var/lib/influxdb"
    networks:
      - kafka

  jobmanager:
    image: ${FLINK_DOCKER_IMAGE_NAME:-flink}
    expose:
      - "6123"
    ports:
      - "8081:8081"
    command: jobmanager
    environment:
      - JOB_MANAGER_RPC_ADDRESS=jobmanager
    networks:
      - kafka

  taskmanager:
    image: ${FLINK_DOCKER_IMAGE_NAME:-flink}
    expose:
      - "6121"
      - "6122"
    depends_on:
      - jobmanager
    command: taskmanager
    links:
      - "jobmanager:jobmanager"
    environment:
      - JOB_MANAGER_RPC_ADDRESS=jobmanager
    networks:
      - kafka

  alert-generator:
    build: ../alertGenerator/.
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - kafka

  producer-python:
    build: ../producer/.
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - kafka
```

```

  monitoring-event:
    build: ../monitoringEventSource/.
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - kafka

  request-python:
    build: ../request/.
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - kafka
    depends_on:
      - end_api

  upload-python:
    build: ../uploads/.
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      - kafka

  end_api:
    build: ../api/.
    environment:
      - PYTHONUNBUFFERED=1
    ports:
      - 6000:6000
    networks:
      - kafka

networks:
  kafka:

volumes:
  dbdata:
  grafana-storage:
  influx-storage:
```

Anexo E

Tabelas de Resultados

Tabela E.1: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser enviado para o servidor WebSockets num cenário com 1 fonte de dados e envio de 1 mensagem/s

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Generator	Out Generator	Generator	In Socket	Web Socket
#1	**1570628680452**	**1570628680453**	**1**	**1570628680463**	**1570628680661**	**198**	**1570740710581**	**1570740710582**	**1**	**1570740710591**	**9**
#2	**1570628690476**	**1570628690476**	**0**	**1570628690490**	**1570628690577**	**87**	**1570740718612**	**1570740718614**	**2**	**1570740718636**	**22**
#3	**1570628698491**	**1570628698492**	**1**	**1570628698502**	**1570628698591**	**89**	**1570740786757**	**1570740786760**	**3**	**1570740786777**	**17**
#4	**1570628700495**	**1570628700495**	**0**	**1570628700499**	**1570628700593**	**94**	**1570740788764**	**1570740788765**	**1**	**1570740788829**	**64**
#5	**1570628702506**	**1570628702506**	**0**	**1570628702516**	**1570628702598**	**82**	**1570740790763**	**1570740790764**	**1**	**1570740790771**	**7**
#6	**1570628706512**	**1570628706512**	**0**	**1570628706522**	**1570628706604**	**82**	**1570740798771**	**1570740798772**	**1**	**1570740798778**	**6**
#7	**1570628708516**	**1570628708517**	**1**	**1570628708526**	**1570628708707**	**181**	**1570740867021**	**1570740867021**	**0**	**1570740867026**	**5**
#8	**1570628718535**	**1570628718536**	**1**	**1570628718547**	**1570628718723**	**176**	**1570740869034**	**1570740869035**	**1**	**1570740869052**	**17**
#9	**1570628720539**	**1570628720540**	**1**	**1570628720551**	**1570628720726**	**175**	**1570740871037**	**1570740871039**	**2**	**1570740871056**	**17**
#10	**1570628728551**	**1570628728552**	**1**	**1570628728568**	**1570628728739**	**171**	**1570740879152**	**1570740879152**	**0**	**1570740879158**	**6**
Média			**0,6**			**133,5**			**1,2**		**17**

Tabela E.2: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser enviado para o servidor WebSockets num cenário com 2 fontes de dados e envio de 1 mensagem/s

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Generator	Out Generator	Generator	In Socket	Web Socket
#1	1570627133211	1570627133212	1	1570627133224	1570627133376	152	1570741593457	1570741593458	1	1570741593473	15
#2	1570627137224	1570627137224	0	1570627137227	1570627137383	156	1570741609281	1570741609281	0	1570741609288	7
#3	1570627141230	1570627141230	0	1570627141233	1570627141389	156	1570741745697	1570741745698	1	1570741745710	12
#4	1570627145235	1570627145235	0	1570627145242	1570627145395	153	1570741749503	1570741749504	1	1570741749511	7
#5	1570627173304	1570627173304	0	1570627173307	1570627173397	90	1570741753605	1570741753606	1	1570741753611	5
#6	1570627177311	1570627177311	0	1570627177314	1570627177394	80	1570741769637	1570741769638	1	1570741769652	14
#7	1570627197344	1570627197345	1	1570627197348	1570627197481	133	1570741873816	1570741873817	1	1570741873823	6
#8	1570627201350	1570627201351	1	1570627201359	1570627201487	128	1570741905867	1570741905868	1	1570741905874	6
#9	1570627233406	1570627233406	0	1570627233410	1570627233537	127	1570741909880	1570741909882	2	1570741909899	17
#10	1570627257452	1570627257453	1	1570627257455	1570627257676	221	1570741913884	1570741913885	1	1570741913898	13
Média			0,4			139,6			1		10,2

Tabela E.3: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser enviado para o servidor WebSockets num cenário com 2 fontes de dados e envio contínuo

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Generator	Out Generator	Generator	In Socket	Web Socket
#1	**1570637618659**	**1570637618659**	**0**	**1570637618667**	**1570637618784**	**117**	**1570742281889**	**1570742281889**	**0**	**1570742281892**	**3**
#2	**1570637650716**	**1570637650717**	**1**	**1570637650726**	**1570637650936**	**210**	**1570742285999**	**1570742285999**	**0**	**1570742286002**	**3**
#3	**1570637654727**	**1570637654727**	**0**	**1570637654735**	**1570637654942**	**207**	**1570742290016**	**1570742290017**	**1**	**1570742290023**	**6**
#4	**1570637658738**	**1570637658739**	**1**	**1570637658747**	**1570637658950**	**203**	**1570742310144**	**1570742310146**	**2**	**1570742310161**	**15**
#5	**1570637662746**	**1570637662746**	**0**	**1570637662748**	**1570637662866**	**118**	**1570742314149**	**1570742314150**	**1**	**1570742314160**	**10**
#6	**1570637666752**	**1570637666753**	**1**	**1570637666764**	**1570637666871**	**107**	**1570742386161**	**1570742386162**	**1**	**1570742386174**	**12**
#7	**1570637690802**	**1570637690803**	**1**	**1570637690817**	**1570637691010**	**193**	**1570742390165**	**1570742390168**	**3**	**1570742390178**	**10**
#8	**1570637718865**	**1570637718865**	**0**	**1570637718875**	**1570637718952**	**77**	**1570742442248**	**1570742442249**	**1**	**1570742442259**	**10**
#9	**1570637722873**	**1570637722874**	**1**	**1570637722882**	**1570637722958**	**76**	**1570742446252**	**1570742446253**	**1**	**1570742446258**	**5**
#10	**1570637754938**	**1570637754938**	**0**	**1570637754948**	**1570637755107**	**159**	**1570742450356**	**1570742450356**	**0**	**1570742450360**	**4**
Média			**0,5**			**146,7**			**1**		**7,8**

Tabela E.4: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser enviado para o e-mail num cenário com 1 fonte de dados e envio de 1 mensagem/s

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Generator	Out Generator	Generator	In Mail	Email
#1	**1570629699921**	**1570629699922**	**1**	**1570629699925**	**1570629700085**	**160**	**1570735353298**	**1570735353301**	**3**	**1570735354227**	**926**
#2	**1570629709938**	**1570629709938**	**0**	**1570629709942**	**1570629710101**	**159**	**1570735383247**	**1570735383247**	**0**	**1570735384282**	**1035**
#3	**1570629717954**	**1570629717955**	**1**	**1570629717965**	**1570629718212**	**247**	**1570735389255**	**1570735389255**	**0**	**1570735390964**	**1709**
#4	**1570629719958**	**1570629719959**	**1**	**1570629719975**	**1570629720216**	**241**	**1570735391264**	**1570735391264**	**0**	**1570735392879**	**1615**
#5	**1570629721965**	**1570629721965**	**0**	**1570629721975**	**1570629722218**	**243**	**1570735393259**	**1570735393259**	**0**	**1570735394889**	**1630**
#6	**1570629725974**	**1570629725974**	**0**	**1570629725981**	**1570629726024**	**43**	**1570735409397**	**1570735409397**	**0**	**1570735411067**	**1670**
#7	**1570629736002**	**1570629736003**	**1**	**1570629736015**	**1570629736140**	**125**	**1570735419503**	**1570735419503**	**0**	**1570735421230**	**1727**
#8	**1570629738007**	**1570629738007**	**0**	**1570629738035**	**1570629738143**	**108**	**1570735421509**	**1570735421509**	**0**	**1570735423183**	**1674**
#9	**1570629750031**	**1570629750031**	**0**	**1570629750040**	**1570629750162**	**122**	**1570735427323**	**1570735427323**	**0**	**1570735428922**	**1599**
#10	**1570629752035**	**1570629752036**	**1**	**1570629752047**	**1570629752170**	**123**	**1570735433322**	**1570735433322**	**0**	**1570735435122**	**1800**
Média			**0,5**			**157,1**			**0,3**		**1538,5**

Tabela E.5: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser enviado para o e-mail num cenário com 2 fontes de dados e envio de 1 mensagem/s

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Generator	Out Generator	Generator	In Mail	Email
#1	**1570618134414**	**1570618134415**	**1**	**1570618134422**	**1570618134651**	**229**	**1570736522918**	**1570736522918**	**0**	**1570736524527**	**1609**
#2	**1570618138424**	**1570618138424**	**0**	**1570618138436**	**1570618138659**	**223**	**1570736583112**	**1570736583112**	**0**	**1570736584117**	**1005**
#3	**1570618142430**	**1570618142431**	**1**	**1570618142442**	**1570618142666**	**224**	**1570736587108**	**1570736587108**	**0**	**1570736588087**	**979**
#4	**1570618150445**	**1570618150446**	**1**	**1570618150457**	**1570618150580**	**123**	**1570736591115**	**1570736591115**	**0**	**1570736592043**	**928**
#5	**1570618162482**	**1570618162482**	**0**	**1570618162485**	**1570618162601**	**116**	**1570736643232**	**1570736643233**	**1**	**1570736644216**	**983**
#6	**1570618178512**	**1570618178512**	**0**	**1570618178516**	**1570618178730**	**214**	**1570736647330**	**1570736647330**	**0**	**1570736648262**	**932**
#7	**1570618186526**	**1570618186527**	**1**	**1570618186539**	**1570618186744**	**205**	**1570736651355**	**1570736651356**	**1**	**1570736652280**	**924**
#8	**1570618190548**	**1570618190548**	**0**	**1570618190557**	**1570618190750**	**193**	**1570736743398**	**1570736743398**	**0**	**1570736745063**	**1665**
#9	**1570618194554**	**1570618194555**	**1**	**1570618194567**	**1570618194756**	**189**	**1570736747510**	**1570736747510**	**0**	**1570736749116**	**1606**
#10	**1570618218605**	**1570618218606**	**1**	**1570618218612**	**1570618218694**	**82**	**1570736751512**	**1570736751513**	**1**	**1570736753449**	**1936**
Média			**0,6**			**179,8**			**0,3**		**1256,7**

Tabela E.6: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser enviado para o e-mail num cenário com 2 fontes de dados e envio contínuo

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Generator	Out Generator	Generator	In Mail	Email
#1	1570632725809	1570632725809	0	1570632725812	1570632725984	172	1570738898389	1570738898390	1	1570738899331	941
#2	1570632729815	1570632729815	0	1570632729821	1570632729990	169	1570738930448	1570738930448	0	1570738932319	1871
#3	1570632745854	1570632745855	1	1570632745871	1570632746118	247	1570738934474	1570738934475	1	1570738936089	1614
#4	1570632761951	1570632761952	1	1570632761963	1570632762145	182	1570738954582	1570738954582	0	1570738956205	1623
#5	1570632781999	1570632782001	2	1570632782012	1570632782077	65	1570738970420	1570738970420	0	1570738972038	1618
#6	1570632802039	1570632802040	1	1570632802052	1570632802208	156	1570739014505	1570739014505	0	1570739016314	1809
#7	1570632818076	1570632818076	0	1570632818083	1570632818338	255	1570739026643	1570739026643	0	1570739028366	1723
#8	1570632866190	1570632866191	1	1570632866202	1570632866427	225	1570739038654	1570739038654	0	1570739040409	1755
#9	1570632870198	1570632870199	1	1570632870209	1570632870433	224	1570739042673	1570739042677	4	1570739044408	1731
#10	1570632874206	1570632874207	1	1570632874226	1570632874440	214	1570739058588	1570739058588	0	1570739060284	1696
Média			**0,8**			**190,9**			**0,6**		**1638,1**

Tabela E.7: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser registado na base de dados, num cenário com 1 fonte de dados e envio de 1 Mensagem/s

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Database	Database
#1	**1570750802766**	**1570750802766**	**0**	**1570750812819**	**1570750812942**	**123**	**1570750812943**	**1**
#2	**1570750804789**	**1570750804790**	**1**	**1570750830859**	**1570750831081**	**222**	**1570750831082**	**1**
#3	**1570750806793**	**1570750806793**	**0**	**1570750844888**	**1570750845108**	**220**	**1570750845108**	**0**
#4	**1570750808797**	**1570750808798**	**1**	**1570750846889**	**1570750846910**	**21**	**1570750846911**	**1**
#5	**1570750810802**	**1570750810802**	**0**	**1570750858911**	**1570750859029**	**118**	**1570750859030**	**1**
#6	**1570750812807**	**1570750812807**	**0**	**1570750878954**	**1570750879179**	**225**	**1570750879179**	**0**
#7	**1570750814817**	**1570750814817**	**0**	**1570750884957**	**1570750885187**	**230**	**1570750885187**	**0**
#8	**1570750816821**	**1570750816821**	**0**	**1570750886960**	**1570750887190**	**230**	**1570750887192**	**2**
#9	**1570750818823**	**1570750818823**	**0**	**1570750892971**	**1570750893201**	**230**	**1570750893202**	**1**
#10	**1570750820826**	**1570750820826**	**0**	**1570750911020**	**1570750911131**	**111**	**1570750911132**	**1**
Média			**0,2**			**173**		**0,8**

Tabela E.8: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser registado na base de dados, num cenário com 2 fontes de dados e envio de 1 Mensagem/s

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Database	Database
#1	**1570749195631**	**1570749195631**	**0**	**1570748967202**	**1570748967319**	**117**	**1570748967320**	**1**
#2	**1570749199640**	**1570749199640**	**0**	**1570748979229**	**1570748979339**	**110**	**1570748979340**	**1**
#3	**1570749203655**	**1570749203656**	**1**	**1570748983240**	**1570748983348**	**108**	**1570748983349**	**1**
#4	**1570749207668**	**1570749207668**	**0**	**1570748995274**	**1570748995471**	**197**	**1570748995471**	**0**
#5	**1570749211675**	**1570749211675**	**0**	**1570749015288**	**1570749015415**	**127**	**1570749015416**	**1**
#6	**1570749215679**	**1570749215679**	**0**	**1570749019298**	**1570749019422**	**124**	**1570749019422**	**0**
#7	**1570749219689**	**1570749219689**	**0**	**1570749091443**	**1570749091658**	**215**	**1570749091659**	**1**
#8	**1570749223700**	**1570749223700**	**0**	**1570749095446**	**1570749095668**	**222**	**1570749095669**	**1**
#9	**1570749227717**	**1570749227717**	**0**	**1570749135530**	**1570749135642**	**112**	**1570749135642**	**0**
#10	**1570749231731**	**1570749231732**	**1**	**1570749147548**	**1570749147767**	**219**	**1570749147768**	**1**
Média			**0,2**			**155,1**		**0,7**

Tabela E.9: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada etapa do sistema até ser registado na base de dados, num cenário com 2 fontes de dados e envio contínuo

#	In MES	Out MES	MES	In Flink	Out Flink	Flink	In Database	Database
#1	**1570747644924**	**1570747644925**	**1**	**1570747484586**	**1570747484834**	**248**	**1570747484839**	**5**
#2	**1570747648941**	**1570747648941**	**0**	**1570747488591**	**1570747488637**	**46**	**1570747488638**	**1**
#3	**1570747668987**	**1570747668987**	**0**	**1570747508626**	**1570747508775**	**149**	**1570747508776**	**1**
#4	**1570747705078**	**1570747705078**	**0**	**1570747544707**	**1570747544742**	**35**	**1570747544744**	**2**
#5	**1570747709089**	**1570747709089**	**0**	**1570747548714**	**1570747548849**	**135**	**1570747548850**	**1**
#6	**1570747548701**	**1570747548701**	**0**	**1570747644941**	**1570747645055**	**114**	**1570747645056**	**1**
#7	**1570747580783**	**1570747580783**	**0**	**1570747648965**	**1570747649065**	**100**	**1570747649066**	**1**
#8	**1570747584789**	**1570747584789**	**0**	**1570747669006**	**1570747669119**	**113**	**1570747669121**	**2**
#9	**1570747588797**	**1570747588797**	**0**	**1570747705107**	**1570747705329**	**222**	**1570747705330**	**1**
#10	**1570747592805**	**1570747592805**	**0**	**1570747709104**	**1570747709141**	**37**	**1570747709150**	**9**
Média			**0,1**			**119,9**		**2,4**

Tabela E.10: Registo dos \emph{timestamps}, em milissegundos, do evento/alerta em cada função no motor Flink, num cenário com 1 fonte de dados e envio de 1 Mensagem/s

#	In Apply	Out Apply	Apply Rule	In Trigger	Trigger Alert	In Send	Send Alert
#1	**1570669966536**	**1570669966729**	**193**	**1570669966730**	**1**	**1570669966730**	**0**
#2	**1570670012648**	**1570670012821**	**173**	**1570670012822**	**1**	**1570670012823**	**1**
#3	**1570670014646**	**1570670014824**	**178**	**1570670014824**	**0**	**1570670014824**	**0**
#4	**1570670016648**	**1570670016829**	**181**	**1570670016830**	**1**	**1570670016834**	**4**
#5	**1570670046731**	**1570670046884**	**153**	**1570670046885**	**1**	**1570670046885**	**0**
#6	**1570670092834**	**1570670092885**	**51**	**1570670092885**	**0**	**1570670092885**	**0**
#7	**1570670094836**	**1570670094888**	**52**	**1570670094889**	**1**	**1570670094889**	**0**
#8	**1570670118903**	**1570670119033**	**130**	**1570670119034**	**1**	**1570670119034**	**0**
#9	**1570670120909**	**1570670121036**	**127**	**1570670121037**	**1**	**1570670121038**	**1**
#10	**1570670126926**	**1570670127147**	**221**	**1570670127147**	**0**	**1570670127147**	**0**
Média			**145,9**		**0,7**		**0,6**

Tabela E.11: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada função no motor Flink, num cenário com 2 fontes de dados e envio de 1 Mensagem/s

#	In Apply	Out Apply	Apply Rule	In Trigger	Trigger Alert	In Send	Send Alert
#1	**1570668941423**	**1570668941649**	**226**	**1570668941650**	**1**	**1570668941651**	**1**
#2	**1570668953474**	**1570668953675**	**201**	**1570668953675**	**0**	**1570668953675**	**0**
#3	**1570668977515**	**1570668977623**	**108**	**1570668977627**	**4**	**1570668977628**	**1**
#4	**1570669005586**	**1570669005780**	**194**	**1570669005781**	**1**	**1570669005783**	**2**
#5	**1570669037643**	**1570669037834**	**191**	**1570669037835**	**1**	**1570669037835**	**0**
#6	**1570669041645**	**1570669041841**	**196**	**1570669041842**	**1**	**1570669041843**	**1**
#7	**1570669045658**	**1570669045847**	**189**	**1570669045848**	**1**	**1570669045849**	**1**
#8	**1570669049666**	**1570669049853**	**187**	**1570669049853**	**0**	**1570669049854**	**1**
#9	**1570669053673**	**1570669053859**	**186**	**1570669053859**	**0**	**1570669053860**	**1**
#10	**1570669073698**	**1570669073792**	**94**	**1570669073793**	**1**	**1570669073793**	**0**
Média			**177,2**		**1**		**0,8**

Tabela E.12: Registo dos *timestamps*, em milissegundos, do evento/alerta em cada função no motor Flink, num cenário com 2 fontes de dados e envio contínuo

#	In Apply	Out Apply	Apply Rule	In Trigger	Trigger Alert	In Send	Send Alert
#1	**1570666513969**	**1570666514017**	**48**	**1570666514025**	**8**	**1570666514028**	**3**
#2	**1570666517964**	**1570666518128**	**164**	**1570666518129**	**1**	**1570666518130**	**1**
#3	**1570666521981**	**1570666522134**	**153**	**1570666522134**	**0**	**1570666522134**	**0**
#4	**1570666546023**	**1570666546298**	**275**	**1570666546298**	**0**	**1570666546298**	**0**
#5	**1570666550031**	**1570666550106**	**75**	**1570666550106**	**0**	**1570666550107**	**1**
#6	**1570666582112**	**1570666582259**	**147**	**1570666582260**	**1**	**1570666582262**	**2**
#7	**1570666586117**	**1570666586265**	**148**	**1570666586265**	**0**	**1570666586265**	**0**
#8	**1570666590110**	**1570666590378**	**268**	**1570666590378**	**0**	**1570666590378**	**0**
#9	**1570666674355**	**1570666674429**	**74**	**1570666674430**	**1**	**1570666674430**	**0**
#10	**1570666678382**	**1570666678434**	**52**	**1570666678435**	**1**	**1570666678436**	**1**
Média			**140,4**		**1,2**		**0,8**