José Miguel Saramago de Carvalho

PhisioStream: A physiology monitoring system using off-the-shelf stream processing frameworks

PhisioStream: Um sistema de monitorização fisiológica baseado em ferramentas de processamento de streams

José Miguel Saramago de Carvalho

PhisioStream: A physiology monitoring system using off-the-shelf stream processing frameworks

PhisioStream: Um sistema de monitorização fisiológica baseado em ferramentas de processamento de streams

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor José Maria Amaral Fernandes, Professor auxiliar do Departamento de Electrónica da Universidade de Aveiro, e do Doutor Ilídio Fernando de Castro Oliveira, Professor auxiliar do Departamento de Electrónica da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor António José Ribeiro Neves

Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Rolando Martins

Professor Auxiliar do Departamento de Ciências de Computadores da Faculdade de Ciências da Universidade do Porto

Prof. Doutor José Maria Amaral Fernandes

Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

agradecimentos / acknowledgements

I would like to take this opportunity to express my deep gratitude to all which, in one way or another, were involved in the development of this work.

First place, to my mentor Professor José Maria Amaral Fernandes, for allowing me to work with him, for his exemplary character not only as a professional but also as a human being, for his constant availability which led to my everyday progress, and also for his scientific and technical guidance which allowed me to explore technologies I would not have been in touch yet. To my co-mentor, Ílidio Fernando de Castro Oliveira, for his constant concern and motivation supporting me whenever I needed along this work.

To my parents, the ones I am sure I can count on under any circumstances. For all your dedication, all the effort put into my education, your endless patience to deal with me and for being the two role models in whom I trust the most.

To my sister, the strongest individual I had the pleasure to meet, for all your words, thank you.

To my fellows, thank you for always being there.

And finally, to this awesome city which I would not forget, for the good friends I made and all the incredible moments I lived here, thanks Aveiro.

palavras-chave

Processamento de Streams; Monitorização; Fisiologia Computacional; Socorristas

resumo

O projeto VR2Market surgiu a partir de um consórcio composto por vários parceiros desde a área da tecnologia à psicologia, incluindo a Universidade de Carnegie Mellow, Estados Unidos, sob o programa CMU-Portugal financiado pelo FCT. O principal objetivo deste projeto é fornecer uma solução de monitorização de equipas de operacionais em profissões de risco, *First Responders*, em relação a aspetos tanto ambientais como fisiológicos. Contudo, a presente solução não oferece suporte à *cloud* e é composta maioritariamente por componentes ad hoc, o que dificulta o processo de evolução para soluções mais distribuídas.
O objetivo do presente trabalho consiste no *refactoring* do VR2Market no sentido de oferecer suporte à *cloud*, a partir de uma arquitetura mais expansível e que possibilite o processamento e visualização de dados sem comprometer as funcionalidades existentes no momento.
As opções tomadas recaem sobre o uso de processamento de *streams* e soluções *off-the-shelf,* tipicamente mais usadas para tarefas de gestão e monitorização de *logs*.
O processamento de streams assente sobre Apache Kafka revelou ser uma boa abordagem para garantir o tratamento e processamento de dados pré-existentes bem como para criar alarmes simples sobre alguns parâmetros. Esta capacidade de processamento poderá ser elevada a níveis mais complexos de *analytics*, nomeadamente através de ferramentas como o *Apache Spark* ou *Storm*, sem comprometer o funcionamento da restante arquitetura. O tratamento dos dados como uma *stream* possibilitou ainda a integração de ferramentas off-the-shelf que possibilitaram a visualização dos dados de forma contínua ao longo do tempo. Ao combinar estas duas abordagens, foi possível garantir a visualização e processamento de dados de uma forma dinâmica e flexível – tanto sobre dados pré-existentes como os que chegam ao sistema.
Foi adotada uma abordagem baseada em *Docker containers* que possibilitou não só uma forma mais simples de instalar o sistema como também chegar a uma solução totalmente *cloud-enabled*.
Apesar de estar diretamente relacionado com o contexto do VR2Market, pela sua natureza, a nossa arquitetura pode ser facilmente adaptada a outro tipo de cenários. Além disso, a integração de novos tipos de sensores pode ser agora feita de forma mais fácil.

keywords Stream Processing; Monitoring; Computational physiology; First Responders

abstract The on-going VR2Market research project emerged from a consortium composed of several partners from technology to psychology, including Carnegie Mellow University, United States under the CMU-Portugal program funded by FCT. The VR2Market main objective is to provide a team-wide monitoring solution over context, environmental aspects, and physiology of operating in hazardous professions, First Responders. However, the current solution is not cloud-enabled and relies on custom-made components within a centralized design which hinders future evolutions towards more distributed situations.
The objective of this work consists in refactoring VR2Market in order to provide cloud support with a more extensible architecture while allowing flexible data handling and visualization without compromising the existing functionalities.
The key architectural option relies on the adoption of a streaming processing approach, applying off-the-shelf log monitoring and management solutions.
Apache Kafka was used to handle and process data flows, both for integrating legacy data sources and to deploy simple trigger alarms. The later can be easily extended to more complex analytics, namely by using Apache Spark or Storm, without any refactoring of the data flow pipeline.
The proposed solution handles simultaneously the processing of data and flexible visualization over both historical and live data. Services are modeled under a container-oriented approach, using Docker, to fully harness cloud-enabled deployments.
Using the VR2Market context as the starting point, we managed to define and implement a new architecture that leverages on off-the-shelf tools to address the system needs. However, due to their general-purpose nature, they can easily be adapted to other scenarios. In addition, the system should support the integration of new types of sensors which can now be made with low effort.

I. Contents

II. List of figures

III. List of Acronyms

API – Application Programming Interface

CO – Carbon Monoxide

CSV – Comma Separated Values

DSL – Domain Specific Language

ECG – Electrocardiography

ELK – Elasticsearch, Logstash, Kibana

ETL – Extract, Transform, Load

FR – First Responder

GPS – Global Positioning System

HTML – Hypertext Markup Language

JAR – Java Archive

JSON – Javascript Object Notation

JWT – JSON Web Token

QOS – Quality of Service

RDBDMS – Relational Database Management System

REST – Representational State Transfer

SQL – Structured Query Language

TSDB – Time Series Database

UI – User Interface

URL – Uniform Resource Locator

VJ – Vital Jacket

WBAN – Wireless Body Area Network

YAML – Ain't Markup Language

1 Introduction

Information plays an important role since it is a powerful and valuable available resource from which it is possible to achieve high purposes. When properly used, information can add value to an organization in different ways.

Log files are an example of information source once they contain hidden data representing vast business value. Log monitoring and management solutions are commonly used when it comes to doing extended analysis and gather insights from these files, which could help in making business decisions.

VR2Market [1] project, in which this work is integrated, provides a team-wide monitoring solution over context, environmental aspects, and physiology of operational in hazardous professions, First Responders. The project is a collaboration of a consortium involving several partners from technology to psychology.

With a vast collection of cloud-based off-the-shelf solutions available for log monitoring and management, why not explore them in the VR2Market scenario?
Despite being designed to be used for handling large amounts of log data, these tools can also be applied to store, search and visualize different types of information.

The main purpose of this dissertation is to refactor VR2Market in order to provide cloud support with a more extensible architecture while allowing flexible data handling and visualization, without compromising the existing functionalities. Besides enhancing the online visualization layer, a new support for basic alarms over physiological data should be added.

The more visible part of the work focused on the refactoring of VRCommander. This component is responsible for the online team monitoring by creating a visualization of data in real time during its acquisition.

However, we also addressed a technological refactoring as the current solution is not fully cloud-enabled and maintainability and reuse were the main pointed issues on a first assessment. Integration of new types of sensors or events would imply partial or full system refactoring. For this reason, we made an effort to give the system the ability to support different contexts and scenarios. Furthermore, in order to get actionable insights from it, we handled data as streams which allowed us to make some processing over it.

1.1 Motivation

Our motivation for this work was the need to establish a way to give online support to operational in hazardous professions. By enhancing the analytical capabilities of the system during missions, we can allow commanders or team managers to take effective and

faster decisions leading not only to better mission operability but also to better levels of health and security among these professionals.

A proper way of maintaining a detailed analysis of the mission's global state during its course is by creating visualizations of data through time along with the triggering of alarms over certain parameters. Some of these features can be achieved when using real-time monitoring tools which allied with some stream processing are able to produce the intended outcome.

After a brief study over some real-time monitoring off-the-shelf solutions more frequently used, we managed to have an insight on a possible container-oriented solution based on data streaming which could fulfil the system requirements while adding some new features to it.

1.2 Objectives

The main objective of the current work is to provide a distributed solution for human monitoring by following an event streaming approach using real-time monitoring off-the-shelf tools. This solution should be part of the refactoring of the existing online visualization layer, VRCommander. In order to do that, some requirements should be addressed, as follows:

- Cloud deployment enabled
- Dashboard
    - Online Viewer
    - Support for instant and historical requests over data
- Online stream processing a solution for
    - Abstraction to handle heterogeneous data sources
    - Support for different types of input sources

1.3 Structure

Including this one, this dissertation is formed by 8 chapters, each one divided by subjects which are described below:

- Chapter 2, describes the current state of the existing system, VR2Market
- Chapter 3, provides a theoretical introduction to the main concepts associated with event-based systems and stream processing
- Chapter 4, presents some most popular off-the-shelf monitoring solutions
- Chapter 5, starts by describing PhisioStream architecture emphasizing its rationale; then it presents implementation options for the refactoring done as well as for the new added features
- Chapter 6, describes the comparison between the refactored PhisioStream with VRCommander; alongside it presents a proof of concept of the system applied to a real-life scenario

- Chapter 7, summarizes the conclusions and analysis of the proposed solution while pointing some possible improvements to be made at future

2 VR2Market

First responders (FR) can be included in high-risk professions as they are prone to be exposed to a vast type of hazardous environments in their daily routines, which has a strong negative impact on their health [2]. However, there is a lack of suitable systems quantifying such risk-prone exposures.

FR exhibit a statistically large occurrence of line-of-duty deaths [3]. The exposure to high levels of physical exertion, chemicals, thermal and emotional stress may add or amplify their risk of death. In this regard, monitoring both physiological and environmental status of the operational in the field could take an important role in supporting teams. In addition, post-operation analysis is also useful to understand the outcome of long-term exposure to hazardous conditions pointing out situations that demand attention in the long run.

In order to tackle this gap, the VR2Market project emerged from a consortium composed of several partners from technology to psychology, including Carnegie Mellow University, United States [1]. The VR2Market system aims to provide monitoring of both environment and physiology data from FR's

The system proposed by the project, i.e. VR2Market, is able to extract different types of useful indicators during firefighting missions that can be used not only to give the commander or team manager a global welfare state of the team but also to each individual member, preventing dangerous situations. Monitoring this type of scenario allows a better track of the status of teams in the field allowing to provide more reliable and effective information to FR's.

There are two main objectives of this system which are 1) the ability to support decisions, based on online information, to react to hazardous situations directly to the operating theatre, and 2) the possibility to store after mission information for post-mission analysis and reporting. Monitoring the environment conditions and FR physical reactions over time can help to a better understanding of the impact that this type of exposure has on their health state. The project aims to promote better levels of health among these professionals.

2.1 Real-life Scenario

The VR2Market system can be applied to a wide range of hazardous professional activities such as oil drilling industry, air traffic controllers, truck and train drivers, surgeons, miners, etc. However, the main target of the project is to monitor FR's welfare in firefighting scenarios.

The firefighting activity involves performing many physically demanding tasks while being submitted to many types of severe conditions, resulting sometimes in high levels of

stress and fatigue. Besides the need of wearing heavy clothing and protective gear in extremely hot environments, heavy lifting or walking through unstable and uneven surfaces are also in the list of existing tasks. Furthermore, another concern to take is the continuous exposure to a mixture of solid and liquid aerosols, vapours, and gases derived from fire smoke and harmful from a respiratory health perspective. Monitoring all these factors can help identify and prevent situations that can be potentially harmful to health and safety of FR's.

A straightforward advantage of sensing this risk related markers is the ability to establish, in real-time, an improved characterization of the surrounding environment of each operational. This knowledge is vital to take tactical decisions in a faster and reliable way that could have life-safety implications. One typical situation that can occur in under-ventilated conditions is the sudden re-introduction of oxygen to combustion, known as backdraft. Measuring changes in oxygen concentration allows operators to anticipate this kind of events.

The ability to locate each team-member in real-time is another benefit of the system. Sometimes the operating theatre could turn into a dangerous maze mainly due to dense smoke conditions. Hence, firefighters are often unaware of potential hazards in the field during firefighting. By collecting location data of each team-member the process of making tactical decisions is easier for commanders allowing them to relocate FR's whenever it is necessary and consequently avoiding hazardous situations.

2.2 Architecture

The VR2Market system could be sliced into several layers, although we can group them into four main groups which are the data collection, monitoring, processing and review units as depicted in Figure 1. The different components are described below:

- VR-Remote –VR-Unit configuration interface
- VR-Unit – Sensing data gather
- Data Collector – Aggregator of data from all VR-Units
- VR-Commander – Visualization of data during acquisition (Online mission monitoring)
- VR-Mission Review – Visualization of data post acquisition (Offline mission review)
- VR-Data – Data storage and retrieval to other services

VR-Unit (personal gateway) supports the data collection as an aggregator of local sensing of environment and psychophysiological data of each individual. VR-Unit is responsible for managing the communication with each sensor node (most using Bluetooth) and provides the time reference assuring, used as local time ground truth when relaying data to the Data Collector.

VR-Unit acts as a local data sink of all sensor data collected by each VR-Unit and simultaneously as a bridge to the remote long-term storage used by monitoring, processing and

review units (VR-Data services). Typically running on a smartphone or a Raspberry PI device, VR-Unit works not only as a personnel data gather but also as a gateway to the network by routing the locally collected data, in the operating theatre, from each WBAN to the Data Collector node. From here it is possible to access missions' data through public services making it accessible for data analytics and third-party applications.

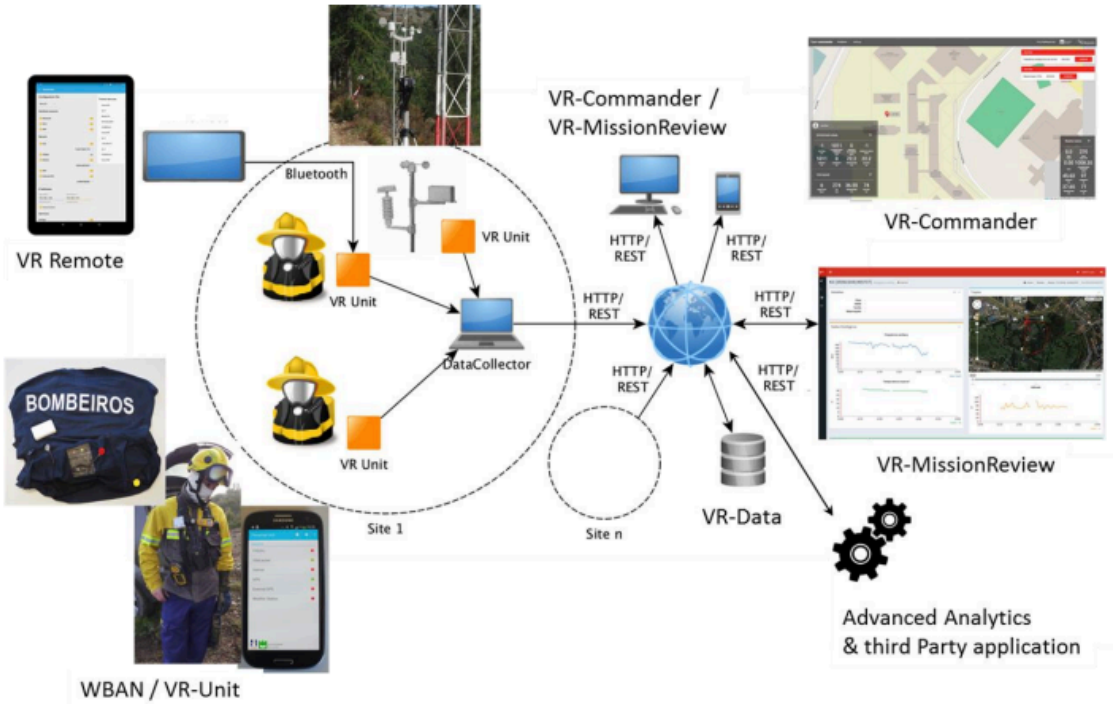

Figure 1 – VR2Market System architecture

VR-Remote allows the team member to do a proper configuration of his VR-Unit which then starts to collect both physiological and ambiance data from the wearable sensors attached to his equipment. Each VR-Unit is then responsible to send the gathered sensor data to Data Collector in order to be consumed by the rest of the system.

The Data collector is the main data input endpoint of the system – acting as middleware between the data collection and the system dashboards VR-Mission Review and VR-Commander.

VR-Mission Review is used for offline mission review, useful to make conclusions about what happened in the field. By the other hand, VR-Commander is used for online visualization of all the FR and collected data in the operating theatre which helps team managers to take faster decisions aiming to optimize critical event approach in a securely and effectively way.

Finally, VR-Data is the sink of all the data temporary allocated to Data collector and provides access to it. It provides also a persistence service of the system both for storage and retrieval, accessible through an application programming interface (API) which can be used to develop third-party applications.

The system is also capable of receiving alerts sent by team-members through their aggregators. This type of events is highlighted due to their need of being properly attended in a promptly and effectively manner.

The following sub-sections briefly specify the sensing devices that compose the WBAN, both the environment and physiological.

2.2.1 VR-Unit sensing

Currently, the VR-Unit supports both individual FR (physiology and movement) and environmental data namely their location, vital signals (heart rate, body temperature, actigraphy and skin perspiration) and body-area environment markers (toxic gases, surrounding temperature, humidity, pressure, and luminosity). By keeping track of these factors, it is easier to correlate its impact on the subject and warn him about any incoming high-risk situation.

VR2Market System vital sensing relies on the use of the VitalJacket [4] t-shirt which is a wearable device developed in order to acquire medical quality multi-lead ECG, actigraphy data and body temperature in ambulatory scenarios. In the VR2Market system, the main focus is on the ECG and heart rate monitoring.

In many scenarios collecting only physiological data is not enough as many surrounding parameters are continuously changing, affecting drastically FR's conditions. Exposure to hazardous elements such as high temperatures, dangerous levels of toxic gases or dry environments for long periods of time could lead to stress or fatigue situations. In order to gather these datasets, FR's are equipped with a set of wearable devices developed to address both physiological and environmental data. The collection of environment information relies on different sensors.

- FREMU;
- AmbiUnit;
- Weather Station sensor;
- GPS sensor (aggregator);
- G2RAYS sensor (external).

FREMU can measure air temperature, carbon monoxide, atmospheric pressure and relative altitude. The AmbiUnit can be seen as a second version of FREMU as in addition to the signals acquired by that device, this one is also capable to collect humidity and luminosity values allowing the user to choose what gases to monitor (carbon monoxide, carbon dioxide, nitrogen dioxide, oxygen, and ozone). This functionality makes the sensor more adjustable to each situation according to the operating environment.

Unlike these two sensors that can work in ambulatory scenarios, the Weather Station was

developed to work as a fixed sensor measuring values of temperature, rain, wind direction and speed, atmospheric pressure, humidity, luminosity and sensor battery.

To keep track of the position of each FR their location is gathered in two ways, one from the internal aggregator GPS and the other from G2RAYS, providing values of altitude and location.

2.2.2 Real-time Data Recording

Successful firefighting is, in most of the times, strongly dependent on two factors: time and information [5].

In a real context, it is estimated that firefighters have less than five minutes to get ready for whatever they will face in the operating field [6]. Bearing this in mind it is important to establish a way to collect and feed as much data as possible in a structured way and in the shortest period of time.

The VR2Market system relies on VR-Data as data recording back-office. VR-Data API provides a data view of the acquisitions loaded and stored in the system. Through VR-Data, it is possible to import, delete and export acquisitions to be used by other applications or systems. In order to assure this service, it is responsible to support a REST API which stands for making information disposable to third-party applications. It relies on a simple UI for management purposes.

2.2.3 System Dashboards: VR-Commander and VR-Mission Review

Visualization is recognized as an independent and important research area with a wide array of applications in both science and day to day life [7].

After data gathering, a key requirement is to turn it readable to operators. In order to be useful in the field, VR2Market data should be presented in a structured way so that its analysis could be fast and effective from FR's side. The VR2Market's visualization layer is composed by VR-Commander (online review) and VR-Mission Review (offline review) as system dashboards for monitoring both FR and environment status. These front-end elements are supported by a Java EE framework, deployed in a Wildfly[1] server and depends on JavaScript for custom UI. Both dashboards rely on VR-Data API.

VR-Commander provides an online visualization of the state of the mission by establishing a view of the FR location, at each instant, overlaid on a map (supported over Google Maps). This live visualization of data streams can be useful to support decisions as the mission commander can access all the incoming data of each operational in real-time.

[1] http://www.wildfly.org/

The other visualization tool is the VR-Mission Review for offline review. The information is the same as in VR-Commander but now supporting navigation through time over pre-acquired missions. VR-Mission Review compiles data from a certain acquisition into a view in which the gathered information is sliced into many charts displaying the different sensor measurements and a map illustrating the path taken by the firefighter. The review of previous missions can be helpful to understand operational related questions which could lead to improving team efficiency over time.

3 Event based & Streaming

An event was defined in [8], as a significant change in the state of the universe. It is seen as something that has occurred within a particular system or domain [9]. Observations of the universe at different time produce unique events, even when time is the only property that is changing. However, among the infinite number of events produced over time, there are some which are more relevant than others to an application extent.

Typically, a monitoring application collects and reacts to changes occurring in a system or measurements produced by sensors, in form of events [10]. This implies the application to have the ability to observe events and being able to consume them. An observation of an event can be seen as a tuple consisting of timestamp and event-related information.

3.1 Event based system

According to [11], the use of event-driven architectures in sensing systems allows reducing the required power for communication, reason why it is becoming the new paradigm.

An event-based system is considered to be a system in which observed events, through some processing, can trigger reactions in the system [10]. These systems consist of three main elements: monitoring, transmission, and reactive components. The monitoring component is responsible for observation and representation, transmission component for event routing and, finally, the reactive component is used to trigger actions in the system through some application logic [12].

Event data is the most important source of information when it comes to decision making from derived insights [13], [14]. Hence, more than collecting, it is crucial to transforming data into real value.

3.1.1 Event processing

When it comes to event-based systems, a common goal is to extract insights from a particular system through the observed events. In these contexts, a processing layer is usually used to perform a set of operations over events including namely reading, creating, transforming and deleting events [9]. The authors in [15] and [16] categorize the processing into three distinct styles: Simple processing, Stream processing, and Complex processing.

3.1.2 Notation of time

Time plays an important role in event processing applications. However, when considering stream processing, several time-related constraints must be taken into consideration. It is stated in [17] that the time-value of information is significant in many use cases where the value of particular insights decreases quickly after the event.

Within any typical data processing system, there are two domains to consider:

- Event time – time at which events occurred, often provided by the event producer
- Processing time – time at which the system becomes aware of the event

Event time versus processing time

Ideally, event and processing time would always be equal, allowing the processing of events as they occur. Nonetheless, what really happens is that the disparity between event and processing time is often influenced by the underlying input sources, execution engine and hardware characteristics.

The plot represented in Figure 1, shows the progress of event and processing time in a real scenario:

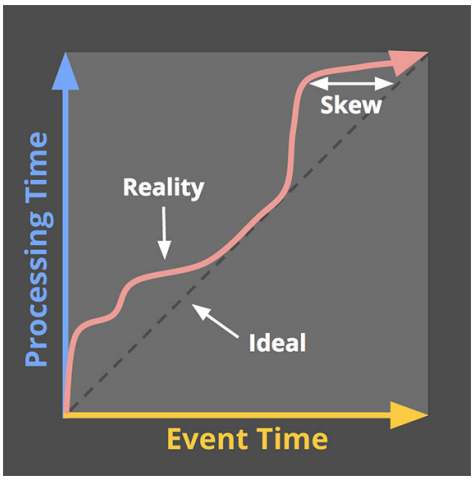

Figure 1 – Time domain mapping. The X-axis relates to the time X up to each all data with event times less than X have been observed. The Y-axis relates the progress of clock time during the processing execution. [18]

The dashed line is associated with the ideal situation, where event and processing time would be equal, while the red line represents reality.

There are two visible moments of lag, the first one at the beginning of the processing time, and the other at the end. In the mid time, there is a moment where event and processing time tend to the ideal line. The represented skew is the difference between processing and event time, which is, essentially, the latency introduced by the processing pipeline.

Typically, due to the infinite nature of event datasets, data processing systems establish some time intervals in the incoming data which work as finite time boundaries. Unfortunately, due to the skew, the time lapse between event time and processing time, static solutions may lead to incorrectness when mapping events and processing timestamps. For that reason, when designing a data processing application, one should be aware of the uncertainty imposed by these complex

data sets – ideally, a profiling on the system skew could be helpful.

In addition, a data processing application should be able to consume, process, and generate results very close to real-time as possible (minimize the skew), in some cases of the magnitude of a few nanoseconds to at most a couple of seconds [19].

3.2 Why stream?

Structuring data as a stream of events is becoming a popular approach for data processing architectures [20]. In many situations, there is a continuous flow of related occurrences over time that can be seen as a stream of events. As a matter of fact, according to Luckham [21], an event stream is a sequence of events ordered by time, such as a stock market feed.

Saxena and Gupta refer to a data stream as a continuous flow of any kind of data through any kind of medium [19]. Among the many potential advantages of handling data as streams, its main purpose is the task of getting actionable insights from a system, almost instantaneously. Most of the systems to be monitored happen as a stream of related events – seismic monitoring, water management, financial market data, machine metrics.

With newly available technologies and computing resources, data can be streamed from many sources and used by a variety of consumers almost immediately or later on [17]. Improvements of message throughput for persistent message queues, resulted in an increment from rates of thousand to millions of messages per second, even while persisting messages.

Nowadays stream processing often appears associated with an online approach to handle big data scenarios, which were more commonly associated with batch processing solutions, namely Hadoop[2].

There are two types of data streams: bounded and unbounded streams, as shown in Figure 2. Bounded streams are finite and unchanging, have a defined start and end and everything is known about the set of data. Hence, data processing stops by the end of the stream which is commonly called batch processing [22]. On the other hand, unbounded streams are unpredictable and infinite. For that reason, data processing starts from the beginning. This is known as real-time processing [23]; the state of events is kept in memory for processing.

Batch processing is a good approach to handle huge amounts of distributed data since data is collected and analysed in batch. The processing occurs on blocks of data previously stored over a period of time. Computation is performed over a single global window which contains the complete data set [24].

One of the best framework for processing data in batches is the Hadoop MapReduce which is illustrated in Figure 3.

---

[2] http://hadoop.apache.org/

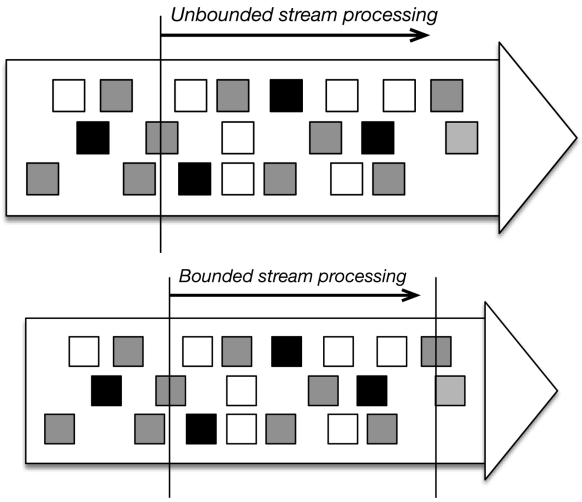

Figure 2 – Unbounded vs bounded stream processing; Unlike of Bounded stream processing where input has a beginning and an end, processing occurs in chunks, with Unbounded stream processing processes data continuously and indefinitely since the input does not have an end. Adapted from [25]

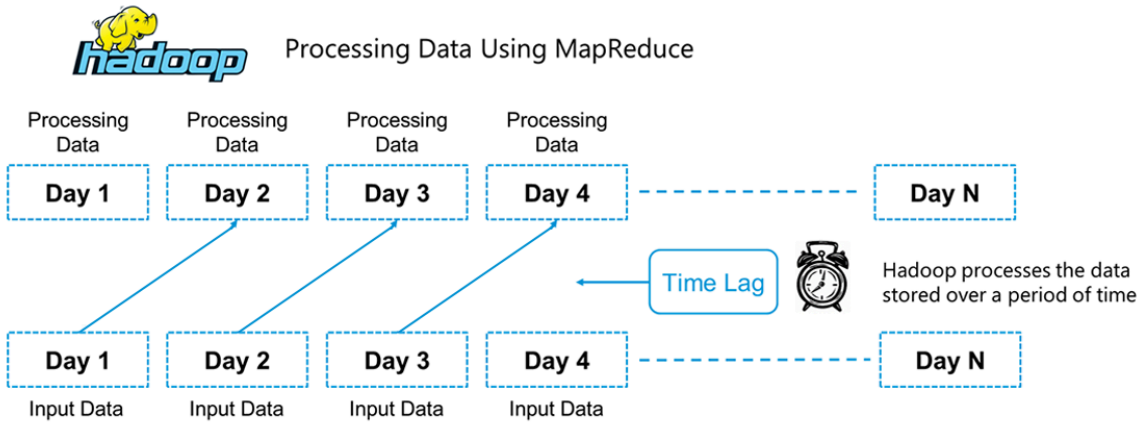

Figure 3 – Processing data using Hadoop MapReduce. Adapted from Spark tutorial at Edureka[3]

This kind of data processing works well when handling large volumes of information is more relevant than taking real-time insights over data. Although, if analytical results are needed in real time, the suitable approach is stream processing.

Stream processing deals with unbounded streams of data and that is why it is considered to be the golden key to turning big data into fast data [26]. With platforms like Spark Streaming[4], data can be processed as soon as it is generated which produces near-instant analytical results. By

[3] https://www.edureka.co/blog/spark-tutorial/

[4] https://spark.apache.org/

using this kind of approach, the time lag issue can be tackled, since the data is analysed in micro batches of few records, as illustrated in Figure 4.

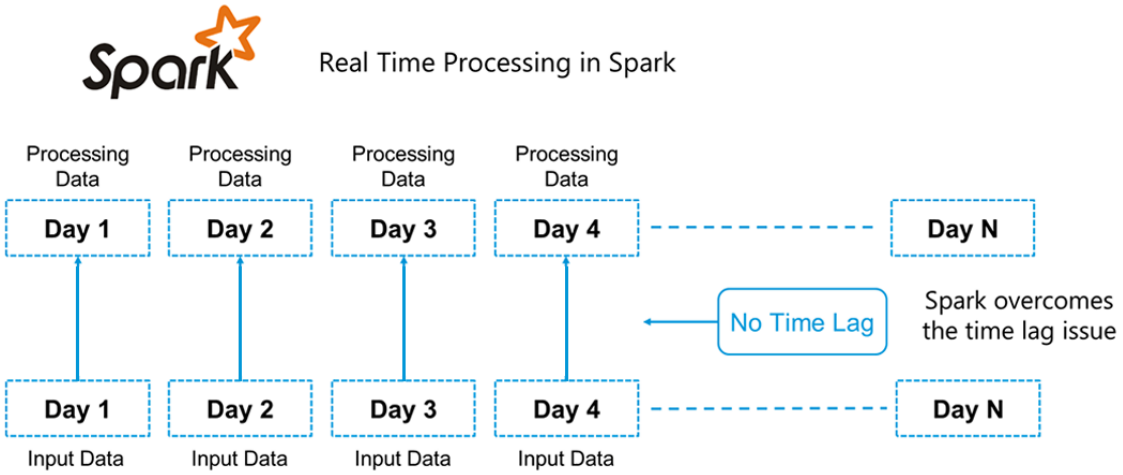

Figure 4 – Processing Data using Spark. Adapted from Spark tutorial at Edureka[5]

3.3 Stream processing

Stream processing is often part of real-time systems and is an approach used to handle the flow of information, the stream, which is usually decomposed in three-stages: 1) Events generated by event producers 2) are subjected to some sort of intermediary processing layer before being 3) routed to their destination, the event consumers, which can either store the events for later use or display them in a suitable interface.

In the typical architecture [27], after collecting data from event producers, the system provides a transport pipeline, which consists, in essence, of a pipeline capable of streaming events from one or multiple sources and routing them to a stream processing application. The stream of data can then be subjected to a wide variety of operations such as transformations or aggregations before, typically, being persisted in the low-latency data store. From here, data can be extracted from the data store and presented to the end user through a wide range of visualization and analytical tools.

3.4 Broker – The core abstraction

Dunning states in [17] that message-passing infrastructures are at the heart of the stream-based approach success.

The capabilities introduced by the improvements made in the way message-passing systems work address not only the need to handle large volumes of data, by being highly scalable but also

[5] https://www.edureka.co/blog/spark-tutorial/

help in the process of decoupling the involved components, which is a key to agility. Furthermore, brokers provide elasticity which proves to be useful when traffic spikes occur since data can be kept in a queue. Another key aspect to consider when designing stream-based systems is its need to handle data from multiple sources and making it available to multiple consumers, which is possible thank to this messaging technologies.

Wampler [27], points message queues as the core integration tool in a stream-based architecture. These first-in, first-out data structures provide the ability to improve scalability through parallelism since data can be arranged into user-defined topics, where each topic has its own queue. Most message queues support an arbitrary number, not only, of producers to insert messages but also of consumers to extract them. Along with the fact of its implementation abstraction, scalability and resiliency features can be often implemented in an easier way.

Typically, this kind of messaging systems allows producers to send messages regardless of the consumers being ready to receive them; messages are kept available until it happens.

In order to fit the needs of a typical stream-based architecture [12] the message-passing infrastructure must regard some features, as follows:

- **Isolation between producer and consumer**: producers do not have to know about the consumers that will process the messages;
- **Persistence**: feature required to ensure producers and consumers decoupling so that data do not disappear in cases that producers and consumers are not coordinated
- **High throughput**: essential to assure the handling of high messaging rates
- **Fault tolerance**: required for production use since data loss is unacceptable

When it comes to streaming data, there are many advantages to storing it. Aside from being possible to extract powerful insights from event streaming data it is required for the message-passing system to persist it so that reading from specific points in the flow can be possible at any time. Therefore, the event stream can act as a "time-machine" which can be useful to perform deeper studies which allow the detection of trends and patterns in datasets.

3.5 Kafka

Kafka[6] is one of the most popular platforms to redundantly store huge amounts of data, keeping a message bus with huge throughput and perform real-time stream processing on the data which goes through it, all at once [28], [29], [30].

[6] http://kafka.apache.org/

With the need of scaling their multiple data pipelines, in 2010, LinkedIn, came up with a distributed system designed to handle data streams. Now, available as an open-source project under Apache, it has grown by the day not only in functionality but also in reach. It is estimated that Kafka is used by one-third of the Fortune 500[7], including seven of the top 10 global banks, eight of the top 10 insurance companies, nine of the top 10 U.S. telecom companies and six of the top 10 travel companies [31].

Often defined as a "distributed commit log" or as a "distributed streaming platform", it is built to be fault-tolerant, high-throughput, horizontally scalable and allows geographically distributing data streams and processing.

The commit log of a filesystem or database is designed to provide a durable record of all occurred transactions in order to be possible to consistently build the state of a system. With Kafka, data is also stored so that it can be read deterministically. Moreover, scaling and fault-tolerance can be achieved with little effort since data can be distributed within the system [32].

Keeping Kafka at the heart of a stream-based architecture provides a distribution point for the system data, because, aside from being very fast it also allows data to be permanently stored. Suitable for both online and offline message consumption [33], Kafka also allows performing some processing job on streams by providing a powerful API [34].

It combines the benefits of traditional log aggregators, as the fundamental abstraction for streaming, with the benefits of message queues.

3.5.1 Kafka as a Messaging System

There are two known paradigms widely used for messaging: Queuing and Publish/Subscribe [10]. Queues are useful for fault-tolerance and scalability. Work is divided through consumers since they act as a worker grouper. For that reason, each message is routed to a single worker process.

Publish/Subscribe approach acts similarly as a notification system, subscribers are independent of each other. When publishing messages, it is not required for senders to specify a receiver. Instead, the messages are somehow categorized so that the receiver subscribes to receive certain classes of messages. Using this approach, the decoupling of systems is possible due to the fact that it allows multi-subscribers.

Combining these two approaches is what turns Kafka into a single robust messaging system. Although, there are also some other features of this system that should be referred.

[7] https://www.confluent.io/blog/apache-kafka-goes-1-0/

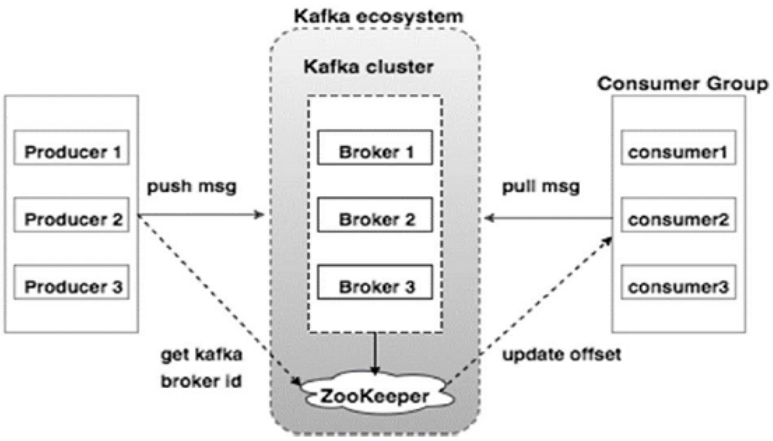

Figure 5 – Kafka cluster architecture [28]

Within its architecture, Kafka is composed of a few building blocks which include brokers, topics, producers and consumers as depicted in Figure 5. A stream of records is defined by a topic where messages can be published from producers. Each topic can have one or more partitions that are essentially separations of ordered and immutable sequences of records inside a topic.

After being published, messages are stored in the Kafka cluster, also known as a broker, which can involve one or more servers. From here, a consumer can subscribe to one or more topics and start to consume the subscribed messages by pulling data from the brokers.

Zookeeper [35], is responsible for the coordination and management of the brokers. Whenever there is a new broker, or a failure of an existing one, in the Kafka system it is used to notify producers and consumers. This synchronization service performs the load balance in topics when the number of consumers changes.

Despite being often described as a messaging system, the key abstraction in Kafka is a structured commit log of updates.

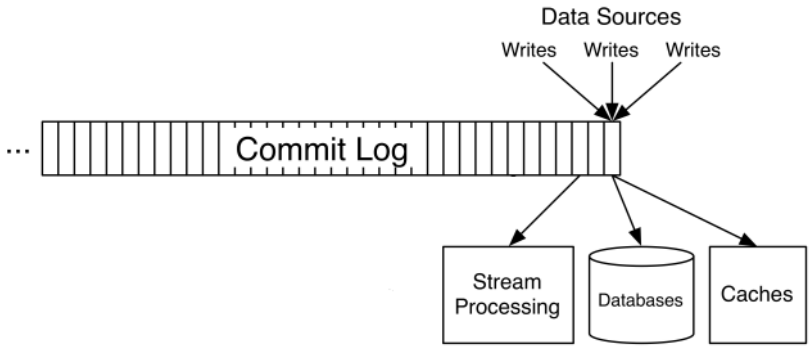

Figure 6 – Kafka's abstraction as a structured commit log of updates [36]

This feature can be better understood when analysing Figure 6 where it is clear that messages can be published by multiple producers, being appended to the log. When appended, each record in a partition is assigned with a unique id, the offset. Simultaneously, a given number of consumers can stream these records off the tail of the log, continuously, with millisecond latency. The distribution of stream updates to each consumer can be ordered, in a reliable way, as each of them has its own position in the log. Once read, messages are not deleted which enable consumers to process them whenever it is needed.

This concept of commit log is a core feature of Kafka that associated with its persistence efficiency makes it applicable well beyond the usage patterns of traditional messaging systems [37].

3.5.2 Kafka message persistence

Although Kafka can be often referred to as a messaging passing solution, it has some unique features which make it different from traditional messaging systems [32].

Typically, in a message queue, multiple subscribers can pull a message or a batch of them. However, after being processed, messages are removed from the queue. Despite being possible to extend the queue with multiple consumers, they will all keep the same functionality. This means that it is not possible to perform different and independent actions within the same event.

In contrast, Kafka persists messages by the time they are published. After being processed, messages can be replayed since they are not removed from the topic. The main advantage introduced by this feature is the ability to add different types of consumers executing different process logic on the same messages. Figure 7 gives a better perspective of this comparison.

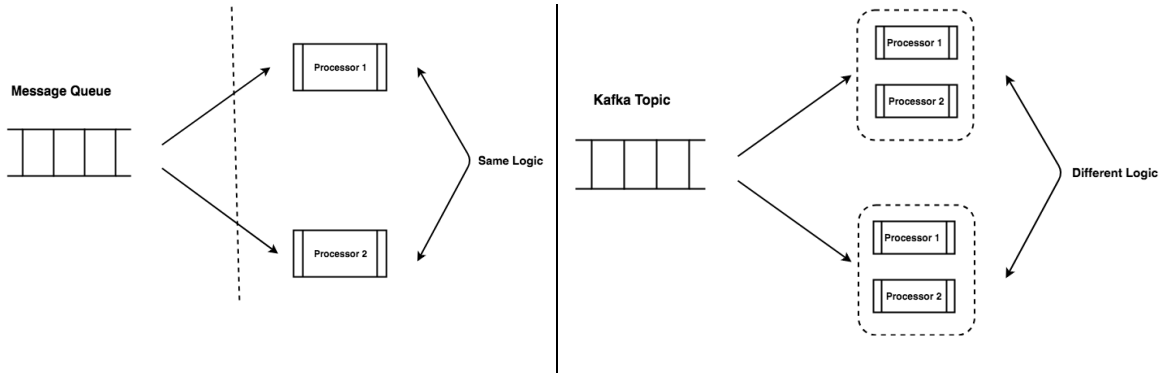

Figure 7 – Typically, with queues the same logic is applied to every message in the same domain. With Kafka different logic can be applied by different domains or systems on the same events.

While other message systems do not allow for the consumption of an already pulled message, Kafka is designed to support message reads from multiple consumers without interfering with each other. That is what makes Kafka a natural backbone for a stream-based architecture.

3.5.3 Kafka and Stream Processing

There are some properties of Kafka which makes it a frequent choice as a message-passing system for most streaming architectures:

- **Persistence queue**: Data is stored in topics, in order.
- **Fault tolerance**: In case of one or multiple brokers fail, there is no risk of data loss. In this scenario, data is served from other brokers containing replicas since Kafka replicates topic partition in multiple brokers.
- **Logical ordering**: Data is stored in order of timestamp, which can be an important feature for critical streaming applications.
- **Scalability**: Kafka can scale up simply by adding more broker nodes to the cluster.
- **Integration**: It can be easily integrated with most stream processing applications through provided APIs.

As mentioned before, message-passing systems represent an essential component of a stream-based architecture. Fulfilling this premise, with a single robust and scalable messaging system like Kafka, which provides the storage and publish/subscribe required features, next stage is to find a suitable way of processing the incoming stream of messages.

Kafka Streams [34], provides just that through a built-in API. This lightweight library is developed, specifically, for building streaming applications that transform input Kafka topics into output Kafka topics. The development of streaming services becomes easier due to Kafka Streams being cluster and framework free [38]. Since it is tightly coupled with Apache Kafka, it is able to

provide some important features when building a stream processing application, as follows:

- **Ordering**: Kafka Streams uses the capability of Kafka and consumes data in order.
- **State Management**: For some state dependent applications, this can be an important feature since data processing can require access to recently processed or derived data.
- **Fault Tolerance**: In case of failure, Kafka Streams restarts the process in another working instance, while managing load balancing internally.
- **Time and Window**: Its windowing approach, keeps each record linked with a timestamp which is helpful not only in the order processing but also for dealing with late arriving data.
- **Scalability**: Applications can be easily scalable, simply by adding more instances while Kafka automatically balance the work over them.

When building stream applications using Kafka Streams, there are some involved components which must be highlighted such as the stream topology, the record cache, and the previously mentioned, local state.

A stream topology consists of stream processor nodes linked with each other in order to perform a determined computation. It is formed by three nodes: Source processor, stream processor and sink processor. The first one is responsible for consuming records from the source topic and forwarding them to the downstream processor. Stream processors are responsible for computing data. A single topology can include multiple nodes of this type. Sink processor is the last node in the topology and it is responsible to consume data from stream processors and route it to the target topic or system.

As a Kafka integrated library, Kafka Streams offers some considerable advantages when it comes to build stream processing applications since it only depends on the Kafka cluster. That is the main reason why it is simpler than other streaming platforms. Lightness, is one of its key features. After bringing Kafka into the architecture, the only thing needed to start the processing job is the application code. There is no need for external dependencies because it uses the API directly.

Other advantages when using Kafka Streams include fault tolerance, millisecond latency in stream processing and its easy deployment. Relying only on Kafka producer and consumer APIs, Kafka Stream based applications does not require extra clusters which makes them deployable on a single-node machine.

3.5.4 Kafka Streams Key Concepts

As previously mentioned, a Kafka Streams application is defined as a processor topology (Figure 8) where streams are represented by edges and processors by each node. It can be defined

with both Kafka Streams available APIs: Kafka Streams DSL (high-level) and Processor API (low-level) [34].

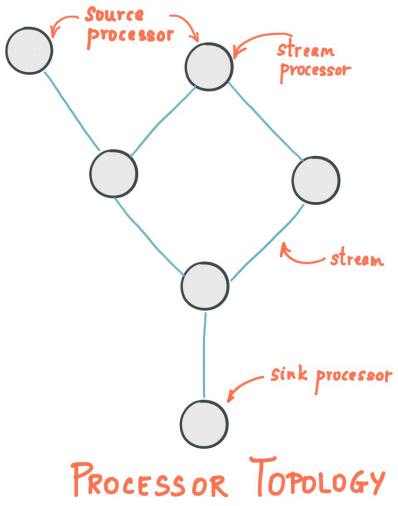

Figure 8 – Kafka Streams Processor Topology [34]

Streams and Tables

Kreps mentions in [38], that tables and streams share some duality which allows one to convert streams into tables and vice versa, as suggested in Figure 9. In Kafka Streams DSL, these abstractions are represented by KStream class (stateless) and KTable class (stateful) [34].

As a matter of fact, a stream can be interpreted with an event sourcing approach [39], since this series of updates for data, when accumulated, form the final state. Typically, these derived aggregations get persisted in a local embedded key-value store (RocksDB[8], by default) and form a KTable.

[8] https://rocksdb.org/

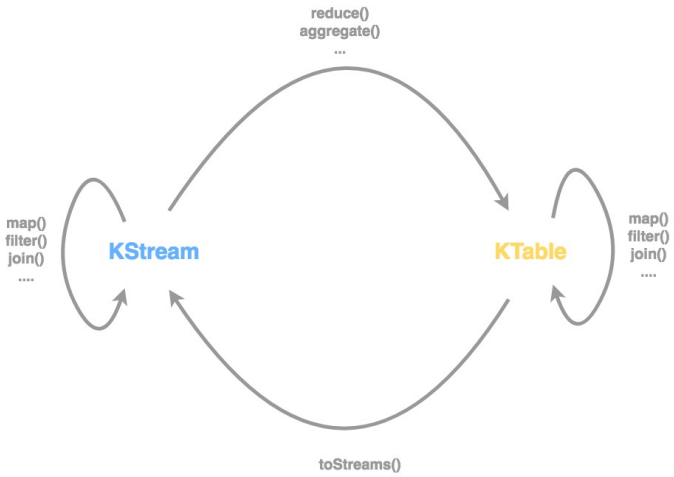

Figure 9 – KStream class (streams) and KTable class (tables) duality. Adapted from [40]

On the other hand, as well as stream records can form a table also table updates can establish a changelog stream. Hence, a table can be interpreted as a cache of the latest value for each key in a stream, as referred by Pat Helland in [41].

Stateful and Stateless Processing

There are some stream processing applications which do not require state, i.e., the processing of a determined message is independent of the processing of other messages. These applications, often called stateless, can be used to transform one message at a time or to perform filtering operations. However, whenever an application needs to perform other kinds of operations like joining, aggregating or windowing data, it is required a state.

It is easier to understand this state mechanism in stream processing when thinking about the kind of operations usually performed in SQL. If queries were performed against a real-time stream of data containing only filtering or single-row transformations (select, where), they would be stateless operations once every row could be processed without needing to remember anything between rows. On the contrary, if the query involves the aggregation or join of many rows, it must keep a state in between rows. For example, when counting, data is being grouped by some field and the state would be the accumulated counts so far in the determined window.

When it comes to stream processing, keeping the state in a fault-tolerant way can reflect an issue whenever processors fail. Once maintaining all state stored in a remote database can slow down the application performance, it should be followed another approach. The previously mentioned duality between tables and streams provides a mechanism for handling this problem by storing the streams in a Kafka broker. When performing stateful operations as aggregations, windows or counts the Kafka Streams DSL allows to create and manage state stores, automatically. This state is kept in a local table and updated by an input stream and if the process

fails it can restore its data by replaying the stream.

Windowing

Kafka Streams provides a computation model based on keys and temporal windows which allows controlling how to group records that share the same key for stateful operations, such as aggregations or joins, into windows.

Considering the given example, in Figure 10, where a sensor performs a counting operation every fifteen seconds the resulting stream could be, as follows:

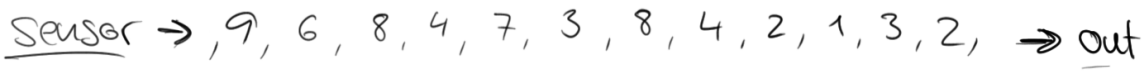

Figure 10 – Stream of sensor transmitted counted values. Adapted from [42]

In order to compute the total number of counts, one would perform the sum of the individual counts, however, due to the nature of a sensor stream, it is not possible since it flows continuously in time.

Windowing operations can be seen as a special case of stream processing where it is applied a batch processing approach. Kafka Streams allows performing four different kinds of windowing operations: tumbling time window, hopping time window, sliding time window and session window.

In Figure 11 is shown how to define a sliding window which basically returns the total number of counts, in the last minute, at every thirty seconds. If it is required for the windows to be non-overlapping, elements can be grouped into finite sets of the stream. Figure 12 refers to a tumbling-window which returns the total sum of counts at every sixty seconds.

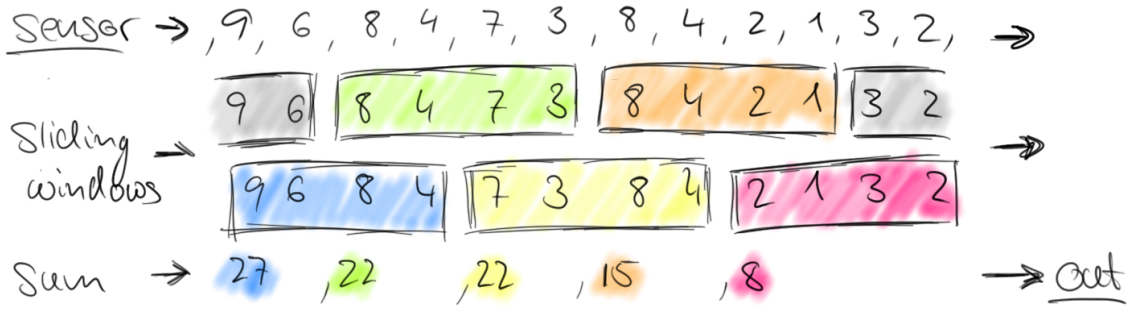

Figure 11 – Example to illustrate the use of sliding windows. Each coloured bucket refers to a sixty second window of the stream. Adapted from [42]

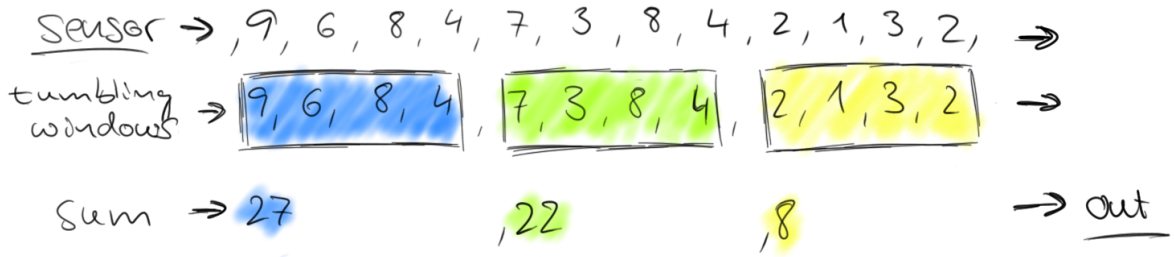

Figure 12 – Example to illustrate the use of tumbling-windows. Each coloured bucket refers to a sixty second window of the stream. Adapted from [42]

Despite also being based on time intervals, hopping time windows are different than sliding windows. They are also defined by the window's size, albeit windows move forward over a certain and determined advance interval ("hop"). On the other hand, session windows are used to group key-based values into sessions which, mainly, consist of periods of activity split by a defined gap of inactivity, i.e., idleness. Processed events that fall within this inactivity gap are merged into existing sessions, in case of falling outside the session gap, a new session is created.

3.5.5 Similar Solutions

Clearly, choosing a stream-processing system is directly dependent on the problem to be solved. As it has been said before, the broker is one fundamental component in stream-based architectures.

A large part of the design behind Kafka was inherited from traditional message queues [37]. However, there are still some differences between Kafka and other available technologies under this segment. Mostly due to scale and speed requirements of stream-based architectures, Kafka proves to be more efficient than conventional message queues. Similarly, to most of them, it employs a pull-based consumption model. However, its focus on log processing gives it not only higher rates of throughput but also integrated distributed support and scaling capabilities. This match between event log abstraction and message queues approach is what is making Kafka widely used as the backbone of fast data architectures [27].

As stated before, some of this functionality can also be achieved with message-passing systems like Apache ActiveMQ[9] or RabbitMQ[10]. Still, persistence, known as a key feature of Kafka, was a high-cost capability for systems like these, leading to performance decrease.

[9] http://activemq.apache.org/

[10] https://www.rabbitmq.com/

From the comparative study carried out at [43], it is possible to infer some Kafka benefits over these two messaging systems. From this work, it is easy to understand which Kafka properties are responsible for enhancing both the producer and consumer performance. The results are compiled in the charts included in Figure 13.

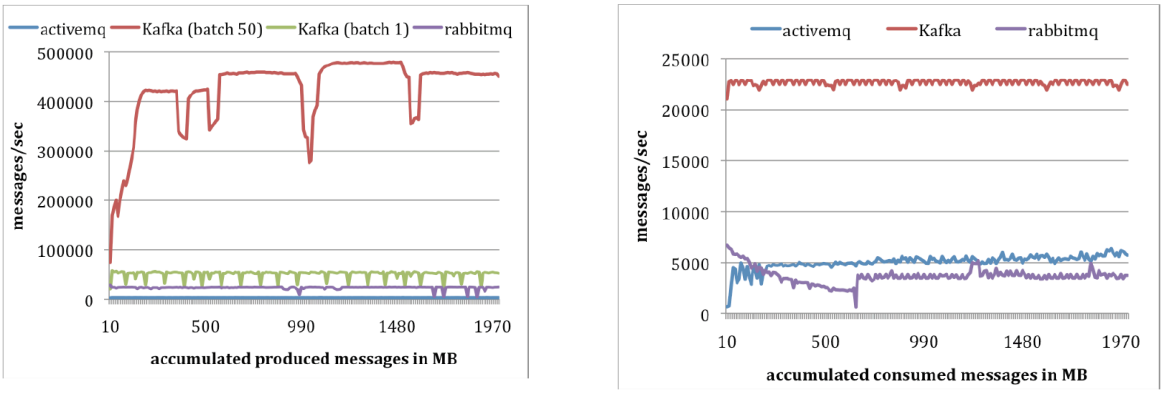

Figure 13 – Producer and consumer performance results when comparing Kafka with ActiveMQ and RabbitMQ [43]

Through a brief analysis, it is clear that Kafka overtakes the other two systems. For the purpose of the experiment, the Kafka producer was configured to send messages in batches of size 1 and 50. Apparently, the same was not possible with ActiveMQ and RabbitMQ, since there was not an easy way to do it. It is visible that even when the considered batch size is the same for the three systems, the throughput rate of Kafka is still higher than the others with rates of 50,000 messages per second. The red line indicates that when in batches of 50 messages Kafka can publish at an impressive rate of 400,00 messages per second, which shows the improvements introduced by batching.

This throughput results can be explained not only with the efficient way Kafka storage works but also with the absence of message acknowledgment in the system. In fact, this feature can sometimes create a tradeoff situation. Some messages can be dropped since the system does not keep a track of which of them are received or not by the broker, what could be potentially problematic when working with critical data.

Relatively to the consumer tests, results pointed again to better levels of performance, particularly of throughput. In contrast to ActiveMQ and RabbitMQ that had to maintain, in disk, the delivery state of each message, Kafka does not. Allied to its referred storage efficiency, on average, Kafka was able to consume messages more than 4 times that the others.

4 Cloud based monitoring solutions

Monitoring plays an important role in systems management. One solution for monitoring systems during runtime is by log observation (e.g. output on custom system domain events to screen, file, etc) and/or indirect measures over runtime side effects of the system (e.g. load on memory, CPU, etc). Kreps states in [44] that logs can be seen as a synonym for stream which places event log as the core abstraction of a stream processing architecture [27]. Cloud-based systems are not different, adding the natural distribution of the system over the cloud. For that reason, big companies like Amazon[11], Netflix[12] or Google[13] adopted several solutions and tools developed with the purpose of monitoring running systems [45]. Most of these solutions are being used widespread due to their ability to handle big flows of information which makes them able to provide a good support for online analytics and not only for log management context but also for system related metrics (e.g. throughput, delays, resources consumption, …).

In this section, we present two of the most popular and proven stacks for log and system monitoring: ELK stack and TICK stack

4.1 The Elastic Stack

Developed by Elastic[14], this is an end-to-end stack which aims to deliver actionable insights, in real time, from a vast array of types of structured and unstructured data sources. Estimated to be downloaded around 500,000 times every month, this stack could be considered the world's most used tool in log management[46].

Composed of three products, under open-source license [47], each component performs a specific task in data management:

- Elasticsearch works as an indexing and searching engine;
- Logstash works as a processing framework & ELK entry point;
- Kibana works as a visualizing framework.

Lately, it was announced a fourth product, called Beats [48], which, when attached to the stack works as a lightweight shipper from hundreds or thousands of devices to Logstash.

[11] https://www.amazon.com

[12] https://www.netflix.com

[13] https://www.google.com

[14] https://www.elastic.co/

Despite of being designed to work as an integrated solution, Elasticsearch and Logstash can be used separately as powerful tools applicable to different use cases.

4.1.1 Elasticsearch

Developed on top of Apache Lucene, Elasticsearch [49] is an open-source, distributable and highly scalable tool for content indexing and web search. It is considered to be the best tool to store data in an effortlessly queryable way. Mainly due to the fact of being multitenant-capable and allowing a full-text search, similar to Google's search engine. It's being used by some major companies like Cisco[15] or Netflix, among many others.

Full-text querying is assured by implementing inverted indices with finite state transducers [49]. In order to store numeric and geodata, it uses BKD-trees [50] and a column store for analytics [51]. Thus, it is quick. Operations on data such as create, retrieve, update and delete (CRUD) can be performed through its comprehensive REST-based API. Some core concepts of this framework are as follows:

- **Field**: The smallest single unit of data stored in Elasticsearh. Similar to a column in a relational database.
- **Document**: Collection of fields. Similar to a row in a relational database.
- Type: List of fields, defined for every document. Similar to a table in a relational database.
- **Node**: Running instance of Elasticsearch.
- Cluster: Collection of nodes, having one or multiple.
- **Mapping**: Defines the fields, the data type for each field, and how the field should be handled. Similar to schemas in a relational database.
- **Sharding**: Since an index can store any amount of data, if it exceeds its disk limit searching speed would be affected. The shard acts as an independent index hosted in a node within a cluster. By doing this it is possible to horizontally scale the content volume while improving performance due to parallel operations across the distributed shards.
- **Inverted index**: Elasticsearch makes an index of the content and looks into indexes as opposed to the contents. Thereby, it searches for an index instead of text, feature useful to provide fast full-text search.

In contrast with RDBDMs, where data is stored in one database leading to vertical scaling when data amount increases, Elasticsearch scales horizontally. For that reason, it can handle large amounts of events per second by distributing indices and queries across the cluster while

[15] https://www.cisco.com

controlling the equality of loading [52]. Another advantage over RDBMSs is the fact of being schema free. It means that when a new document with a new field is added to an index, Elasticsearch will automatically update the mapping since it does not impose a schema to a document. Otherwise, in RDBMSs, the update process implies schema modifications which than must be followed with its appliance to the existing records.

In spite of generally being used for log monitoring, there are many other scenarios where Elasticsearch can be applied [53].

4.1.2 Logstash

Logstash [54] is a plugin-based event forwarder for log collection, centralization, parsing, storage, and search. The main advantages of this framework are the possibility to simultaneously ingest from multiple sources, its performance, scalability and the fact of being easy to use.

Aside from allowing to pull data from a wide range of sources, it also provides tools to filter, massage and shape the data. Written in JRuby[16] and running on a Java Virtual Machine (JVM), Logstash is designed atop a simple event-based architecture. Each event is processed in the pipeline that can be sub-divided into three blocks:

- **Log input** – One of the most valuable features of Logstash is its ability to take inputs from various sources. Among many others available is the input plugin to read events from Twitter, Amazon Web Services Cloudwatch[17] and TCP/UDP sockets. By default, if the input is not previously defined, Logstash automatically creates a stdin input. There is also an open source tool, from Elastic, called Beats, useful to gather data from distinct sources and ship it to Logstash.
- **Log filtering** – Next, when events enter the Logstash server, it is possible to apply filters to modify, manipulate or transform them. There are multiple types of filters able to add additional information or remove dispensable one. More complex operations like parsing unstructured data can be performed with the use of Grok filter [54].
- **Log output** – Lastly, by the time of outputting data, as with inputs, Logstash comes with a vast number of output plugins. For that reason, it is possible to send events to a huge range of destinations, including TCP/UDP sockets, files, SQL databases, etc. It can be integrated with metric engines, alerting tools, graphing suites or in a custom application by creating an own plugin, in a simple way.

[16] http://jruby.org/

[17] https://aws.amazon.com/cloudwatch

Taking advantage of the variety of plugins for input and output it is possible to build a data pipeline capable of centralizing data processing, which allows one to convert many different input sources into a single common format. Moreover, custom plugins can be developed and published.

Combining the multiple available plugins, a higher range of different environments can be covered which makes Logstash a really versatile tool.

4.1.3 Kibana

Kibana [55] represents the visualization layer of the Elastic stack. Entirely written in HTML and Javascript, this web application is used as a front-end interface for data stored in an Elasticsearch cluster [56]. The hard task of analytical processing remains in Elasticsearch side, while Kibana's role stands on exposing the processed data through beautiful histograms, geomaps, pie charts, graphs, tables, along with others.

It queries the Elasticsearch cluster through its RESTful API and displays the data with powerful graphics which simplifies the job of understanding large volumes of data. Since Kibana is just a rendering of processed data from Elasticsearch, it is possible to build a real-time visualization at scale. Whenever data volume grows, the cluster is accordingly scaled to offer the best latency.

This framework adds the visual power to Elasticsearch aggregations offering features like:

- Real-time analysis, summarization, charting, and debugging capabilities;
- Highly customizable, intuitive and user-friendly interface;
- Possible to have multiple dashboards;
- Integration in different systems by sharing or embedding dashboards.

Dashboards are nothing more than an interface for JSON[18] documents. Simple to set up and use, they supply an easy and fast manner to visualize data stored in Elasticsearch with the absence of coding.

4.2 The TICK Stack

The TICK stack is a collection of open-source projects combined to deliver a time series platform for storing, capturing, monitoring and visualizing data in real time. In addition, it offers an API to build custom implementations to suit different types of approaches. Developed by InfluxData [57], its name is an acronym of the four tools involved:

- **Telegraf**: lightweight and efficient. With over 40 plugins available, it is able to ingest and sink from and to multiple data sources. Integration with messaging systems such

[18] https://www.json.org/

as Apache Kafka or with third-party API from Google and Amazon are some examples, along with others.

- **InfluxDB**: time-series database with high availability and performance. Schema-less and with a SQL-like query language it is capable of handling unstructured data.
- **Chronograf**: web-based application integrated with InfluxDB for data visualization.
- **Kapacitor**: monitoring and alerting system able to trigger events over the data and materializing metrics back to InfluxDB, using either stream or batch mode for metric processing.

The stack is optimized to work as a single unit, yet each component can provide significant value as a standalone installation.

Key Concepts of InfluxDB

A time series is an ordered sequence of data points (indexed, listed or graphed) of a variable at equally spaced time intervals [58]. A time series database (TSDB) is a database which is optimized for time series data.

Written in GO, InfluxDB is a time series database whose distinctive factor from other types of databases is its optimization to maintain time indexes on a very large amount of records. Some of its core concepts are, as follows:

- **Measurement**: Similar to a table in a relational database. Time series are persisted into measurements.
- **Point**: A point contains mandatory field(s) and a timestamp.
- **Timestamp**: Represents the time a point was created. If no timestamp is provided by the time a point is created, InfluxDB automatically assigns one to it.
- **Tags**: Indexed as metadata of the point.
- **Series**: Collection of measurements and tags.

InfluxDB is schema-less which means that series, measurements, and tags can be added at any time. Two relevant features of this TSDB are downsampling and data retention.

Handling hundreds of thousands of data points per second can lead to storage concerns over time. Keeping the lower precision, summarized data and getting rid of high precision raw data, can be a solution to this problem [59]. Data life-cycle is ruled by retention policies which describes how long data is kept and how many copies are stored in the cluster (replication factor). Downsampling of data is performed with the use of continuous queries. When implemented, these queries are automatically and periodically executed while the results are stored in a measurement for future use.

An important feature provided by InfluxDB is the ability to tag metrics. Tags are indexed while fields are not. This means that querying series by tags is more performant than by

fields. For branching purposes (e.g. GROUP BY) this feature proves to be useful. However, it should be avoided to store high cardinality data in tags since this highly increases the size of the index.

4.3 Similar alternatives

When it comes to visualizing log data, Kibana is a commonly used tool mainly for its native integration with ELK stack which provides a simple and user-friendly setup. Running on top of Elasticsearch makes Kibana a powerful tool to create comprehensive log analytics dashboards with little effort from users. Otherwise, combining InfluxDB and Grafana [60] proves to be a proper way for time series and metrics data visualization [61].

Grafana is an open source project, most commonly used for visualizing time series data. Its dashboards are highly customizable and make it possible to represent large amounts of data from many different storage back-ends. It has a specific query editor for different types of data sources that are customized for the features and capabilities of each one. Besides that, it has a pluggable design which allows users to integrate custom data sources [62]. Some of the key features of this web-based dashboard builder are, as follows:

- Multi-tenancy;
- Integration with LDAP authentication;
- Fully-interactive;
- Alerting system: possibility to apply threshold rules on metrics that can trigger notifications;
- Annotations: Graphs can be enriched with events from different data sources;
- Support for several data-sources: Graphite[19], InfluxDB, Prometheus[20], Elasticsearch, and plenty more;

Grafana natively provides numerous ways to manage time ranges for the data being visualized, presenting not only real-time data but also making possible to seek through historical records, as shown in Figure 14.

[19] https://graphiteapp.org/

[20] https://prometheus.io/

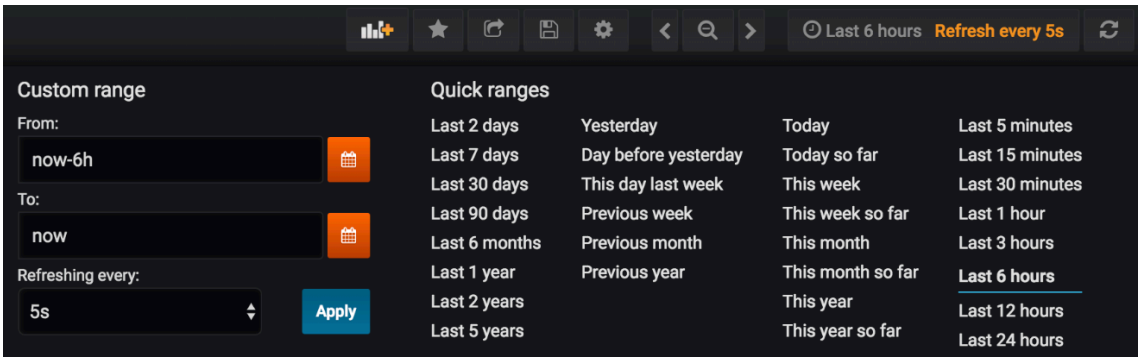

Figure 14 – Grafana time range controls. Adapted from [63]

It is possible to choose either preset values in the “Quick ranges” list, or create a custom time range. When enabled, the auto-refresh feature will reload the dashboard at the chosen time rate. Another useful feature of Grafana are the variables. By using variables, Figure 15, it is possible to create more interactive and dynamic dashboards since changing data that is being displayed is easier. Variables work as placeholders for values that can be used in metric queries. Whenever these values are changed, through the dropdown menu, the dashboard is reloaded reflecting the values of the new metric queries. This feature allows one to create a global view of a whole system while simultaneously having the ability to slice it into different selections relying on server, application or sensor name, for example.

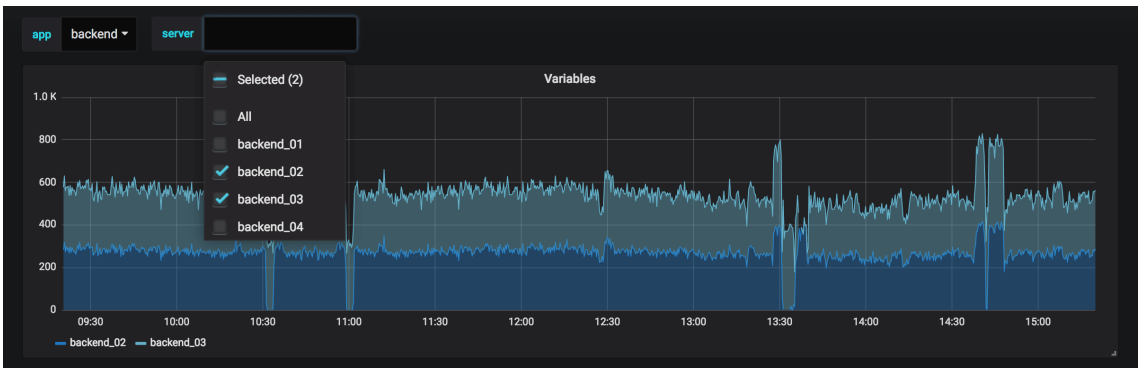

Figure 15 – Grafana Variables feature. Adapted from [63]

5 Refactoring VR2Market with Streams

VR2Market is a system to monitor FR's welfare state in firefighting scenarios by making use of a set of both environmental and physiological sensors. The current implementation of VR2Market:

- Relies on an integration solution that lacks cloud support and it is not easily extendable or scalable due to its original centralized design – any evolution, already identified as necessary implies a major refactoring
- Lacks a monitoring layer to measure the QoS system and/or support flexible and customizable analytics over incoming information.

VR2Market, given its design and domain, is prone to fit the typical streaming approach if we handle data from FR missions as an unbounded stream of events of both physiological and environment sensing measurements.

A combination of streaming and cloud-based monitoring solution could be a good option due to:

- Natural abstraction to approach different sources of information
- Proven and open solutions to support either streaming or monitoring solutions

5.1 Objectives

Our system main objectives are shared with the original VR2Market's online dashboard (VRCommander), i.e., to support the visualization and generation of alerts over the streams of data. On the one hand, data can become more valuable when properly displayed, on the other hand, alarms play an important role when it comes to giving instant feedback on upcoming hazardous situations.

In this scenario the system refactoring, besides maintaining existing functionalities should address some new features:

- Cloud deployment enabled
- Dashboard
    - Online Viewer
    - Support for instant and historical requests over data
- Online stream processing as a solution for
    - Abstraction to handle heterogeneous data sources
    - Support for different types of input sources

5.2 Needs & opportunities

In original dashboard implementation (VRCommander) there is no data persistence and data from event producers (VRUnit) is fed directly to the online visualization dashboard (VRCommander). For this reason, temporal queries were not supported and, consequently, it was only possible to visualize incoming information on-the-fly. To note that the data is persisted elsewhere but not on the online dashboard. Since collected data is associated with a timestamp during acquisition, the online dashboard historic feature can be introduced by assuring storage capability for incoming time series.

This feature constitutes one opportunity to extend VR-Commander regardless of the ability to perform mission offline review using VR-Mission reviewer.

Another core feature we aim to introduce into the system is the capability of trigger alerts over the incoming data. To perform this, it should be added a processing component to serve this need through some analytical tasks. A typical use case could be a situation in which the battery of a collector unit is running low, without awareness from the commander side. For being a measurement that typically does not require much attention as the others, it can go unnoticed. However, with the alerting feature, this kind of situations can be prevented, assuring the correct work of the system.

5.3 Proposed Architecture

This section gives an overall view of the proposed architecture which is illustrated in Figure 16. At a very high-level, the proposed data flow pipeline can be seen as a typical real-time analytic pipeline [19], that consists of four major categories: data collection, storage, analysis, and visualization.

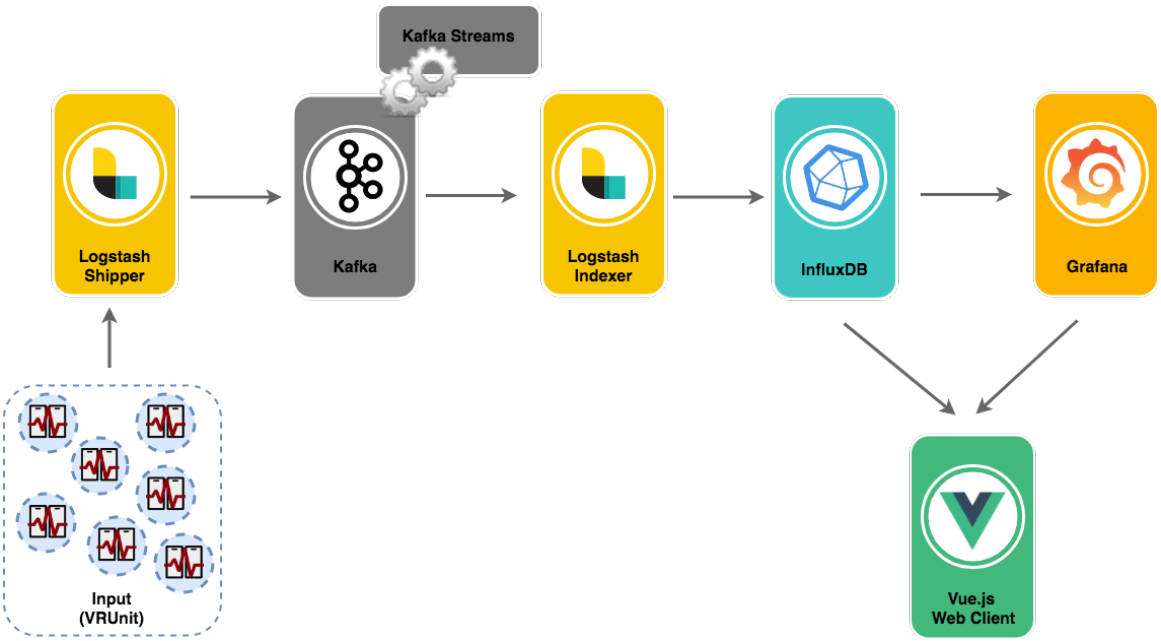

Figure 16 – PhisioStream Architecture

Data is collected from aggregators, VRUnit, through the push/pull collection agent, Logstash. This agent is included in two stages of our system, serving different purposes. The first one (shipper) is responsible to ingest data from the input source (VRUnit) immediately into Kafka, which makes it a producer.

Keeping the data in Kafka allowed us to introduce some processing power to the pipeline. Making use of the available API for stream processing, Kafka Streams, we designed a processing component which is used to set alerts for certain kind of events, such as exposure to high levels of CO or situations on which FR's heart rate reach hazardous levels. The outcome of this processing tasks is then reinserted into Kafka, although separated from the other types of data. As a matter of fact, Kafka's ability to split data among different topics proved to be useful when it comes to consuming it, which will be later explained.

From this point, we rely on the second instance of Logstash (indexer) to route data from Kafka to InfluxDB. Acting as a Kafka consumer, at this stage, Logstash does not work as a shipper but rather as an indexer of each message into the database. Events are indexed according to its corresponding sensor and with its associated timestamp, which relates to the acquisition time.

The symbiosis between this time series database and Grafana turns possible to deliver powerful dashboards and reports over the incoming data. When querying InfluxDB, Grafana is able to represent data through powerful and customizable graphs, making it easily readable by whom it may concern. With this tool, mission analysis can be performed in real time providing content that can be used to take actions and decisions, aiming for a better response to hazardous situations.

Finally, there was a need to establish a platform in order to deliver all the content in a structured and clean way so that it could be reviewed during missions. To accomplish this task, it was developed a web-based dashboard based on Vue.js[21]. Grabbing data from both InfluxDB and Grafana, it is possible to gather all the captured data from each operational and present it separately. Divided into three main sections this dashboard regards information related to physiological and environmental status along with potentially triggered alerts from the processing engine.

5.4 Monitoring Stack

Monitoring stacks like the one provided by Elastic, ELK, were designed to handle large amounts of data, typically for log monitoring purposes. However, due to their features, their use can be extended to other contexts. Given that messages in our system share a format similar to the typical log structure, it makes sense to explore the use of the stack in our approach.

Despite being built to work as an integrated solution, this set of tools can be split and serve different needs by working separately. As a matter of fact, the only tool we inherited from ELK stack was Logstash. Frequently mentioned as an ETL pipeline, Logstash added our architecture the ability to ship data while performing some transformation operations over it. This tool acts as a bridge between several components in our system due to ingesting and sinking data from and into different sources.

Elasticsearch proved to be an extremely powerful tool mainly for its unique feature of full-text search which makes it optimal for the purpose of log analysis. However, a great part of the functionality this tool offers was not required for our system, which led us to remove it from our architecture. In spite of storing data containing textual records, within our system scope, there is no need to perform queries by content. Instead, our data visualization relies on time which required us to store the measured values in time series. For this purpose, we chose InfluxDB as our storage engine proven to be more suitable to handle time series data, as one can notice from the benchmark study described in [64].

Consequently, we also had to drop Kibana from our implementation once it runs on top of Elasticsearch. This tight symbiosis, make it a primary choice when it comes to log analysis but ineffective when working with data sources other than Elasticsearch. In place of it, we chose Grafana which offers a rich variety of visualization types and allows to combine multiple data sources at the same dashboard. Consuming data from InfluxDB, made it possible to create widely customizable views over our system's data.

[21] https://vuejs.org/

5.5 Stream-based approach to address extensibility of the system

VR2Market can be seen as a sensor network [12] which is continuously collecting information from FR's in the field. Bearing this in mind, it makes sense to look at this system with an event-based approach since its purpose is to analyse and react to the sensing generated events.

In the refactored system to support multiple FR's monitoring, we followed a stream-based approach to create a pipeline capable of providing actionable insights over the information. For its need to handle data between multiple sources and consumers, messaging systems are at the core of this kind of architectures reason why we included Kafka in ours. As a matter of fact, using a message passing approach to support event streaming is the natural solution.

According to information provided in [17], due to the coupling of components in traditional architectures, performing changes in the system can create an undesirable impact on other components. Although the original system also used message passing technologies (RabbitMQ, Mosquitto[22] brokers), Kafka serves a different role in our implementation.

As referred earlier, Kafka supports message persistence and is able to manage concurrent access from consumers allowing the same event data to be made available to different processors/consumers simultaneously and at a different pace, in contrast to RabbitMQ and Mosquitto used in the original solution. By design, it also allows extending the system with new consumer/producers without performing changes to other modules, besides extending topics in Kafka and/or publishing their reference.

5.5.1 Refactor to streams

The past sections described some of the main advantages of establishing a stream-based architecture. However, there are some modifications which should be made in the current system so that we could follow this kind of approach.

One major goal of our system was the creation of a way to react, in form of alarms and alerts, to the incoming information gathered in each aggregator. For that reason, our architecture should follow some core aspects of a stream-based architecture, in order to add a stream processing component into our system.

As mentioned before, message-passing systems are at the core of this kind of architectures. Due to its unique features, discussed above, we decided to use Kafka as our messaging layer, which in fact proved to be convenient in two ways. In one hand, this distributed publish-subscribe messaging system is fast, scalable and durable which allows to broker high amounts of messages with extremely low latency, a requirement to assure one of our goals: FR's welfare monitoring in

[22] https://mosquitto.org/

real time. On the other hand, Apache Kafka provides a rich API, Kafka Streams, which enables to perform low-latency transformations on a stream of events.

By keeping the messages divided into different topics, it is easier to provide the information to different services since each of them only has to subscribe to the intended topic. Messages from each sensor are grouped in different topics so that different processing logic could be applied to them. Thereby, the event handler can subscribe one or multiple of these topics, consume them and apply some processing over the events. The resulting output from Kafka Streams is then inserted in a new topic containing the generated alerts.

We tried to design an architecture which could be adapted to different scenarios. So, we needed to choose an agent able to push data from different types of sources. To accomplish this, we used Logstash which allows one to push data from many sources, filter it through some transformations, and then pull it to a vast range of outputs.

The first instance of Logstash in our system is used to collect messages that arrive at RabbitMQ, derived from VRUnits. It is then responsible to route them into different topics in Kafka. Once each message corresponds to a different type of sensor it makes sense split them into different topics.

We found Logstash a useful component to deal with integration between components in our system. As it can receive data from multiple sources and route it to a vast range of outputs it can sometimes work as a plug element. In our architecture, Logstash plays the role of connecting event producers to Kafka and later to the storage component.

In order to provide a historical view of each mission, our system should provide a way to sink the information. Storing the information in a time series database makes possible to query it over time which proves to be crucial when it comes to giving end users a way to analyze past events in a mission.

Thus, Logstash works both as a shipper and an indexer, since it collects data from aggregators to Kafka and routes it from Kafka to InfluxDB. Since data is already ordered by sensor type, it is indexed from each topic to a corresponding measurement in InfluxDB. Thereby, this data can be queried from Grafana's side since it is already indexed in time series.

5.6 Semantics and messages

Once the pipeline was designed, we had to establish a system-wide messaging semantics. By doing this, we could assure the system internal integrity while enhancing its scalability. Maintaining a common structure among all messages flowing through the system it is easier to integrate new external components with our system.

VR2Market, in contrast, was designed with its own message format which was not standardized. Data was stored in files, in plain text. Each line of the file corresponded to a record while the different values, included in each record, were separated by commas. As a

matter of fact, this file format, most known as CSV is widely used to exchange tabular data since it is supported by many consumer, business and scientific applications [65]. However, besides of keeping a message format similar to CSV approach, VR2Market also had a code mapping, as follows:

```
248,2017-08-0910:42:02.109; 109,355921042020741; 150,BVAlb;151,1; 149,
9b24201d00d592910cd45c11440028a9; 253,40.70725077385496,-8.492235404560027;
118,10:41:54.725; 257,202.0; 153,gps
```

Figure 17 – Sample of GPS sensor measurement in CSV format

The example above represents a line of one file which corresponds to a single measure of the GPS sensor. From Figure 17, it is possible to infer that fields are separated from each other with a semicolon, while in each field there is a comma-separation between the code and its corresponding value.

This approach introduces some limitations to the system when it comes to adding new components to it. While internally there is a knowledge of this mapping, which turns the parsing of messages easier, whenever it is needed to extend the system new modules should be designed taking this constraint in consideration.

For this reason, we put some effort in unifying the system messages format so that integrations of new components could be made in an easier way. For this purpose, we chose JSON which mainly consists of a file format that uses human-readable text to exchange data objects.

```
{
  "bat": "64",
  "timestamp": "2018-04-30T19:21:11.145224194Z",
  "Altitude": "70",
  "devId": "f97561bba0a205b2",
  "Longitude": "-8.66",
  "Latitude": "40.63",
  "sensor": "GPS"
}
```

Figure 18 – Sample of GPS sensor measurement in JSON format

By using this notation of attribute-value pairs, messages became more human-readable, as one can notice from Figure 18, which can be useful for debugging purposes, for example. Another advantage introduced by this modification is the possibility of working with the data as JavaScript objects, which turned out to be useful later in the frontend development.

This message structure allowed us to establish data relationships in the system, which were crucial for organising information accordingly to its specifications. Therefore, messages can be related to each other both by context or by time. Once associated with a timestamp they can be ordered by time which is essential when following a stream-based approach. On the other hand,

messages can also be divided by context which can be made with regard to its source type or identifier. Each aggregator, i.e. VRUnit, has its own unique device identifier (devId) which is used in our system to separate messages accordingly to their origin. In addition, each message carries a field identifying the sensor used to retrieve the measurement which is also used to split messages into different groups. Sharing these two identifiers, there are six types of messages flowing in the system providing different information, as follows:

- Fremu message – provides measurements for altitude, carbon monoxide (CO), nitrogen dioxide (NO2), environment temperature (envTemp), atmospheric pressure (atmPressure)
- VJ message – comprises ECG timeseries, heart rate estimations (heartRate), body temperature (temperature)
- GPS message – when available, GPS from smartphone, provides altitude, latitude and longitude measurements
- External GPS message – deriving from external source GPS, provides the same measurements as the included in smartphone
- Helmet message – provides measurements for altitude, atmospheric pressure, environment temperature, carbon monoxide, nitrogen dioxide, humidity and luminosity
- Alert message – relates to generated alerts in the system distinctly identified (alertType)

Each one of these messages enters in the system through Logstash and from here is routed to Kafka. Since we aim to have a system which can provide easy integration with new modules, it was created a topic for each type of message. Hence, one can start receiving data just by connecting to Kafka and subscribing to the intended topic.

5.7 Deployment

One requirement of the system was turning it into a cloud-enabled solution. Following a container-based approach was a suitable way of achieving this functionality. Containers allow creating consistent environments, isolated from other applications, from development to production stage. Besides, it can package up an application with all it needs, namely software libraries and dependencies.

From the developer's perspective, environment consistency can be assured regardless of where the application is deployed. Actually, containers are able to run virtually anywhere, regardless of any customized settings of a particular machine could have which could differ from the one where the application was created. Containers share their operating system, running as isolated processes

regardless of the host operating system. In contrast with the virtual machines approach, where the entire hardware stack is virtualized, this virtualization at the OS-level provides a sandboxed view of the OS logically isolated from other applications.

At the heart of container-based architectures lies Docker[23], an open-source platform built to automate the deployment of applications within containers. Within our system scope, the main advantage of adopting Docker containers approach is the fact that it provides the ability to deploy applications anywhere, including cloud. Since containers are independent of the underlying OS and infrastructure, they can be quickly moved with minor changes. Additionally, relevant cloud computing providers such as Amazon Web Services (AWS)[24], Microsoft Azure[25] or Google Compute Platform (GCP)[26] offer support to Docker cloud deployment.

This technology carries plenty other benefits that increased our developing productivity, namely the mentioned environment parity and its configuration customization. Dockerfile, a text file which contains all the required commands to build a given image, can specify additional resources to be loaded or invoked depending on the needs of the application. For this reason, containers can be kept immutable while Dockerfile performs the needed customizations at load time. On the other hand, by establishing a consistent development environment it reduced the amount of time wasted in system configuration or debugging and increased the amount of time available for development.

We chose to use Docker compose, a simple deployment orchestration script that through YAML configuration files deploys a multi-docker environment with related resource configurations, namely network and filesystem. For debugging purposes, we split the stack into two compose files the first one including the data collecting and processing layer (Logstash and Kafka containers) and the second one regarding the storage and visualization layer (Logstash, InfluxDB, and Grafana containers). Since the applications we were working with were already included in official repositories in Docker Hub[27], we simply had to pull their images from there.

```
version: '2'
services:
  ...
  kafka:
    image: wurstmeister/kafka:0.11.0.1
    ports:
      - "9092:9092"
```

[23] https://www.docker.com

[24] https://aws.amazon.com

[25] https://azure.microsoft.com/

[26] https://cloud.google.com/

[27] https://hub.docker.com/

```
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    links:
      - zookeeper
    depends_on:
      - zookeeper
    container_name: kafka
    volumes:
      - ./kafka:/var/log
    command: java -jar kafkaStreamsApp-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Snippet 1: Sample of Docker Compose file

Docker gives the possibility to declare volumes on the compose file that can be reused across multiple services. In order to include, mainly, configuration files and data related to each container, we created some volumes. This feature also proves to be useful to insert the produced JAR files (regarding Kafka Streams application) into Kafka container. We created a network so that containers could be linked and establish communication with each other (Snippet 2).

Once we were using some plugins which were not included in Logstash's official image (namely output and input plugins from Logstash to Kafka and InfluxDB), we created some Dockerfiles which are responsible to pre-install them before containers initialisation (Snippet 3). This functionality was also used to create the InfluxDB database. Thus, in case of launching the containers for the first time it is assured that all the components are already included in each container.

Consequently, in order to deploy the whole system, it is only required to execute both the Python modules (messages parsers) and launch compose files.

```
networks:
  default:
    external:
      name: kafkalogstashdocker_default
```

Snippet 2: Sample of Docker Compose file which relates to the network creation

```
FROM docker.elastic.co/logstash/logstash:5.6.3

RUN bin/logstash-plugin install logstash-output-influxdb && \
    bin/logstash-plugin install logstash-input-kafka
```

Snippet 3: Dockerfile which is responsible for pre-installing the required Logstash plugins

5.8 PhisioStream Implementation

Our work was based on the refactoring of the VR2Market system real-time component, the VRCommander. One of our initial objectives was to use cloud-enabled off the shelf solutions to implement the new VR2Market data pipeline to support the refactored dashboard while being integrated with the actual. The main concern was that implementation options on the refactored

VR2Market data pipeline would not compromise the system extensibility and scalability while proving the existing and new features.

Above, we discussed some high level and conceptual decision. Here, it is described in detail how we managed to develop each module of our system. Taking into consideration the system's requirement to handle different types of data sources from the ones VR2Market provides, we start by giving some context in this integration.

5.8.1 From sources to web UI

Before diving into the system implementation details, it was essential to assure the integration with the existing VR2Market system. For that reason, a translation module was needed to convert the structure of the incoming messages from VRUnits into the new structure of our new system. This was implemented as Python scripts that carry out the message parsing tasks and translation through a RabbitMQ client used to consume messages from the existing VR2Market message exchange. There are different Python modules for each type of sensor that convert messages to JSON objects which are, then, routed to a new RabbitMQ queue. At this stage, we assure that every message which arrives at our system is already identified by its sourcing sensor. As a matter of fact, the previous implementation did not cover this aspect which was preventing us to split messages by context.

To add some extensibility to our system, we selected Logstash as the data collector. For being a plugin-based event forwarder, Logstash allows one to ingest data from multiple sources and route it to different types of output. Whenever data input or output changes the system could be rearranged just by changing the plugins in use.

Logstash is configured through a config file, illustrated in Figure 19 that specifies which plugins to use along with its related settings. The file is divided into three separate sections for each type of plugin, as follows:

```
input {
  ...
}

filter {
  ...
}

output {
  ...
}
```

Figure 19 – Sample of a Logstash configuration file

As our system data source was based in a RabbitMQ queue, we opted to use Logstash's RabbitMQ input plugin for accessing the queue where converted messages are added.

This first Logstash instance is responsible to collect converted messages from RabbitMQ to Kafka, routing them according to criteria based on message type and content. In our case different topics were created for each type of sensor, therefore, messages should be sorted from this field.

Logstash is able to perform many transformation and parsing operations in data thanks to its many available filter plugins. Since messages reach Logstash in JSON format, we used the JSON filter plugin to parse them. By using it, we can access each field of the message which helps us to understand its corresponding sensor type. With conditionals, we inspect the sensor field and place a tag to the message with its sensor type. These tags are added by a mutate plugin which appends them to the message tags list, which is just a regular Logstash field.

For routing messages to Kafka, we used conditionals in Logstash which, based on the sensor type included in the message tags field, send them to their corresponding Kafka topic. The reason why we use tags to perform this filtering, instead of just comparing sensor fields before routing the messages, is because message fields comparison is not allowed in the Logstash output field.

This tag comparison mechanism is used once again by the time data need to be indexed from each topic to its corresponding InfluxDB measurements. The second instance of Logstash connects to Kafka, this time using the input plugin, and is responsible to subscribe to the previously created topics. Since the added tags, related with the sensor type, remain in the messages, now, when filtering them we only need to do it relative to the alert type, in case of messages derive from the alerts topic. Thus, before being indexed, messages are subjected to the same conditionals already used to send them to Kafka. With respect to the way that data is routed to InfluxDB there are some aspects that should be taken into consideration.

InfluxDB configuration

```
if "VJ" in [tags] {
    influxdb {
      send_as_tags => ["devId"]
      data_points => {
        "Sensor" => "%{[sensor]}"
        "devId" => "%{[devId]}"
        "heartRate" => "%{[hr]}"
        "temperature" => "%{[temp]}"
        "battery" => "%{[bat]}"
        "time" => "%{[timestamp]}"
      }
      coerce_values => {
        "heartRate" => "integer"
        "temperature" => "float"
        "battery" => "integer"
      }
      db => "VRUnit"
      measurement => "vj"
      host => "influxdb"
      allow_time_override => true
    }
  }
```

Snippet 4: Configuration required for the indexing of Vital Jacket measured values into InfluxDB

Snippet 4 shows an example of how we configured our Logstash so that it could output data to InfluxDB in a proper way. First of all, we created our database “VRUnit” which of composed by five different measurements: “vj”, “fremu”, “gps”, “gpsExt” and “alerts”.

Tags in InfluxDB are used to store commonly-queried metadata or to perform “Group By” clauses. The “Group By” clause does not work with fields, reason why we defined the device ID (“devId”) variable as a tag. By performing our queries with aggregations of data corresponding to this specific tag, we can assure the information sorting by collector device id. Otherwise, we would not be able to create visualizations regarding information of a single user.

In “data_points”, we set the fields of each measurement which need to be converted to the appropriate type before posting. This operation is performed with “coerce_values”, where we define the type of each field either as an integer or float. The ones which are not defined are sent as string format. By setting “allow_time_override” as true, we allow the override of the time column. When not defined, any column with a name of “time” will be, by default, ignored which will lead to time being replaced with the value of “@timestamp” (corresponding to Logstash processing time) instead of with the device timestamp.

Setting up Grafana

Since the system collects data from multiple users, in order to establish a data visualization of a single one, we should include some distinctive identifier in the InfluxDB query. As mentioned before, for this purpose, we used the device ID which was indexed as a tag.

However, keeping this value hard-coded in queries was not possible due to the fact that new devices could later be attached to the system. Hence, queries should be made in a dynamic way. For this purpose, Grafana provides the templating feature. The figure below shows how we made use of this feature to tackle our dashboard needs.

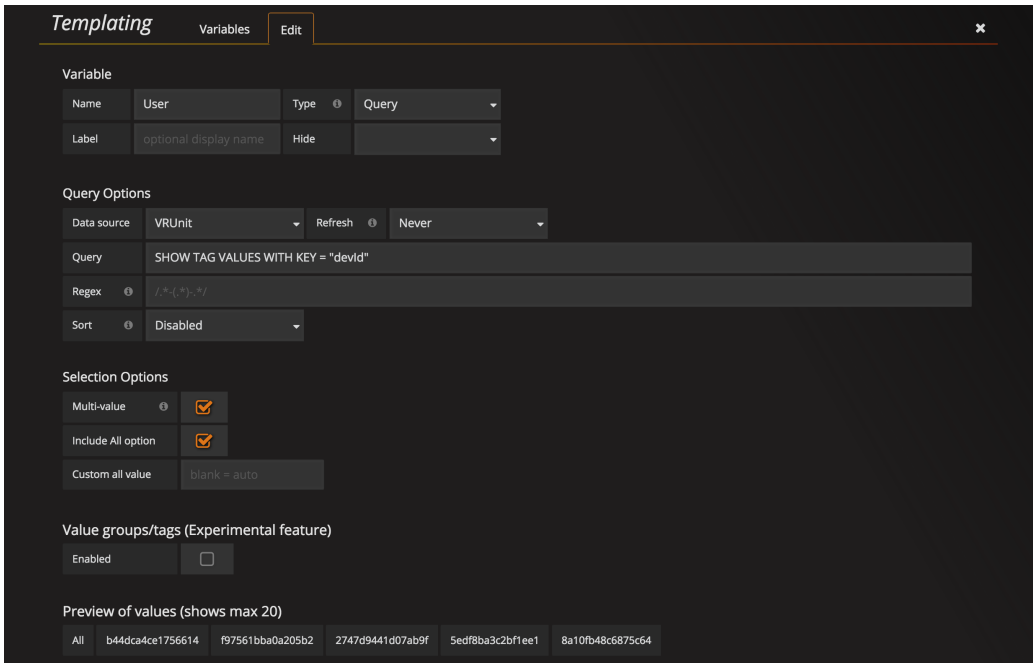

Figure 20 – Variable setting

From Figure 20 it is possible to understand how we created the variable User, of type query, which aims to return a list of all device ID's included in "VRUnit" database. Each device ID is shown as a dropdown select box (Figure 21) which make it easy to change the displayed data. Since we checked the options for "Multi-value" and "Include All option" it is possible to display data from one or multiple users, at the same time. Later, this variable can be included in queries from Grafana to InfluxDB to keep a relation between returned values and device ID (Figure 22).

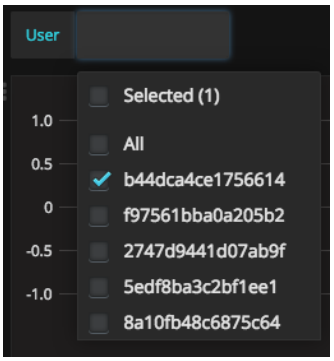

Figure 21 – Variables dropdown, regarding values of each device ID

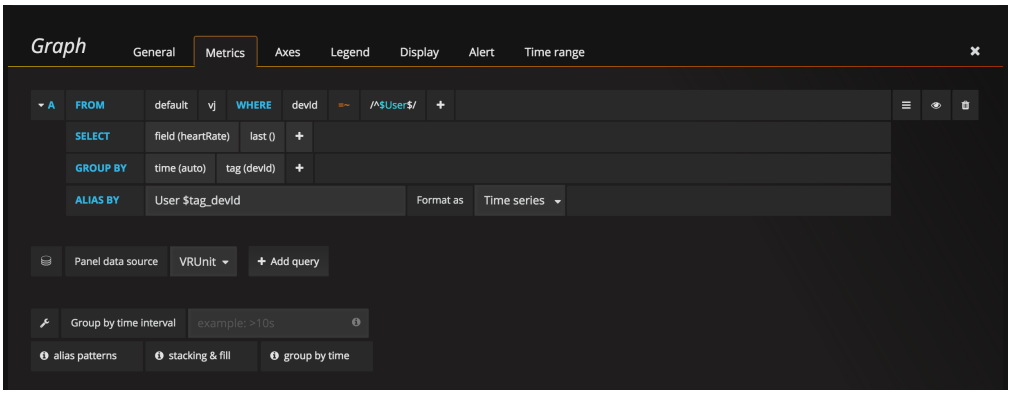

Figure 22 – Sample of an InfluxDB query from Grafana, relating to Vital Jacket measurement and including restriction by device ID.

Grafana made it possible to add a new feature to our system which was the ability to relate, at every moment, the measured data with each FR position. Since Grafana keeps the same time for all panels in a dashboard, when moving over a graph, e.g., heart rate level graph, the position of the user for each instant is shown as a dot in the map. In order to do that we included the "grafana-trackmap-panel" plugin and configured the query that feeds it (Figure 23).

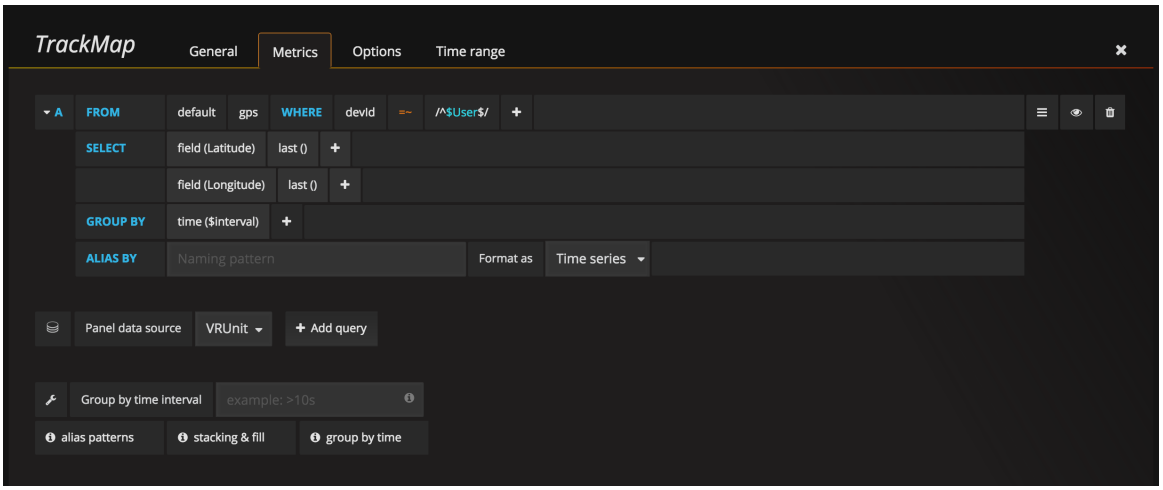

Figure 23 - Sample of an InfluxDB query from Grafana, relating to GPS measurement

VRCommander Refactoring

After reaching to a fully functional data pipeline, the gathered information should be displayed in a structured way which could be easily interpreted by end-users. Furthermore, VRCommander, the VR2Market frontend layer responsible for representing measured data in real-time, needed some refactoring. We focused mainly on enhancing the existing features of the platform while adding new features such as authentication and historical view over data as currently

VRCommander only displays measured values at each instant.

We developed an interface by using the typical web development technologies such as HTML, CSS, and JS. However, since we put more focus on the arrangement of the application structure, we chose a JS framework to develop it instead of just using pure JS. This way, we could experience less complexity level while taking advantage of many of its benefits such as code reuse and faster performance. With frameworks, web page content updates are only written when it is necessary, i.e., only when changes occur.

There are plenty of available JS frameworks and among the most popular ones there is Vue.js, React[28] and AngularJS[29]. Due to its pointed friendly learning curve, we chose Vue.js as our developing framework.

The dashboard implementation is divided in two layers: backend and frontend. Backend layer comprises the authentication mechanism while the frontend is responsible for displaying the intended data. It is fully responsive as it is implemented with Bootstrap 4.0.

Backend Layer

We implemented a token-based authentication using Passport.js[30] which relies on a signed token that is sent to the server on each request. This Node.js[31] authentication middleware has different steps, as follows:

- Authentication via passport through its strategies
- User serialization – find or create a new user in the database, MongoDB[32] in our case
- Token generation - JWT
- Send everything back to the user

Users registration relies on three parameters: email, password and its member ID which is, in fact, its corresponding device ID. With this authentication mechanism, we can manage the information retrieval based on the logged user by being them member ID's the distinctive factor in sessions. After logged in, the user is prompted only with related information and corresponding team while its member ID gets stored at Vuex[33]. This management library allows holding the application state in a reactive way.

---

[28] https://reactjs.org/

[29] https://angularjs.org/

[30] http://www.passportjs.org/

[31] https://nodejs.org/en/

[32] https://www.mongodb.com/

[33] https://vuex.vuejs.org/guide/

Frontend Layer

The frontend layer of our web application consumes data both from Grafana and InfluxDB and is composed of six main components:

- VitalJacket
- Fremu
- Helmet
- Weather Station
- Team Location
- Notifications

Since data need to be related with the user logged in, we implemented a method to retrieve the device ID stored in Vuex ("*getDevId()*"), as shown in Figure 24.


```
getHr () {
 const influx = new Influx.InfluxDB({
   host: 'localhost',
   database: 'VRUnit'
 })
 influx.query(`SELECT heartRate FROM vj WHERE devId=${Influx.escape.stringLit(this.devId)}
ORDER BY DESC LIMIT 1`).then(rows => {
   rows.forEach(row => {
     this.heartRate = row.heartRate
   })
  })
},
getTemp () {
 ...
},
getBat () {
 ...
},
getDevId () {
 this.teamId = this.$store.getters['auth/currentUser'].teamId
}
```

Figure 24 – Dashboard upper panel which retrieves information from the current logged user and following methods which provide the information. (*getTemp()* and *getBat()* code is omitted for sharing the same logic as *getHr()*)

Every component has an area on top of the page which displays instant values of user heart rate level, surrounding environment temperature and aggregator device battery level. Each one of these values is obtained by three methods we implemented that query InfluxDB for the values according to the logged in device ID (Figure 24). Queries return the last value of the measurement and are executed at every ten seconds through updated hook (Figure 25).

```
updated () {
  this.$refs.circleProgress.$data.value = parseInt(this.bat)
  setTimeout(function () {
    this.getHr()
    this.getTemp()
    this.getBat()
  }.bind(this), 10000)
},
```

Figure 25 – Method from Vue lifecycle hooks, responsible for updating DOM when data changes occur. For our purpose, it updates information retrieval methods

Components are subdivided into tabs related with the different measurements of each type of sensor. Each tab is formed by two graphs, one displaying the corresponding measurement values over time, and the other displaying the user's position. Tabs are mapped with an id used to select which Grafana panel to share.

As one can notice, Grafana definitely plays an important role in our dashboard, as most of the information we display comes from it. It provides different ways of sharing a dashboard or a specific panel, being the embed panel the one we are using. This feature enables to generate an iframe that can be included in any web page just by using the provided HTML code (Figure 26).

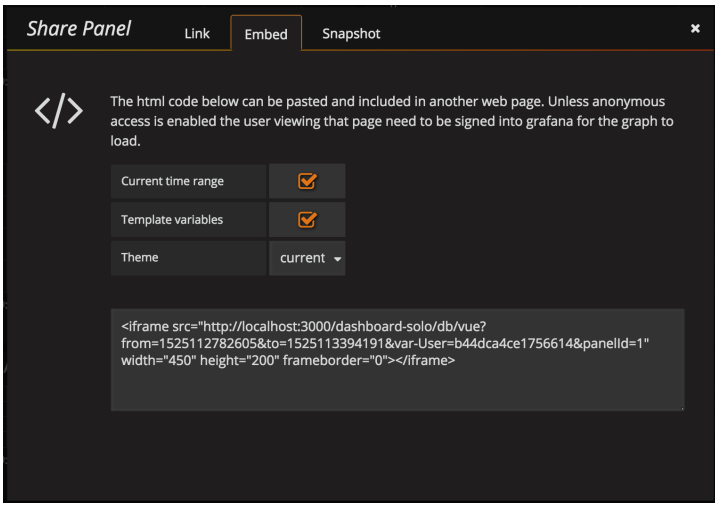

Figure 26 – Grafana sharing modes.

Despite being a great feature, unfortunately, there are some limitations when sharing a panel this way. Since iframes are independent of each other, they are not time synced which results in the loss of some Grafana features. On one hand, we noticed that, for this reason, graphs did not show feedback when other is hovered. On the other hand, since time range could not be imported we should implement our own (Figure 27) and find a way of manipulating the iframe in order to change the values.

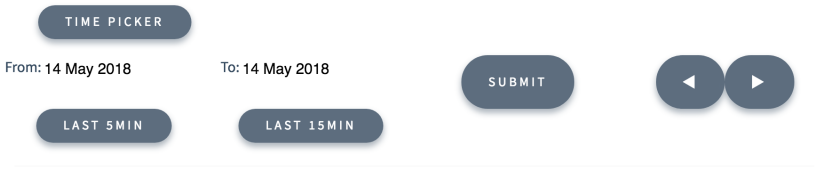

Figure 27 – Custom time picker to interact between Vue and Grafana.

Since Grafana panels are ordered with id's, we created a method which returns the id of the currently selected tab. This way, we could establish a way to import the desired panel to each tab.

We implemented the method "*inFram()*" which takes the time range as parameter, with default values of "now-5m" and "now", creating a visualization of data from the elapsed five minutes to the current time. This method gets the HTML element used to display the Grafana panel and change its content by providing a different iframe. Thereby, we can choose which panel to share by providing values for time range, device id, to select the user, and finally the tab id which refers to the panel id to import. Whenever time values are changed through time picker, this method is called again, iframe source URL is changed and the element content is reloaded.

```
inFram (fromTime, toTime) {
  var devId = this.$store.getters['auth/currentUser'].teamId
  document.getElementById('tCont').setAttribute('style', 'height: 650px')
  document.getElementById('tCont').innerHTML = '<iframe src="http://localhost:3000/
    dashboard-solo/db/vue?from='+fromTime+ '&theme=light&to='
    +toTime+ '&var-User=' +devId+ '&panelId=' +this.currentId+'"width="100%"
    height="200" frameborder="0"></iframe><iframe src="http://localhost:3000/
    dashboard-solo/db/vue?panelId=8&tab=general&from=' +fromTime+
    '&theme=light&to=' +toTime+ '&var-User=' +devId+ '" width="100%" height="400"
    frameborder="0"></iframe>'
},
```

Figure 28 – Method responsible to adjust Grafana's shared iframe.

Since we implemented the dashboard with Vue.js we could reuse many parts of the implemented code. As we can see in the image below, by keeping this code as Vue components we simply have to call them in the HTML side, which otherwise would force us to replicate it.

Team Monitoring

As mentioned before, by using the grafana-trackmap plugin, we can provide, in each instant, the user's position. However, due to the way it is implemented it does not offer a way of displaying multiple users at the same time. The plugin just allows displaying one marker which makes the position to be overwritten with other users' data.

For this reason, we used leaflet real-time plugin to achieve this feature. This plugin allows displaying several markers in a leaflet map by receiving them as a set of GeoJSON[34] points. In

[34] http://geojson.org/

fact, leaflet supports all of the GeoJSON types, though *Features* and *FeatureCollections* are more appropriate as they enable to describe features with a set of properties.

To be aware of when new features are added, old ones are removed and which of them were updated, leaflet real-time relies on a unique feature, by using an id. Typically, this can be achieved by using one of the feature's properties.

Since we did not have the points in GeoJSON format, they are converted after being retrieved from InfluxDB. We created an array on which we append the last ten entries of the database and, then, by accessing each index of the array we create the GeoJSON point and return it to leaflet realtime.

5.8.2 Processing Engine - Kafka Streams

As mentioned before, in order to generate alerts over the incoming information we had to add some processing mechanism to our system. To fulfil this requirement, we chose Kafka Streams. This lightweight Java library allows performing stream processing on top of Apache Kafka topics. The outcome of this processing tasks can easily be published to other topics or to external destinations.

For the purpose of our system, we aimed to generate some alerts whenever FR's heart rate level reached certain potentially hazardous values. We established a range of values which relates to the relevance of each alert and is defined as follows:

- Yellow Alert – values of heart rate comprehended between 125 and 144
- Orange Alert – values of heart rate comprehended between 145 and 159
- Red Alert – values of heart rate comprehended higher or equal to 160

Since we already had messages split into different topics we could stream them directly from vital jacket's corresponding topic, using a KStream.

```
    public class VjStream {
        static public class VjMessage {
            public String devId;
            public String sensor;
            public String bat;
            public String temp;
            public long hr;
        }
        private static final String APP_ID = "vj-streaming-analysis-app";

        public static final String MAIN_TOPIC = "vj_topic";

        public static void main(String[] args) {
            // Defining vjMessageSerde
            Map < String, Object > serdeProps = new HashMap < > ();
            final Serializer <VjMessage> vjMessageSerializer = new JsonPOJOSerializer <>();
            serdeProps.put("JsonPOJOClass", VjMessage.class);
            vjMessageSerializer.configure(serdeProps, false);

            final Deserializer <VjMessage> vjMessageDeserializer = new JsonPOJODeserializer
< > ();
            serdeProps.put("JsonPOJOClass", VjMessage.class);
```

```
            vjMessageDeserializer.configure(serdeProps, false);
            final Serde <VjMessage> vjMessageSerde = Serdes.serdeFrom(vjMessageSerializer,
vjMessageDeserializer);

            // Building Kafka Streams Model
            KStreamBuilder kStreamBuilder = new KStreamBuilder();

            //Start Sorting Job
            KStream<String, VjMessage> source = kStreamBuilder.stream(stringSerde,
vjMessageSerde, MAIN_TOPIC);
```

Snippet 5: Sample of Kafka Streams application code

As messages come in form of JSON objects, in order to access data, we had to deserialize them. We created vjMessageSerde which is basically a Serde that carries a serializer and deserializer based on JSONPOJODeserializer and JSONPOJOSerializer, using the Jackson library. So that we could access each field we created a class for the message which gathers all the fields included in the JSON object.

We managed to sort the messages into three different topics related to the above-mentioned range, using the branch method (Snippet 6). For each of these topics, we created a KStream so that we could, then, group the messages by its unique identifier, the device id.

```
Predicate<String, VjMessage> yellow = (k, vj) ->
     (int)vj.hr    > 125         &&    (int)vj.hr    <= 144;

Predicate<String, VjMessage> orange = (k, vj) ->
     (int)vj.hr    > 145    &&    (int)vj.hr    <= 159;

Predicate<String, VjMessage> red = (k, vj) ->
     (int)vj.hr    >= 160;

// Branching processor
KStream<String, VjMessage>[] values = source.branch(yellow,orange,red);
values[0].to(stringSerde, vjMessageSerde, "yellow");
values[1].to(stringSerde, vjMessageSerde, "orange");
values[2].to(stringSerde, vjMessageSerde, "red");
```

Snippet 6: Sample of Kafka Streams application code (Branching Processor)

As a matter of fact, these topics are formed by alerts, since their content are messages in which heart rate level exceeded the limits imposed by predicates. For this reason, we just needed to count the number of incoming messages in the KStream over a certain period of time.

Since we aimed to perform counts in defined intervals of time, we needed an immutable window. Once with tumbling-windows, records belong to one and only one window, we established a five-minute tumbling-window in which the count of alerts is computed (Snippet 7).

```
KStream<String, VjMessage> vjStream_r =
kStreamBuilder.stream(stringSerde, vjMessageSerde, "red");

KTable<Windowed<String>,Long> numAlert_r = vjStream_r
               .map((k, vj) -> new KeyValue<>(k, vj))
               .groupBy((k, vj) -> vj.devId,stringSerde, vjMessageSerde)
               .count(TimeWindows.of(TimeUnit.MINUTES.toMillis(10)))
              .filter((windowedVal, count) -> count >= 3);
               ;
```

Snippet 7: Sample of Kafka Streams application code (Time Window)

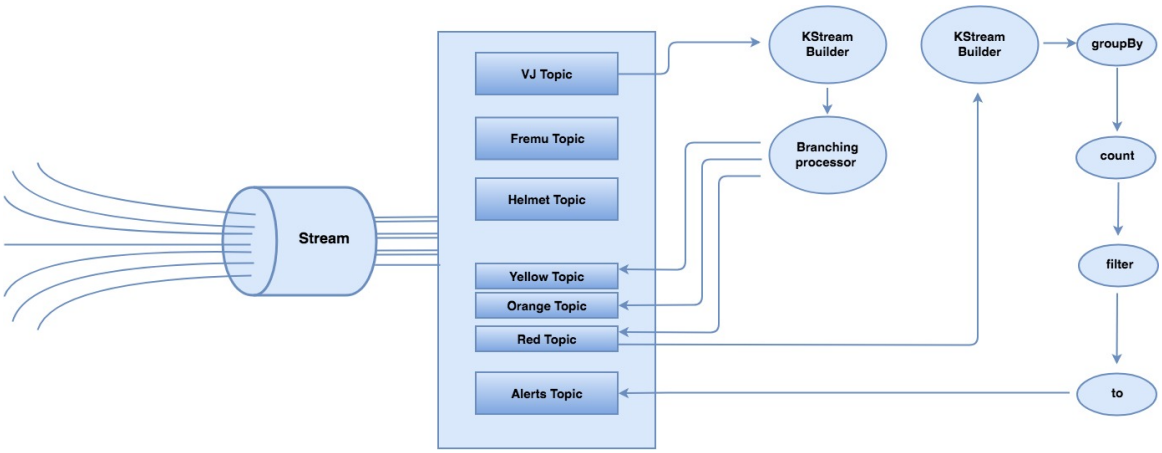

Figure 29 – Kafka Streams application sample diagram illustrates the entire flow of a message in the processing engine, related with a red alert. The same logic is applied to the remain types of alerts.

Thereby, we already had our alerting system built, however, due to potential system failures such as sensor reading errors, some of these alerts could be related to false positives. In order to minimize this, we make sure that we only sink alerts to the main topic, the alerts list, when they occur at least three times in a row, by applying a filter to the KTable.

The alerts topic contains all types of system generated alerts, reason why we established a code to identify each of them, as follows:

- Alert 001: Heart Rate Level Over 125
- Alert 002: Heart Rate Level Over 145
- Alert 003: Heart Rate Level Over 160

6 Evaluations

One of the objectives of the current work was to refactor the VR2Market system with special focus on the online monitoring dashboard UI – the VR-Commander. Our main concern was to ensure that the existing features were also present in the refactored version. This implied two main requirements:

- Compare UI visual features – map and verify existing features in both of them
- Ensure that the new version is operational equivalent to the old version - verify if new VR-Commander is operational compatible with existing solution and provides the same information

6.1 Visual features

We carried out an evaluation of our implementation keeping the old version as a reference (Figure 30). The purpose of this comparison was to verify if the existing features were still available in the new implementation while pointing out the new ones added by our system.

The current VRCommander UI is designed to display information related to FR. As one can infer from Figure 30, it is composed by a single view where are shown instant values for both physiological and surrounding environment data related with connected team members. From the navigation bar, it is possible to access the settings menu (change connection parameters) and choose which team to be displayed from a given list of the current ones which are active. The focus of this UI is to show the current positions of each team member by displaying markers on a map for each one of them. When clicking over one of these markers, a popup window is shown displaying an instant status of the selected member-related data (physiological and environmental).

The refactoring work introduced not only some aesthetic modifications but also changed in some way the system's operability.

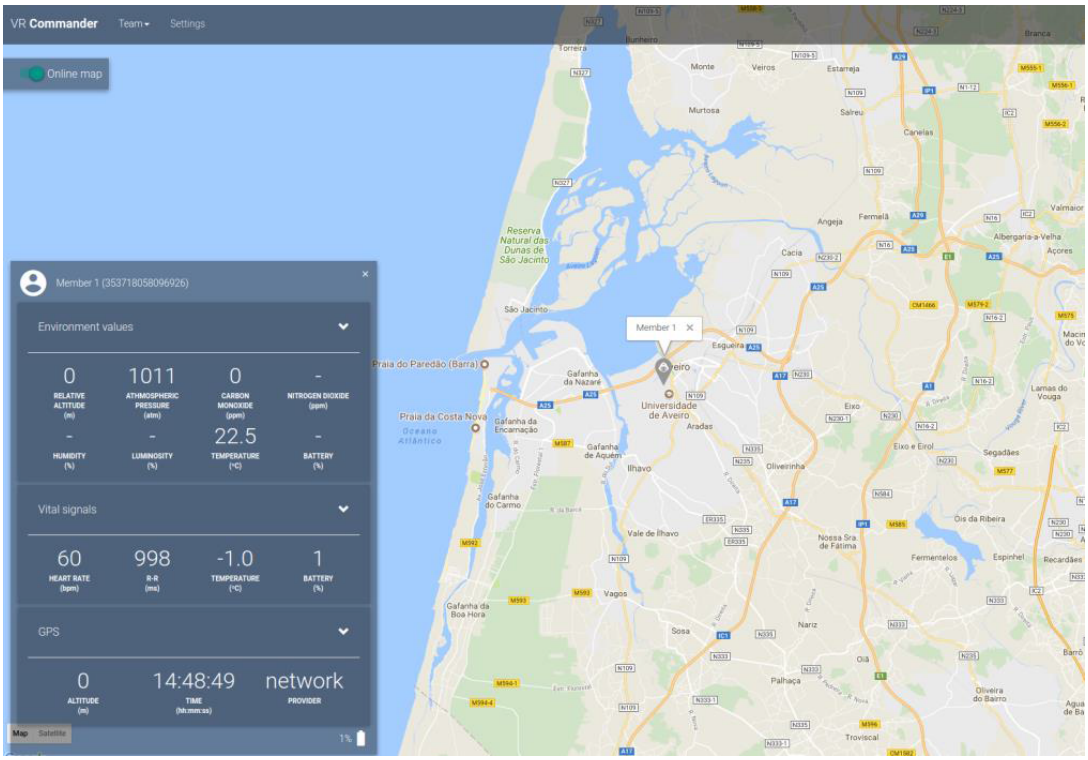

Figure 30 – Current VRCommander UI

In contrast with the old version, our implementation is formed by two different ways of performing online mission analysis. From our UI, it is possible to monitor a given user in detail while in Grafana it is established a global view of the team. Thus, the intention of our authentication module (Figure 31) is not to create an access control layer to the dashboard. Instead, it is the way to establish a different session for each member. For this reason, after logged in, a user can whether access to data related to a particular member or move to Grafana in order to have an overall view of all team members.

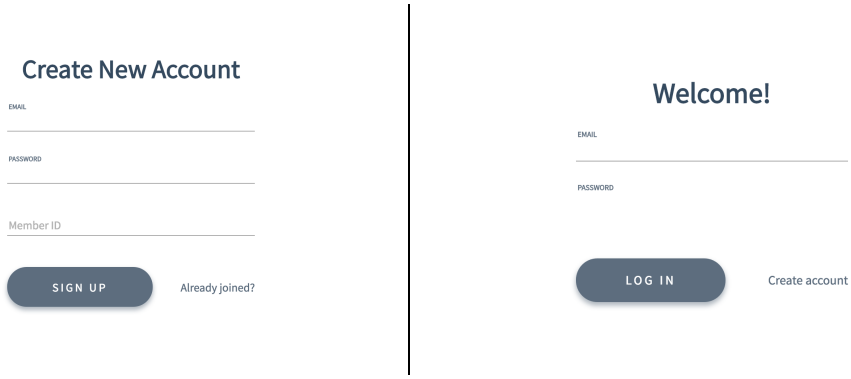

Figure 31 – Refactored UI – Authentication mechanism

Instead of gathering all the data in the same window, our UI prompts the user with different

sections, divided by content nature, which can be accessed through a side navigation menu. Some values considered to be more relevant, such as the surrounding environment temperature and the FR's heart rate level, are displayed in every section in the upper side of the window, as shown in Figure 32. Each section refers to a different sensor and, similarly to the old version, it is possible to relate the measured values with FR's location. However, instead of an instant view of these values, it is now possible to analyse them over time.

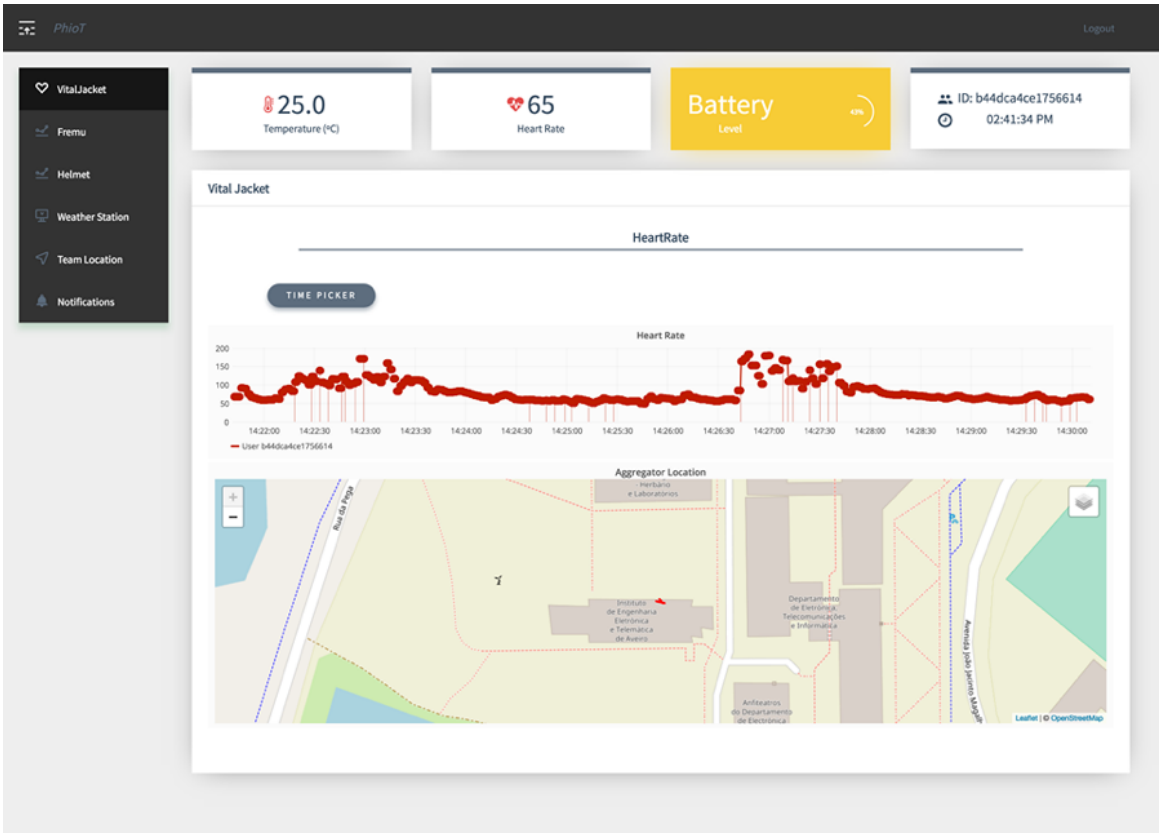

Figure 32 – Refactored UI – Logged in view

As mentioned before, to establish an overall view of all team members it is possible to access Grafana from our UI. Through "Team location" section, illustrated in Figure 33, it is possible to view each member's position. When clicking over a displayed marker the user is prompted with a popup window which indicates the correspondent device ID, measured coordinates and provides a direct link to a new window with Grafana's view (Figure 35).

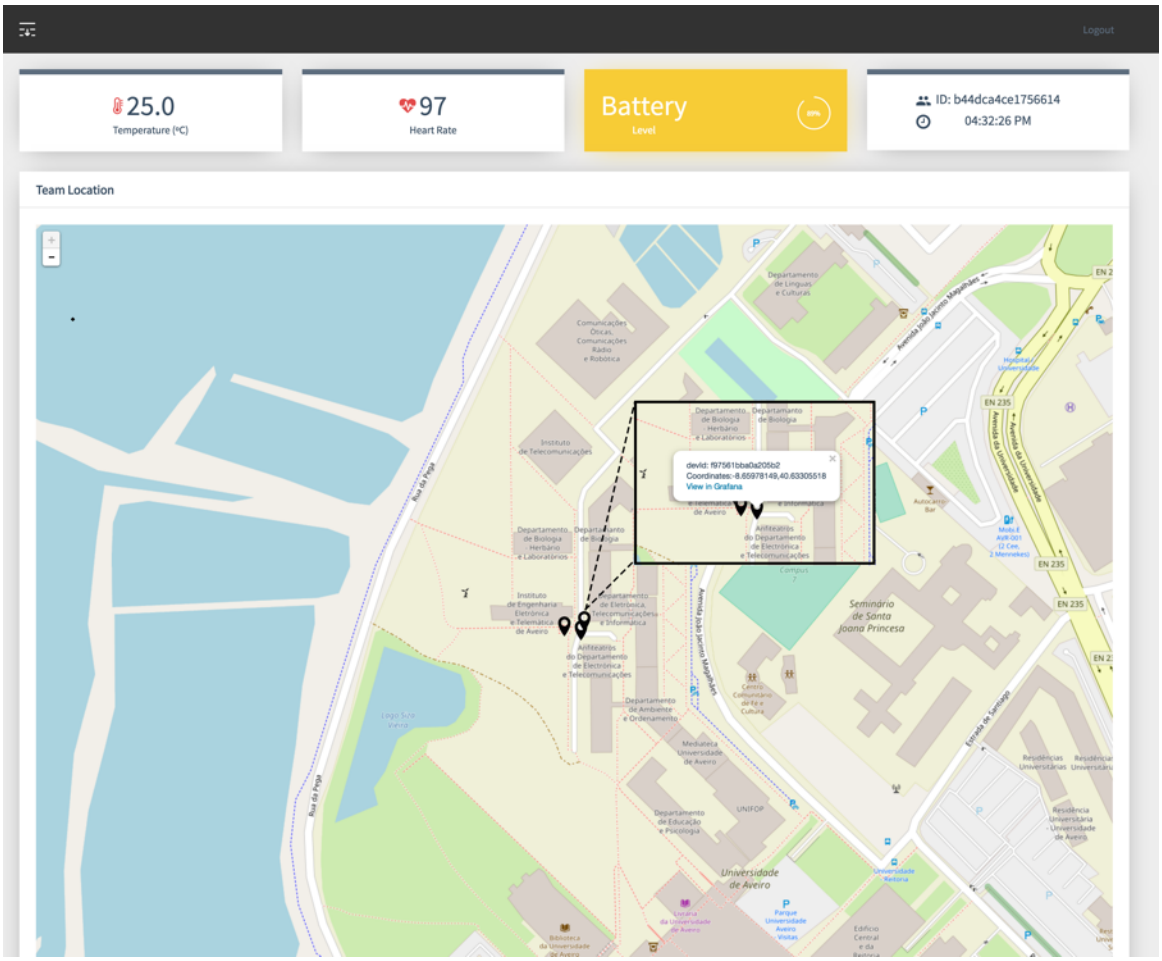

Figure 33 – Web client view of team members positions. When clicking over a team member, it is possible to access to related information (as illustrated in overlay on main view)

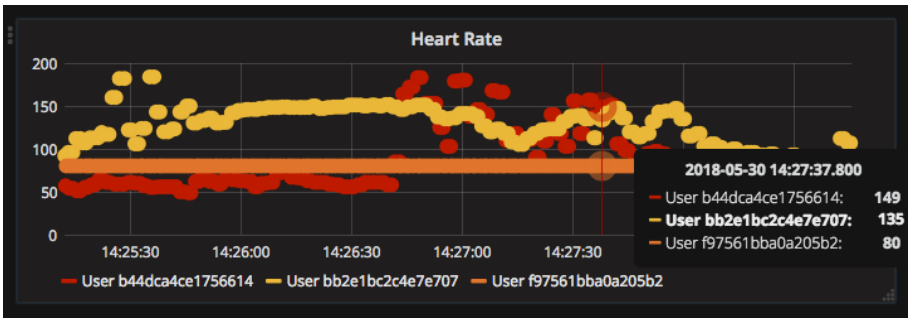

Figure 34 - Sample of Vital Jacket visualisation in Grafana. Each line concerns with a different user, each one carrying a VJ sensor. Contrarily to the others, orange line does not show variations over time since this one relates to simulated data.

Figure 35 - Grafana global team view.

Using Grafana-based VR Commander UI it is possible to perform comparisons between multiple members retrieved information, as shown in Figure 34. This feature can become useful, for example, to manage FR physical activity.

Our system's alerting feature is included in "Notifications" section where the user is regarded with a table containing the last triggered alarms.

6.2 PhisioStream Integration with VR2Market

In our VRCommander refactoring, regardless of the new features, we had to ensure that past features (or equivalent) were also available. As major changes were made to data flows from VRUnit to VRCommander, some verification was performed. The translation components ensure that our implementation supports not only the past generation of VRUnit but also the new one, and its integration with the VR2Market occurred without compromising its functionality or performance.

In order to establish a proof of concept, we performed some field experiments trying to simulate situations as close to real scenarios as we could. Despite not being possible to run experiments during fire scenarios, some of the conditions experienced for the FR could be replicated, namely physiological reactions.

The following trial was performed with three volunteers which carried, each one, an acquisition set composed by one data collector and FREMU and Vital Jacket sensors, included in their clothing (Figure 37). Data collection relied on a new generation of VRUnit which provided measurements of surrounding environment, participants' vital signals, and their location, at each instant.

The experiment took place in the campus of Universidade de Aveiro in a predefined route, as shown in Figure 36:

Figure 36 – Experiment route

The whole experiment lasted forty-five minutes since the route took fifteen minutes to

cover and was repeated three times. Our intention in repeating the path was to vary the levels of physiological measurements. Since we could not change the surrounding environment, FREMU ended up presenting a homogeneous range of values, however, we could vary the heart rate levels simply by increasing the intensity of participants' physical activity. For this purpose, we indicated the participants to walk calmly in the first lap, to jog in the second lap and, finally, we asked them to run through the defined route.

Figure 37 – VRUnit collecting data in the field

By performing these variations in the physical activity rhythm, we were able to notice heart level changes on participants which led to the triggering of the created alarms, as follows:

- Lap 1: Since none of the defined values was reached, there were no triggered alarms
- Lap 2: It was triggered a yellow alarm in the last part of the first half of the route, followed by an orange alarm triggered at the end of the path
- Lap 3: The beginning of the route was marked by several orange alarms; from here, the system reported red alarms for every participant

Simultaneously with the acquisition work in the field, there was a monitoring team in the lab responsible for the mission analysis (Figure 38). This team was able to take some insights over the acquired data during the experiment.

Through Grafana dashboard, it was possible to create an overall view of the three participants while in Vue the displayed data corresponded only to the logged user. This visualization duality, allowed the team to perform different kinds of analysis. As an example, if one of the participants experiences a hazard situation during the mission, the monitoring team can be aware of this from

the Grafana side and, then, point their attention only to this participant by logging in with his device id in Vue dashboard. Once there the view refers only to an individual, the monitoring team can perform a more precise and detailed analysis to find what went wrong.

Figure 38– Monitoring of team members during experiment

7 Conclusions

With this dissertation, we aimed to refactor VR2Market's online visualization layer, the VRCommander. Without compromising the remaining parts of the system, we managed to enhance the existing components while adding it some new capabilities.

Apart from the features provided by the current system, which we can keep functional with our implementation, we were able to introduce some new ones.

The opportunity to explore stream processing, although within a "hype", was a viable option and, in fact, our work shows that it provides a good approach to handle and process data flows. In our system, this was illustrated with simple trigger alarms but can be easily extended to more complex analytics without any refactoring needed at the architecture and data flow pipeline. Streaming also allowed an easy integration with off-the-shelf solutions used for logging and monitoring to provide a continuous visualization of data. By combining both approaches, we could simultaneously add processing and flexible views over data – incoming and historical.

An important requirement to address was the need to establish a fully cloud-enabled solution which was possible due to the fact of keeping our whole architecture based in Docker containers. Containerizing our implementation added the possibility to deploy our system in the cloud since plenty of the major cloud computing providers, including Amazon Web Services (AWS), Microsoft Azure or Google Platform (GCP) support Docker hosting.

Due to relying on tools which are prone to be scaled, we can ensure that, if needed, our system can be scaled up in order to fulfil growing demands.

Despite being related with VR2Market context, our system should be able to adapt itself to other types of contexts and scenarios which in fact can now be made without the need of major system refactoring. Mainly, our architecture relies on a set of off-the-shelf tools which were properly linked and configured in order to deal with our system's needs. However, due to their general-purpose nature, they can easily be adapted to other scenarios. In addition, the system should support the integration of new types of sensors which can now be made with low effort.
As a matter of fact, this mentioned flexibility represents one key advantage added to the system with our implementation, since the made choices allow to integrate new modules to the system effortlessly.

7.1 Extending the system

Our system was designed taking in consideration its need to adapt to different types of context and scenarios. A clear advantage of this approach is the ability to extend the system without performing major changes to it.

Taking our processing component as an example, there are plenty of other kinds of alarms that can be created in order to address different needs. The abstraction provided by our system allows one to develop new features since information is easily accessible through Kafka topics, so, if needed, integration work should only include small modifications. Thus, different processing approaches can be applied to the system by other processors such as Apache Spark, Storm[35] or Flink[36], just to name a few. Having Kafka at the core of our implementation allows to easily integrate this kind of tools which can lead to reaching more complex levels of processing and analytics.

In addition, the UI layer can also be adapted to better suit other scenarios once it is based atop of Grafana. This visualization tool is highly customizable allowing to create dynamic dashboards. Extensibility to serve other purposes can be easily achieved since it relies on a plugin-based approach. By designing new plugins or using the existing ones, it is easy to address a vast range of situations, from ingesting from different sources to creating custom visualizations over data.

7.2 Future work

From an overall view, it is fair to claim that the proposed requirements for this project were accomplished. However, there are still opportunities for further improvements which can be done by adding new features or just by enhancing the existing ones.

Despite being designed to be cloud deployable this scenario was not implemented nor tested. Fully containerized, the system gives this functionality which, however, is not being used yet.

On the other hand, there are also some improvements which can be made to the processing engine of our system. On one hand, as previously mentioned, it is possible to add different processors which can allow performing more complex operations over the incoming data. Moreover, there are some different types of alerts that can be generated, as the one presented above works only as a proof of concept. Namely, QoS related alarms can be implemented so that both the commander and FR can be aware of the system status.

Finally, the section of our work which can be subjected to major modifications, aiming to become more valuable, is the visualization layer. The designed UI can be improved not only in functionality but also in appearance. As our main concern was to display all the gathered data to the end user we ended up giving more relevance to functionality over styling. However, even this aspect can be enhanced, namely, the notification's section. Currently, the user is regarded with a list containing the last triggered alerts, however, there is not any notification mechanism implemented in the system. With this, it would be possible to display the incoming alerts

[35] http://storm.apache.org/

[36] https://flink.apache.org/

regardless of which section the user was browsing. In addition, in case of being related to a FR alert, this notification could also display the measured physiological data as well as the location of the operational by the time the alert occurred. As a matter of fact, data relationships in our system already allow implementing this feature.

References

[1] "VR2Market: Towards a Mobile Wearable Health Surveillance Product for First Response and other Hazardous Professions - INESC TEC." [Online]. Available: https://www.inesctec.pt/en/projects/vr2market-PG08007#about. [Accessed: 15-Feb-2018].

[2] M. Skogstad, M. Skorstad, A. Lie, H. S. Conradi, T. Heir, and L. Weisaeth, "Work-related post-traumatic stress disorder," *Occup. Med. (Chic. Ill).*, vol. 63, pp. 175–182, 2013.

[3] "CDC - Fire Fighter Fatality Investigation and Prevention Program: Main Page - NIOSH Workplace Safety and Health Topic." [Online]. Available: https://www.cdc.gov/niosh/fire/. [Accessed: 28-Feb-2018].

[4] "VitalJacket - Biodevices." [Online]. Available: http://www.vitaljacket.com/?lang=pt-pt. [Accessed: 04-Feb-2018].

[5] F. Guerrini, "How The Internet Of Things Can Help Firefighters Save Lives." [Online]. Available: https://www.forbes.com/sites/federicoguerrini/2015/06/29/firefighters-and-the-internet-of-things/#539948945e72. [Accessed: 02-Mar-2018].

[6] B. Marr, "Is Big Data Analytics The Secret To Successful Fire Fighting?" [Online]. Available: https://www.forbes.com/sites/bernardmarr/2017/01/13/is-big-data-analytics-the-secret-to-successful-fire-fighting/#795eae7045d0. [Accessed: 02-Mar-2018].

[7] E. R. Tufte and P. Graves-Morris, *The visual display of quantitative information*, Vol. 2. Graphics press Cheshire, 1983.

[8] K. Mani Chandy, "Event-Driven Applications: Costs, Benefits and Design Approaches. Gartner Application Integration and Web Services Summit, ummit 2006, San Diego, CA, June 2006."

[9] O. Etzion, P. Niblett, and D. Luckham, *Event Processing in Action*. .

[10] A. Hinze, K. Sachs, and A. Buchmann, "Event-Based Applications and Enabling Technologies."

[11] M. Wolf, "The Physics of Event-Driven IoT Systems," *IEEE Des. Test*, vol. 34, no. 2, pp. 87–90, Apr. 2017.

[12] D. Namiot, "On Big Data Stream Processing," *Int. J. Open Inf. Technol.*, vol. 3, pp. 48–51, 2015.

[13] W. M. P. Van Der Aalst, "Data Scientist: The Engineer of the Future," *Mertins K., Bénaben F., Poler R., Bourrières JP. (eds) Enterprise Interoperability VI. Proceedings of the I-ESA Conferences, vol 7. Springer, Cham*.

[14] Y. Chen, P. Xu, and L. Ren, “Sequence Synopsis: Optimize Visual Summary of Temporal Event Data,” *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 45–55, Jan. 2018.

[15] R. Elchaama, B. Dafflon, R. K. Chamoun, and Y. Ouzrout, “Toward a traffic regulation based on event processing agent system,” in *2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, 2017, pp. 1350–1356.

[16] P. S. Group, B. M. Michelson, S. Vp, and S. Consultant, “Event-Driven Architecture Overview Event-Driven SOA Is Just Part of the EDA Story Event-Driven Architecture Overview,” 2006.

[17] T. Dunning, *Streaming Architecture.* O’Reilly Media, 2016.

[18] T. Akiday, “The world beyond batch: Streaming 101 - O’Reilly Media.” [Online]. Available: https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101. [Accessed: 28-May-2018].

[19] S. Saxena and S. Gupta, *Practical real-time data processing and analytics : distributed computing and event processing using Apache Spark, Flink, Storm, and Kafka*. .

[20] J. Stenberg, “Making Sense of Event Stream Processing.” [Online]. Available: https://www.infoq.com/news/2015/03/events-streams-processing. [Accessed: 19-May-2018].

[21] D. Luckham, “What’s the Difference Between ESP and CEP? – Real Time Intelligence & Complex Event Processing.” [Online]. Available: http://www.complexevents.com/2006/08/01/what’s-the-difference-between-esp-and-cep/. [Accessed: 19-May-2018].

[22] S. Shahrivari and Saeed, “Beyond Batch Processing: Towards Real-Time and Streaming Big Data,” *Computers*, vol. 3, no. 4, pp. 117–129, Oct. 2014.

[23] M. Stonebraker, U÷. Çetintemel, and S. Zdonik, “The 8 Requirements of Real-Time Stream Processing.”

[24] B. E. Friedman and K. Tzoumas, *Introduction to Apache Flink : stream processing for real time and beyond*, 2016th ed. O’Reilly Media.

[25] E. Friedman and K. Tzoumas, “Batch Is a Special Case of Streaming - MapR.” [Online]. Available: https://mapr.com/ebooks/intro-to-apache-flink/chapter-6-batch-is-a-special-case-of-streaming.html. [Accessed: 24-Oct-2017].

[26] R. Estrada, “From big data to fast data - O’Reilly Media.” [Online]. Available: https://www.oreilly.com/ideas/from-big-data-to-fast-data. [Accessed: 12-Mar-2018].

[27] D. Wampler, *Fast Data Architectures For Streaming Applications*. .

[28] 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)*, 2017, pp. 4803–4806.

[29] 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)*, 2017, pp. 1–5.

[30] G. M. D’silva, A. Khan, Gaurav, and S. Bari, “Real-time processing of IoT events with historic data using Apache Kafka and Apache Spark with dashing framework,” in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, 2017, pp. 1804–1809.

[31] “Confluent Grows Subscriptions By Over 700 Percent in 2016.” [Online]. Available: https://www.confluent.io/press-release/confluent-grows-subscriptions-700-percent-2016-businesses-seize-power-real-time-data/. [Accessed: 04-Apr-2018].

[32] N. Narkhede, G. Shapira, and T. Palino, *Kafka : the definitive guide: real-time data and stream processing at scale*. .

[33] G. M. D’silva, A. Khan, Gaurav, and S. Bari, “Real-time processing of IoT events with historic data using Apache Kafka and Apache Spark with dashing framework,” in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, 2017, pp. 1804–1809.

[34] “Kafka Streams - The easiest way to write mission-critical real-time applications and microservices.” [Online]. Available: https://kafka.apache.org/documentation/streams/. [Accessed: 07-Apr-2018].

[35] P. Hunt, M. Konar, Y. ! Grid, F. P. Junqueira, B. Reed, and Y. ! Research, “ZooKeeper: Wait-free coordination for Internet-scale systems.”

[36] N. Narkhede, “Confluent - Real-time Data Streams.” [Online]. Available: https://confluentinc.wordpress.com/page/3/. [Accessed: 28-May-2018].

[37] S. Patro, M. Potey, and A. Golhani, “Comparative study of middleware solutions for control and monitoring systems,” in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, 2017, pp. 1–10.

[38] J. Kreps, “Introducing Kafka Streams: Stream Processing Made Simple - Confluent.” [Online]. Available: https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/. [Accessed: 09-Apr-2018].

[39] M. Fowler, “Event Sourcing.” [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html. [Accessed: 18-Apr-2018].

[40] M. Sax, “Stream Application Development with Apache Kafka.”

[41] P. Helland, “Immutability Changes EVERYTHING.”

[42] “Apache Flink: Introducing Stream Windows in Apache Flink.” [Online]. Available: https://flink.apache.org/news/2015/12/04/Introducing-windows.html. [Accessed: 05-Jun-2018].

[43] J. Kreps, L. Corp, N. Narkhede, and J. Rao, “Kafka: a Distributed Messaging System for Log Processing,” 2011.

[44] J. Kreps, *I [love] logs : [event data, stream processing, and data integration]*. O’Reilly Media, 2014.

[45] “logz.io - 15 Companies Using the ELK Stack.” [Online]. Available: https://logz.io/blog/15-tech-companies-chose-elk-stack/. [Accessed: 24-Jan-2018].

[46] A. Yigal, “The Complete Guide to the ELK Stack - Logz.io.” [Online]. Available: https://logz.io/learn/complete-guide-elk-stack/. [Accessed: 08-Mar-2018].

[47] “Apache License, Version 2.0.” [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0. [Accessed: 08-Mar-2018].

[48] “Beats - Lightweight Data Shippers.” [Online]. Available: https://www.elastic.co/products/beats. [Accessed: 08-Mar-2018].

[49] “Elasticsearch 6.1 online documentation - Elasticsearch product description.” [Online]. Available: https://www.elastic.co/products/elasticsearch. [Accessed: 09-Mar-2018].

[50] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter, “Bkd-Tree: A Dynamic Scalable kd-Tree,” in *Advances in Spatial and Temporal Databases*, 2003, pp. 46–65.

[51] A. Grand, “Elasticsearch as a column store.” [Online]. Available: https://www.elastic.co/blog/elasticsearch-as-a-column-store. [Accessed: 09-Mar-2018].

[52] R. Kuć and M. Rogoziński, *Elasticsearch server : create a fast, scalable, and flexible search solution with the emerging open source search server, ElasticSearch*. Packt Pub, 2013.

[53] S. Bagnasco, D. Berzano, A. Guarise, S. Lusso, M. Masera, and S. Vallero, “Towards Monitoring-as-a-service for Scientific Computing Cloud applications using the ElasticSearch ecosystem,” vol. 664, 2015.

[54] “Logstash 6.2 online documentation - Logstash product description.” [Online]. Available: https://www.elastic.co/products/logstash. [Accessed: 10-Mar-2018].

[55] “Kibana 6.2 online documentation - Kibana product description.” [Online]. Available: https://www.elastic.co/products/kibana. [Accessed: 10-Mar-2018].

[56] Saurabh Chhajed., *Learning ELK Stack.* Packt Publishing, 2015.

[57] “InfluxData | Time Series Database Monitoring & Analytics.” [Online]. Available: https://www.influxdata.com/. [Accessed: 15-Mar-2018].

[58] D. R. Brillinger, *Time series : data analysis and theory*. Society for Industrial and Applied Mathematics, 2001.

[59] “InfluxData | Documentation | Downsampling and data retention.” [Online]. Available: https://docs.influxdata.com/influxdb/v1.5/guides/downsampling_and_retention/. [Accessed: 15-Mar-2018].

[60] “Grafana - The open platform for analytics and monitoring.” [Online]. Available: https://grafana.com/. [Accessed: 19-Mar-2018].

[61] J. Verona, P. Swartout, and M. Duffy, *Learning DevOps : continuously deliver better software : learn to use some of the most exciting and powerful tools to deliver world-class quality software with continuous delivery and DevOps : a course in three modules.* Packt Publishing, 2016.

[62] G. Avolio, M. D’Ascanio, G. Lehmann-Miotto, and I. Soloviev, “A web-based solution to visualize operational monitoring data in the Trigger and Data Acquisition system of the ATLAS experiment at the LHC,” *J. Phys. Conf. Ser.*, vol. 898, no. 3, p. 32010, Oct. 2017.

[63] “Grafana documentation | Grafana Documentation.” [Online]. Available: http://docs.grafana.org/. [Accessed: 13-Nov-2017].

[64] V. Hajek, T. Klapka, and I. Kudibal, “InfluxDB vs. Elasticsearch for Time Series Data and Metrics Benchmark.” [Online]. Available: get.influxdata.com/rs/972-GDU-533/images/InfluxDB 1.4.2 vs. Elasticsearch 5.6.3.pdf. [Accessed: 22-Jan-2018].

[65] “CSV Comma Separated Value File Format - How To - Creativyst - Explored,Designed,Delivered.(sm).” [Online]. Available: http://creativyst.com/Doc/Articles/CSV/CSV01.htm. [Accessed: 03-May-2018].

Appendices

A.1 Public Presentations

The developed work in this dissertation was revealed in a public event, *students@deti,* in Universidade de Aveiro (Figure 39).

Figure 39– Poster used to present PhisioStream system at students@deti 2018