Master's Degree in Informatics Engineering

Master's Thesis

Uma plataforma para tratamento de eventos de segurança

Author:
Daniel Alexandre Sobral Fernandes

Supervisor:
Prof. Paulo Simões

Co-Supervisor:
Prof. Tiago Cruz

September 2019

This page is intentionally left blank.

Acknowledgements

To the end of the biggest step of my life,

I thank my mom, for always supporting me, letting me live as I wished and letting me make my own choices, and for all the effort, time, and dedication put in my education.

I thank my friends, for all the support during this exhausting part of my life, particularly my colleague Rui Silva, who helped me in many occasions, not only during the thesis, but during the whole masters.

I thank my colleague Pedro Quitério, who always helped me in many development endeavors and integrated me to the skeleton template used in the project. Without his help, development would have taken considerably more time.

I thank my Co-Supervisor, Professor Tiago Cruz, for helping me integrating in the project and clarifying me of any doubts and problems I had, always in a lighthearted and cheerful way.

I thank Professor Vasco Pereira and colleague Jorge Proença, for their time, effort, and energy they invested in me during this thesis. My work couldn't be done without their help and feedback.

I thank Professor Paulo Simões and Professor Marília Curado, for all the help integrating in the project and LCT.

I thank the people at OneSource who stood by me in the first meeting and watched me be completely oblivious to what was going on, doing their best to thoroughly explain to me what was going on and how things worked.

Finally, I thank my best friend, Jorge, who always helped me keep a good state of mind, when my life was going through a rough patch.

To all of you,

from the bottom of my heart,

Thank you!

This work is supported by the European Regional Development Fund (FEDER), through the Regional Operational Programme of Lisbon (POR LISBOA 2020) and the Competitiveness and Internationalization Operational Programme (COMPETE 2020) of the Portugal 2020 framework [Project 5G with Nr. 024539 (POCI-01-0247-FEDER-024539)];

Cofinanciado por:

UNIÃO EUROPEIA
Fundo Europeu
de Desenvolvimento Regional

This page is intentionally left blank.

Abstract

In a world of constant technological growth, in which more and more things are being virtualized, there are opportunities to create ambitious and innovative projects that can open ways for many new technologies and businesses to be created.

The 5G architecture is one of these advancements. By integrating many aspects such as virtualization and software defined networking, it can is designed to provide a substantial leap regarding its ancestor technology, 4G. 5G makes use of these paradigms to enable technologies such as, for example, robotic surgery, autonomous driving and low latency Virtual Reality (VR).

However, even though this architecture greatly benefited from the software defined networking and virtualization paradigms, the lack of intermediary software has become a problem, heavily affecting the practical use of this type of networks. There is a lack of monitoring software, management software, and middlewares for the interaction of certain (usually important) network components. This compels the developers and network operators to create their own middlewares and applications that allow the platform to work as intended.

This thesis has the goal of creating such software, in this case, an event collector service, plus a dashboard which will aggregate and display data about the other nodes and services in the 5G platform.

The tests done to the created software ensure the performance levels obtained are up to par with the project's demands, and the components should allow for a performance scaling later down the road, if necessary.

Keywords

Virtualization, Software Defined Networking, Middleware, Monitoring

This page is intentionally left blank.

Resumo

Num mundo de constante crescimento tecnológico, em que cada vez mais é virtualizado, surgem oportunidades de criar projectos ambiciosos e inovativos, que abrem caminho para o crescimento de novas tecnologias e negócios.

Tal como a arquitectura 5G, que, integrando aspectos da virtualização e de Software Defined Networking, consegue ultrapassar a sua tecnologia anterior, o 4G. O 5G toma partido destes paradigmas para permitir o crescimento de tecnologias como a medicina robótica, a condução autónoma, e realidade virtual de baixa latência.

No entanto, mesmo que esta arquitectura seja fortemente beneficiada dos paradigmas de Software Defined Networking e da virtualização, há um problema geral de falta de software intermediário, que afecta directamente o uso prático deste tipo de redes. Não existem softwares de monitorização, gestão, e middlewares de comunicação entre componentes (por muitas vezes críticos) da rede. Isto obriga os programadores e operadores de rede a tomar benifício da flexibilidade e programabilidade das Software Defined Networks, para criar as suas próprias aplicações, a fim de satisfazer as necessidades da sua rede em particular.

Este projecto tem o objectivo de criar um software desse tipo. Neste caso, um colector de eventos, juntamente a um dashboard, que irá agregar e exibir dados sobre outros nós e serviços da plataforma 5G.

Os testes feitos aos componentes desenvolvidos garantem que os níveis de performance obtidos estão a par com as necessidades do projecto, e além disso, existe a possibilidade de escalar estes níveis de performance dos componentes no futuro, dado a necessidade.

Palavras-Chave

Virtualização, Software Defined Networking, Middleware, Monitorização

This page is intentionally left blank.

Contents

Acronyms

ACL Access Control List. 36, 38

AMQP Advanced Message Queuing Protocol. 21, 22

API Application Programming Interface. 20, 48

ATENA Advanced Tools to assEss and mitigate the criticality of ICT compoNents and their dependencies over Critical InfrAstructures. 36, 54

CLI Command-Line Interface. 15

DEI Department of Informatics Engineering. 64

DNS Domain Name System. 42

FCTUC Faculdade de Ciências e Tecnologia da Universidade de Coimbra. 52, 82

Gbps Gigabits per second. 9

GSSAPI Generic Security Services Application Program Interface. 38

HTTP HyperText Transfer Protocol. 22

IDPS Intrusion Detection and Prevention System. 78

IoT Internet of Things. 9

IP Internet Protocol. 35, 49

JMS Java Message Service. 21

JMX Java Management Extensions. 21

JSON JavaScript Object Notation. 30, 34, 35, 49, 68–70

JVM Java Virtual Machine. 21

MitM Man in the Middle. 36

MQTT Message Queuing Telemetry Transport. 21, 22

NAT Network Address Translation. 49

NEF Network Exposure Function. 42, 43, 53, 54, 56, 59, 60, 65, 66

NFV Network Functions Virtualization. 41

REGEX Regular Expression. 38

Rkt CoreOS Rocket. 14

SASL Simple Authentication and Security Layer. 36, 37

SHA-1 Secure Hash Algorithm - 1. 38

SIEM System Information and Event Management. 10

SMS Short Message Service. 10

SSD Solid State Drive. 64, 67

SSL Secure Socket Layer. 36–38

STOMP Simple (or Streaming) Text Oriented Message Protocol. 21, 22

TLS Transport Layer Security. 15, 36

UC University of Coimbra. 78

UI User Interface. 22, 44, 45, 85

VAS Vulnerability Assessment System. 42, 43

vCPU Virtual Central Processing Unit. 52

VM Virtual Machine. 7, 8, 11, 64, 67, 77

VNF Virtual Network Function. 49

List of Figures

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

This document consitutes the Master Thesis in Informatics Engineering of the student Daniel Alexandre Sobral Fernandes, during the academic year of 2018/2019, in the Department of Informatics Enginneering of the University of Coimbra.

1.1 Context

This thesis fits in the context of the Project Mobilizador 5G (POCI-01-0247-FEDER-024539), more specifically in the scope of PPS2 - "Products and services for the network core" working group. PPS2 is devoted to the core functions that receive information from other nodes in the 5G network, and displays them in a consolidated, simple manner, is needed.

The 5G project itself is divided into multiple "layers" of development, each one dedicated to a certain component in the overall architecture. I was integrated in the PPS2 development effort, which focuses on the development of products to be used in the network core. In this scope, I was assigned with the task of developing a dashboard to be used in the core of the architecture as a monitor station for all the remaining equipment, and as a security tool, to help prevent or act towards network attacks

In this perspective, there is the need to develop a group of innovative mechanisms that allow network operators to react quickly upon any interference or problem in the network. This mechanism will involve many other technologies, working together to offer the platform a seamless monitoring station with small overhead on the rest of the platform, while providing valuable data to the network operators.

1.2 Motivation

The motivation behind this project comes from the need of 5G network operators to have adequate tools to be used with the 5G platform. While most of the existing tools are either bound to proprietary equipment or paid for, there's the need of having a tool that's developed specifically for this project.

There are a lot of development tools, most of them even open-source, that enable this highly programmable paradigm to shine, and enable developers and network operators to create applications that use many of these tools simultaneously, or in a cooperative manner.

The tools themselves have a very wide range of utility, and even modularity, sometimes. This allows for the applications created using these tools to be highly optimized towards the solution of a particular problem, rather than a bundle of tools put together that solve many problems, but cause a significant overhead in the network.

For the 5G platform, there's the need of a monitoring dashboard, that collects, sorts, and filters information from multiple nodes belonging to the network, and displays them, in a simplified manner, to the network operator(s).

1.3 Goals

For this thesis, I will integrate in the PPS2 working group, familiarizing myself with the Mobilizador 5G project and its scope, objectives, technologies, as well as tools. It was also important for me to learn about the mechanisms that will directly impact my project, as well as the nature of the data that's supposed to be directed to the dashboard.

The objective of the project is to create a software security tool that will allow the 5G network administrators the aggregated view of the events happening in the network, incorporating specific functionalities for the monitoring of the platform, data visualization and event transport/persistence.

This last aspect will involve research on data visualization tools, methods, and solutions that can be relevant for the creation of the application, and are adequate to the nature of the data flows that arrive to the dashboard, in real time.

The main objectives that are planned include:

- Learning about the 5G project and its state of the art, in relation to its tools and components, as well as how a data visualization application would help the architecture in regards to its overall security and monitoring capabilities;
- Proposal for the front-end design to implement, based on the pre-established requirements for the platform;
- Analysis and selection of the adequate visualization sources catered to the necessities of the platform;
- Implementation of the web front-end and data visualization solutions;
- Usability evaluation of the developed application.

1.4 Document Structure

The document comprises of several chapters that detail the entirety of the work done during the thesis' duration:

- State of the Art: where the most currently used technologies that are relevant to the project will be discussed;
- Research Objectives: research done regarding the requisites for the application will be discussed, as well as an introduction to the scope of the application itself, and what options discussed on the state of the art fit the application to solve the problem

at hand.It will then mention any reference software that might be used as a base for any component of the application;

- Preliminary Work: experimentation, or specific research that has been done related to a topic that's important to the development of the application, will be detailed;
- Architecture: there will be some detailing regarding the proposed architecture of the application, and how it fits in the overall project, as well as some prototypes that were built to help imagining the design of the dashboard;
- Solution: this chapter will detail the steps to create the idealized architecture debated in the previous chapter, as well as list any changes that were done to the proposed architecture during development. It will also display the final work, including screen shots of the platform, and how it was set up;
- Validation: here, the platform will be tested against performance levels who are similar to a production environment, while its behaviour will be recorded and analyzed;
- Work Plan: this chapter details all the work that has been done during this thesis, and how long each task took in the overall scope;
- Conclusion: concluding notes about the project and thoughts for future work.

This page is intentionally left blank.

Chapter 2

State of the Art

In this chapter the main topics covered by the thesis will be detailed upon. Firstly, with some concepts about monitoring, since the objective of the work done in the thesis is to apply what is going to be done into a real-life monitoring scenario involving equipment and software from several different companies. Afterwards, there will be a description of each of the candidate technologies that are going to be used and , directly or indirectly, and a comparison between them and the best in their area of work.

2.1 Concepts

The following concepts detail the baseline knowledge required to understand the work related directly to this project.

2.1.1 Monitoring

Considering that the thesis revolves around creating a monitoring system for the 5G architecture, out of the several types of monitoring that exist, this work integrates in the area of system monitoring.

System monitoring is the act of collecting information, statistics, state data, etc... from one or more systems which who are usually involved in a mutual task or purpose. This system data usually comes in the form of hardware or software-specific metrics that are important to be notified to the network operator. In the 5G platform, examples of this include container and/or service operation metrics, and specially security metrics that are sent from another components.

However, even though monitoring in itself has expanded greatly in this past decade or so, there is still a lack of solutions to monitor systems or networks that are not based on a single brand's equipment, and they're usually very expensive, with very scarce open source projects capable of dealing with carrier-like requirements.

The main goals of a monitoring plaform are [13]:

- Provide information about the states and/or variables of equipment and/or software;
- Identify and categorize problems in the scope of the project and notify the administrator;
- Guarantee the accessibility of the data;
- Obtain vital statistics, such as efficiency indicators;
- Improve upon the design on the topology, the systems being monitored, or the monitoring system itself;
- Integrate stakeholders in the project, since monitoring systems usually offer human-perceivable data.

In this project's case, the monitoring will follow a centralized architecture, in which all the equipment that take part of the topology will send data containing metrics or another relevant events to a centralized viewer of events - the Dashboard.

2.1.2 Virtualization

One of the biggest costs a company has to face is usually their IT infrastructure, and it is important this budget is spent wisely and allows for future adapting due to organizational changes, to support new business initiatives, and above of all, is used in a cost-effective way [25].

The fact that the technological advances are happening so quickly allows the creation of many types of solutions to nowadays' problems, particularly solutions that are versatile and don't require dedicated hardware. This is a very interesting topic for the enterprise level, since it causes the infrastructure to be much more flexible, while allowing future growth without the possible impairment of being "out-of-date" or "not compatible" with proprietary hardware or software.

There will still be constraints in using older equipment, of course, but a very small number when in comparison to the use of virtualization techniques.

Here's where virtualization comes in:

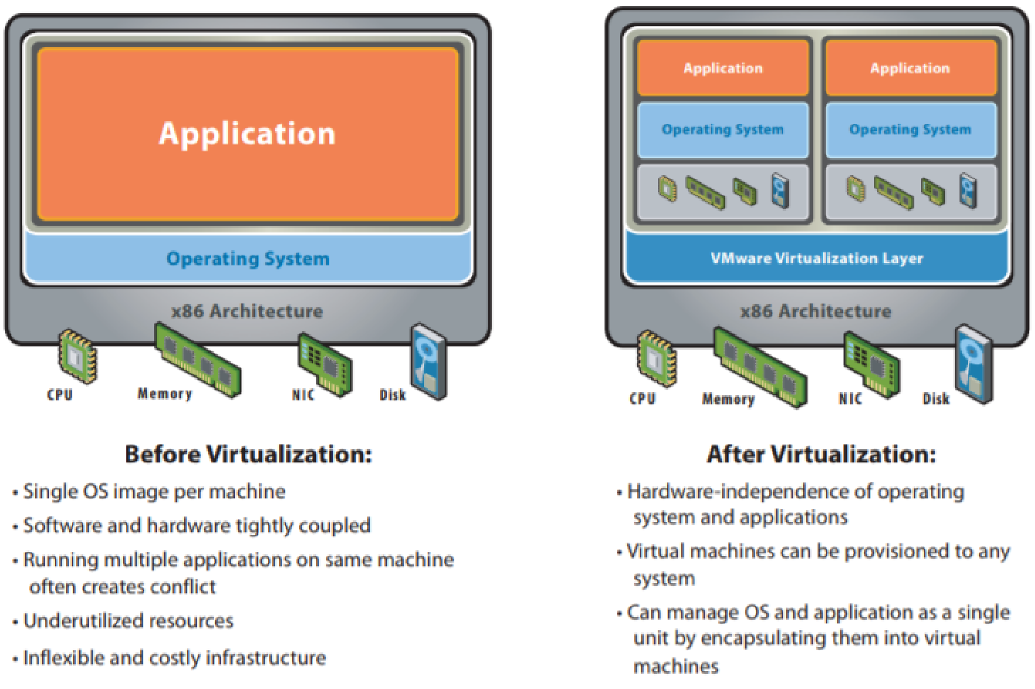

Application
Operating System
x86 Architecture
CPU
Memory
NIC
Disk
Before Virtualization:
• Single OS image per machine
• Software and hardware tightly coupled
• Running multiple applications on same machine often creates conflict
• Underutilized resources
• Inflexible and costly infrastructure
Application
Operating System
Application
Operating System
VMware Virtualization Layer
x86 Architecture
CPU
Memory
NIC
Disk
After Virtualization:
• Hardware-independence of operating system and applications
• Virtual machines can be provisioned to any system
• Can manage OS and application as a single unit by encapsulating them into virtual machines

Figure 2.1: How virtualization acts before a host system [26]

By separating the logical resources and the underlying physical delivery of those resources through simulated environments, we achieve a method to abstract the hardware from the software running over it, while with the help of a Hypervisor, which is a software used in virtualization that is directly connected to the hardware itself, we can create multiple instances of simulated environments that use the same abstracted pooled hardware.

By doing this hardware abstraction as a Host, all the simulated environments that are instanced as Virtual Machine (VM)s are called Guests. All the Guests see physical resources, like memory, CPU, storage, and even Network as a shared pool of resources, which can be allocated freely at their will, assuming they follow the restrictions imposed by the Hypervisor, which were in turn imposed by the user.

Virtualization is also a very important game changer in relation to deployment, since is it non-disruptive to either the Host system or the user experience, and the creation/deletion of new VMs can be done easily via scripts or through the Hypervisor itself. Since the resources are abstracted and pooled together, this also gives infrastructure managers the advantage of being able to manage the pooled resources, for example, to add new resources with no or little downtime, if needed.

2.1.3 Microservices

The idea behind the creation of microservices came from the thought that an application would be easier to manage, build and maintain, if it was operating separately in its smaller, broken-down state. These applications would then work together and the total system would be the "sum" of all the separate applications [8]. This contradicts the "Monolithic" application architecture, in which there is a focus on building everything into a single application.

This makes applications easier to deploy, debug, and allows an easier understanding of the code for the developers who have just started to work on it. It can also be paired with virtualization technologies, to truly make a dedicated environment for each specific application - Containers, as will be discussed further on. Apart from this, there are other benefits, for example [8]:

- *Scalability*, since the applications are smaller, and use fewer resources, they can be increased according to demand;
- *Resilience*, since all the applications run independently from each other;
- *Isolation*,since in case of a crash or unexpected termination of the application, only that instance of the application will be terminated;
- *Monitoring*, since microservices are usually associated to real-time monitoring of the application;
- *Availability*, taking advantage of the low resource usage, multiple instances of the same app may be running and providing better availability to the main service;
- *Fault Tolerance*, since applications should be created in a way that they can tolerate the failure of services already running. Adding to this, microservices are most of the times autonomous, and so, there is a lower chance they'll cause system failures when something wrong happens;
- *Containerization*, for example, using Docker, where services can be created and deployed with lower overheads than using an entire guest VM containing an operating system [25].

Microservices also allow for applications to be more modular, and in doing so, it is possible to create standalone applications and components where previously there were only proprietary solutions available. Many of these components are just tools, but they integrate into projects that can do the same (or more) as proprietary solutions, are more versatile, modular, and can play many of the different types of microservice architectures (perhaps even simultaneously) such as: authentication, logging and monitoring, load balancing, etc...

2.2 Technologies

In this section are presented the relevant technologies and tools to be mentioned for the development of the project. These technologies will be strongly based on topics discussed in the previous section. The main topics that will be discussed include containers, message brokers and intrusion detection/intrusion prevention systems.

2.2.1 5G

5G is the fifth generation of mobile networks, still in ongoing development. It has the objective of meeting the ever growing number of devices being connected to the internet and the also growing number of services, industries and applications that require a highly reliable, low-latency network connection.

By itself, 5G is not a new standalone technology, but rather it is an aggregation of new and existing networks: mobile, fixed and wireless, to create a more flexible environment. 5G is being built with an approach favoring open access and multi-connectivity, so it may create a base for many new services, solutions and applications. By having substantially less latency than its predecessors, 5G provides a flexible platform to make significant advances in several businesses and industries, such as automotive, manufacturing, energy, health, entertainment and will pave the way for smart city architectures. Apart from that, it also provides about ten times the throughput 4G offers, which bumps it up to about 10 Gigabits per second (Gbps), while providing ubiquitous coverage [24].

The areas of interest that 5G is going to propel forward are mainly areas that are very latency-sensitive, in which a 100ms delay would be a very significant problem, such as autonomous driving, robotic surgery and virtual/augmented reality. This is very useful for paradigms such as Internet of Things (IoT), where the internet connection is extended past the regular devices, such as home and city electronics. 5G also offers many new features to IoT networks, mainly communication and coverage-wise, but also creating new opportunities for new businesses and for the growth of society in general.

For this purpose, network slicing shows potential as a promising future-proof framework, fulfilling the technological and business needs of a wide range of industries. There won't be the need of external systems and third party applications concerning security [24], since 5G has its security architecture natively integrated into its overall architecture, granting the security requisites of services and applications for which data integrity and confidentiality is critical [14].

One of the currently proposed 5G architecture follows the outline in Figure 2.2:

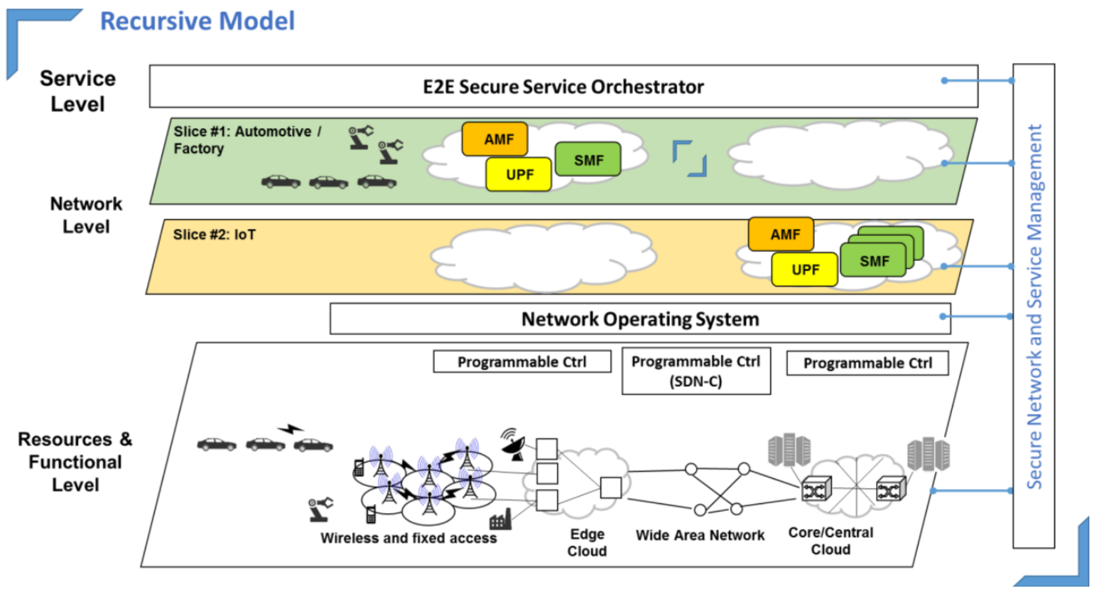

Recursive Model
Service Level
E2E Secure Service Orchestrator
Slice #1: Automotive / Factory
AMF
UPF
SMF
Network Level
Slice #2: IoT
AMF
UPF
SMF
Network Operating System
Programmable Ctrl
Programmable Ctrl (SDN-C)
Programmable Ctrl
Resources & Functional Level
Wireless and fixed access
Edge Cloud
Wide Area Network
Core/Central Cloud
Secure Network and Service Management

Figure 2-1: Overall Architecture

Figure 2.2: 5G architecture [24].

Network slicing, which is a virtualization technique and a core 5G capability that will enable flexibility, provisioning and cost-efficiency of networks, allowing multiple logical networks to be created and mapped on top of a physical infrastructure [14]. This technique was created to fulfill the demand of vertical sectors that require dedicated networks, by creating "customer facing" end-to-end logical networks called slices [24].

Before 5G, slice managing tasks were usually manually performed and there were less types of services that required the use of network slices, such as mobile broadband, voice service, or Short Message Service (SMS). Given the foreseen increase of business branches and customer requests with 5G, it is safe to say that a significant increase in automated mechanisms to create and manage slices is needed.

2.2.2 Event Collectors

The objective of this thesis is to develop a monitoring tool that is adequate to the 5G architecture under development, and is able to be scaled to the event processing needs of said architecture. This tool falls into the category of System Information and Event Management (SIEM) software, in which all the event and log data created by several applications in a given network is centrally obtained, interpreted and correlated, therefore creating a central point of access to all information about every device in the network. This allows system managers to act more efficiently and identify the existing problems more quickly.

However, there are already some tools that offer similar services, but are not adequate to solving our problem, since most of them are proprietary, have low flexibility regarding what type of systems they can monitor and what types of data can be sent, received and interpreted as events. Also, regarding scalability, these systems usually offer low scalability options, being usually necessary to buy additional probes, and their respective software licenses.

Since the collector needs to be able to be highly configurable, highly flexible, easy to deploy, and easy to scale, it was necessary to create a solution based on existing technologies and software. For a better understanding of the available options for event collecting software on the market, some of the most notorious ones will be detailed below.

IBM QRadar

IBM QRadar is a dedicate structure of appliances, being constituted by multiple pieces of hardware, working to achieve network event log centralization. It is used to detect anomalies and identify potential threats by analyzing the network flows, while keeping the number of false positives low.

It correlates log data to find problems that need intervention, while cross referencing the logs with lists containing potential malicious IP addresses.

Is is constituted by:

- QFlow Collector, which purpose is to gather network flows for statistical analysis;
- Event Collector, which gathers events from the network, sending them to the processor;

- Event Processor, which receives events from the collector and processes them, by correlating the event information from QRadar products;
- Console, which provides the end user with real-time information regarding events, flows, and other relevant information for the system;
- Data Node, which enables extra storage and processing power to other appliances, and are able to be added on demand, as necessary;
- Magistrate, which is the service that runs on the end-user console, that provides analytic data, events, reports and security data about the network traffic.

QRadar seems to be a powerful solution, with dedicated hardware, and impressive statistics regarding throughput (which is about fifteen thousand events per second), and its capability to scale thanks to its Data Nodes, which can be added on demand for more storage and processing power. However, its limitations are clear: its price, since it is a very costly solution, and its deployment would be slow, arduous, and probably generate problems with the rest of the topology. Also, there is already a predefined list of log source types and protocols they use, so the introduction of a new event data type and protocol would be hard. Ultimately, IBM QRadar doesn't offer the flexibility this solution requires, and scaling it to the desired proportions would be very costly, while also not taking advantage of the existing infrastructure.

SolarWinds Event & Log Manager

This software allows for the collection of logs from multiple windows servers or workstations (VM), in which the logs are organized, categorized, correlated and stored in a local server, while generating informative reports and sending e-mail alerts.

It possesses a (very limited) free trial version, with the full license going for five thousand dollars, but since it is only capable of gathering logs provided by machines or workstations that are running Windows, it does not fit the problem at hands, since most nodes in the projected system will most likely run a Unix distribution.

Windows Event Collector

This system allows system administrators to obtain event logs from remote machines and keep them into a central one, through message subscription methods, which can be defined based on:

- the source, without defining the event source computers, who set up their event forwarding rules to the collector later on, or usually according to a group policy;
- the collector, if all machines that are pretended to send events to the collector are previously known.

Just like the previous one, this system only gathers event information from Windows-based systems, and is therefore of no use to the project at hand.

2.2.3 Containers

The concept behind containers is to create a virtual environment that only contains the target application and all its dependencies. This container can then be run in most computing environments, mainly because they're usually very small in size and can be heavily optimized not to have unnecessary data in them. This is an optimized way of running standalone applications, and are commonly used over virtual machines due to the fact that a virtual machine usually runs an entire operating system over it, and containers look to maximize performance while minimizing resource usage, hence why usually there can be a larger number of containers running in an environment compared to virtual machines.

Application deployment inside a container environment is typically expected to have similar performance to a bare metal system, while comparing to the deployment inside of a virtual machine, the performance is significantly higher, due to the containerized environment being able to access the system's resources in a more efficient way. This feature also affects the start-up time of containers, which is typically much shorter than virtual machine based solutions.

There are a few companies that design containers and software related to creating, deploying, managing and developing containers, and each of them has a slightly different take on the paradigm and approach. Some of them will be discussed in the following sections.

Regarding development, containers also help the developers, since it is very helpful when paired with microservices. These can be run in separate containers, which allows developers to break down applications in smaller parts, through multiple containers. This means that development teams can work on different parts of the application simultaneously and independently from one another, which causes development to be more efficient, and to be done faster and with a lower rate of bugs.

Security-wise, the container attack surface will be smaller, as well as scattered, which improves overall system security. Container environments also help in using less overall data, given the usual small size of containers. Since they're usually integrated into bigger applications or systems, they can also perform other types of tasks, for example, redundancy or load-balancing purposes, or even to isolate certain components of the application to facilitate development, deployment and management. They can also run different applications with different functions, but acting towards one general vision or purpose of a system. To manage containers, or these clusters, there are a few choices of software created specifically for that purpose, some of which will be detailed upon further.

Container Creators - Docker

Docker is a containerization software used to create, run and deploy applications using Linux containers. This allows developers to create a package with only the application and all its dependencies, and easily deploy it.

Docker calls this "package" an image, making it easier to share an application or a group of services, bundled with all the necessary dependencies. These images can be built and shared manually or through a Dockerfile. Dockerfiles are a script-like file which is interpreted by Docker itself and allows the creation of a complete environment, bundled with all its dependencies (assuming these exist in an online repository). Docker can also automate the deployment of the application inside the container environment itself with scripts or the help of Docker Compose, which manages containers and clusters.

Some of the advantages of Docker include [20]:

- Modularity, since Docker's take on containerization grants the developer the ability to remove certain application components, either for repairing bugs or making changes, without having to take down the entire application;
- Image Version Control, since every docker image is constituted by layers, and each time the image changes, a new layer is created. The image is nothing more than the combination of all layers. This allows for version control, since previous layers are unaffected by new changes to the image. Furthermore, this layering system improves image building speed, lowers its size, and overall increases its efficiency;
- Rollback, taking advantage of the layering system. The user can roll back the image state to any layer he wants;
- Rapid deployment, since docker containers don't have an operating system, deployments can be reduced to mere seconds, and the overhead is a very minimal issue, unlike monolithic apps.

However, docker does not include management capabilities by itself, and since the number of containers is greatly scalable, this can easily get out of control. Hence the need for a management tool for containers and clusters of containers, which will be detailed upon further in this document.

Container Creators - LXC

LXC, or Linux Containers, is a container solution to support virtualization of software at the operating system level, inside a Unix Kernel host. LXC can be used to virtualize either standalone applications or entire Unix-based operating systems [10].

Instead of creating a regular virtual machine to host content, LXC instead creates a virtual environment that has its own process and network space, while sharing the resources of the host machine. Its main advantage is the possibility of controlling its virtual environments from the host, via userspace tools. This reduces the overhead significantly compared to the use of a Hypervisor. LXC was the core foundation of future software such as Docker and CoreOS Rocket, which eventually branched to their own paths and detached from the use of LXC. Another one of its advantages is the fact that it allows multiple distributions to be run simultaneously, in the same host.

Some of the advantages of using LXC for container creation and deployment are [23]:

- Good, well documented configuration capabilities, with a method for using straightforward templates for the creation and configuration of containers. These configurations usually come from a configuration file rather than command line parameters;
- Optional external network container exposure: containers created with LXC are effectively isolated from the outside networks until the user specifically specifies otherwise, and to access or to allow exposure of services within a container, it requires manual iptables forwarding to allow outside connections;
- Significant user base, community support and its indirect security benefits: Since LXC possesses a large number of users and directly contributes to general container advances and improvements as an Open Source project, there are more updates/bug fixes to the tool, and allows quicker access to newer features.

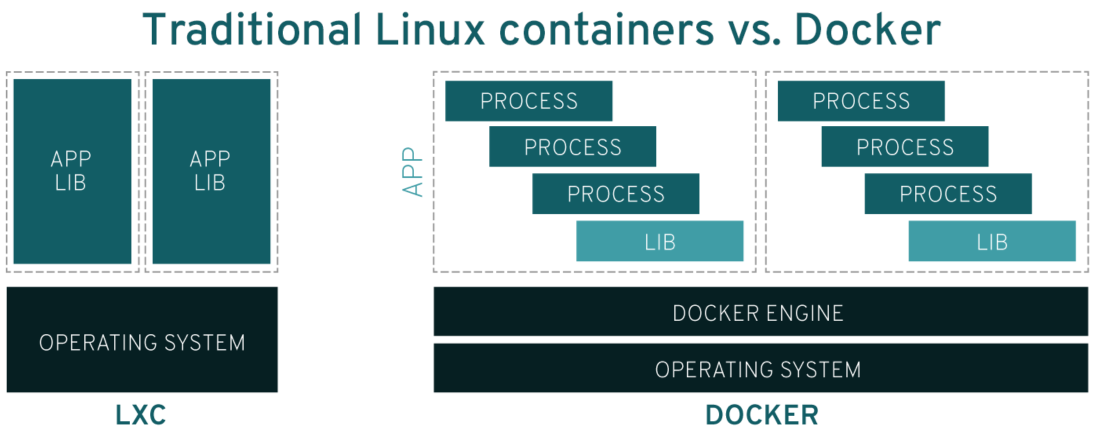


Figure 2.3: Difference between Docker and LCX container architectures

While LXC is more flexible, due to the existence of user tools, and the virtual environments being created at the operating system level, Docker is undoubtedly the better choice between the two, with better options regarding portable deployment across different systems, shared libraries and versioning.

Container Creators - CoreOS Rocket (Rkt)

CoreOS Rocket is another take in the virtualization of containers, it is a containerization software engine created to run applications in an isolated way from the underlying infrastructure. CoreOS Rocket is the main competitor to Docker in the container market [19]. CoreOS Rocket (Rkt) is built upon an open container standard called "App Container" or "appc", which allows Rkt images to be compatible across other container systems. The design objectives of Rkt include simplicity, speed and security. The latter is one of the areas in which Rkt developers believe Docker is fundamentally flawed, and contains many security problems (which are still slowly being corrected to this day), and therefore Rkt was developed to be a more secure container technology, while fixing many of Docker's container model issues [23].

Some of the advantages of using Rkt are:

- Increase of security and reduction of attack surfaces: The simplicity of its design will help keeping a better security and network visibility;
- Compatibility of image formats, meaning that in case a new and better container technology appears, it will be compatible with all Rkt images, as long as it follows the open source container format "appc";
- Hardware-level isolation to each container - provides security in the same level of a true virtualization system, like VMWare;
- Composability - all the necessary tools for the creation and management of Rkt containers should exist but need to be independent;

Docker was chosen over Rocket due to its popularity and general wider community support and larger number of publicly available pre-built containers, that allow for an easier set up of environments. There are some open-source tools that when used with Docker, effectively increase its security, for example Docker Bench, Clair, Cilium, or Anchore [15].

Container Managers - Docker Compose

Compose is a Docker tool created to define multi-container Docker applications via .YAML files; then, with a single command, it can start up all the services, control configurations and set variables (both application's and environment) as needed.

Through .YAML files, Compose can create multiple isolated environments on a single host, while it can also create networks of environments running a service or application. All the relevant configuration for the network and other application-specific settings must be explicitly detailed in the .YAML file. Compose allows the preservation of volume data when containers are created, to minimize build times and to reduce unnecessary resource usage.

At the time of deploying, Compose will check if there were any changes made to the .YAML that reflect in an alteration to the image. If so, it will recreate the image. If not, it will use the same image as the last build, to avoid overhead and extra resource usage on the host system.

Compose has the following functionalities:

- Start and stop services;
- (Re)Build images;
- View the status of running containers;
- View and dump the log output of running containers;
- Execute commands inside the container.

Container Managers - Docker Swarm

Docker swarm is a tool that allows native management of a cluster of Docker containers. It also has the capability to create swarms, deploy application services to swarms, and manage swarm behaviour [17]. The cluster management tools are integrated with the main Docker engine, which allows the user to use the regular Docker Engine Command-Line Interface (CLI) to create a swarm, where applications can then be deployed, without the need of additional software to perform the orchestration of the swarms.

Swarm can also control the scale of running services. If another container running the same service is needed, it can easily be adjusted, either automatically or manually, This can be paired with other swarm functionalities such as load-balancing.

Regarding security, Transport Layer Security (TLS) is enforced in the swarm, to assure authentication and encryption of any in-flight traffic between nodes. As for updates, given the functionalities of Docker Engine discussed previously, it is possible to apply updates to a node or groups of nodes. They can also be reversed to previous versions, or layers.

Container Managers - Kubernetes

Kubernetes is an open-source, portable platform dedicated to the management and monitoring of containerized services, that has an emphasis in automation and active configuration. Google created this project, and later open-sourced it, back in 2014, with all the experience gained in the last decade or so, from running it in production workloads at scale [21].

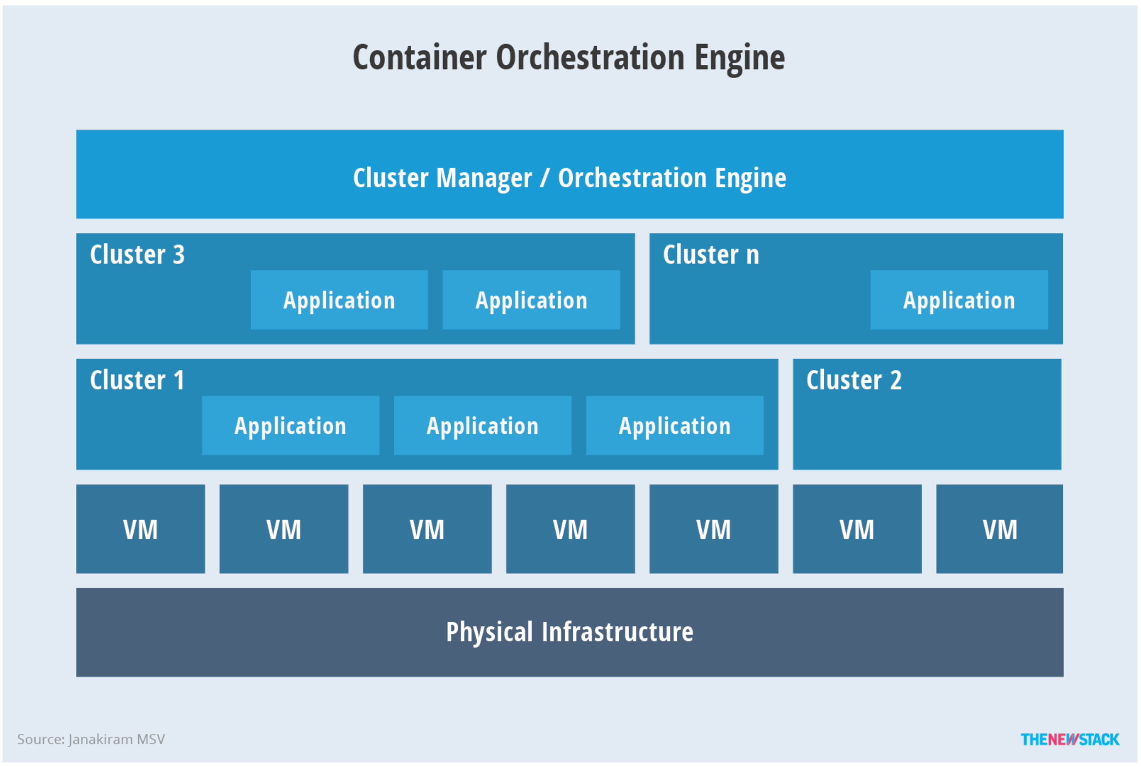


Figure 2.4: Where the orchestration role sits in a containarized architecture [9]

Kubernetes provides a centralized management environment for container solutions. It can orchestrate computing, networking, and storage infrastructure.

A container management such as this is needed to complement Docker, since it doesn't have any dedicated monitoring and cluster management tools, apart from Swarm, which is very limited. It also tracks other useful information, like availability, deploying updates in real time and failure management [9].

Some of the best advantages of using Kubernetes include:

- *High* speed when providing continuous deployment of new features without downtime;
- *Ease* to deploy software updates at scale;
- *Declarative* configuration capabilities, provides the user alerts for erroneous states in the system;
- *Monitoring capabilities*, that provide availability of the nodes of the cluster, while doing self-checks and auto-replacement if needed.

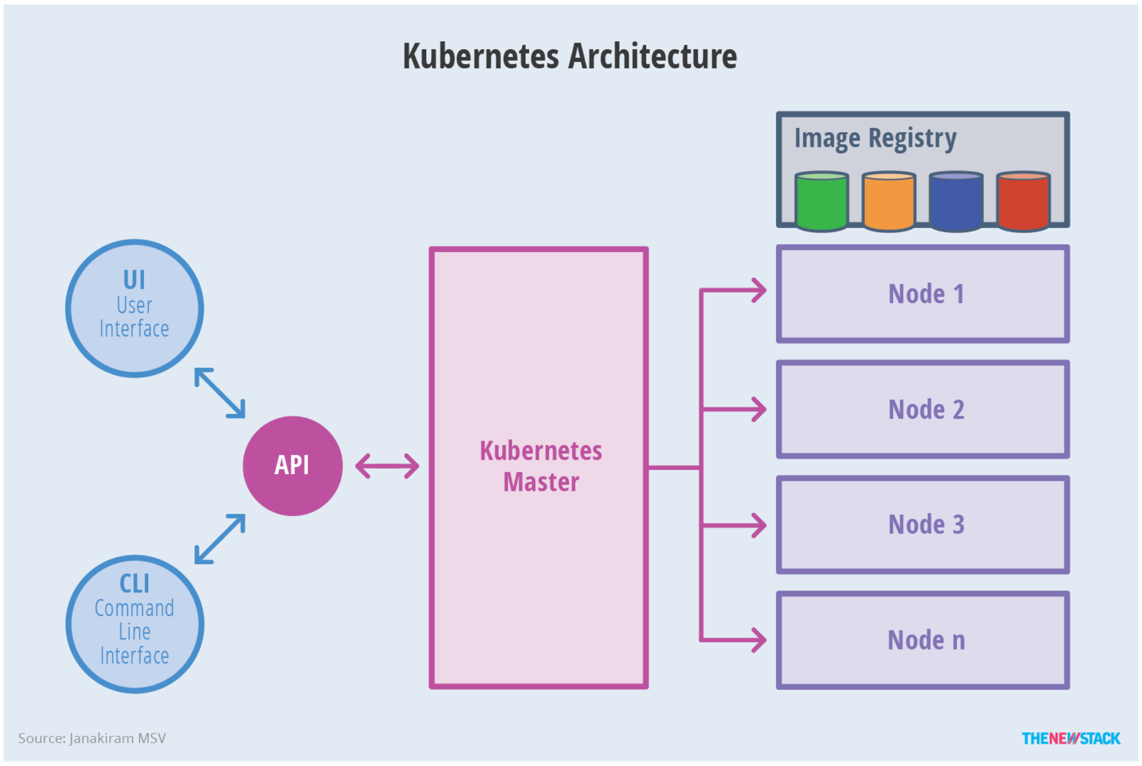


Figure 2.5: The Kubernetes architecture broken down into its components [21]

Kubernetes provides a flexible mechanism for service discovery, where a master node is elected to perform tasks such as exposing the API, scheduling deployments and managing the cluster. As for the remaining nodes, each one runs its code, along with a node that communicates with the master. This node also does logging and monitoring of the entire cluster.

Container Managers - Apache Mesos

Mesos is a cluster manager that allows efficient resource sharing between distributed applications or frameworks. Mesos' objectives are a bit different from the previous cluster managers, since Mesos is oriented towards individually managing a wide array of workloads, like real-time analytics, stateful distributed data services, stateless Docker microservices, etc...

This wide coverage of workloads happens due to its architecture, which allows "application-aware" scheduling, which creates a own-purpose built application scheduler for each workload, that understands its specific operational requirements for deployment, scaling and upgrading [5].

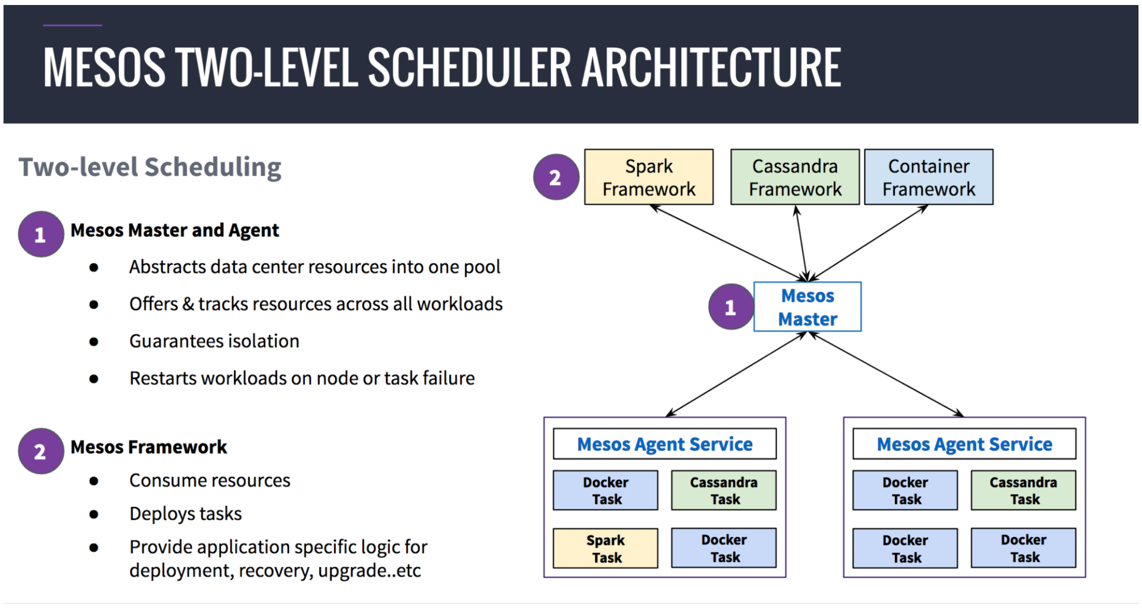


Figure 2.6: Depiction of Mesos' two level scheduler architecture [6]

Mesos' ability to create a purpose-built application scheduler for each type of workload has enticed many companies to use Mesos as a unified platform meant for running microservices and data services together. The common name given to the architecture that runs data-intensive applications is "SMACK Stack" [11].

In the end, docker compose was the chosen tool, due to the containers not having to operate in a managed way, since they'll be deployed directly with all the configuration necessary, and also a GUI will most likely be deployed to manage the cluster itself, so there won't be a need for an extra container manager and its consequent additional overhead to the system.

2.2.4 Message Brokers

A message broker is a intermediary software that often acts as a middleware within a system, that provides communication via data messages between distributed systems or applications. Its flexibility in configuration, whether the brokers themselves or the message parameters and format, have distinguished themselves in the networking and microservices areas in the past few years.

It is a viable option for the communication between heterogeneous applications or systems, as well as proprietary equipment or operating systems. It reduces the complexity of communications in said systems, since the developer has the freedom and flexibility to program many of the aspects of the message itself, and how it is going to be delivered, as well as program scalability features, like routing patterns within the message broker node mesh (cluster) and/or monitoring features.

Typically, these messaging systems allow two types of asynchronous messaging:

- The message queue style, also known as point-to-point, where there can be many messages produced to the same queue, but only a maximum of one consumer will retrieve each message;

- The publish-subscribe style, which uses topics to allow the submitted messages to be received by more than one consumer. It acts as a 1-to-n queue, in which all consumers subscribed to the topic will have access to the same produced messages;

The following subsections will describe the most relevant message brokers currently available, and detail some of their advantages.

Message Brokers - Apache Kafka

Apache Kafka is a distributed streaming platform which uses the publish-subscribe style of messaging. This means its main functionalities are [1]:

- Act as a message broker, in which other applications may publish and subscribe to streams of data;
- Process in-flight streams of data;
- Store streams of data for fault tolerance purposes, in categories called topics.

Kafka Consists of five main elements: the Broker(s), the Topic(s) The Zookeeper(s), the Producer(s) and the Consumer(s):

- The Brokers (or simply, the nodes), are places where data will be received, processed, then forwarded. Each broker can have multiple topics, which act as a separation between "conversations", and can be used to filter unwanted entities from accessing messages;
- The Topics are feed names in which messages are published by producers and then retrieved by consumers. It contains messages, which are byte sequences that can take any format desired by the programmer.

  There is also the matter of partitions and replication: Each message in a partition is assigned and identified by its unique offset. Partitions allow consumers to read from topics in parallel from one or more brokers.

  As for replication, it is implemented at the partition level. Each partition usually has at least 2 replicas, meaning the messages that go through are replicated to other nodes to assure data availability;
- The Producer is the application that sends messages into a broker's topic;
- The Consumer is the application that consumes messages from a broker's topic;
- The ZooKeeper, used for electing a controller. The controller is going to be elected from one of the nodes and he's responsible for maintaining the node hierarchy for all partitions. The ZooKeeper also keeps track of which clusters (and nodes within the clusters) are active, what topics exist, how many partitions it has and where are its replicas located. Apart from that, the ZooKeeper also deals with establishing quotas and enforcing authorization via Access Control Lists on the brokers.

The number of brokers can be scaled up, and they can even form clusters, where tasks like load-balancing and redundancy can be configured, as well as scaling the maximum throughput of the message processing and delivering to the needs of the application or system.

More Zookeepers can also be set up, to either help with the replication process by load-balancing, if the processing gets too heavy for a single instance because of a very high message throughput, for example. They can also be used for redundancy and availability, since the zookeeper is often kept in a separate container than the brokers themselves, and in the case of a zookeeper crash, the whole cluster will stop working, unless another zookeeper takes its place.

Kafka also possesses an Application Programming Interface (API) written in Java called Kafka Streams, which allows for the creation of Java applications or microservices that directly interact with data from Kafka, and allows them to be highly scalable, flexible and fault-tolerant. It combines the ease of programming and deploying of Java applications with the benefits of the Kafka cluster functionalities, for example, data aggregation and treatment at the broker level.

A stream is a undefined influx of data that occurs in real time. This data can then be manipulated by the Streams API for many different purposes, for example processing and windowing/aggregation.

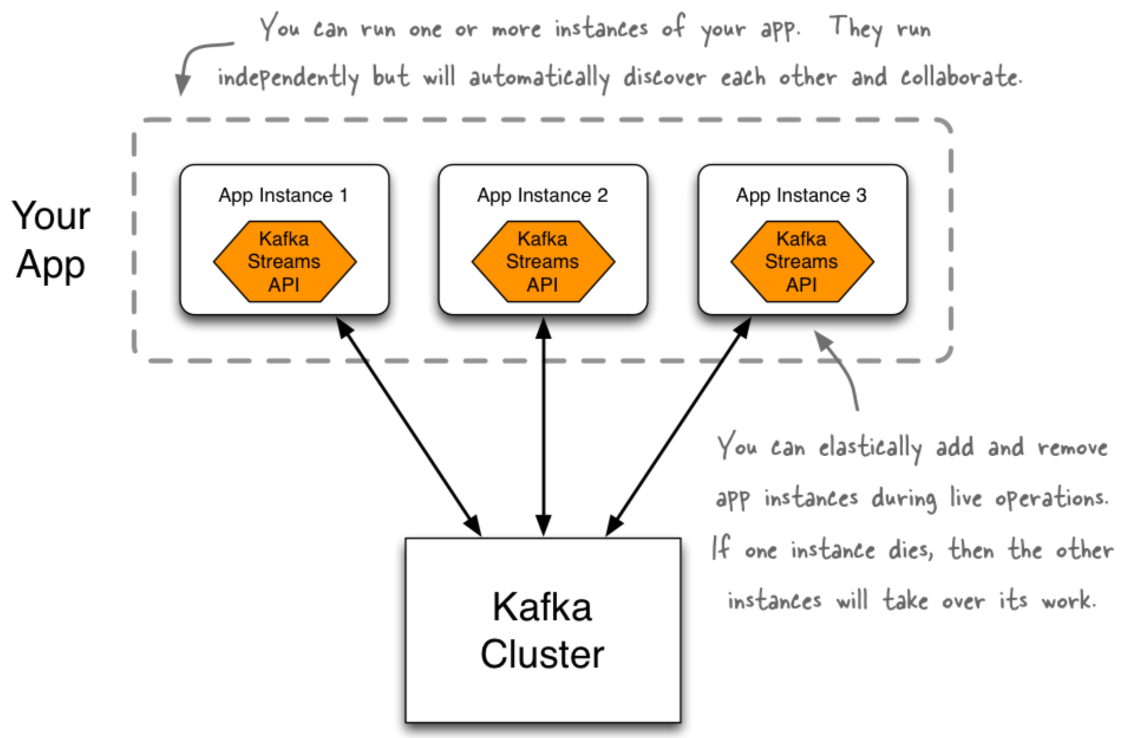

You can run one or more instances of your app. They run independently but will automatically discover each other and collaborate.
Your App
App Instance 1
Kafka Streams API
App Instance 2
Kafka Streams API
App Instance 3
Kafka Streams API
Kafka Cluster
You can elastically add and remove app instances during live operations. If one instance dies, then the other instances will take over its work.

Figure 2.7: Kafka Streams basic functional architecture [2]

It was originally developed to allow the stream processing to branch out of the Big Data environments into the mainstream application development scenarios. Stream processing is a paradigm that allows some applications a limited form of parallel processing. The API also allows the developers to run one or more instances of the same application, and they will automatically work with each other collaboratively, and provide fault tolerance of the general application scope.

Some of its key aspects include [4]:

- Application scaling to small, medium and large environments;
- Fault Tolerance;
- Stateful and Stateless processing;
- Data aggregation;
- Low programming barrier to entry;
- Ease of deployment;
- No external dependencies besides Kafka;
- Runs on private/public clouds, containers, etc...;
- Possesses the low Kafka latency and its high throughput;
- Automatic handling of out-of-order data;
- Supports the security mechanisms found in Kafka (Encryption, authentication and authorization).

These characteristics allow the developers to build powerful, yet simple applications and microservices with Kafka as its messaging backbone, without requiring a dedicated cluster to operate, all that while inheriting all the main Kafka benefits.

Message Brokers - ActiveMQ

Apache ActiveMQ is a open-source, general purpose message broker that supports a fair number of messaging protocols such as Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT) and Simple (or Streaming) Text Oriented Message Protocol (STOMP). It allows communication from more than one client or server. It supports more complex message routing patterns than other message broker software. It is mainly used for integration in services that use Service Oriented Architectures. It uses several strategies to allow high availability, achieving true replication using Apache Zookeeper. It is widely regarded for its configuration flexibility in the enterprise world.

It seems to be favored by some developers, due to its straightforwardness, great compatibility with Java Virtual Machine (JVM) related languages and its integration with Java Message Service (JMS). It allows for the use of a Java Management Extensions (JMX) console to visualize topics and queues in the broker, even create and send messages without the need of a producer. More brokers can be added under load in a seamless way for the users already using it, meaning that the queue name being used won't change.

The use of persistent messages can be configured, allowing the brokers to keep message data stored in their own database instance, which prevents data loss from possible system (or broker-specific) failures.

Since there are many pre-built Dockerfiles to quickly create and deploy containerized ActiveMQ environments, without the need of extensive or complicated configuration, it is very used for Research and Development purposes at enterprise levels.

Message Brokers - RabbitMQ

RabbitMQ is a distributed-messsage queue system, because it is normally run in a cluster where the queues are distributed by the nodes, while keeping a single logic broker visible to the clients. It can also optionally replicate data to ensure fault tolerance, high availability and reliability. RabbitMQ supports either asynchronous or synchronous communication, as needed. Messages are sent from the producers to the exchanges, then to the queues, where finally they'll be consumed by the consumers [18]. Decoupling producers from queues by using the exchanges ensures that producers aren't overloaded with extra hardcoded routing decisions, and therefore decreasing the overhead on the system itself.

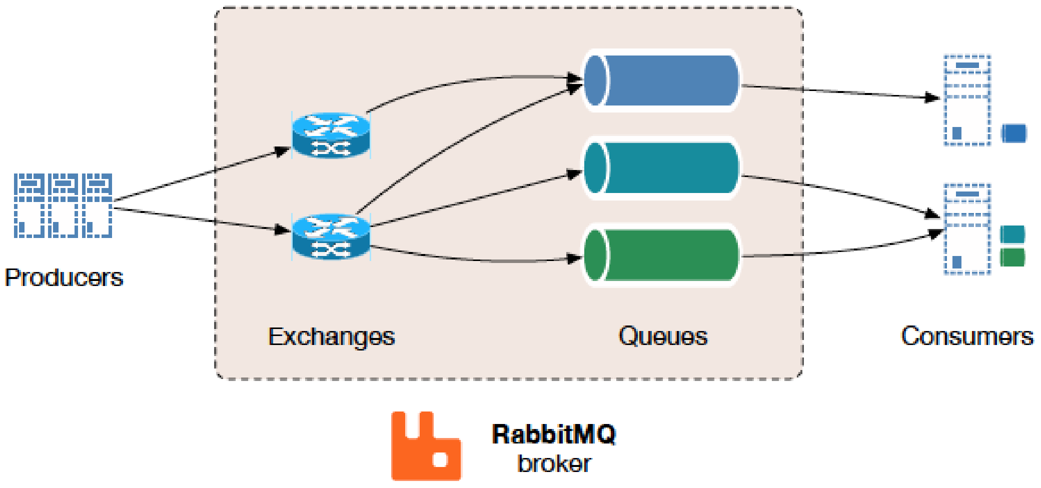


Figure 2.8: Simplified RabbitMQ architecture [18]

It supports protocols such as AMQP, STOMP, MQTT and HyperText Transfer Protocol (HTTP). The difference in RabbitMQ is the fact that it uses its distributed nature to its advantage, via a complex routing capability, to employ a fast, scalable and reliable messaging system [18].

The plugin system allows RabbitMQ to extend its capabilities in many different ways, and allows developers to write their own plugins to fit their needs. Plugin use advantages include, for example, enabling the system or application to access internal RabbitMQ functionalities, which are not normally available by choosing any of the supported protocols. These may also be used, for example, for system state monitoring and node federation, which consists in the transmission of messages between un-clustered brokers. It includes a management User Interface (UI) that allows control and monitoring over the brokers, as well as tracing support, for debugging problems via logs.

The message broker chosen to be used in this project was Apache Kafka, given its high throughput, ease of deployment and containerization, and dedicated API used to develop applications that take direct advantage of its features.

2.2.5 Message Formats

There were two considered types of data interchange format that the events sent to the dashboard would be created with, namely:

JSON

JSON, or JavaScript Object Notation is a lightweight data-interchange format used to represent data structures in an universal way, easily interpreted by many of the programming languages used, such as C, C++, C#, Java, JavaScript, Perl, Python, and others [7].

It achieves this simplistic format by using name/value pairs and lists to describe array structures. This means the message can then get interpreted as an *object*, which is vastly used in the languages mentioned.

Given this, messages in JSON format are highly flexible and can be easily modified or adapted to the programmer's needs.

Avro

Avro is an open-source serialization system that employs communication compatibility across different frameworks, systems and programming languages [22].

Avro is commonly used on Kafka-based systems since it allows the definition of schemas, and is compatible with popular message formats, like JSON. Schemas are user-created message format profiles that are compared to the messages at the time of arrival to the Kafka broker. These are put against the defined schema(s), and if they pass its integrity check, they're then forwarded to its destination.

They allow for more robust stream processing systems, data clarity and compatibility. They also protect downstream applications and consumers from erroneous and malformed data, only allowing data that matches the schema to leave the broker.

In the end, JSON was chosen mainly due to its flexibility paired with the fact that the dashboard would receive some data formats that were undefined, which will be explained in further chapters.

2.2.6 Summary

Regarding solutions that are similar to the one being developed, there are a few available, however they either require dedicated hardware or are limited to certain operating systems, which would not be adequate to the problem this project intends to solve.

Tables 2.1 and 2.2 try to summarize the features of both container creators and message brokers, so that all important characteristics can be easily compared between them.

Functionality Tool	Allows image layering/ Version Control	Has good container security	Open-Source	Containerization Limitations	Scalable	Lightweight?	Helpful Documentation	Compatibility with the image formats, following the open source container format "APPC"
Docker	Yes	No, still gets regular security updates	Yes	Applications can run at scale, in production, on VMs, bare metal, OpenStack clusters, public clouds and more [19]	Yes	Yes	Yes, even though it seems to be out of date sometimes	Some
LXC	No	No, but large user base contributes to its development	Yes	Allows virtualization of applicaitons at the OS level, inside a Unix Kernel host. It can be used to virtualize applications or entire Unix-based operating systems.	Yes	Yes, because of the virtualization at the operating system level technique	Yes, the large user base helps founding a good set of documentation	Some
CoreOS Rocket	No	Yes, was designed with security in mind	Yes	Created to run applications in an isolated way from the underlying infrastructure. Its objectives include simplicity, speed and security.	Yes	Yes, but has additional overhead due to the container security mechanisms	Yes	Most

Table 2.1: Feature comparison - Docker vs. LXC vs CoreOS Rkt

Functionality Tool	Easy to configure / Set up	Requires other applications?	Open-Source	Supports JMS?	Scalable	Messaging Types Supported	Has management GUI	Best used for:
Apache Kafka	No	Yes, Zookeeper	Yes	No, uses own non-standard protocol and clients (3rd party clients exist)	Yes (very good horizontal scaling)	Asynchronous	No (with 3rd party options available)	Real-time data streaming
RabbitMQ	Yes	No	Yes	Yes, via plugin	Yes	Asynchronous, Synchronous	Yes	Integration between applications/services (especially in a Service Oriented Architecture)
Apache ActiveMQ	Yes	No (can use Zookeeper to achieve replication)	Yes	Yes	Yes	Asynchronous	Yes	Backbone for an architecture of distributed applications

Table 2.2: Feature comparison - Kafka vs. RabbitMQ vs ActiveMQ

This page is intentionally left blank.

Chapter 3

Research Objectives and Approach

This chapter details what the solution intends to solve, what is its place in the architecture, as well as what are its requirements regarding software/hardware components, tactics or strategies. After that, any reference architectures or software that will be adapted or used to the benefit of this project.

3.1 Objective

The software will take the form of a dashboard, that will display, in real time, relevant information about status or errors sent from all other equipment in the 5G platform, and also the underlying system that will handle and feed the dashboard with the messages sent by the other hardware in the network. This message data flow that feeds the dashboard must be unidirectional, as it should only serve as a monitoring point with no control over the other machines in the network.

The software will also act as a security mechanism to the overall architecture, allowing the network administrators to get notified quickly of problems that may occur with the platform's hardware or software, or even prevent/stop network attacks.

First of all, we'll need to define some concepts to thoroughly understand the main components of the system and how they'll function.

3.1.1 Events

An event consists of one or more (in case of aggregation) messages sent to the dashboard, from other nodes in the 5G network, which contain information about an occurrence, or diverse statistical metrics from that system that are to be displayed to the operator.

Statistical or numerical data can be altered or processed before it is sent to the dashboard, with the objective to simplify the visualization process of this data, and to not overwhelm the operation with raw, untreated information.

3.1.2 Pre-processing

The events that arrive from the Kafka brokers will suffer a form of pre-processing, which will be done at the middleware level. The middleware will be taking care of unpacking the data, which follows a standardized JSON format, processing and even aggregating the events at times, to allow simple, more intuitive visualization of the data at the dashboard level.

The pre-processing could include, for example, averages, ratios, time-related statistics, and other types of data transformation techniques that are relevant for a good monitoring of the platform.

3.1.3 Dashboard

The dashboard's role is purely to visualize data that is being sent from other 5G network nodes and later on processed by the middleware. It has no control over any of the nodes that it receives information from.

After the data has been processed, it will then be decided which is the best way to output this information. So, by using a certain type of chart or types of visualization that are specifically catered to the data that's being shown, this will allow the operator to quickly obtain the most information of what's happening in the 5G network.

3.1.4 Architectural Considerations

Since the 5G architecture uses different types of equipment, even proprietary, it is necessary for the communication method not to be impaired by the fact that most hardware/software is of proprietary nature. This calls for a method of abstraction of the messages, and a definition of a common message structure that every equipment will use when sending data to the dashboard, otherwise it would be unfeasible for the dashboard to interpret that many messages, due to the amount of different types of messages, and their specific encoding.

The dashboard is intended to be a central point of the project, being connected to the large part of the platform, assuming the role of security monitor.

All the components that will send information to the dashboard are directly out of my control and are responsibility of many other project participants, however, the middleware must be able to support the event flow throughput sent by all the other nodes that send data to the dashboard.

3.2 Requirements

The 5G platform is expected to be used by a large mass of users simultaneously, and its nature implies that some general, non-functional requirements must be met in order to integrate the project in it, such as:

- Adjust to a high client volume - Given that many users will use this platform at the same time, there must be dynamic scalability measures in place in the event that too many messages are in queue. Otherwise, this could affect the monitoring system's accuracy and data reliability;
- Ignore proprietary hardware/software barriers & allow heterogeneous application communication - The solution must be able to send data to the dashboard in a

standardized way, this being agreed upon prior with all the related entities, so that extra processing overhead can be reduced at the dashboard.

Therefore, the dashboard won't have to be programmed with the logic to decipher multiple types of messages, making it a lighter application, and increasing overall throughput of data, and consequent data accuracy and reliability;

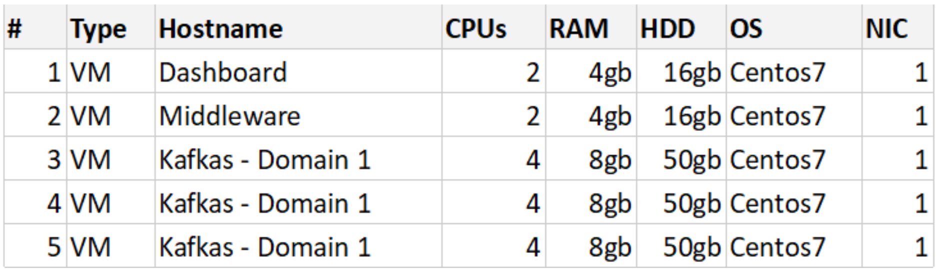
- Possess a high throughput capability - The middleware that will take care of message pre-processing and forwarding to the dashboard must have a high message throughput (which will be scalable if the first requirement in this list is met), to handle the large number of events that will be created, and to make sure the messages won't be bottlenecked by raw message processing speed;
- The system should keep working in an independent way of the general platform, and it should have recovery / redundancy mechanisms in case of general software or hardware failure.

  The dashboard system should have an equal or higher reliability / upkeep than the rest of the 5G platform, since should it fail, the administrators' reaction time to a failing component would be much higher, decreasing overall up-time, diminishing profit and overall quality of service / experience of the platform for its clients;

The dashboard should offer an interface that displays all relevant 5G platform information regarding the network infrastructure and assets, its performance, security indicators and events. Therefore, and for a more complete comprehension of what the system should deliver, some functional requirements must also be listed:

- The dashboard must be connected, at all times, to the remainder of the platform, preferentially by redundant links, so that the lost number of events and overall downtime is greatly diminished;
- The dashboard must be able to retrieve and interpret event data from the middleware, and then display it, in an intuitive, easy-to-read form to its users;
- The dashboard should be optimized to be able to handle a large influx of events per unit of time, without affecting data integrity and avoiding out-of-date notifications;
- The implementation of the tool should be done with security in mind, not to compromise any sensible platform data to any attackers on unauthorized personnel. For this reason, the incoming data will run through integrity checks and profiling, so that only verified platform nodes are allowed to send messages to the dashboard.

The more technical, numeric requirements are stated in the table below:



#	Type	Hostname	CPUs	RAM	HDD	OS	NIC
1	VM	Dashboard	2	4gb	16gb	Centos7	1
2	VM	Middleware	2	4gb	16gb	Centos7	1
3	VM	Kafkas - Domain 1	4	8gb	50gb	Centos7	1
4	VM	Kafkas - Domain 1	4	8gb	50gb	Centos7	1
5	VM	Kafkas - Domain 1	4	8gb	50gb	Centos7	1

Figure 3.1: Hardware Requirements for the Event Collector System

Given the array of both software and hardware products depicted in the state of the art, virtualization, more specifically, containerization will be a highly valuable tool in meeting the performance demands of the 5G platform. Docker was chosen as the software to be used for containerization, while Apache Kafka was chosen to be the message broker to be used for communication. It was chosen for Kafka to have three dedicated VMs to not limit the scalability options of the Kafka cluster, since the platform is expected to have a high number of users, therefore increasing the general number of events that are sent to the dashboard.

The use of containerization allows the infrastructure owners to re-utilize resources, rather than having to buy brand new equipment, meaning that in some ways, the same equipment used for the hosting of the 4G platform may be used in the hosting of the new 5G platform. This reduces costs of implementation by a significant margin.

The Dashboard will be modified to the needs of this project to reflect the data that it will receive from other nodes in the 5G platform. This dashboard will connect to a middleware that will take care of forwarding the already processed data [3].

For testing purposes, it will also be necessary to develop a small application that will serve as an event generator. This application will help test both the message channels, as well as the data received by the Dashboard. The process of extracting data from the messages will also be possible with this tool, which will be using a similar format to the proposed one.

Finally, regarding the standard message format to allow communication between heterogeneous applications, a draft of a proposal will be made, which will be delivered to all entities involved in the project, to be discussed. The structure should be based on a JavaScript Object Notation (JSON) format, since it allows messages format to be very flexible, which is something required due to the unknown nature of the metrics that will be sent to the dashboard by its client nodes.

In the end, Docker was chosen to be the containerization software used in the project, since it allows for massive horizontal scaling of the message brokers, which will be very necessary to the architecture, given the amount of clients that will use it, as well as giving developers the ability to version control docker images, due to its layering system.

Regarding message brokers, Apache Kafka was chosen, given that it has a significantly higher throughput than the rest of the available choices, and is also able to scale horizontally up to the needs of the architecture. The fact that the message structure itself can be created from scratch or heavily customized is also a very important factor in this architecture, since it will have to operate with data coming from machines with different operating systems and even proprietary hardware and software.

3.3 Reference Software

Aside from the rest of the platform, which will be monitored via incoming events sent by the nodes, it is also important to get some data on the Kafka cluster and its status, specifically data on the topics that are being used to feed the middleware data.

The dashboard receives data from Prometheus to obtain detailed statistics regarding the Kafka topics and its usage over time.

Prometheus is a open-source monitoring system and toolkit, written in GO, that was originally built at SoundCloud.

It is composed of [16]:

- Prometheus Server, which stores time series data;
- Client Libraries, to instrument application code;
- An Alert Manager, to notify and manage users of alerts and events;
- Graphical Exporters for services like StatsD, Graphite, HAProxy, etc...;
- Various other support tools.

While its main features include:

- Multi-dimensional data model with time series identified via key-value pairs, supported by PromQL, an adequate query language for this multidimensionality;
- Server node autonomy, removing the need of reliance in distributed storage;
- Time series collection over HTTP;
- Target discovery via specific service discovery or static configuration;
- Graphing and dashboard data exporting support.

This page is intentionally left blank.

Chapter 4

Preliminary Work

A number of experiments and scenarios had to be built in order to make sure the chosen software worked together as a whole without problems.

4.1 Familiarization with the project and its scope

This included learning about the 5G project and its scope, studying and reading articles and other information sources on the web about all the relevant existing technologies, as well those that the project was going to use, such as containerization, message brokers, microservices, container management, heterogeneous application communication, the dashboard itself, and how to link all three together to create the solution required.

As previously stated in the first chapter, the 5G project is subdivided in multiple layers of development, in which I was assigned a position in the second layer, or PPS2, more precisely, building a dashboard system that allows monitoring data to be centralized for security purposes.

To get a better comprehension of the scope of the project, and its architecture, I participated in a meeting with nearly all the companies involved in the 5G project. In this meeting it was presented the current state of the project, what was being developed and how was it being done. Here I got a superficial idea of how the architecture proposal was being molded to the requirements of the technology and what were the problems being faced by different parts of the team.

After this meeting, it was then decided that the solution to develop was intended to be run in containers, therefore using Docker, and to use Apache Kafka for the message forwarding part, since it offered a higher throughput than its competition. I then began tinkering with some virtual machines, setting up some virtual environments for testing purposes.

Create a test environment to containerize a Kafka solution, so it would be according to the following architecture:

The test environment was made using Dockerfiles of solutions made public by other users, specifically solutions that would integrate both the Kafka and Zookeeper, since they are two separate pieces of software, however, Kafka won't run without an associated zookeeper instance.

The Dockerfile that contains the environment I used, which is currently hosted at `https://github.com/confluentinc/cp-docker-images`, allows for the deployment of a single

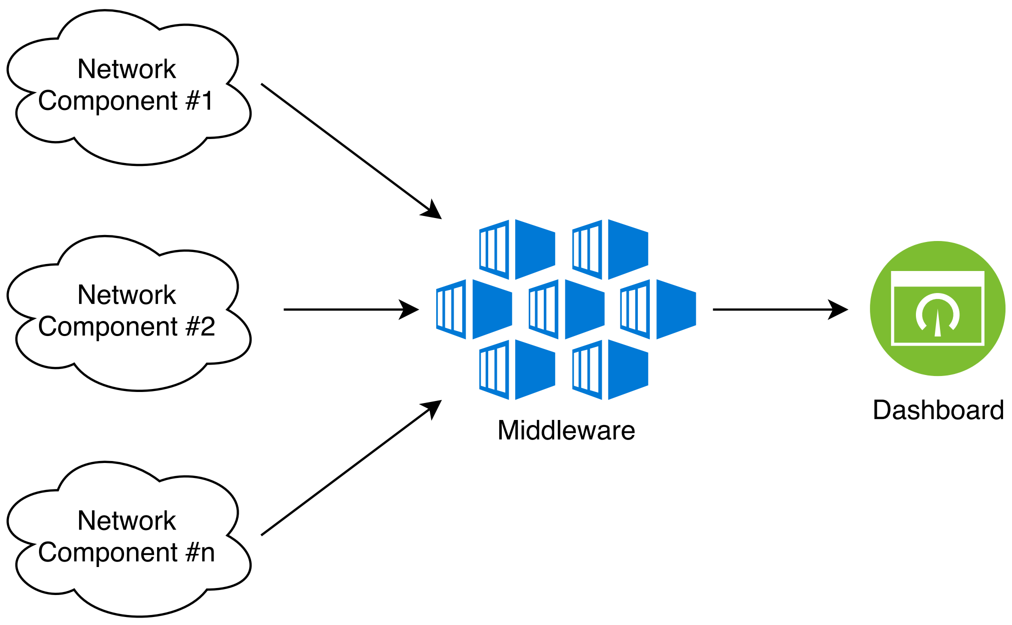


Figure 4.1: A view on the proposed architecture

Kafka instance and a zookeeper, each in a Docker container. These can then be scaled up, by creating new brokers, one per container directly from the original broker, that associate themselves to the cluster. It also allows to perform partitions and message replication between them.

Regarding its configuration, the environment is highly flexible, and some parameters can be configured directly in the dockerfile, as well as environment variables.

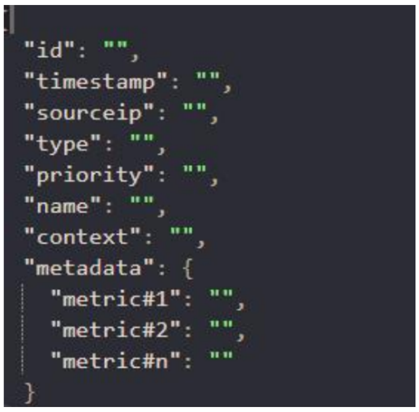
In this test environment, a Kafka cluster was deployed, together with an instance of Zookeeper, since it is a mandatory part of the Kafka architecture. It was then attempted to send messages from a Kafka producer to a topic using the Synthetic Event Generator, and consume them using a Kafka consumer in the middleware.

4.2 Creation of a Message Format Standard Draft

Since the whole point of my work is to create a mechanism to allow multiple different applications to send information, following a standard, to the dashboard, it was vital to define a message format that would be used by every machine when sending data to the dashboard.

This way, a lot of overhead could be avoided as well as other potential problems of incompatibility, while still taking advantage of all the Kafka functionalities.

The proposed message format was written in JSON. While JSON is considered to be slightly slower at message creation and decryption, it allows for a highly flexible package where many types of information can be put in. Pairing this with the high throughput of Kafka allows for the extra compression/decompression overhead to be less noticeable.



```
{
"id": "",
"timestamp": "",
"sourceip": "",
"type": "",
"priority": "",
"name": "",
"context": "",
"metadata": {
  "metric#1": "",
  "metric#2": "",
  "metric#n": ""
}
```

Figure 4.2: A view on the proposed message format

The following list details each field:

- id: Unique event Identifier;
- timestamp: Temporal Identifier of the Event, according to the ISO8601 norm;
- sourceip: The source Internet Protocol (IP) of the event;
- type: Type of Event (INFO, EVENT, ERROR, etc...);
- priority: How important the event is (LOW, HIGH, CRITICAL);
- name: Name of the event. Assigned by the administrator;
- context: Info detailing the context of the event;
- metadata: Object containing metrics that are sent for processing.

4.3 Synthetic Event Generator

Since it was not only needed to test the message flow within Kafka itself, but also its throughput and scalability, considering that the messages would all be coming from an outside source, an application that would generate events and feed the Kafka topics was created.

This application was written in Python, since it allowed me to create it quickly, given it already has Kafka libraries available online that allow the direct connection of a Python-based Kafka producer/consumer to a Kafka broker.

4.4 Middleware Integration

The middleware that feeds raw data to the dashboard will need to be integrated so that it consumes the messages from the Kafka topic, and read the JSON formatted message onto data objects it can use to feed the dashboard. Since this is code developed by someone else, it implies that I will have to spend time studying it and figure out a way to adapt it into the project's needs. This application is written in NodeJS which, like Python, has the tools to allow Kafka communication. This middleware is inspired from the Advanced Tools to assEss and mitigate the criticality of ICT compoNents and their dependencies over Critical InfrAstructures (ATENA) project middleware, which is also feeding its dashboard.

4.5 Security mechanisms of Apache Kafka

After a test environment was created and working properly, I studied what security mechanisms Kafka offered, which are detailed below:

4.5.1 Context

Apache Kafka is a middle layer message broker, allowing systems to share real-time data with each other through queues called topics. By default, Kafka has no security configured, which means any user can read and write from and to any topic and get the messages within, even if they're not the messages' destination. This compromises basic security standards in modern networks. Kafka however, does allow for security mechanisms within its cluster, mainly encryption, authentication and authorization. This sub-chapter aims to explain the different security mechanisms available to Kafka, as well as the advantages they bring to an enterprise-level network.

4.5.2 Components

Apache Kafka security revolves around the following three main components [1]:

- Encryption of in-flight data using Secure Socket Layer (SSL)/TLS: Since the default Kafka communication between consumers and producers uses no form of encryption (plain text), it is a good idea to use encryption methods to avoid being target of common network Man in the Middle (MitM) attacks such as spoofing or sniffing;
- Authentication using SSL or Simple Authentication and Security Layer (SASL): This allows both producers and consumers to authenticate themselves onto the Kafka cluster, which verifies their identity. This prevents unwanted access from unknown third parties to either the producer or the consumer;
- Authorization using Access Control List (ACL): Typically placed after the authentication, authorization via Access Lists manages which users can access which topics, be it to write, read, or both. Without these, even if the user is authenticated, he can just access whatever topics he wishes with no kind of control;

4.5.3 Encryption

Any Kafka-transmitted data, by default, will not be encrypted. Therefore, there's the need of encrypting the data between consumer and producer. This can be done with SSL certificates, when only consumer and producer can verify the content of the encrypted packets with their certificates, signed by a verified authority. This method can put a halt to most man-in-the-middle attacks, because they don't have access to any of the certificates used to encrypt the packets.

However, encryption has a cost, and that comes in the form of CPU usage overhead, which can in turn slow the overall Kafka throughput, since it has to leverage the CPU for the clients' messages' and for the broker itself to take care of the encryption and decryption process. Still, using for example Java 9 over Java 8 for this endeavor, performance cost is decreased by a substantial amount [1]. There is a way around the increased CPU usage in the cluster, however. By delegating the responsibility of encrypting and decrypting messages to the producers or consumers - this performance overhead is only applied once, and does not affect the overall performance of the Kafka cluster.

4.5.4 Authentication

There are two ways to authenticate Kafka users, SSL and SASL.

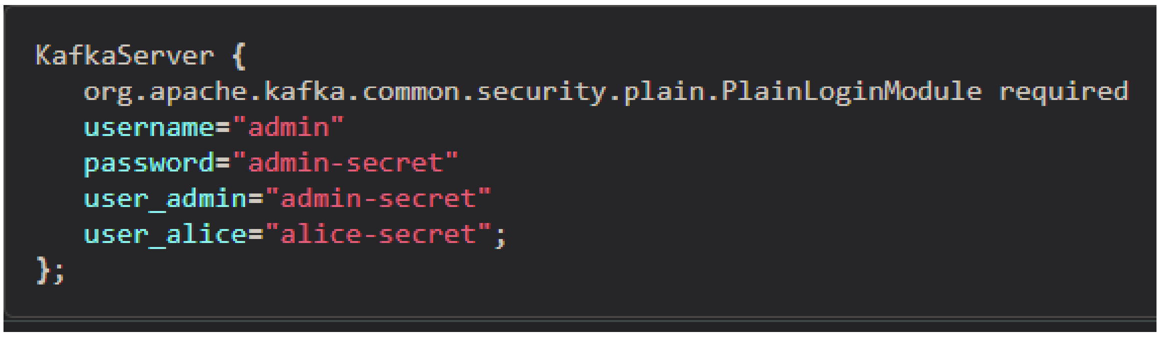
SSL Authentication is done using a feature from SSL called two-way authentication, by supplying the client endpoints with SSL signed certificates by a trusted certificate authority. These will be checked at connection time to match against a SSL certificate stored in the server.

SASL (Simple Authorization Service Layer) is the service which separates authentication from the Kafka protocol, being most popular with Big Data Systems.

Along the many types of SASL available, the following are supported by Kafka:

- PLAINTEXT: Classic username/password combination. These are stored inside a file, typically with the .jaas extension, which are kept in the Kafka brokers in advance. The downside is, each change to the file requires a restart to the Kafka Cluster to take effect. It should be paired together with SSL in-flight encryption to avoid credentials being sent in plain text over the network.

  The following figure shows an example of a .jaas file, where "username" and "password" are the credentials used for inter-broker communication, and inside that broker, two users are defined: "alice" and "admin", each with his own password;



```
KafkaServer {
   org.apache.kafka.common.security.plain.PlainLoginModule required
   username="admin"
   password="admin-secret"
   user_admin="admin-secret"
   user_alice="alice-secret";
};
```

Figure 4.3: .jaas authentication file example

- SCRAM: Classic username/password combination with a challenge, hence the name, Salted Challenge Response Authentication Mechanism. Like PLAINTEXT, the account data ins stored inside a .jaas file in the broker. However, for the challenge to work, all clients and servers need to support the Secure Hash Algorithm - 1 (SHA-1) hashing algorithm. SCRAM, unlike CRAM-MD5 or DIGEST-MD5, is independent from the underlying hash function. The P8KDF2 mechanism used by SCRAM also increases its resistance against brute force attacks;

- Generic Security Services Application Program Interface (GSSAPI) (Kerberos): This uses a Kerberos ticket mechanism, which is a very secure way of providing authentication. It is a great choice for big enterprises as it allows the companies to manage security from within their Kerebros Server. Additionally, all communications are automatically encrypted with SSL. The only drawback of this approach is the difficulty of configuration and implementation presented;

- OAUTHBREAKER (KIP 255 - Kafka Improvement Proposals): This will allow the cluster to leverage OAUTH2 tokens for authentication of users. OAUTH2 is a flexible framework with pluggable implementations and flexibility in its configuration to provide clients with a secure way of authentication. This proposal has currently been accepted.

4.5.5 Authorization

As per regular user interaction in a system, there are times in which users cannot interact with certain elements, due to the lack of permission to do so. Access lists aid the Kafka Cluster security in a way that it will let the administrators filter who can read and/or write at a given topic.

Currently with the packaged SimpleACLAuthorizer included with Kafka, ACLs are not implemented to have Group rules or Regular Expression (REGEX)-based rules. Therefore, each and every security rule must be written in full, except for the cases in which all are affected. These ACLs are stored in the zookeeper, so it becomes a liability in the network, since it becomes much more susceptible to attacks. By storing the ACLs externally, we're effectively increasing the attack surface, therefore increasing security.

To add ACLs, consider using the "kafka-acls" command, since it is oriented to managing Access Lists regarding the Kafka consumers and producers.

An example of adding permissions so that the user "alice" can produce messages to the topic named "test":

```
$ kafka-acl --topic test --producer --authorizer-properties
zookeeper.connect={IP}:2181 --add --allow-principal User:alice
```

Since this method is pretty inconvenient and hard to use in the long run or for creating a larger rule set, there is a small utility called the Kafka Security Manager, which runs in a separate docker container. This allows the user to provide the ACLs from an external source, while synchronizing them continuously with zookeeper.

Feature	Type	Pros	Cons
Encryption	Plain Text	No extra performance impact on the CPU	In-flight data is not encrypted, and is vulnerable to attacks such as MitM and sniffing
	SSL Certificates	Provide encryption of the in-flight data by the use of certificates signed by a certified authority.	Extra taxing on the CPU (This can be lessened by using Java version 9 over 8)
Authentication	Plain Text	Easier to set up and to deploy. Can be used simultaneously with SSL for encryption of in-flight data	User data is unencrypted in a raw text file. Any changes done to this file require a cluster restart to take effect.
	SCRAM	Easier to deploy and set up. Supports a Salt Challenge using SHA-1, which must be supported by clients and servers alike. PSKDF2 mechanism increases the resistance against bruteforce attacks.	User data is unencrypted in a raw text file. Any changes done to this file require a cluster restart to take effect.
	GSSAPI (Kerberos)	Most secure of all the options. Advised for enterprise use. All communications automatically encrypted with SSL	Requires an Active Directory (Kerberos Server) Hard to configure and set up.
	OAUTHBREAKER	Allows the leverage of OAUTH2 tokens for authentication. Flexibility of configuration in the OAUTH2 framework.	Still in proposal phase, but has been accepted. No existing implementations yet.
Authorization	SimpleACLAuthorizer	Comes bundled with kafka. Allows the creation of authorization rules.	Each rule must be written in full. Doesn't allow the import of external ACLs. The ACLs are stored in the Zookeeper, which may become a liability in the network.
	kafka-acls	Specifically oriented to managing ACLs of kafka consumers and producers.	Each rule must be written in full. Doesn't allow the import of external ACLs. The ACLs are stored in the zookeeper, which may become a liability in the network.

Table 4.1: Security Characteristics - Summary

This page is intentionally left blank.

Chapter 5

Architecture

5.1 Proposed Architecture

The 5G architecture requires some high-profile changes in the way the network is managed and monitored, given its flexibility, as well as the introduction of technologies such as Network Function Virtualization Network Functions Virtualization (NFV) The general architecture for the PPS2 components and tasks is divided in three categories, the Management Plane, the Service Plane, and the 5G Core.

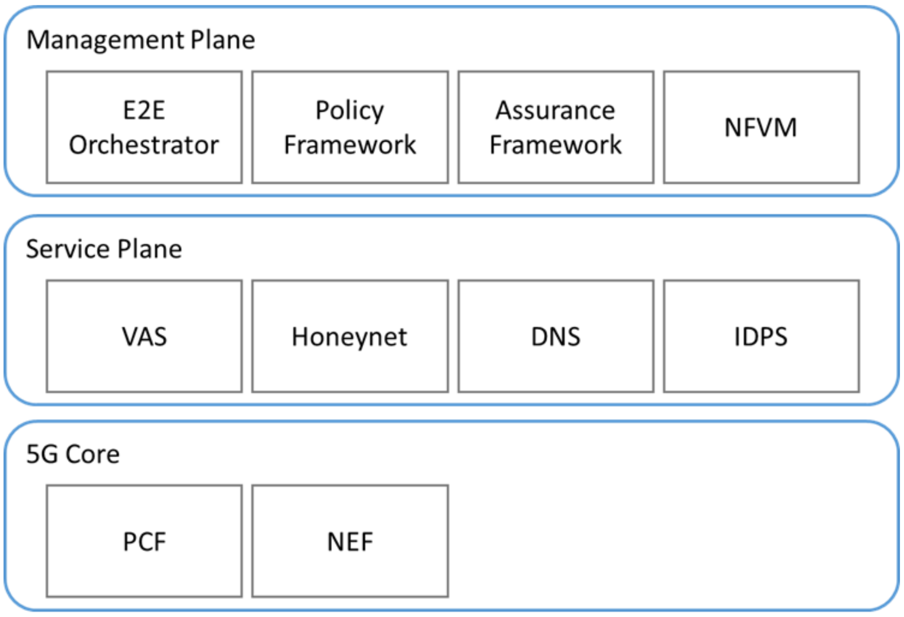


Figure 5.1: A view on the different PPS2 layers and its components [12]

Each layer is assigned with a specific role within the PPS2 task structure, namely:

- The Management Plane contains all operation and support systems that provide the necessary functionalities to network operators to manage the platform;
- The Service Plane hosts the applications that allow network operators to enable other layers' services, such as security related functions or name resolution;

- The 5G Core is, as the name implies, the layer that contains the core functionalities of the 5G platform, such as policy control and network exposure functions.

The work hereby described fits in the Service Plane, where a dashboard will be developed to receive data in the form of events from various other platform nodes, and display them in real time, helping network operators to debug and act quicker upon any anomaly. Its place in the architecture will be in the Service Plane, alongside other components such as the HoneyNet, which is a security functionality used to get information on intruder attacks; the Domain Name System (DNS), the Network Exposure Function (NEF), which takes care of exposing some network functions to outer networks/services, and the Vulnerability Assessment System (VAS), which provides information about potential security threats [12] .

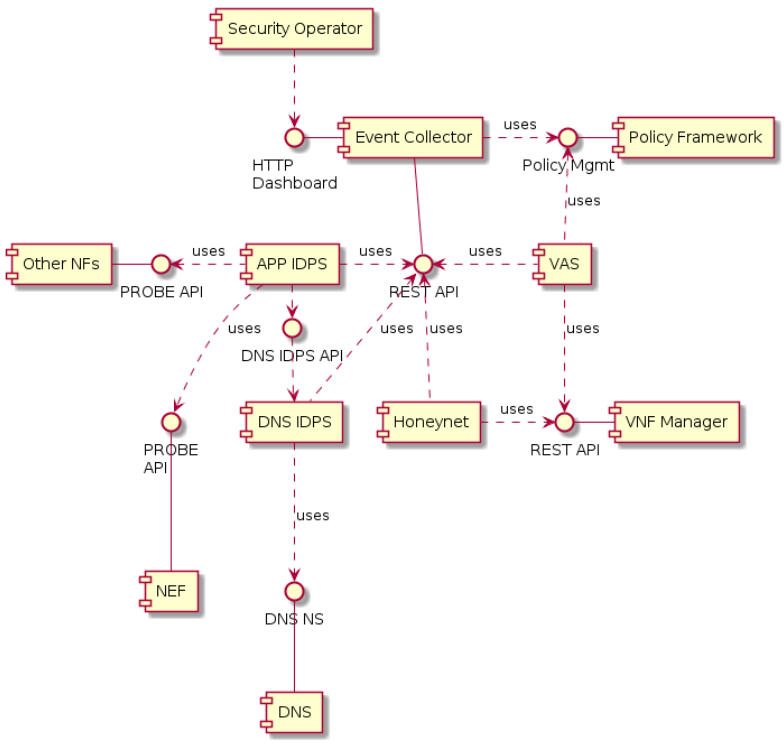


Figure 5.2: The external view of the Security Framework [12]

The proposed architecture (see Figure 5.3) merges the capabilities of a high throughput message forwarding system provided by Kafka, with the scalability and deployment features allowed by containerization using Docker. The messages are then consumed by the middleware, which takes care of the processing and makes this data available to the dashboard, where the data is going to be presented to the user. This data may contain events, notifications or statistical data from the rest of the 5G platform components.

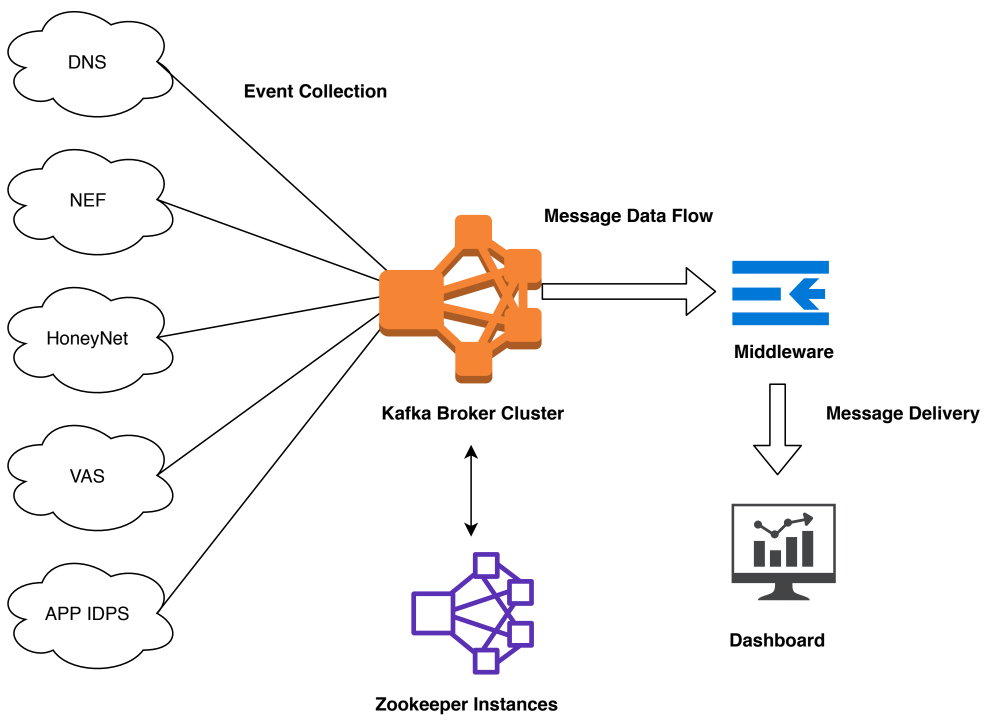


Figure 5.3: A view on the proposed architecture.

This architecture, as shown in Figure 5.3 was considered the optimal way of solving the problem, taking into account the rest of the platform and the nature of its data, which can be easily collected in the dashboard with little overhead to the remainder of the components.

By separating the message brokers and the data processing, we can achieve higher throughput numbers without compromising data integrity or causing bottlenecks at the dashboard, keeping all its data relevant and up-to-date.

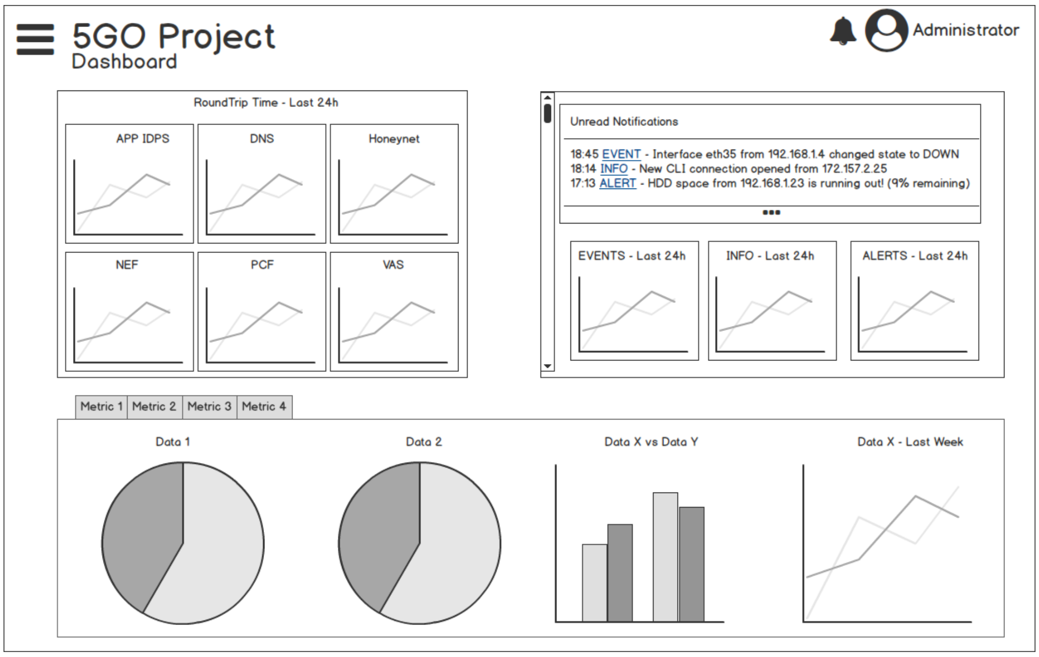
The expected 5G components to use the platform are the VAS, NEF and HoneyNet, since they're the main sources of security-related data in the platform, and the nature of their data is the most adequate to be displayed in the dashboard.

Given the different types of data we're dealing with, there will be different kinds of visualization techniques employed, such as:

- Temporal series, with grouped indicators, i.e.: number of events per type, per time slice, or both;
- Visualization of the events and its attributes;
- Visualization of the grouped quantitative indicators per event type, i.e.: number of events per level of severity;
- Event distribution per category;
- Other relevant techniques that are relevant to the type of data dealt with but are yet to be found.

The chosen visualization method will depend on which data it is trying to be visualized, for example, a simple XY line graph with real-time update will suffice to deal with the event visualization, however, for its attributes and metrics, tables will be needed.

As part of the UI planning and design effort, a series of mockups were drawn, in order to test some of the ideas for the layout of the dashboard. These mockups are shown in the next figures.



18:45 EVENT - Interface eth35 from 192.168.1.4 changed state to DOWN
18:14 INFO - New CLI connection opened from 172.157.2.25
17:13 ALERT - HDD space from 192.168.1.23 is running out! (9% remaining)

Figure 5.4: An idea of how the dashboard could be designed #1

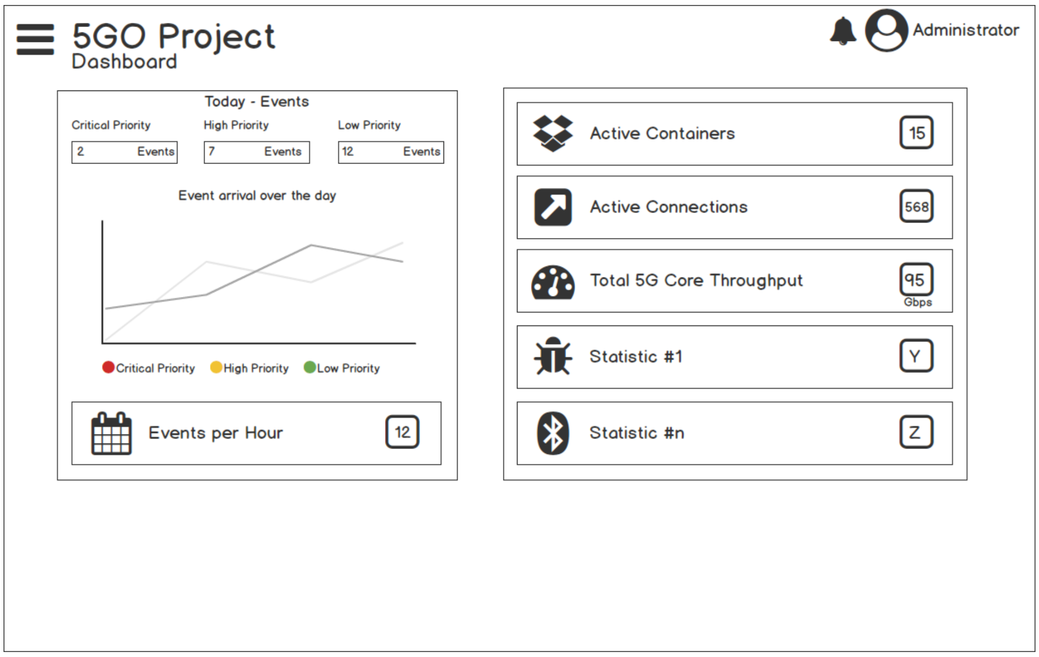


Figure 5.5: An idea of how the dashboard could be designed #2

These mockups were created based on existing monitoring software and their respective UI components. Some elements that were also relevant for this project were also debated and inserted in the mockups. These would then be debated upon in the following months, refining the UI and the components needed, which would then take the form of the final dashboard design.

This page is intentionally left blank.

Chapter 6

Solution

In this chapter we'll discuss the decisions that were taken to create the final version of the project, given the project's requirements and scope, as well as any changes that had to be done from the proposed architecture.

6.1 Architectural Decisions

After a series of refinement and testing steps, the final architecture layout for the proposed solution was reached. In its stable form, the event processing, transport and visualization subsystem includes 4 main components, namely:

- The Kafka Cluster, which would take care of making the events available to the middleware, and posteriorly, the dashboard;
- The Middleware, which reads data directly from the Kafka topics and transforms it into a state which is ready to be accessed and mapped by the dashboard. It is also tasked with processing some of the metrics;
- The Kafka Streams App, which will take care of the event windowing and some metrics processing, while dumping the results in a separate, easy-to-access Kafka topic;
- The Dashboard, which will consume the data fed by the middleware, and display it to the user in appropriate and intuitive ways.

This resulted in the architecture being slightly changed from the one represented in Figure 6.1, changing to the architecture represented in the following figure:

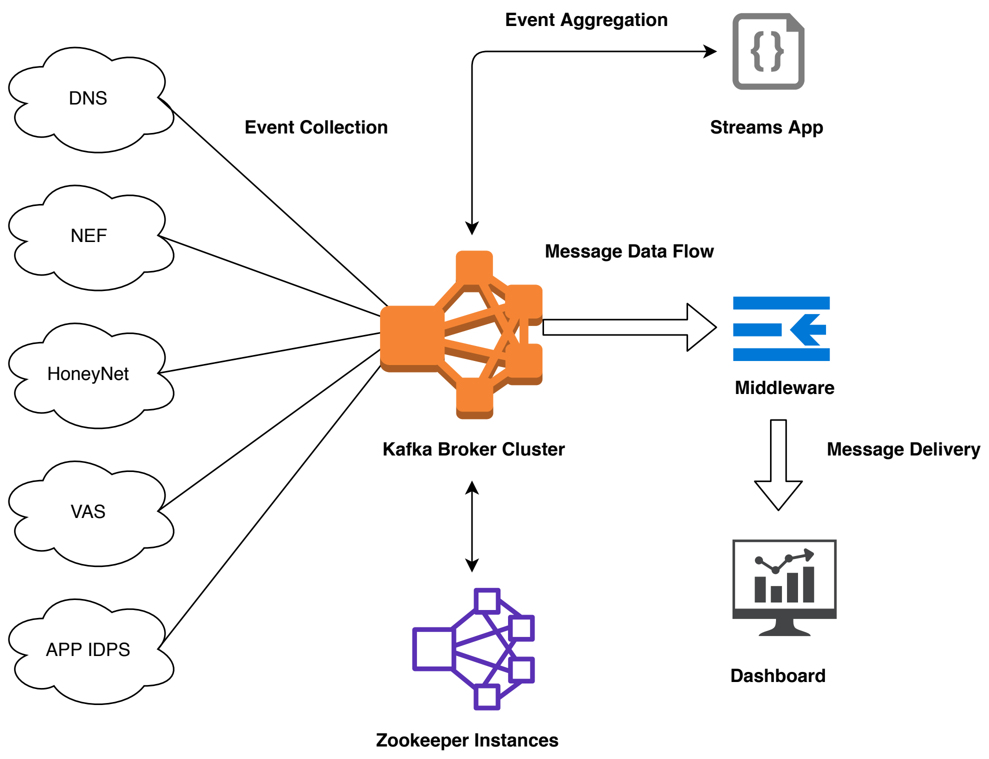


Figure 6.1: Details of the updated architecture.

This implied that one extra component was added to the system:

6.2 Kafka Streams App

It is a Java application that takes advantages of the Kafka Streams API, which, as described previously in the State of the Art chapter, manipulates the data flowing through the Kafka topics in real time, and in this case is used for two distinct functions:

- Windowing - Kafka Streams has the tools to make message aggregation possible, and given that the nodes might send a large influx of events, it is important to aggregate these in order to keep the dashboard as "clean" as possible, with only the most relevant information;
- Metrics Processing - The Streams App will also be in charge of processing the incoming data into temporal statistics and make them available to the dashboard. This function however, will only be used when metrics require processing within a defined time interval.

Both these processing features will require the Streams App to dump the processed data onto another dedicated topic, which will then be consumed by the middleware with all the data it needs.

Regarding the aggregation, this will be done according to the needs of the application, but for now it will aggregate the events which are categorized as ”Low” priority, to prevent low priority messages (which are assumed to be the most common) from flooding the dashboard.

6.3 Message Encoding - JSON vs Avro

Two possible message encodings were considered to be used with the platform: JSON and Avro.

In the end, it was decided to use JSON, since it allows for more flexibility of the message format and doesn't require additional formatting or variable types to be declared. This would help down the road, since we don't have direct control over what type of variables we'll receive as metrics for processing.

This would turn to be a loss in other departments however, since using JSON over Avro would mean that the Kafka broker now couldn't run schema checks, and therefore the JSON message integrity/parameter checking needed to be done at the middleware rather than at the broker, which is an extra step of data flow, not to mention additional coding.

6.4 Updated Data Model and Details

The data model was reviewed multiple times, mainly trying to fix the problem of dealing with unknown formatted data being sent to the dashboard for metrics processing. This lead to a few debates between myself and the Coordinators, in which we discovered some problems:

- There is the need to hard-code the base message format, so both the Streams App and the Middleware can interpret it and de-serialize it accordingly.

  However, this is out of our control, because we do not know which type and how many metrics each node will send, and the dashboard has the need of interpreting the metrics according to the node it is sending;

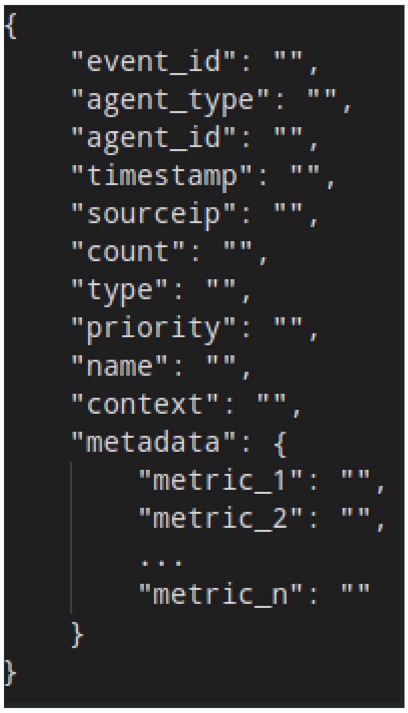
- Selecting the node via IP is not also a viable solution, since the nodes might be hidden behind forwarding mechanisms (Network Address Translation (NAT), VMs, Virtual Network Function (VNF), Port Forwarding, etc...);

- Since the events are to be sent by other partners, we cannot hardcode the metrics in the components, therefore there is the need to do profiling.

  Profiling was the solution that we concluded was the most fitting to the problem, and it is based on using a dedicated parameter in the message format - ”agent” to dictate which profile will be used to de-serialize the message. This will, of course, mean that there will be two significant drawbacks:

  – The increase in processing time, due to both the middleware and the Streams App now have the need to check which profile is being used by the event sent;
  – The fact that each profile will have to be manually created and added to the collection, on demand.

Given this, the data model document has since been updated, with more detailed explanations about each field: its intended uses and limitations, which will be detailed below:



```
{
    "event_id": "",
    "agent_type": "",
    "agent_id": "",
    "timestamp": "",
    "sourceip": "",
    "count": "",
    "type": "",
    "priority": "",
    "name": "",
    "context": "",
    "metadata": {
        "metric_1": "",
        "metric_2": "",
        ...
        "metric_n": ""
    }
}
```

Figure 6.2: Details of the updated Data Model.

Regarding the global structure of the data model, there will be two new fields - ”agent_type” and ”agent_id”:

- The ”agent_type” field will be used to allow the user to see which 5G network component the event came from, as for the dashboard, this will allow it to know which calculations should be done with the metrics received along with this event, without the need of a specific data model for each type of node;
- The ”agent_id” field will allow the dashboard user to see which instance of the service the event came from, in the case of a service being hosted by multiple instances/-containers/VNFs.

As for the other fields:

- The ”count” field was added in the previous Data Model iteration, and was introduced mainly because of the need of aggregating data, this way, the events can be aggregated seamlessly to the dashboard, without having the need of creating a different data model specifically for aggregated events;
- The ”type” field will dictate the type of the event, and its value will be selectable from a static list of pre-programmed values, that follow the criticality values used in the Syslog Protocol, according to RFC 5424;

  Its possible values are:

  - Emergency: The system is inaccessible;

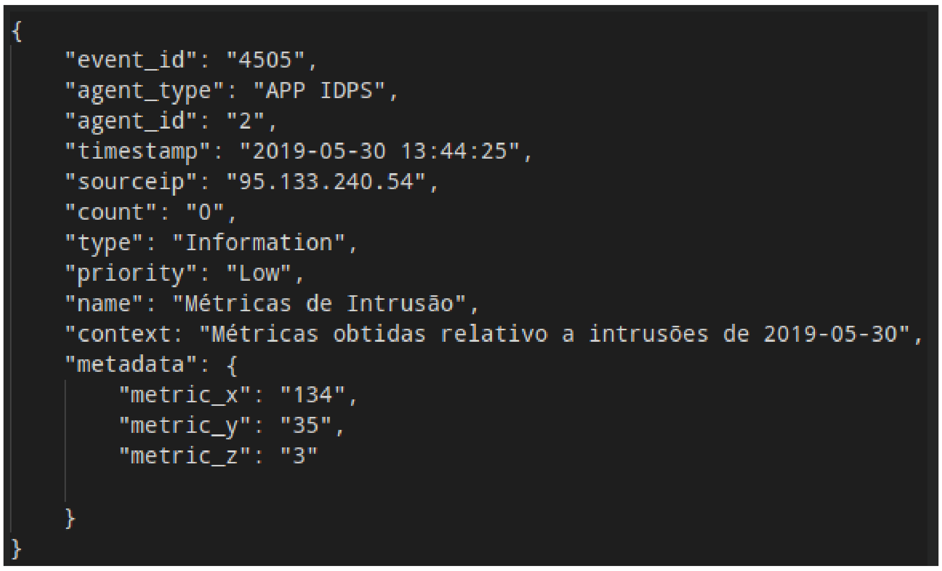
  - Alert: Immediate action should be taken;
  - Critical: Critical condition;
  - Error: Occurrence of an error;
  - Warning: Warning about possible problems;
  - Notice: Information about an event of possible significance;
  - Information: Informational messages;
  - Debug: Debug Messages.

- The fields ”name” and ”context” will contain detailed information about the event. These will be filled by the producer, and while ”name” is a short description, with a 25 character limit, ”context” allows for a more detailed description, up to 250 characters;

- The ”metadata” field is where all the metrics would be stored. As it was concluded earlier, the different nodes will use different metrics, and different operations will be performed upon them. The dashboard needs to know how to identify which node is sending these metrics and do the calculations accordingly.

  This is where the ”agent_type” fields come in, which allows both the middleware and Streams App to differentiate the incoming event based on this value, effectively creating a profiling mechanism to be used for each kind of node.

The following image is an example of a filled event, at the time of arrival at middleware:



```
{
    "event_id": "4505",
    "agent_type": "APP IDPS",
    "agent_id": "2",
    "timestamp": "2019-05-30 13:44:25",
    "sourceip": "95.133.240.54",
    "count": "0",
    "type": "Information",
    "priority": "Low",
    "name": "Métricas de Intrusão",
    "context: "Métricas obtidas relativo a intrusões de 2019-05-30",
    "metadata": {
        "metric_x": "134",
        "metric_y": "35",
        "metric_z": "3"

    }
}
```

Figure 6.3: Example of an event at the time of arrival at the middleware.

These changes, field descriptions and requirements were all compiled and sent to the partners, so that they could start producing actual data to be used at the dashboard. Clients were also asked to fill a form containing the agent identification and all the relevant metrics that they would send. This would then, in turn, allow me to create profiling mechanisms for the middleware to separate which metrics to use which operations on, and which display methods to use.

6.5 Testbed Deployment

A demo is planned, to showcase the integration between the components implemented by Faculdade de Ciências e Tecnologia da Universidade de Coimbra (FCTUC), Ubiwhere and OneSource, which are the other partners involved in the project. This was scheduled for early October, and I was asked to deploy the dashboard and its components into a testbed owned by the project, hosted in Aveiro. The objective of this deployment was to test the compatibility of the software with the other partners' tools.

This deployment was made in OpenStack, using two separate instances: one for the Kafka cluster, and another for the middleware, dashboard and Streams App. There were some problems in the logistics part of the deployment, mainly with the lack of resources that were allowed to the quota given to the FCTUC group, which delayed the total operation by about two weeks.

The objective after the deployment was completed was for the dashboard to interact with real data, provided by OneSource, instead of data generated by the event generator that was created. This meant that it was requested to OneSource to specify which metrics it would send to the dashboard, as well as which types of data processing would be useful. This processing would then be programmed in the Streams App.

Of the two OpenStack instances that were created:

- One contained the dashboard, middleware, and Streams App, and was given:
  - 8GB RAM
  - 4 Virtual Central Processing Unit (vCPU)
  - 50GB SSD
- The other exclusively contained the Kafka cluster, and was given:
  - 16GB RAM
  - 4 vCPU
  - 50GB SSD

  It was decided to have the Kafka cluster separated from the rest of the dashboard elements in case of a higher Kafka throughput necessity, since it could be easily scaled to demand on the same machine.

After the deployment was made, the Data Model document was updated to include some information that was necessary for the partner to successfully connect to the Kafka broker that was missing in older versions, as well as some troubleshooting solutions for future partners from smaller problems that happened during the deployment with OneSource.

6.6 Metrics Processing Decisions

For now, the platform is programmed to do processing for two partner entities: OneSource and Ubiwhere, where they would then be contacted to provide them the Data Model, which was necessary to understand the data format and its characteristics.

After each partner filed the requirement with the metric-specific parameters and their use in their specific platform, a method of allowing the end user to get the most information while minimizing clutter in the GUI would be devised. For this effort, charts and tables were mostly used, as well as some global indicators.

Most information sent by these partners is going to be somewhat session-based, however, since there wasn't a specific time-slice required for the processing, most of the processing is going to be done in the middleware rather than in the Streams App, since the latter requires a specific time to be set for aggregation purposes.

Some time was also spent talking individually to the partners to mostly learn more about the nature of the data that was going to be sent to the dashboard, to ensure that both the design and data processing was according to their liking and in context of the monitoring of each of the partner's platforms, and finally to debate which visualization methods would be the best to demonstrate the processed data in an easy to understand way. By having this personal interaction with the partners, a lot of development by trial and error was avoided, and allowed the reuse of some code and paradigms that were created for the data processing and session-grouping.

Below, the details of each of the partners' proposes and how I handled their requests will be detailed.

6.6.1 OneSource - NEF

For the NEF, the following parameters would be sent by OneSource:

- api_route: this parameter indicates the API route which is under attack;
- attack_count: The number of attacks made to the paired API route, which serves as a blocking threshold.

After some analysis of this data and the ways it could be represented, I chose a graph to represent the attacks across the time, including time-slice filters for easier visualization, as well as a table that listed all the routes, and kept track of the cumulative number of attacks made to each one. All of this was complemented by a general counter, that displayed the sum of all threats from all routes.

By categorizing and storing the events based on which route they occur, I can create datasets which then can be used to populate either the graphs or the tables. These show more details about each route in specific, for example: all the time stamps of each event and the corespondent number of attacks.

6.6.2 Ubiwhere - HoneyNet

For the HoneyNet, the following parameters would be sent by Ubiwhere:

- session_id: this identifies an open attacking session, and will be used to uniquely separate datasets;
- intrusion_type: identifier of the type of intrusion made;
- source_ip: source IP address of the intrusion;
- dest_ip: destination IP address of the intrusion;
- service: what services are being attacked;

- action: what command the attacker performed.

Each event that is received from the HoneyNet will represent either an ongoing attacking session or a new attacking session, with the finality of storing the commands performed by the attacker in the entirety of a session. Like the NEF case, since there's a session paradigm happening, we can use the session identifier to categorize and separate each session's info and actions performed, which can then be used to either populate charts or tables.

Here it is slightly harder to do a session based chart, since the Y axis isn't quantifiable, however that allows us to use it as the session identifier, solving a few problems. The visualization methods here would comprise of three methods: the session listing table, where all the sessions and its main characteristics would be listed, followed by a session-actions graph, which highlights the periods of activity during an attack session.

Finally the actions table, which displays commands done by the attackers in all sessions, ordered chronologically, which can then be filtered per unique session, to show each session's attacks and respective time stamps.

6.7 Dashboard Design and Component Decisions

As previously mentioned, a skeleton template of the ATENA project dashboard was used as a base to create this project's dashboard. It was decided to go for a simplistic approach, having as much relevant information displayed in a accessible manner, in a way that the user could get the most of some general system statistics, as well as some of the latest happenings in relation to event arrivals.

For navigation purposes, the dashboard also has a sidebar menu for easy access of its different parts, currently having three tabs, with the metrics tab containing all the pages related to metrics processing.

The following pages detail the contents of each page, as well as the features that were added to allow the user to have a more efficient navigation across the dashboard:

- Main Page: here's where all the most relevant information about the events and Kafka broker is located, and is used as a general page to collect statistical data on all the nodes sending events to the dashboard, as well as Kafka topic usage;

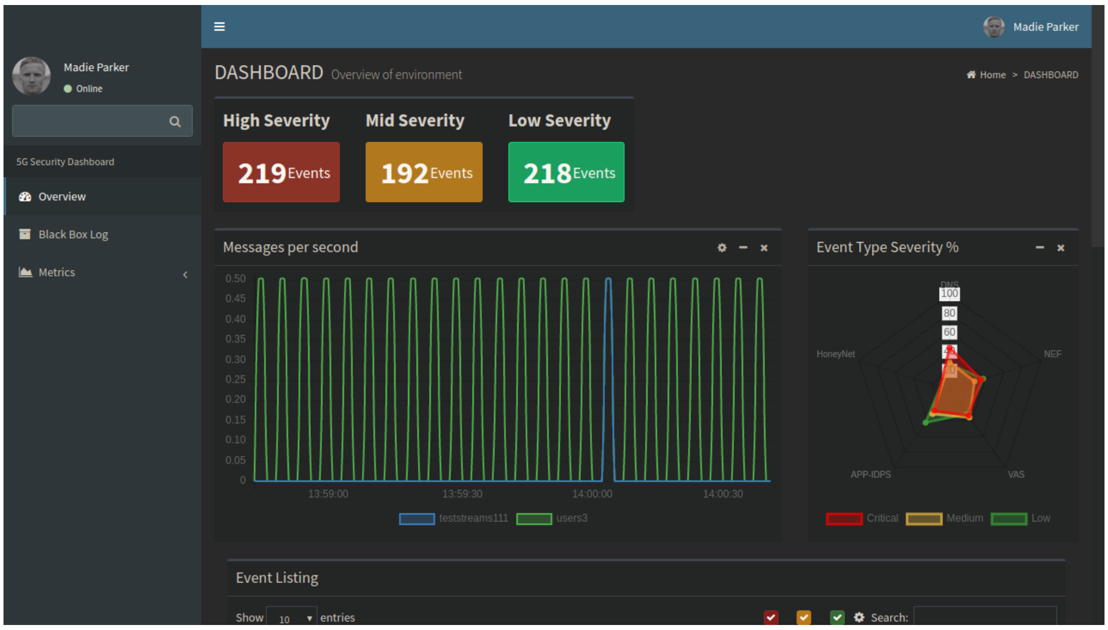
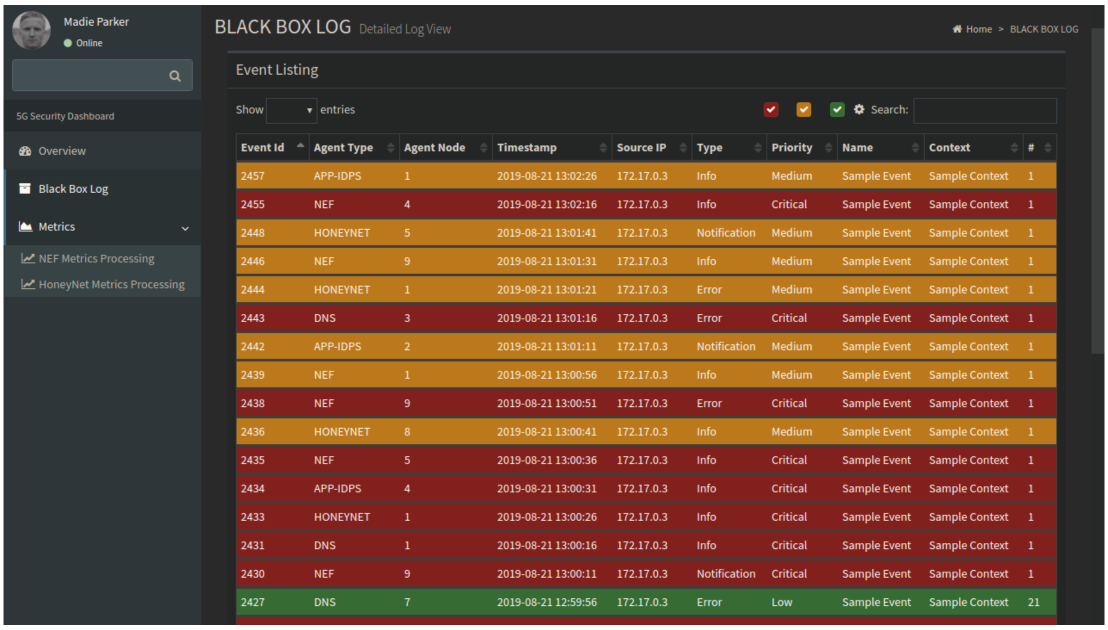


Figure 6.4: Main page of the dashboard.

- BlackBox Log: this page contains a more extensive log used to search up patterns or old events;



Event Id	Agent Type	Agent Node	Timestamp	Source IP	Type	Priority	Name	Context	#
2457	APP-IDPS	1	2019-08-21 13:02:26	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2455	NEF	4	2019-08-21 13:02:16	172.17.0.3	Info	Critical	Sample Event	Sample Context	1
2448	HONEYNET	5	2019-08-21 13:01:41	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2446	NEF	9	2019-08-21 13:01:31	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2444	HONEYNET	1	2019-08-21 13:01:21	172.17.0.3	Error	Medium	Sample Event	Sample Context	1
2443	DNS	3	2019-08-21 13:01:16	172.17.0.3	Error	Critical	Sample Event	Sample Context	1
2442	APP-IDPS	2	2019-08-21 13:01:11	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2439	NEF	1	2019-08-21 13:00:56	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2438	NEF	9	2019-08-21 13:00:51	172.17.0.3	Error	Critical	Sample Event	Sample Context	1
2436	HONEYNET	8	2019-08-21 13:00:41	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2435	NEF	5	2019-08-21 13:00:36	172.17.0.3	Info	Critical	Sample Event	Sample Context	1
2434	APP-IDPS	4	2019-08-21 13:00:31	172.17.0.3	Info	Critical	Sample Event	Sample Context	1
2433	HONEYNET	1	2019-08-21 13:00:26	172.17.0.3	Info	Critical	Sample Event	Sample Context	1
2431	DNS	1	2019-08-21 13:00:16	172.17.0.3	Info	Critical	Sample Event	Sample Context	1
2430	NEF	9	2019-08-21 13:00:11	172.17.0.3	Notification	Critical	Sample Event	Sample Context	1
2427	DNS	7	2019-08-21 12:59:56	172.17.0.3	Error	Low	Sample Event	Sample Context	21

Figure 6.5: Black Box log of the dashboard.

- Metrics: here's where the metric processing pages are located. For now, this menu is composed of the two entities that worked with us so far - NEF (OneSource) and HoneyNet (Ubiwhere);

The following pages try to describe each page and its functionalities, starting with the main page. The main page contains the following components:

- General Event Counters: there are a total of three counters on the top part of the dashboard, each indicating how many events per priority type are currently stored in the dashboard;

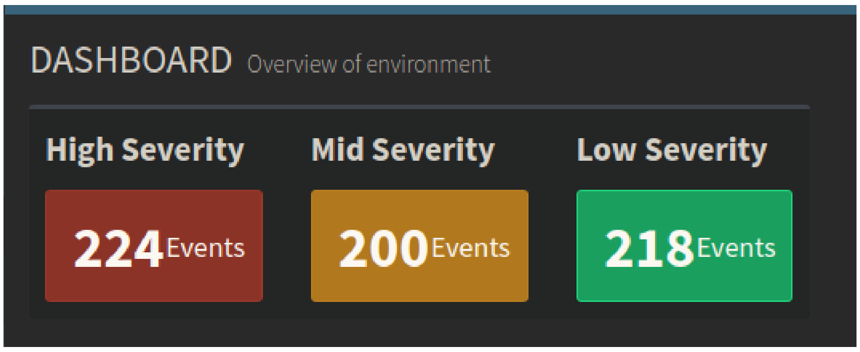
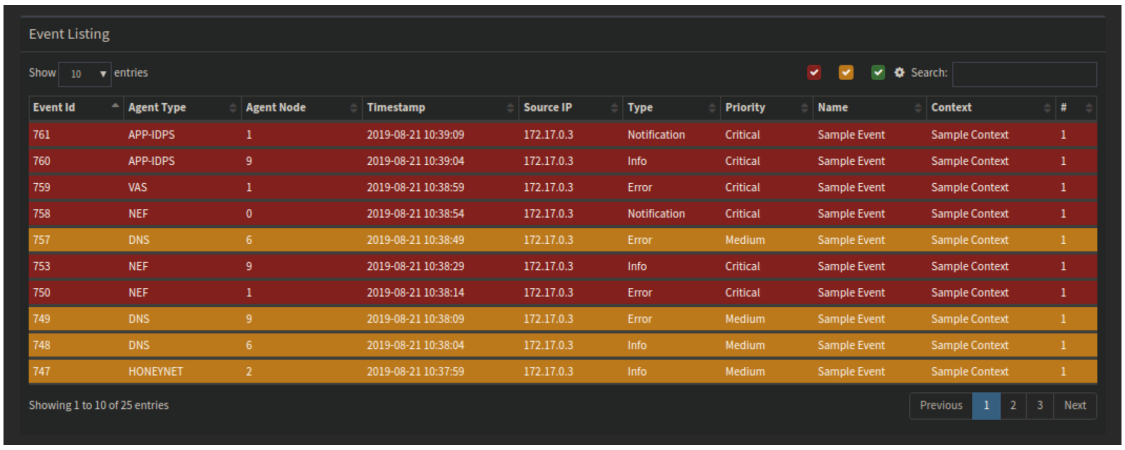


Figure 6.6: Dashboard event counters by severity.

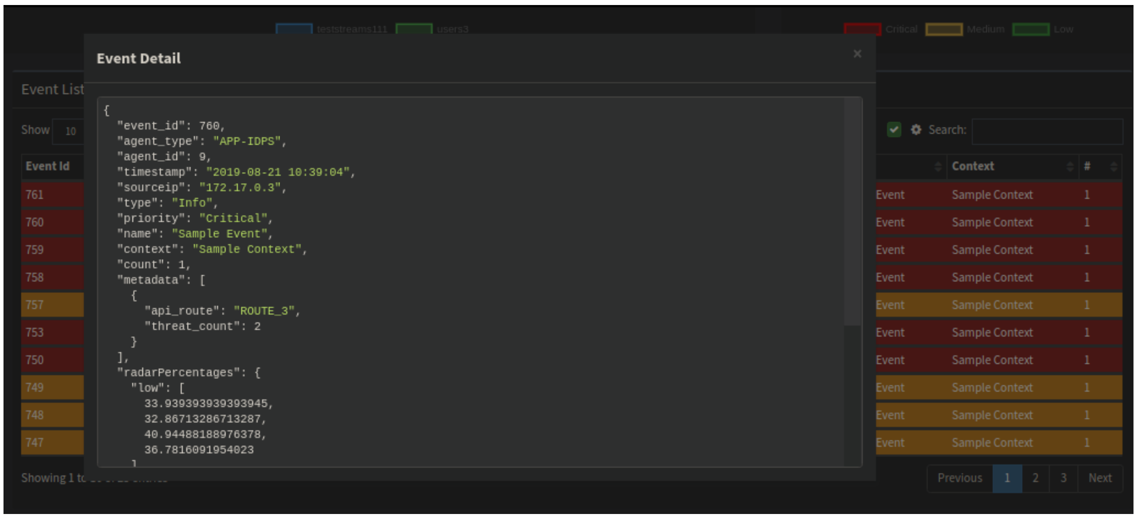
- Main Event Arrival Log: this log contains the latest 500 events arriving at the dashboard, categorized by colour according to their priority (Low - Green, Medium - Yellow, Critical - Red), with a dedicated search bar and color filter checkboxes, for an in-depth search. The main parameters of each content are listed in this table, while the full contents of each event can be seen by clicking each event on the log, which displays a window with all the event information in a raw format;



Event Id	Agent Type	Agent Node	Timestamp	Source IP	Type	Priority	Name	Context	#
761	APP-IDPS	1	2019-08-21 10:39:09	172.17.0.3	Notification	Critical	Sample Event	Sample Context	1
760	APP-IDPS	9	2019-08-21 10:39:04	172.17.0.3	Info	Critical	Sample Event	Sample Context	1
759	VAS	1	2019-08-21 10:38:59	172.17.0.3	Error	Critical	Sample Event	Sample Context	1
758	NEF	0	2019-08-21 10:38:54	172.17.0.3	Notification	Critical	Sample Event	Sample Context	1
757	DNS	6	2019-08-21 10:38:49	172.17.0.3	Error	Medium	Sample Event	Sample Context	1
753	NEF	9	2019-08-21 10:38:29	172.17.0.3	Info	Critical	Sample Event	Sample Context	1
750	NEF	1	2019-08-21 10:38:14	172.17.0.3	Error	Critical	Sample Event	Sample Context	1
749	DNS	9	2019-08-21 10:38:09	172.17.0.3	Error	Medium	Sample Event	Sample Context	1
748	DNS	6	2019-08-21 10:38:04	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
747	HONEYNET	2	2019-08-21 10:37:59	172.17.0.3	Info	Medium	Sample Event	Sample Context	1

Showing 1 to 10 of 25 entries

Figure 6.7: Main event log of the dashboard.



```
{
  "event_id": 760,
  "agent_type": "APP-IDPS",
  "agent_id": 9,
  "timestamp": "2019-08-21 10:39:04",
  "sourceip": "172.17.0.3",
  "type": "Info",
  "priority": "Critical",
  "name": "Sample Event",
  "context": "Sample Context",
  "count": 1,
  "metadata": [
    {
      "api_route": "ROUTE_3",
      "threat_count": 2
    }
  ],
  "radarPercentages": {
    "low": [
      33.939393939393945,
      32.86713286713287,
      40.94488188976378,
      36.7816091954023
    ]
```

Event Id		Context	#
761	Event	Sample Context	1
760	Event	Sample Context	1
759	Event	Sample Context	1
758	Event	Sample Context	1
757	Event	Sample Context	1
753	Event	Sample Context	1
750	Event	Sample Context	1
749	Event	Sample Context	1
748	Event	Sample Context	1
747	Event	Sample Context	1

Figure 6.8: In-depth viewing of an event's details.

- Prometheus Topic Monitoring: this chart displays the activity of each Kafka topic being used over the last two minutes. The main goal of this tool is to be able to analyze and react to discrepancies of topic usage and sudden increases in overall Kafka usage and consequent event arrival.

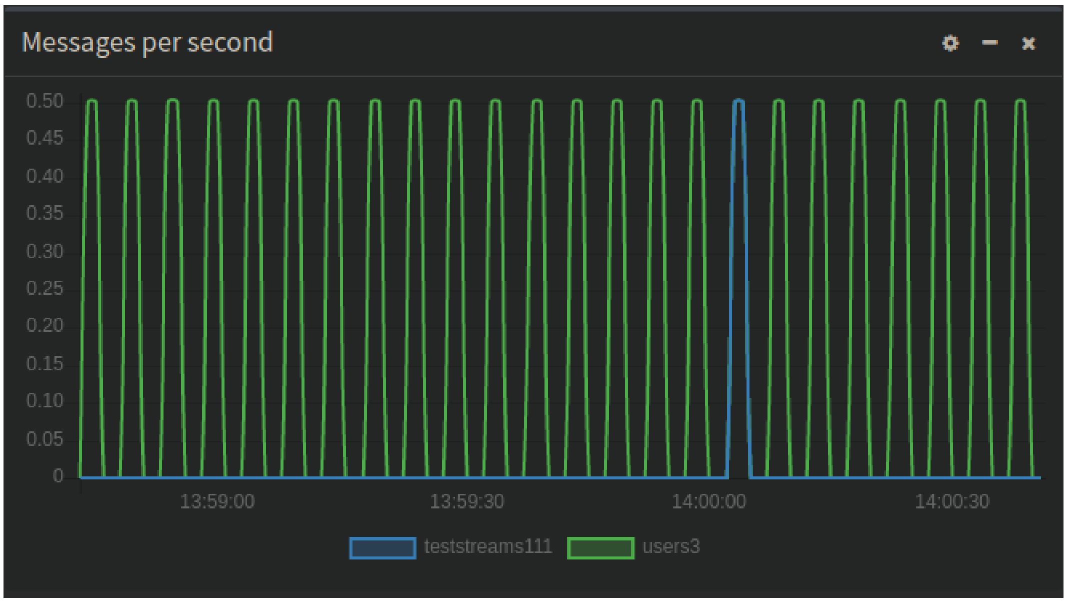


Figure 6.9: Prometheus Kafka topic monitoring.

To set up the Prometheus monitoring tool, it was also necessary to add a Prometheus agent to the Kafka brokers, which was configured in the Docker-Compose file. Then, the monitoring can be set up by directly accessing the data that is collected by that agent in the dashboard itself;

- Event Statistics Radar: this radar displays, in a simple manner, the amount of events currently on the dashboard. These are categorized by arriving agent and per priority (colour) in percentage form. It is also possible to hide and show layers of the radar to avoid confusion, in this case a layer being a type of severity;

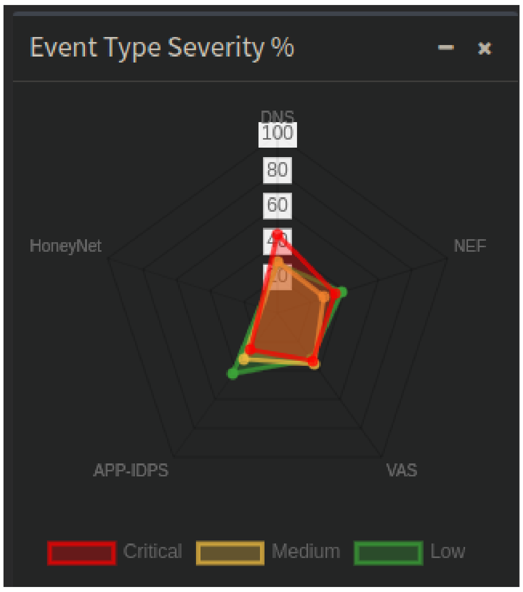
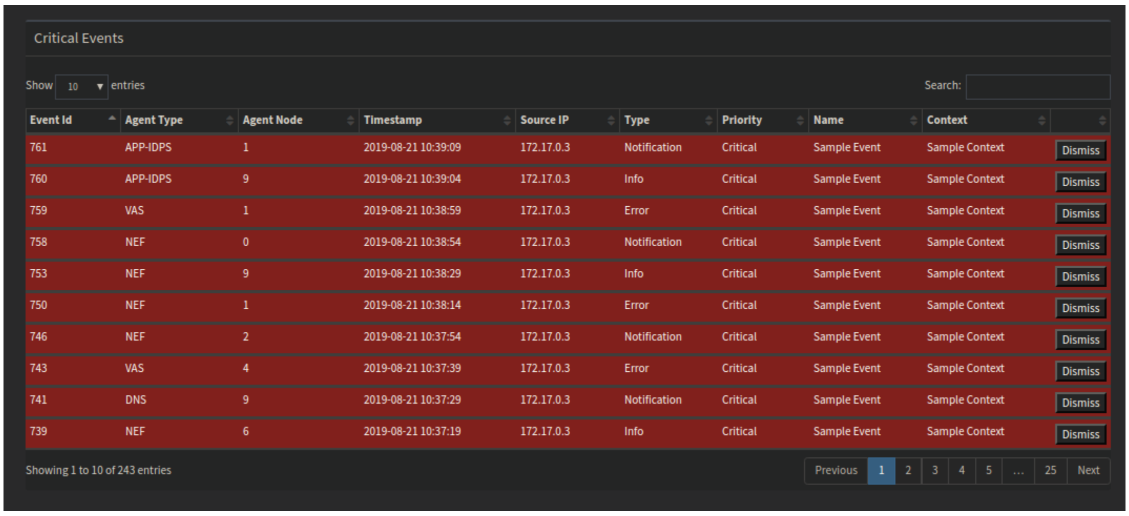


Figure 6.10: Dashboard radar chart displaying severity statistics of the registered nodes.

- Critical Event Log: This tool is used as a separate log that only displays the critical events, for a more simple and condensed view of these more important type of events, which usually require immediate action. Any event on this log can then be acknowledged, which removes it from the log, however, it will still remain on the other logs (Main Events Log & Black Box Log).



Critical Events

Show 10 entries

Search:

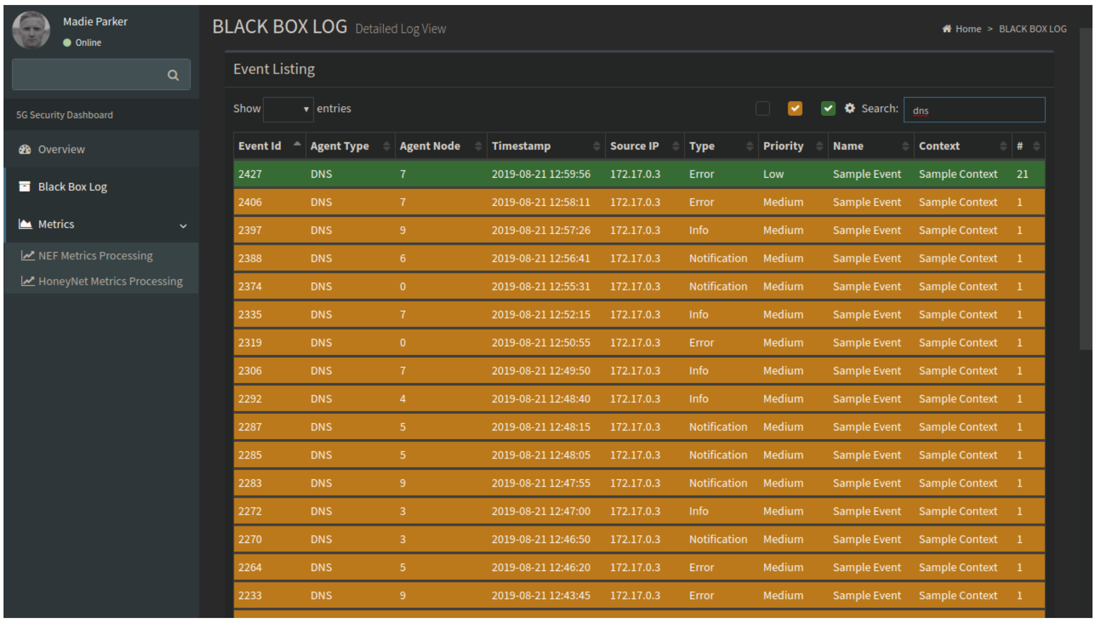
Event Id	Agent Type	Agent Node	Timestamp	Source IP	Type	Priority	Name	Context	
761	APP-IDPS	1	2019-08-21 10:39:09	172.17.0.3	Notification	Critical	Sample Event	Sample Context	Dismiss
760	APP-IDPS	9	2019-08-21 10:39:04	172.17.0.3	Info	Critical	Sample Event	Sample Context	Dismiss
759	VAS	1	2019-08-21 10:38:59	172.17.0.3	Error	Critical	Sample Event	Sample Context	Dismiss
758	NEF	0	2019-08-21 10:38:54	172.17.0.3	Notification	Critical	Sample Event	Sample Context	Dismiss
753	NEF	9	2019-08-21 10:38:29	172.17.0.3	Info	Critical	Sample Event	Sample Context	Dismiss
750	NEF	1	2019-08-21 10:38:14	172.17.0.3	Error	Critical	Sample Event	Sample Context	Dismiss
746	NEF	2	2019-08-21 10:37:54	172.17.0.3	Notification	Critical	Sample Event	Sample Context	Dismiss
743	VAS	4	2019-08-21 10:37:39	172.17.0.3	Error	Critical	Sample Event	Sample Context	Dismiss
741	DNS	9	2019-08-21 10:37:29	172.17.0.3	Notification	Critical	Sample Event	Sample Context	Dismiss
739	NEF	6	2019-08-21 10:37:19	172.17.0.3	Info	Critical	Sample Event	Sample Context	Dismiss

Showing 1 to 10 of 243 entries

Previous 1 2 3 4 5 … 25 Next

Figure 6.11: Critical event log of the dashboard.

Next up is the BlackBox Log page. This page contains only one element, the black box log. This is a log that keeps records to a larger extent, in case the user needs to look up older data for either event pattern purposes, or just for debugging problems. Since its size is considerably larger than the Main Event Log located on the main page, it was decided that it should have its own page, to not affect overall page performance.

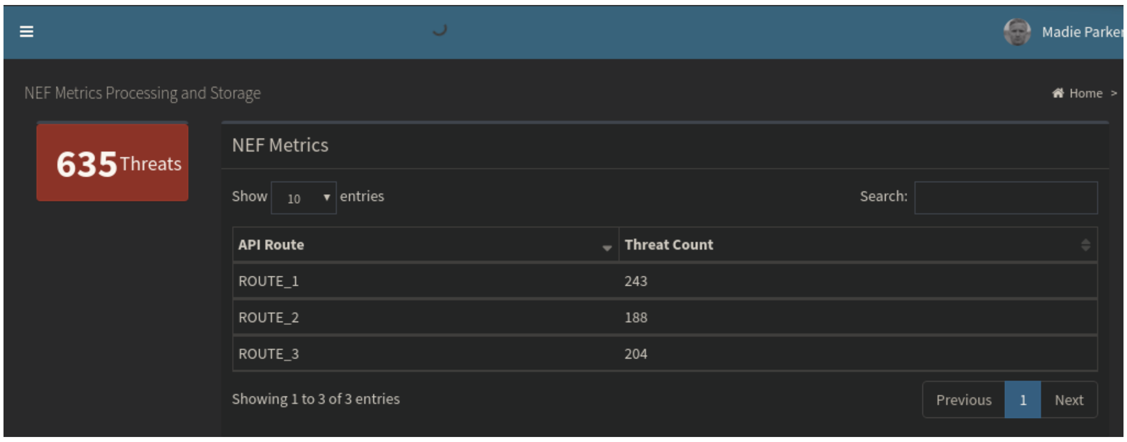


Event Id	Agent Type	Agent Node	Timestamp	Source IP	Type	Priority	Name	Context	#
2427	DNS	7	2019-08-21 12:59:56	172.17.0.3	Error	Low	Sample Event	Sample Context	21
2406	DNS	7	2019-08-21 12:58:11	172.17.0.3	Error	Medium	Sample Event	Sample Context	1
2397	DNS	9	2019-08-21 12:57:26	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2388	DNS	6	2019-08-21 12:56:41	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2374	DNS	0	2019-08-21 12:55:31	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2335	DNS	7	2019-08-21 12:52:15	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2319	DNS	0	2019-08-21 12:50:55	172.17.0.3	Error	Medium	Sample Event	Sample Context	1
2306	DNS	7	2019-08-21 12:49:50	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2292	DNS	4	2019-08-21 12:48:40	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2287	DNS	5	2019-08-21 12:48:15	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2285	DNS	5	2019-08-21 12:48:05	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2283	DNS	9	2019-08-21 12:47:55	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2272	DNS	3	2019-08-21 12:47:00	172.17.0.3	Info	Medium	Sample Event	Sample Context	1
2270	DNS	3	2019-08-21 12:46:50	172.17.0.3	Notification	Medium	Sample Event	Sample Context	1
2264	DNS	5	2019-08-21 12:46:20	172.17.0.3	Error	Medium	Sample Event	Sample Context	1
2233	DNS	9	2019-08-21 12:43:45	172.17.0.3	Error	Medium	Sample Event	Sample Context	1

Figure 6.12: BlackBox log being used with color (severity) and agent filtering.

Finally, the metrics processing pages, where we can look up some of the received metrics, as well as some statistics calculated by the middleware:

- NEF (OneSource): As previously mentioned, the metrics that originated in the NEF would consist of two variables, "api_route" and "attack_count". This meant that with each event received, a total number of "attack_count" attacks had occurred to the API route located at "api_route". So, it was important to keep track of how many attacks had occurred, per route, as well as the total of attacks so far,



API Route	Threat Count
ROUTE_1	243
ROUTE_2	188
ROUTE_3	204

Showing 1 to 3 of 3 entries

Figure 6.13: NEF Metrics Page - Total and per-route threat counters.

For this purpose, the NEF processing page contains a total attack tracker, located at the top of the page, which displays the number of total attacks done to date, as well as a table detailing how many attacks per route have occurred so far. For an easy temporal identification of each event arrival, the page also possesses a chart that displays a timeline for each API route, in which the user can check how many and at what times did attacks for each route happen. This chart can then be filtered to display only events from the last 1/5/24 hours (configurable), to reduce the amount of irrelevant data shown to the user;

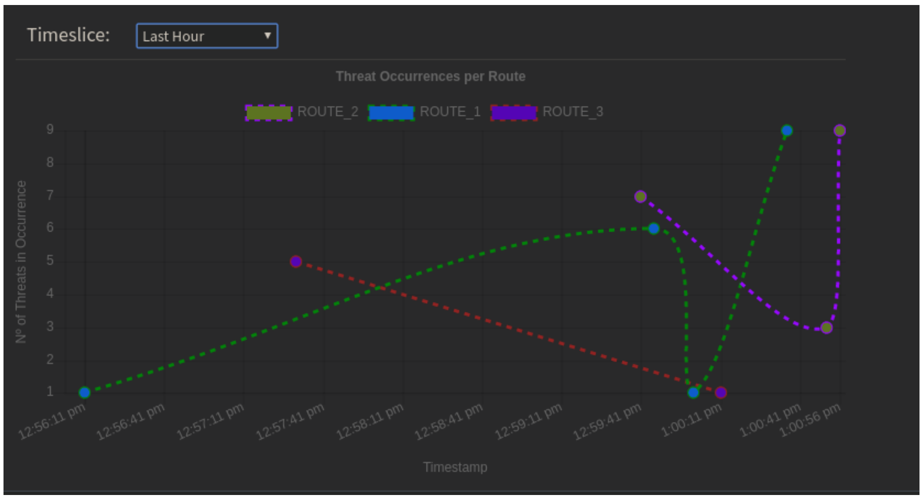
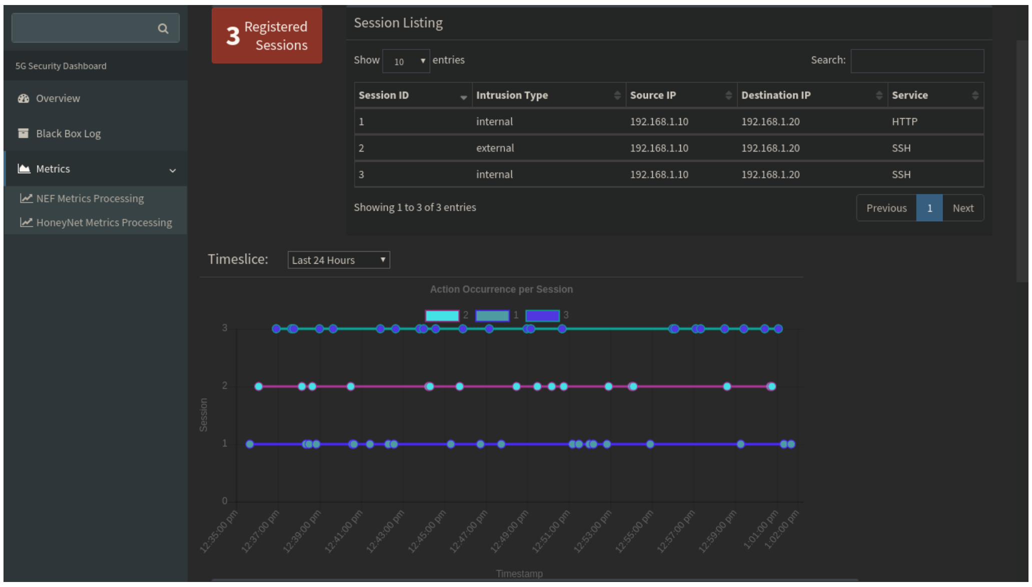


Figure 6.14: NEF Metrics Page - Chart displaying threat occurrences per route.

- HoneyNet (Ubiwhere): Regarding HoneyNet, the dashboard will be receiving data regarding attack sessions ("session_id"), in which it is important to keep track of every command performed by the attacker during the session ("action"). Therefore, the same paradigm as the one used in NEF also applies, in which we have a session-based system, and we need to aggregate data from incoming events to their respective sessions, to then perform some calculations to obtain relevant data to the end-user.



Session Listing

Show 10 entries Search:

Session ID	Intrusion Type	Source IP	Destination IP	Service
1	internal	192.168.1.10	192.168.1.20	HTTP
2	external	192.168.1.10	192.168.1.20	SSH
3	internal	192.168.1.10	192.168.1.20	SSH

Showing 1 to 3 of 3 entries

Previous 1 Next

Figure 6.15: HoneyNet Metrics Page - Session details and activity throughout time.

For this purpose, this page possesses a general counter displaying the number of ongoing sessions, a table that displays session information in a more detailed way, such as type of intrusion and source/destination IPs. Like the previous page, it also features a chart that displays each session's activity on a timeline, with each action associated with its timestamp. This chart's usage was thought to be more towards gauging each session's activity across time, which can be helpful debugging problems or identifying attacker behaviour/patterns or even detecting system backdoors and faults.

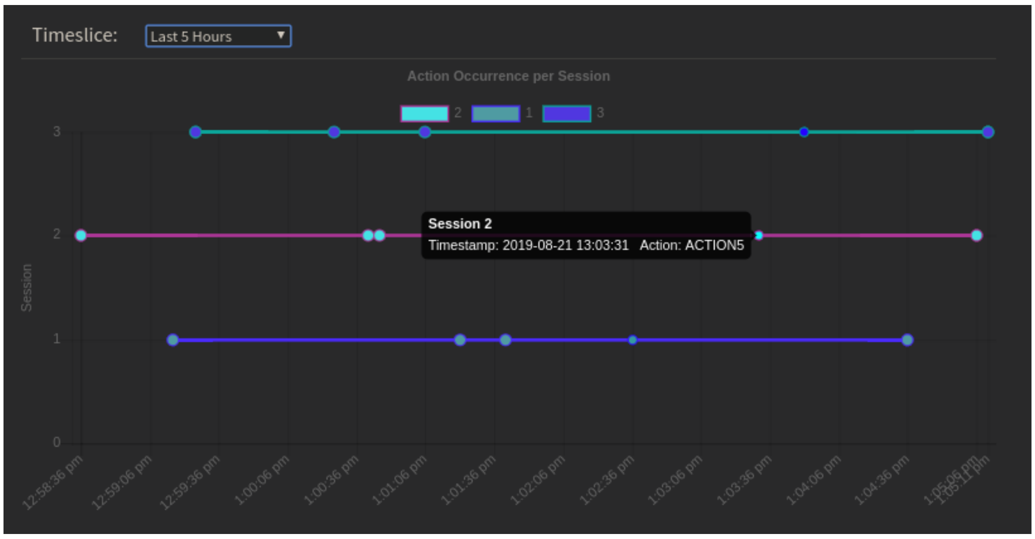
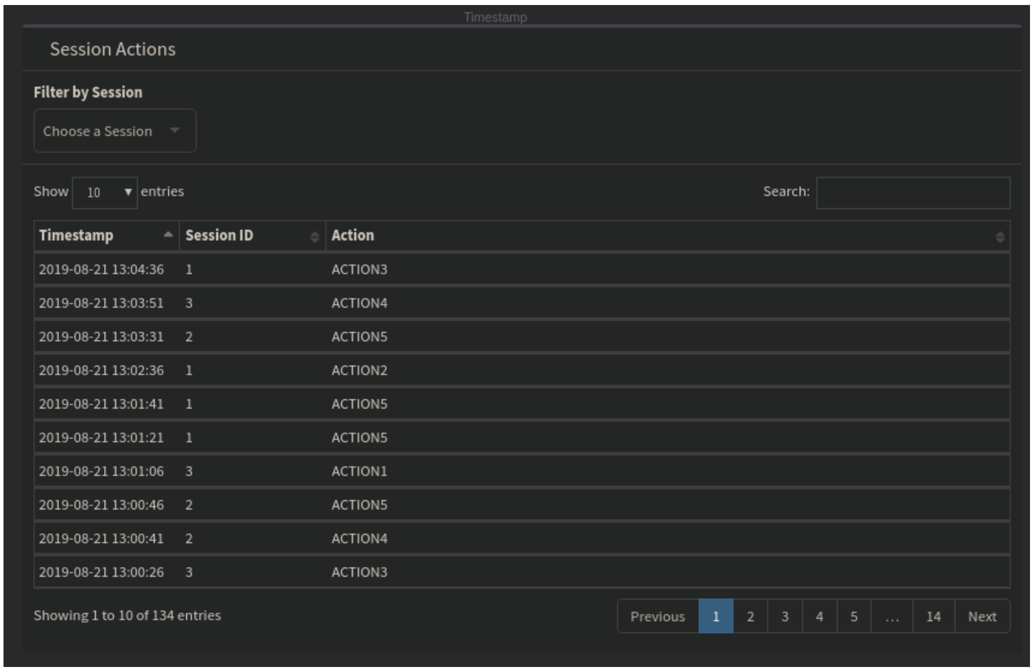


Figure 6.16: HoneyNet Metrics Page - Session activity chart, in detail.

Finally, we have a chronologically ordered table that displays each action, and corespondent timestamp and session. This can be then filtered to display only attacks from a certain session.


Session Actions

Filter by Session

Choose a Session

Show 10 entries Search:

Timestamp	Session ID	Action
2019-08-21 13:04:36	1	ACTION3
2019-08-21 13:03:51	3	ACTION4
2019-08-21 13:03:31	2	ACTION5
2019-08-21 13:02:36	1	ACTION2
2019-08-21 13:01:41	1	ACTION5
2019-08-21 13:01:21	1	ACTION5
2019-08-21 13:01:06	3	ACTION1
2019-08-21 13:00:46	2	ACTION5
2019-08-21 13:00:41	2	ACTION4
2019-08-21 13:00:26	3	ACTION3

Showing 1 to 10 of 134 entries

Figure 6.17: HoneyNet Metrics Page - Attack listing, chronologically ordered.

The dashboard was created with expansion in mind, so that more metrics page can be freely added and the data structures used for both HoneyNet and NEF could be re-utilized or reprogrammed to fulfill very different objectives.

Chapter 7

Validation

7.1 Overview

This dashboard was designed with the goal of supporting a heavy event flow, and that was made clear by the choice of using Kafka, which supports a heavy message throughput. However, the remaining components that take part in the architecture must be tested, to ensure that the performance level provided by Kafka is not wasted with upstream bottlenecks.

Regarding Kafka itself, there are plenty of comparison tests available in the literature, and has been proven to perform consistently with performance levels much higher than needed for this case in particular, Reason why there is no need to perform further tests to its performance in the scope of this project.

There are two components which must be tested to ensure that the whole architecture provides an acceptable degree of performance. These two components will of course be the middleware and the Streams App.

Given that the middleware doesn't store data in a static database or file, instead using a buffer to store all the events and the data it processes, there will be direct performance limits imposed. Still, for the intended task, the expected performance of both middleware and Streams App should suffice.

7.2 Setup for the validation tests

During the development and testing of the application, all the components were hosted in a server running bare metal VMware ESXI 6.7 located at Department of Informatics Engineering (DEI), with the following characteristics:

- Manufacturer: Dell Inc.;
- Model: PowerEdge R430;
- CPU: 8 CPUs x Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz;
- Memory: 63.78 GB DDR5;
- Storage: 900GB Solid State Drive (SSD).

To make sure that the test results weren't being compromised by hardware limitations, the Kafka cluster was set up in a VMWare ESXI 6.7 Virtual Machine, with the following specs:

- 16GB RAM
- 4 vCPUs
- 35GB SSD

All of the packages installed were to the usage of project's components, to prevent clutter and resource misusage.

As for the rest of the components, namely dashboard, middleware and Streams App, they'll be running on a separate VM, with the following specs:

- 32GB RAM (16 of which added after VM ran out of memory running certain sets)
- 4 vCPUs
- 35GB SSD

It is important to mention that during the tests, the server resources were solely allocated for that purpose, meaning no other resource-heavy tasks were running on another VM in the same hardware.

7.2.1 Planning

The decision to have the Kafka cluster separate from the rest of the components was made due to the fact that the middleware uses the buffer system, rather than a database. This implies that for large volumes of events, the impact that the middleware can have on the system resources may hamper the Kafka cluster performance, which could affect or even invalidate the performed tests. Also, this allows the replication of a production environment, which will increase the quality of the results.

As previously mentioned, there is nothing new to gain in testing Kafka itself, as it has already been tested numerous times by other papers, and is, in its whole, a *"battle-tested"* application.

The same does not apply for the remaining of the components, mainly the two possible data bottlenecks, the Streams App and the middleware. These will have different testing scenarios best suited to the operations they perform: for instance, the Streams App will deal with scenarios in which only events with Low priority are created, since these are the only ones it processes at the moment. Meanwhile the middleware, which will normally handle a larger amount of data, will deal with scenarios with heavy event flows.

The following subsections detail the tests that will be made, as well as the metrics that are to be collected, to then be reflected upon.

7.3 Test Scenarios - Introduction

All scenarios described in this subsection will use the event generator created for the purpose of testing the platform, since it has more availability than data created by partners, and is flexible enough to be changed at my will, for any testing purposes that might occur.

This generator creates data that mimics those of the partners', staying loyal to the defined formats. At this point, it can even generate random metric datasets based on the two partner profiles that were implemented in the dashboard: NEF and HoneyNet. These are sent to the dashboard, processed by the middleware, and shown to the end user according to the visual representation method chosen that I thought was best suited, given the data's nature.

Given the previous statement, it is fair to say that the event generator can be used for stress testing of the platform without any issues, provided it can keep up with the production throughput that is required. After testing the generator in an isolated environment, it was concluded that it is suitable to be used in the testing, given it can create the quantity of messages specified in the test scenarios.

7.3.1 Considerations

Since the platform does not use a database system to store the events, it will fall to me to decide in which manner to handle the buffer, a topic that will be debated later in this chapter.

Regarding the tests, the buffer size will be a major factor, due to the sheer amount of events it can handle at a given time (which will also be dictated by system RAM, mostly), so it will be taken in consideration when modifying the non-static variables of the tests.

For performance sake, the visual debugging on the console was kept to a bare minimum, to prevent visual bottlenecking caused by console *stdout* throughput, which would affect overall application cycle speed.

For a better testing flow, the following method will be applied to the test datasets:

Producing X (varied by testing set) events, it is important to know:

1. Arriving time of the event (middleware)

2. Time after processing has been done to the event (middleware)
3. Time between 1 and 2 - total effective processing time

As for the Streams App, the same rules will apply, but since it currently only processes the traffic categorized as Low Priority, constituting about 33% of the data, the generator will have to be tweaked for flows consisting of only data from that type.

The minimum time between events will follow a specific events per minute ratio, which will then be increased to test the actual middleware throughput. If the middleware cannot handle event arrival at the proposed speeds, this be will also be documented and analyzed, to see whether the middleware and Streams App can keep up with full production speed of the Kafka producers.

Regarding the Kafka cluster and its usage by both middleware and Streams App, there will be two topics, one that contains the raw data sent from the generator, simulating data coming from all the partners, called ”main”, and another topic, which will be used to store the Streams App processed data, ”streams”.

The middleware will then read from both these topics simultaneously, getting all the data it needs for its computational purposes.

7.4 Test Scenarios

A few test scenarios were thought of, namely:

Total Capacity - These will test the buffer’s capacity, and how its size affects overall platform’s performance:

- Set 1: 10000 events;
- Set 2: 100000 events;
- Set 3: 500000 events.

Events per Minute - These will test raw throughput of the platform, and how its performance will behave with different arrival times between events:

- Set 1: 1000 events per minute;
- Set 2: 10000 events per minute;
- Set 3: 100000 events per minute.

There will also be two categories/variants of the scenarios proposed above:

- Variant 1: The test will comprise of data that mimics the nature of the data that will be sent to the dashboard, with each events’ content randomized, as is normally done by the generator;
- Variant 2: The test will comprise exclusively of data that suffers extra processing (simulated data that originates from NEF and HoneyNet - OneSource and ubiWhere, respectively).

These two variants are separated into the stated due to the fact that it allows the tests to stay loyal to the data's nature, achieving a high degree of fidelity and a good representation of how it will perform under a real production environment (Variant 1), but also how it will perform under the highest load it can possibly have at a given time (Variant 2).

Regarding repetitions, each combination of test (Set + Variant) **will be done 10 times**, and the final result will comprise of its average and standard deviation, as well as analysis to other factors, such as the buffer in case of the middleware.

As for the Streams Apps, the same scenario sizes will be used, however, the production will be changed to have only low priority events, since they're the only ones (at the moment) being processed by this application.

Regarding the capacity tests, on larger sets, there might be an issue regarding the buffer size, in which it may be needed to expand on resources of the machine. If that does happen, it will be mentioned, as well as what new resources are introduced.

To get the effective processing time of each event done by the component in question, be it the Middleware or Streams App, there's no need to take in account the time frame since the event is produced up to the moment he reaches the middleware, to avoid introducing extra variables to the testing, such as network speed, SSD read/write speed, and inter-VM communication processes. Therefore, the testing method used will be:

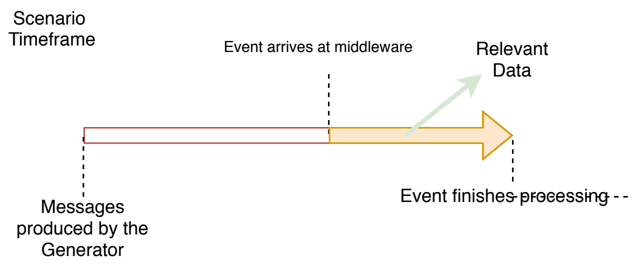

Scenario Timeframe
Event arrives at middleware
Relevant Data
Messages produced by the Generator
Event finishes processing

Figure 7.1: Sequence of actions before an event is sent to the dashboard.

7.5 Event Integrity Testing

Two types of tests other than performance tests can be performed to assure the quality of the software, these being:

- Invalid JSON: Events produced by the generator will be intentionally erroneous and contain fatal structure errors to the point that are ”unparseable” by the JSON parsing function;

  To allow for a better understanding of the middleware's behaviour during this integrity check, a Black Box test using the Equivalence Class Partitioning technique was done, of which the test cases are detailed below:

Input	Valid Equivalence Classes	Invalid Equivalence Classes
JSON message	Input 01: Message with all mandatory fields and 3 metadata variables; Input 02: Message with all mandatory fields and 16 metadata variables; Input 03: Message with all mandatory fields but no metadata variables.	Input 04: Message lacking one or more mandatory fields.

Table 7.1: Black Box Test test cases for invalid JSON formats using the Equivalence Class Partioning technique.

- Invalid Data format: Events that will not follow the parameters or variable limitations defined in the Data Model document, which the middleware will check before doing any kind of processing.

  For this purpose, the same test was done, but this time the parameters tested were the mandatory variables and their type. The following table describes the test cases of this test:

Input	Valid Equivalence Classes	Invalid Equivalence Classes
		Input07: event_id = 1
		Input08: event_id = “!”
		Input09: event_id = “a”
		Input11: agent_type = 1
		Input12: agent_type = “245”
		Input13: agent_type = “#%”
		Input15: agent_id = 2
		Input16: agent_id = “hello”
		Input17: agent_id = “&/(“
	Input05: event_id = “1”	Input19: timestamp = 2
	Input06: event_id = “2147483647”	Input20: timestamp = “May”
	Input10: agent_type = “NEF”	Input21: timestamp = “=)(“
	Input14: agent_id = “5”	Input23: sourceip = 1
JSON message	Input18: timestamp = “2019-05-30”	Input24: sourceip = “jah”
	Input22: sourceip = “95.133.240.54”	Input25: sourceip = “&º^”
	Input26: count = “0”	Input27: count = 0
	Input30: type = “Alert”	Input28: count = “zero”
	Input33: priority = “Low”	Input29: count= “?!?”
	Input37: name = “Sample Event”	Input31: type = 7
	Input41: context = “Sample Context”	Input32: type = “**?”
		Input34: priority = 0
		Input35: priority = “7”
		Input36: priority = “:;:;:”
		Input38: name = 9
		Input39: name = “8”
		Input40: name = “><><”
		Input42: context = 8
		Input43: context = “1”
		Input44: context = “/&%$#”

Table 7.2: Black Box Test test cases for invalid JSON parameter validation using the Equivalence Class Partioning technique.

7.6 Results

The test results were taken as an output to a text file, and then exported to an Excel spreadsheet, where they would be used to calculate several useful metrics and create charts to illustrate some of the key points of this experience. However, there were a couple points that could be taken right after the tests had ended, without even looking at the reported values. We'll discuss these further in this section, but let's first take a look at the global test results.

7.6.1 Blackbox Test Results

Regarding the JSON message format tests, all tests concluded with a success rate of 100%, guaranteeing that an erroneous event containing fatal data structure errors are discarded by the middleware, and do not undergo any type of processing.

The same applies to the JSON parameter validation tests, which also concluded with a success rate of 100%, guaranteeing that events that contain data that does not match the specified limitations and standards defined in the Data Model document are not processed.

7.6.2 General Test Results

The first question that needed answers was: "Can the software perform under the proposed test sets?", meaning, could the software complete all the proposed tests? Table 7.3 illustrates whether the tests could be completed or not for the major test sets:

Test / Application		Middleware	Streams App
Buffer Capacity	10k	V1: Yes V2: Yes	No buffer.
	100k	V1: Yes V2: Yes	
	500k	V1: Yes V2: No	
Event Throughput	1k per minute	V1: Yes V2: Yes	V1: Yes V2: Yes
	10k per minute	V1: Yes V2: Yes	V1: Yes V2: Yes
	100k per minute	V1: No V2: No	V1: Yes V2: Yes

Table 7.3: Completion results for Capacity and Throughput tests done.

7.6.3 Capacity Tests

As previously mentioned, these tests aim to test the capacity of the buffer used by the middleware, as well as the effect that a high number of events have on its overall performance. Chart 7.2 and table 7.4 illustrate the values obtained for the capacity tests done:

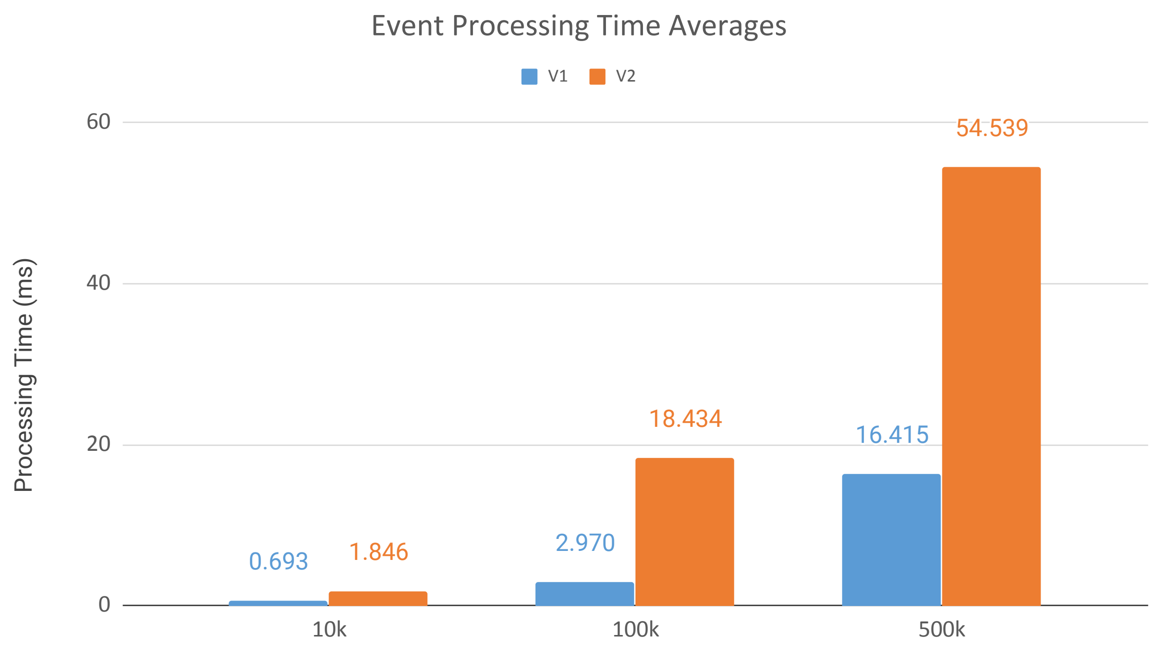


Figure 7.2: Capacity/Buffer Tests - Average event processing time, in milliseconds, per set/variant combination.

Set	Variation	Minimum	Quartile 1	Median	Quartile 3	Maximum	Range	Average	Std. Deviation
10k	**V1**	0.0102	0.0708	0.1224	1.1662	6.5794	6.5692	0.6926	0.9539
	V2	0.0741	0.9153	1.7543	2.6013	7.3469	7.2728	1.8457	1.1060
100k	**V1**	0.0089	0.0465	0.1437	5.1858	83.5888	83.5799	2.9701	4.6042
	V2	0.0723	8.6799	17.4786	26.8181	137.6163	137.5440	18.4342	11.8192
500k	**V1**	0.0086	0.1272	0.6090	27.4776	449.7409	449.7323	16.4145	26.0342
	V2 (300k out of 500k sample size)	0.0749	24.7074	51.8328	79.1531	2971.9506	2971.8856	54.5388	36.1368

Table 7.4: Middleware capacity (buffer) test results, in milliseconds.

As mentioned in chart 7.3, the 500k set for Variant 2 wasn't completed due to hardware limitations. Therefore, the data shown in the table 7.4 that refers to that test has a sample size of 300k, however, this sample is still large enough that allows the observation of the middleware behaviour to large data sets.

As we can see, there's an exponential increase in the average event processing time as the buffer grows, more so if the event has extra metrics to be processed (Variant 2). This behaviour happens because the middleware keeps track of all the metrics collected and uses them to populate the respective tables/charts. After a couple hundred thousand occurrences, this causes the cycle speed of the middleware to drop substantially because it uses mostly arrays of objects to store the events, iterating through these arrays with lengths in the hundreds of thousands every other cycle.

Since the events contain metrics that are created in a session-based paradigm, there's an importance to keep track of previous events, either for debugging purposes or to facilitate general problem-solving. Even though Variant 2 consists of cases of the highest middleware usage possible, and by no means should occur in a production environment, the values obtained by these tests are useful to estimate a "ceiling" or limit to the buffer, as well as discuss in which way to overcome these limitations.

The buffer limits may be implemented in a few different ways which will be discussed later on in this chapter, but for now, let's analyze the processing time growth in-between the sets, to try to estimate a limit number to the events the buffer should support, in a way that preserves performance, but also allows storage for a sufficient number of records and metrics.

The following charts illustrate and compare the growth of the processing time in quartile 1 (25th percentile), 2 (50th percentile / Median), and 3 (75th percentile), for all test sets and its variants:

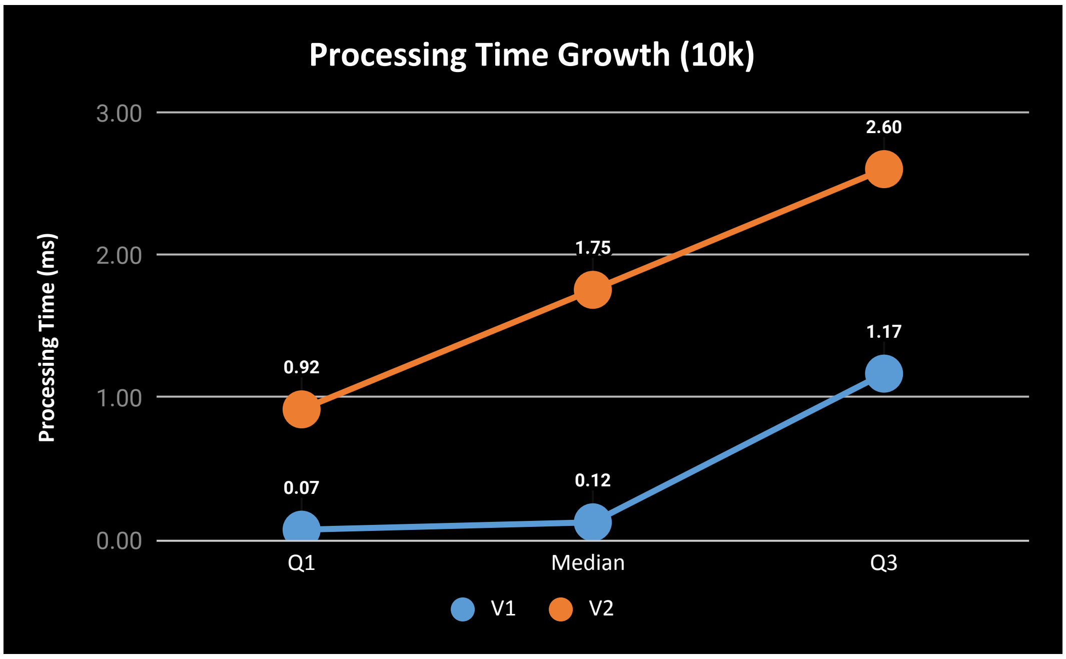


Figure 7.3: Processing time growth across the test, for the 10k set and both variants.

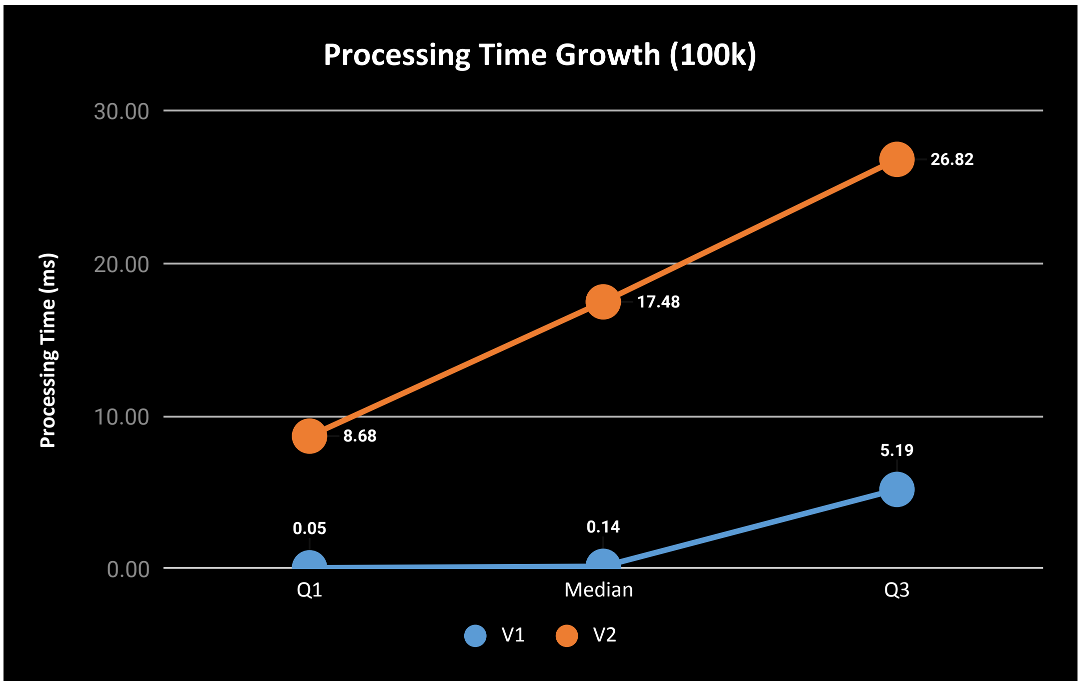


Figure 7.4: Processing time growth across the test, for the 100k set and both variants.

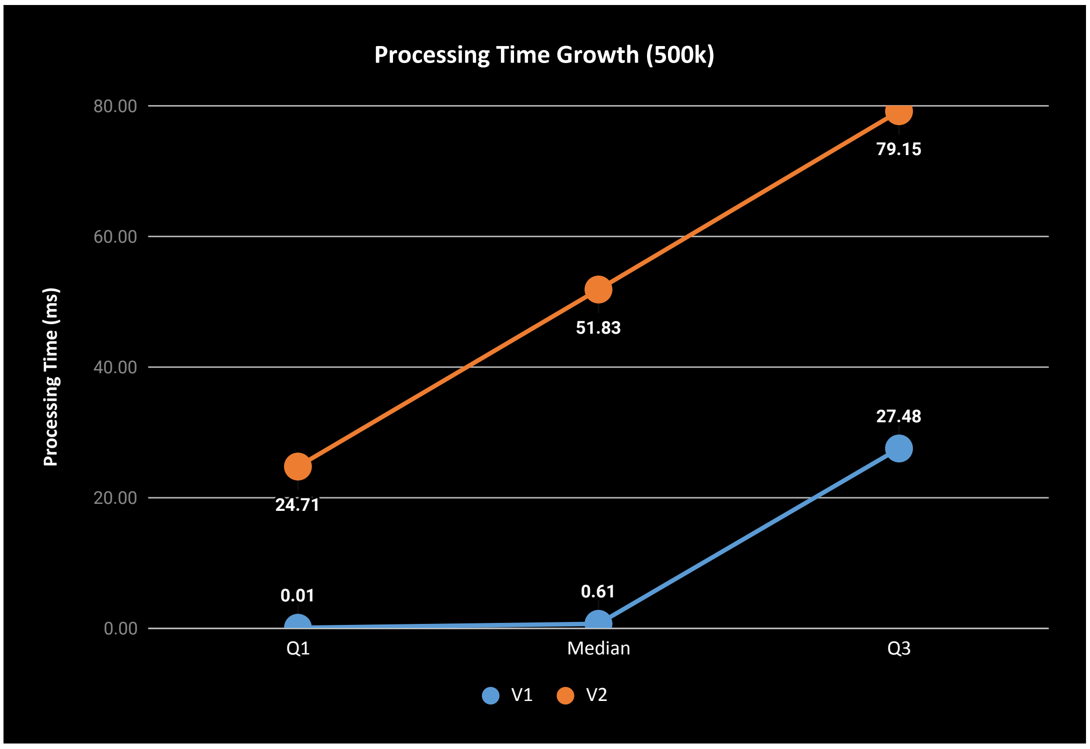


Figure 7.5: Processing time growth across the test, for the 500k set and both variants.

With these charts, we can conclude that the increases in processing time happen the most during the third quartile for Variant 1, while increasing at a steady rate since quartile 1 for for Variant 2. However, for the 500k set, these increases in performance time are much more harsher. Therefore, for a stable middleware performance, I estimate that no more than 100k events should be kept in the middleware at a given time: this guarantees that incoming events have a good processing time while keeping the middleware resource usage at an acceptable level.

In relation to the buffer limits that should be imposed to guarantee a high level of middleware performance, the buffer limiting, or "event purging" can be done in a couple ways:

- Session-based purging: It is agreed with a producer to send an event with a special flag to signal the middleware to purge all the events from that session from the buffer. This requires additional programming and coordination from both the middleware and the partners, so it should be agreed upon before event production from that producer even starts, to confirm that event production is within the defined structure (according to the data model document);

- Timeframe-based purging: Events will be deleted after a certain period of time, cyclically. This timeframe could eventually be agreed with the partner, and in practice, each partner could set his event "purging cycle", depending on whether they needed their data to remain longer on the dashboard. However, this could result in critical data loss in cases of a shorter purging cycle, and reversely, could contribute to starvation of some partners: in cases where one or more partners would choose a rather long "purging cycle", which meant the partners with low cycles had overall less buffer capacity to work with;

- Buffer-Size-based purging: When new events reached the middleware and the buffer was full, or near its capacity, the middleware would automatically purge the oldest data in store, in order to make space for the new data. This could also result in critical data loss since now the data wouldn't be prioritized, but would however solve the problem of buffer starvation.

The implemented method as to now is the Buffer-Size-based purging, since there's no data on how long the partner's data should be kept in the dashboard. In the future, the other possibilities can be explored and programmed accordingly to the partner's needs.

7.6.4 Throughput Tests

Middleware

These tests objectively test both Middleware and Streams App event throughput, with the finality of getting an estimate of how many events per minute these platforms can support, and if these values are suitable to the project's demands.

Referring to table 7.3, we can check that test sets for 1k and 10k per minute concluded successfully, while the 100k per minute concluded non-successfully. The first variant tested in each set was always variant 2, because this implied that if the test was successful for that variant, which is the one with the heaviest data flow, by proxy, it would also be successful for variant 1. Since the 100k per minute test wasn't successful for variant 2, variant 1 was also experimented with, without any success, however.

Figure 7.6 represents the results obtained during the tests:

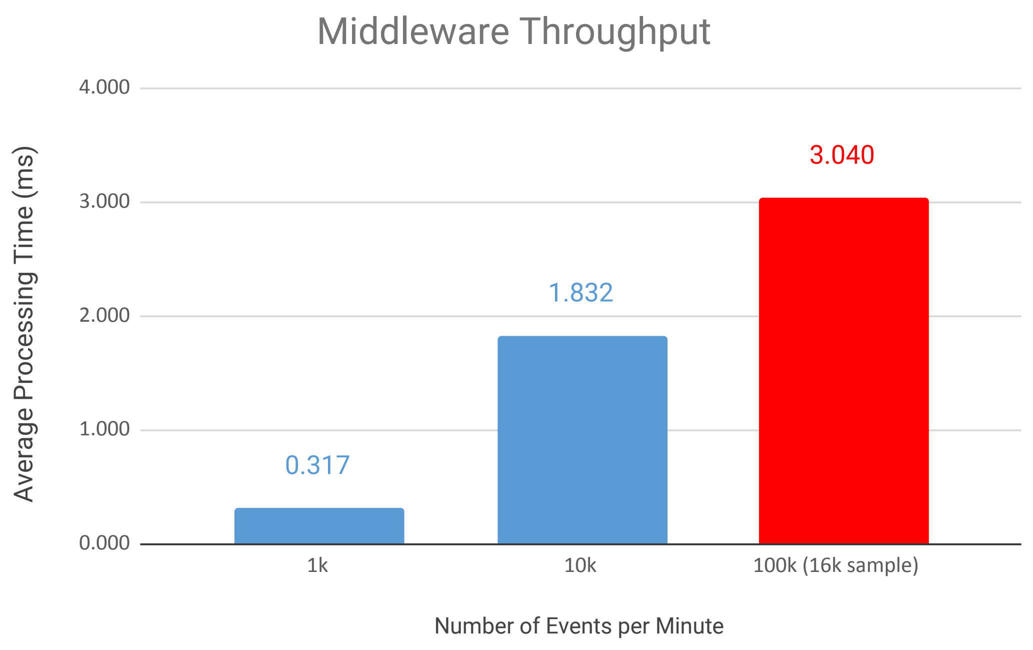


Figure 7.6: Average event processing time of the different sets over periods of one minute, in nanoseconds.

From figure 7.6 we can see that the increase in processing time aligns with the data from the previous capacity-related charts, as it gets progressively higher as the buffer gets filled with events.

For the 100k events per minute case, as it was previously stated, the middleware couldn't handle an influx of events of that dimension, and from the 100k events, only managed to process approximately 16 thousand, making it the sample size for the 100k per minute test.

This means that the Middleware could eventually be optimized to allow for a bigger flow of events, but for the project's demands, it's safe to say that the performance levels are sufficient.

Streams App

In relation to the Streams App, all tests concluded successfully, and it was specially important to test this to make sure the generator could actually create all the ratios stated (1/10/100k per minute), this would erase any uncertainties in relation to the 100k per minute case Middleware test.

Since it was created with the dedicated Kafka Streams API, the performance expected is to be quite superior to the Middleware's. Figure 7.7 represents the results obtained during the tests.

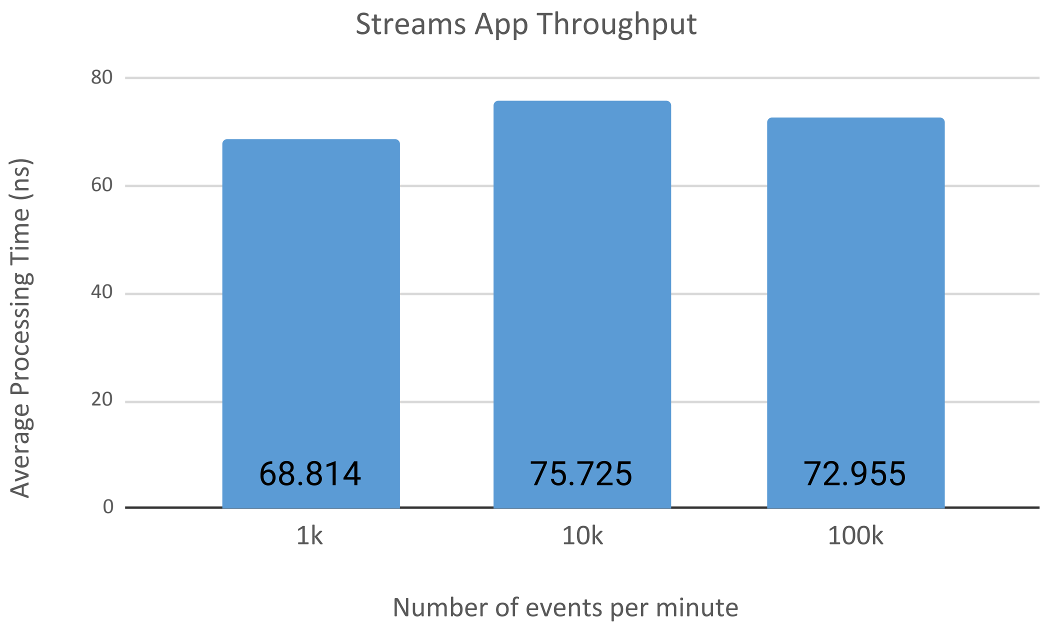


Figure 7.7: Average event processing time of the different sets over periods of one minute, in nanoseconds.

The Streams App completely overthrew the performance levels of the Middleware, getting substantially lower processing times, and eliminating any problem of throughput, allowing for rates of at least 180k events per minute, almost double the amount of the highest set tested. With these results, it is safe to say the Streams App won't need any optimizations to work at the project's demands of performance, and should suffice even if the dashboard was expanded upon.

7.6.5 Observed Behaviour Mid-Test

Middleware

The middleware was visibly impacted by a large buffer size, which was shown during the 500k event sets: while the Variant 1 sets could be completed, it took a massive amount of both time and system memory to hold all that data in a buffer, and the event processing time got severely affected the bigger the buffer got. As to the Variant 2, since it included only events that would suffer extra processing, the system would run out of memory much sooner (not even 300k events in), which would force me to kill the process all-together.

To even be able to complete the Variant 1 of the 500k set, the allocated 16 GB of RAM of the VM were not enough to hold all the data. Therefore, to guarantee that the test could be concluded, 16 additional GB of RAM were allocated (to a total of 32).

As for the throughput impacts in the middleware, the 100k per minute tests could not be concluded successfully, as the middleware could process a maximum of 39k events in a minute for Variant 1, and about half that value for Variant 2, implying that events used for metrics processing take about double the processing time than a regular event.

Streams App

Regarding the Streams App, the throughput test results were the expected, and the application aced all the performance tests, also guaranteeing that the producer used could indeed provide production rates of more than 100k events per minute. Since the Kafka Streams API was built to work directly at the broker level, the App had no problem pushing 100k events per minute, and in fact, could push almost double that value if the event production ratio was left uncontrolled.

There were no buffer tests done to the Streams App because it uses only one event as a tuple to aggregate incoming data. This tuple is then sent to a topic after the defined timeslice has expired, and therefore would make no sense in testing its storing capacity. For the intended purposes of metrics processing, as previously said, the Streams App isn't being used to process extra metrics, due to the fact that these require a session-based paradigm rather than a timeslice-based, which is the one used by the Kafka Streams API. So, for now, the Streams App is purely aggregating incoming low priority events and sending them to the dashboard in intervals of 5 minutes (configurable).

7.7 Partner integration meeting

There was a meeting scheduled for integration purposes with OneSource, to make sure both components were up and running, and could be used in the upcoming App-Intrusion Detection and Prevention System (IDPS) Demo in October. This meeting consisted of partners from OneSource, UbiWhere, our group from University of Coimbra (UC) and several other entities, and served as a discussion ground for development status and what steps to take next.

This meeting happened in the University of Aveiro, in the Institute of Telecommunications, and for my part of development, it consisted of two parts:

- Meeting with our partners from ubiWhere, to talk about the metrics that they would send to us, and what processing needed to be done.
- Meeting with our partners from OneSource, to integrate and test our tools, as well as the version of the dashboard that was deployed in Aveiro's Testbed. This instance of the dashboard would then be used for demonstration purposes in the implementation demo in October.

The partners at OneSource were generally pleased with the integration efforts that were placed creating the platform, and managed to send events to the dashboard with minimum issues. Regarding the metrics processing part, there was also a green light after the partner from OneSource said the processed information was displayed in a simplified and intuitive manner.

When debating what kind of information display to use with the ubiWhere partners, a demonstration was done to OneSource regarding the metrics processing component, which was already implemented at the time. They did then use this demo to try and come up with some ideas of how to display the processed information. We would then discuss issues about the nature of the data they would send for the remainder of the time.

7.8 Validation Test Conclusions

The tests performed helped to gauge the performance level of both Middleware and Streams App, and helped conclude that for the predicted project's load, the developed components should suffice.

However, for larger event flows, it is important to set up a good event "purging" system, in a way that doesn't harm the important data kept in the dashboard, while also removing as much irrelevant data as possible.

With this said, the results estimated that for a good performance level of the dashboard, a limit of one hundred thousand events should be established at the buffer. The tests have proven that higher buffer capacity does affect the event throughput time significantly, and consequently, the increase in system resource usage.

In the future, the buffering mechanism may be expanded upon, or it may change to a different type of data storage method, but for the intended purpose, the performance levels are far better than acceptable. To reduce cluttering and misusage of the buffer space, further aggregation techniques can be employed at the Streams App level to lower the number of events arriving at the dashboard.

This page is intentionally left blank.

Chapter 8

Work Plan

This chapter displays, via Gantt charts, the activities done during each semester and respective duration.

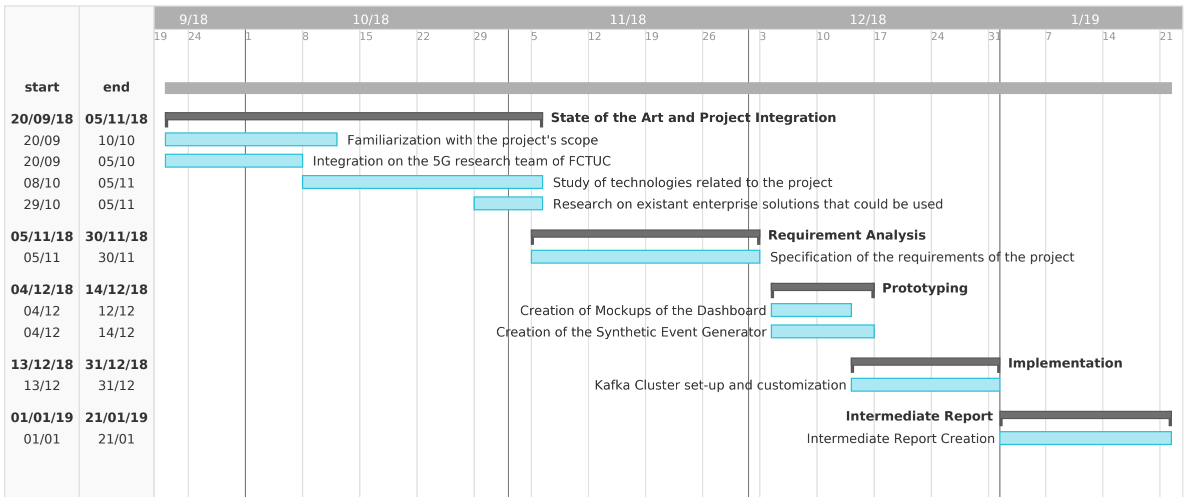


Figure 8.1: Gantt chart displaying the activities done during the first semester.

The first semester consisted of the research part, where a study of the different relevant technologies was done, and debated on which set of options to use. Some tools that would be fundamental to develop and test the platform later on were also created, like the Synthetic Event Generator. Other than that, the time was spent learning to operate with the Kafka cluster, and setting it up to be usable with the remaining future components.

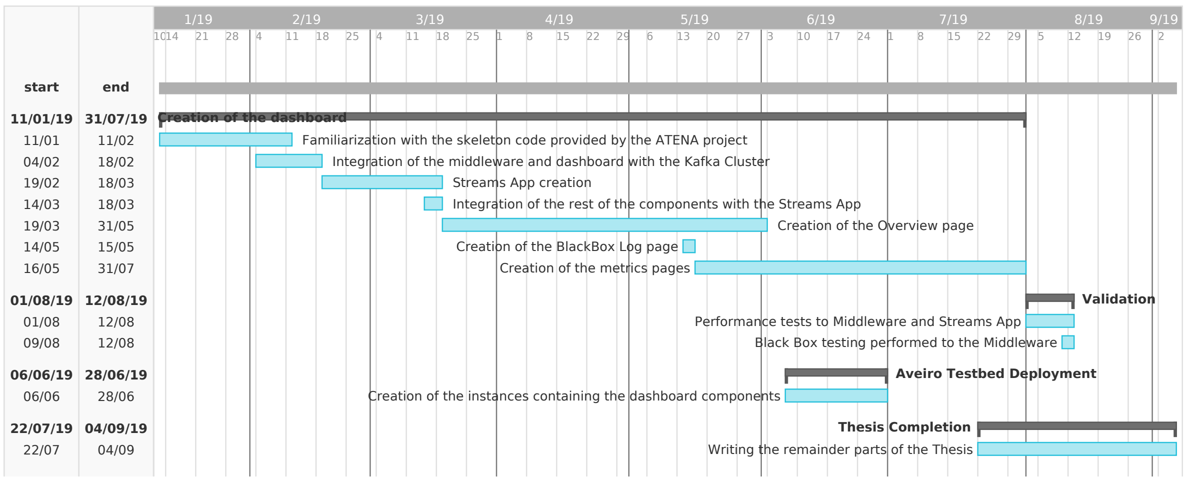


Figure 8.2: Gantt chart displaying the activities done during the second semester.

As for the second semester, after a colleague introduced me to the dashboard skeleton code, a time of adapting and familiarization to both the code and the programming languages occurred, given that I had no prior experience of most of the programming languages used throughout the project.

Then, a simplified middleware and dashboard were built to integrate with the other components of the architecture, namely the Event Generator and the Kafka cluster. After the event flow was tested and working properly, it was time to begin improving the middleware and dashboard.

For aggregation purposes, the Streams App was created, but given that the programming language and the Streams API was unknown to me, some time was spent setting up the programming environment and learning about the language itself and the API, as well as learning how to integrate the API into the code and adapt it for the project's needs. Also, at the time it was unknown to me what processing should be applied to the metrics received (mainly because the data types and context of such metrics were unknown to the FCTUC group at the time), so the processing part was to be developed later once we received more information regarding this topic from the partners.

This took about a month, which was more than expected, and caused some delay on some other parts of the dashboard's development.

After this was done, it was then integrated onto the remaining components, and then the work turned towards deciding what visual elements to put on the main dashboard page. This implied that it was necessary to study to some extent other dashboards, to try and take some relevant elements and implement them in our project.

Meanwhile, the metrics pages were also being separately developed, as we got more information about the metrics that would eventually arrive at the dashboard. These pages got a development green light from the partners at the Aveiro meeting, as they were pleased to see some statistical information about the metrics they sent. Some weeks prior, the testbed deployment was also done, to test the integration with other partners, which was done at this same meeting.

After the metrics pages were finished, a Prometheus agent integration was done with the dashboard's main page, allowing the monitoring of the topics used by the platform.

After the main development of the dashboard was completed, the validation process was done, to ensure that the performance was up to par with the project's needs.

Finally, the remaining time was used to the writing of the final document.

This page is intentionally left blank.

Chapter 9

Conclusions

The final product of this work included all the intended features that were planned, allowing users to easily and efficiently obtain details about the platform's status, as well as access to relevant data used to diagnose and solve problems in the 5G network.

The choice of each component that was created for the dashboard was made with the help of the study of the many technologies available in the market, to create a flexible and modular, yet scalable solution.

The testing conducted to the main dashboard components gave us some indicators of how it would perform in a production environment, and it was concluded that the performance levels do indeed meet the expected processing load of the project.

Some work for the future may include more metric processing profiles, a better buffering system, with the implementation of one or more of the purging methods that were discussed. The UI can also be refined, as well as some quality of life improvements added.

Concluding, during the duration of my part in the project, I came in contact with many new technologies, people and experiences. All of these helped nourish my knowledge and perception of the project's scope and the individual pieces that together form the 5G architecture.

Notably, I personally enjoyed learning about Kafka and its applications, since it is an emerging technology and will very likely impact the microservices world a lot more than we currently think. The soft skills I acquired during my work in the project will surely help me in the future.

This page is intentionally left blank.

References

[1] Apache kafka documentation. https://kafka.apache.org/documentation/#introduction. Accessed: 2018-12-16.

[2] Apache kafka streams- a closer look. https://docs.confluent.io/current/streams/introduction.html#the-kafka-streams-api-in-a-nutshell. Accessed: 2018-12-16.

[3] Atena (advanced tools to assess and mitigate the criticality of ict components and their dependencies over critical infrastructures) h2020 project (h2020-ds-2015-1 project 700581). https://www.atena-h2020.eu. Accessed: 2019-01-14.

[4] Big data dummy - kafka streams. https://bigdatadummy.com/2018/10/15/kafka-streams/. Accessed: 2019-01-14.

[5] Docker vs kubernetes vs apache mesos. https://mesosphere.com/blog/docker-vs-kubernetes-vs-apache-mesos/. Accessed: 2018-12-16.

[6] Docker vs. lxc vs. rkt. https://stackshare.io/stackups/docker-vs-lxc-vs-rkt. Accessed: 2019-01-03.

[7] Introducing json. https://www.json.org/. Accessed: 2019-08-20.

[8] An introduction to microservices. https://opensource.com/resources/what-are-microservices. Accessed: 2018-12-14.

[9] Kubernetes: An overview. https://thenewstack.io/kubernetes-an-overview/. Accessed: 2018-12-16.

[10] Lxc and lxd: Explaining linux containers. https://www.sumologic.com/blog/code/lxc-lxd-explaining-linux-containers/. Accessed: 2018-12-16.

[11] Mesosphere dc/os: The operating system of the distributed cloud. https://softwareengineeringdaily.com/2018/11/30/mesosphere-dc-os-the-operating-system-of-the-distributed-cloud/. Accessed: 2018-12-16.

[12] Mobilizador 5g - deliverable 2.2. https://5go.pt/resultados. Accessed: 2019-01-14.

[13] Monitoring and evaluation. http://web.mit.edu/urbanupgrading/upgrading/issues-tools/tools/monitoring-eval.html. Accessed: 2018-12-14.

[14] Network slicing. https://www.ericsson.com/en/digital-services/trending/network-slicing. Accessed: 2018-12-15.

[15] Open source tools for docker security. https://techbeacon.com/security/10-top-open-source-tools-docker-security. Accessed: 2019-08-25.

[16] Overview — prometheus. `https://prometheus.io/docs/introduction/overview/`. Accessed: 2019-01-06.

[17] Swarm mode overview. `https://docs.docker.com/engine/swarm/`. Accessed: 2018-12-16.

[18] Understanding when to use rabbitmq or apache kafka. `https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka`. Accessed: 2018-12-16.

[19] What is a rkt container technique? should you use it? `https://bobcares.com/blog/rkt-rocket-container-technology-use/`. Accessed: 2018-12-16.

[20] What is docker? `https://www.redhat.com/en/topics/containers/what-is-docker`. Accessed: 2018-12-15.

[21] What is kubernetes? `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. Accessed: 2018-12-16.

[22] Why avro for kafka data? `https://www.confluent.io/blog/avro-kafka-data/`. Accessed: 2019-08-20.

[23] Understanding and hardening linux containers. In *Understanding and Hardening Linux Containers*, page 123. NCC Group, 2016.

[24] 5GPPP Architecture Working Group. View on 5g architecture. In *View on 5G Architecture (Version 2.0)*, page 113. 5GPPP, 2017.

[25] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246. IEEE, 2017.

[26] VMWare. Virtualization Overview. Technical report, VMWare, 2018.