Universidade de Aveiro
2017

Departamento de Eletrónica,
Telecomunicações e Informática

Nuno
Henriques

Configuração Automática de Plataforma de Gestão de Desempenho em Ambientes NFV e SDN

Automated Configuration of a Performance Management Platform in NFV and SDN Environments

Universidade de Aveiro Departamento de Eletrónica,
Telecomunicações e Informática
2017

Nuno Henriques

Configuração Automática de Plataforma de Gestão de Desempenho em Ambientes NFV e SDN

Automated Configuration of a Performance Management Platform in NFV and SDN Environments

"Continuous effort - not strength or intelligence - is the key to unlocking our potential."

— Winston Churchill

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2017

Nuno Henriques

Configuração Automática de Plataforma de Gestão de Desempenho em Ambientes NFV e SDN

Automated Configuration of a Performance Management Platform in NFV and SDN Environments

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica da Professora Doutora Susana Sargento, Professora Associada com Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Pedro Miguel Naia Neves, Gestor Tecnológico na Altice Labs.

Dedico este trabalho ao meu avô, pais, amigos e
em particular à Anabela pelo apoio incondicional

o júri / the jury

presidente / president	Prof. Doutor João Paulo Silva Barraca professor auxiliar da Universidade de Aveiro
vogais / examiners committee	Doutor Pedro Miguel Naia Neves gestor tecnológico na Altice Labs (co-orientador)
	Prof. Doutor Paulo Alexandre Ferreira Simões professor auxiliar da Universidade de Coimbra (arguente)

agradecimentos / acknowledgements

Em primeiro lugar gostaria de agradecer aos meus orientadores, Professora Susana Sargento e Pedro Neves, pela orientação rigorosa, por terem acreditado, confiado e apostado em mim, proporcionando-me a possibilidade de poder participar num projeto desafiante e enriquecedor que tanto contribuiu para o desenvolvimento das minhas competências pessoais, profissionais e académicas.
Agradeço também às equipas da Altice Labs e dos restantes parceiros do SELFNET com as quais trabalhei pelo seu espírito de equipa e entreajuda, pelas discussões produtivas e partilha de conhecimentos, levando a bom porto os objetivos do projeto e em última análise do trabalho que desenvolvi e que é apresentado nesta dissertação.
Aos amigos que fiz ao longo do meu percurso académico e com os quais tive o prazer de trabalhar e partilhar momentos determinantes das nossas vidas, pelo apoio mútuo quando as coisas não corriam de feição, e não menos importante pelos momentos de convívio e acima de tudo a amizade que perdurará.
Finalizo com um agradecimento especial ao meus pais pelo esforço financeiro e educação que me proporcionaram fazendo de mim a pessoa que sou hoje, e em particular à minha cara-metade pela motivação constante e apoio incondicional.

Palavras Chave

5G, SON, SDN, NFV, SELFNET, Virtualização, Monitoria, Gestão, Catalog-Driven

Resumo

Com o 5G prestes a chegar nos próximos três anos, esta próxima geração de redes móveis irá transformar a indústria de telecomunicações móveis com um impacto profundo nos seus clientes assim como nas tecnologias e arquiteturas de redes. As redes programáveis (SDN), em conjunto com a virtualização de funções de rede (NFV), irão desempenhar papéis vitais para as operadoras na sua migração do 4G para o 5G, permitindo-as escalar as suas redes rapidamente. Esta dissertação irá apresentar um trabalho de investigação realizado sobre este novo paradigma de virtualização e programação de redes, concentrando-se no domínio da gestão de desempenho, supervisionamento e monitoria, abordando cenários de redes auto-organizadas (SON) num contexto NFV/SDN, sendo um destes cenários a deteção e predição de potenciais anomalias de redes e serviços. O trabalho de investigação foi enquadrado num projeto de I&D designado SELFNET *(A Framework for Self-Organized Network Management in Virtualized and Software Defined Networks)* financiado pela Comissão Europeia no âmbito do programa H2020 5G-PPP, sendo a Altice Labs um dos parceiros participantes deste projeto. Avanços em sistemas de gestão de desempenho em cenários 5G requerem agregação, correlação e análise de dados recolhidos destes elementos de rede programáveis e virtualizados. Ferramentas de monitoria *open-source* e ferramentas *catalog-driven* foram integradas ou desenvolvidas com este propósito, e os resultados mostram que estas preencheram os requisitos do projeto SELFNET com sucesso. As plataformas de gestão de desempenho das operadoras de rede atualmente em produção estão concebidas para redes não virtualizadas (non-NFV) e não programáveis (non-SDN), e o conhecimento adquirido durante este trabalho de investigação permitiu à Altice Labs compreender como a sua plataforma de gestão de desempenho (Altaia) terá que evoluir por forma a preparar-se para a próxima geração de redes móveis 5G.

Keywords 5G, SON, SDN, NFV, SELFNET, Virtualization, Monitoring, Management, Catalog-Driven

Abstract

With 5G set to arrive within the next three years, this next-generation of mobile networks will transform the mobile industry with a profound impact both on its customers as well as on the existing technologies and network architectures. Software-Defined Networking (SDN), together with Network Functions Virtualization (NFV), are going to play key roles for the operators as they prepare the migration from 4G to 5G allowing them to quickly scale their networks. This dissertation will present a research work done on this new paradigm of virtualized and programmable networks focusing on the performance management, supervision and monitoring domains, aiming to address Self-Organizing Networks (SON) scenarios in a NFV/SDN context, with one of the scenarios being the detection and prediction of potential network and service anomalies. The research work itself was done while participating in a R&D project designated SELFNET (A Framework for Self-Organized Network Management in Virtualized and Software Defined Networks) funded by the European Commission under the H2020 5G-PPP programme, with Altice Labs being one of the participating partners of this project. Performance management system advancements in a 5G scenario require aggregation, correlation and analysis of data gathered from these virtualized and programmable network elements. Both open-source monitoring tools and customized catalog-driven tools were either integrated on or developed with this purpose, and the results show that they were able to successfully address these requirements of the SELFNET project. Current performance management platforms of the network operators in production are designed for non virtualized (non-NFV) and non programmable (non-SDN) networks, and the knowledge gathered while doing this research work allowed Altice Labs to understand how its Altaia performance management platform must evolve in order to be prepared for the upcoming 5G next generation mobile networks.

Contents

List of Figures

List of Tables

Glossary

ACID	Atomicity, Consistency, Isolation, Durability
ACM	Aggregation Configuration Manager
AE	Aggregation Engine
AMQP	Advanced Message Queuing Protocol
ANM	Autonomic Network Management
API	Application Programming Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BAF	Batch Aggregation Framework
BGP	Border Gateway Protocol
CapEx	Capital Expenditures
CnC	Command & Control (of a Botnet)
CDP	Cisco Discovery Protocol
CEP	Complex Event Processing
CEPF	Complex Event Processing Framework
CI	Congestion Index
CLI	Command Line Interface
CQL	Cassandra Query Language
CRUD	Create, Read, Update and Delete
CS	Circuit Switching
CSP	Communication Service Providers
DB	Database
DC-PoP	Data Center Point-of-Presence
DNS	Domain Name System
DPI	Deep Packet Inspection
DSL	Digital Subscriber Line
E2E	End-To-End
EPC	Evolved Packet Core
FMA	Flow Monitoring Agent
FMC	Follow-Me Cloud
FTTX	Fiber To The X
HoN	Health of Network
HSS	Home Subscriber Server
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol
IPTV	Internet Protocol Television
IPS	Intrusion Prevention System
IPTV	Internet Protocol Television
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LLDP	Link Layer Discovery Protocol

LTE	Long-Term Evolution
MC	Monitoring Catalog
ML	Machine Learning
MCC	Mobile Cloud Computing
MCDM	Monitoring Catalog Data Model
MME	Mobility Management Entity
MNO	Mobile Network Operator
MPLS	Multiprotocol Label Switching
NFaaS	Network Functions-as-a-Service
NFV	Network Function Virtualization
NMS	Network Management System
OAI	OpenAPI Initiative
OAS	OpenAPI Specification
OF	OpenFlow
OpEx	Operational Expenses
OSPF	Open Shortest Path First
OTT	Over-The-Top
PDN	Packet Data Network
PNF	Physical Network Function
PS	Packet Switching
QoE	Quality of Experience
QoS	Quality of Service
RADM	Raw Aggregation Data Model
RAN	Radio Access Network
REST	Representational State Transfer
RRH	Remote Radio Heads
SDH	Synchronous Digital Hierarchy
SDN	Software Defined Network
SH-UC	Self-Healing Use Case
SNMP	Simple Network Management Protocol
SON	Self-Organizing Network
SO-UC	Self-Optimization Use Case
SP-UC	Self-Protection Use Case
SQL	Structured Query Language
TCO	Total Cost of Ownership
TSM	Time-Structured Merge Tree
UHD	Ultra High Definition
UMF	Unified Management Framework
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
UUID	Universally Unique Identifier
VM	Virtual Machine
VNF	Virtual Network Function
WSGI	Web Server Gateway Interface

CHAPTER 1

Introduction

With 5G set to arrive within the next three years, this next-generation of mobile networks beyond the 4G LTE mobile networks existing today will transform the mobile industry with a profound impact both on its customers as well on the existing technologies. In a new era in which connectivity will become increasingly fluid and flexible, networks will need to adapt to applications and performance will be tailored precisely to the needs of the user.

The definition of 5G is still very fluid, but as we move closer to 2020, its vision is becoming clearer, with many experts saying it will feature speeds exceptionally fast (20 Gbps or higher) [1][2] and extremely low latencies (mere milliseconds), allowing not only people to be connected to each other as it happens today, but also machines, city infrastructures, automobiles, among other device-to-device communications, envisioning 5G as a network for the Internet-of-Things (IoT).

With such a vast amount of devices connected to the networks, it is expected a growth on traffic, although it will depend upon how the network will be used. For instance, video traffic is expected to grow significantly making it necessary to provide much higher speeds for applications such as streaming video, video conferencing, and even virtual reality.

Self-Organizing Networks (SON) will likely be a key factor in the radio access portion, and Software-Defined Networking (SDN) together with Network Functions Virtualization (NFV) are also going to play key roles for the operators as they prepare the migration from 4G to 5G allowing them to scale their networks quickly.

This document will present a research work done on this new paradigm of virtualized and programmable networks focusing on the performance management, supervision and monitoring domains. The research work itself was done while participating in a R&D project designated SELFNET (A Framework for Self-Organized Network Management in Virtualized and Software Defined Networks) [3] funded by the European Commission under the H2020 5G-PPP programme [4][5], with Altice Labs[1] being one of the participating partners of this project.

1.1 Altaia

Altaia[6] is an unified and convergent end-to-end assurance solution for Communication Service Providers (CSP) developed by Altice Labs, which provides a full set of tools defining an

[1] `http://www.alticelabs.com/en/`

integrated and unified performance management platform for multi-service, multi-technology and multi-vendor environments.

It comprises a scalable performance analytic engine and an extended range of technology packs designed to collect performance statistics and configuration management data from Network Management Systems (NMSs) and external data sources, providing end users with an easy and flexible access to performance data.

This solution supports multiple network technologies and services such as 2G, 3G and Long Term Evolution (LTE) Radio Access Networks (RANs) with full support for a vast number of vendor products, xDSL and FTTX access networks, IP/MPLS, Mobile Core (CS, PS and EPC), ATM, SDH, and so on.

The system is devised to collect, calculate and analyze millions of indicators per hour, using probes to provide QoS and QoE analysis; however, it is tailored for traditional networks and services, but not for non-virtualized (non NFV) and non-programmable (non SDN) environments.

1.2 SELFNET

SELFNET is a project supported by the European Commission Horizon 2020 Programme with the purpose to design and implement an autonomic network management framework with self-organizing capabilities in managing network infrastructures, by automatically detecting and mitigating a range of common network problems that are currently still being manually addressed by network operators, thereby significantly reducing operational costs and improving user experience.

It explores a smart integration of state-of-the-art technologies in Software-Defined Networks (SDN), Network Function Virtualization (NFV), Self-Organizing Networks (SON), Cloud computing, Artificial intelligence, Quality of Experience (QoE) and next generation networking to provide a novel intelligent network management framework that is capable of assisting network operators in key management tasks:

– automated network monitoring by the automatic deployment of NFV applications to facilitate system-wide awareness of Health of Network metrics to have more direct and precise knowledge about the real status of the network;

– autonomic network maintenance by defining high-level tactical measures and enabling autonomic corrective and preventive actions against existing or potential network problems.

The project is driven by use cases, specifically designed to address major network management problems and to substantially reduce operational costs of network operators, by automating a significant number of current labour-intensive network management tasks:

- **Self-Protection** - Capabilities against distributed cyber-attacks;
- **Self-Healing** - Capabilities against network failures;
- **Self-Optimization** - Capabilities to dynamically improve the performance of the network and the QoE of the users.

Further details on these use cases are described in section 4.3, as well as their impact on the architecture of the project.

1.3 Motivation and Objectives

The Altaia performance management platform in production supports a big set of tools to make performance measurements on the network (e.g. LTE) and its underlying services (e.g IPTV, High-Speed Internet). The platform gathers, processes and analyzes information coming from non virtualized (non-NFV) and non programmable (non-SDN) resources.

The static nature of these non virtualized and non programmable resources allows the configuration processes of the information models to be enforced manually. However, in a NFV/SDN context, the agility required for the onboarding and deployment of new functions increases considerably, with the support of automatic configuration processes for the information models of the network functions and virtualized services becoming necessary.

In catalog-driven architectures, the information models of the virtualization network functions are typically stored on a central catalog of the network operator. After they are stored/onboarded on the catalog, mechanisms of publish/subscribe (Pub/Sub) are used to propagate the information models to the components of the architecture that require them.

The SELFNET project provides the necessary research platform and environment to develop a proof of concept for these requirements, allowing Altice Labs to understand how the Altaia performance management platform must evolve in order to be prepared for the upcoming 5G next generation mobile networks.

Furthermore, the performance management system advancements in a 5G scenario also requires aggregation, correlation and analysis of data gathered from these virtualized and programmable network elements. The data itself can be processed either in batch (non realtime processing) and streaming (realtime processing) to create indicators which will be used to generate Health-of-Network (HoN) symptoms to be processed by Machine Learning (ML) algorithms, endowing the network with the necessary intelligence to automatically react upon eminent threats and failures.

With that being said, the scope of this thesis will be focused on three specific components or services of the SELFNET's architecture that were either developed from scratch or studied and integrated within its logical structure throughout the research work:

- **Monasca**: A multi-tenant, highly scalable, performant, fault-tolerant Monitoring-as-a-Service with threshold capabilities, used to store the produced batch aggregated metrics - discussed in sections 3.3.1, 4.2.2.2, 5.1.1.1, 5.1.1.3, 6.1 and 7.3.1;

- **Monitoring Catalog**: a catalog devised to comprise the information on the sensors available for deployment on the platform (its integration with the orchestration layer is out of the scope of this work though), as well the batch/stream aggregation and thresholding rules that need to be enforced on the aggregation layer (within the scope of this work) - discussed in sections 4.2.1.1, 5.1.2, 6.2 and 7.3.2;

- **Aggregation Configuration Manager**: the component responsible for the enforcement of the aggregation and thresholding rules on the aggregation layer - discussed in sections 4.2.2.1, 6.3 and 7.3.3.

1.4 Outline

In order to better understand the remaining structure of this document, a short description of the chapters will follow:

- **Chapter 2 - State of the Art**: presents the state of the art work and developments on the area taking into account some relevant research projects, products and research papers, followed by a discussion;
- **Chapter 3 - Technologies**: summarizes and compares key technologies related to the scope of this work and provides the reasoning behind the choices that were made on each functional area;
- **Chapter 4 - Architecture**: provides the high level architecture of the SELFNET project, focusing on the core components of this thesis work: Monasca, Monitoring Catalog and the Aggregation Configuration Manager;
- **Chapter 5 - Interfaces, Data Sources and Data Models**: the most relevant interfaces and data sources will be described here, as well as the data models that were used to normalize the data that flows inwards and outwards the Monitoring & Analysis layer;
- **Chapter 6 - Implementation**: covers the architecture of the Monasca component in detail, as well as the architecture and implementation of the Monitoring Catalog and the Aggregation Configuration Manager;
- **Chapter 7 - Application and Results**: describes the practical applications of the developed components and presents the results obtained;
- **Chapter 8 - Conclusion and Future Work**: the last chapter provides the conclusions over the developed work including the obtained results, and discusses the future work.

CHAPTER 2

State of the Art

This chapter will approach some of the commercial products available in the market today, as well as ongoing or already done research work in the 5G field that can relate to some degree or extent with the work presented in this dissertation.

Many researchers and vendors of network equipment/solutions already have their eyes set in the future of networking, engaging the surfacing 5G research initiatives to gather the best possible knowledge on this area of expertise, in order to align and evolve the existing products and solutions or simply to contribute to the upcoming 5G standards.

With that in mind, section section 2.1 will be dedicated to research projects, section 2.2 to research papers and section 2.3 to commercial products, followed by a discussion at the end of the chapter.

2.1 Research Projects

2.1.1 UNIFY

UNIFY [7] was a project co-funded by the European Commission that aimed to increase the flexibility in traditional telecommunication infrastructures, envisioning full network and service virtualization to enable rich and flexible services and operational efficiency in order to reduce operating costs.

Research, development and evaluations were done to achieve solutions with the ability to orchestrate, verify and observe end-to-end service delivery from home and enterprise networks through aggregation and core networks to data centres.

The service architecture comprised a unified view of the infrastructure, covering both the traditional network and the data centre entities, aiming for reduced operational costs by removing the need of on-site hardware upgrades, taking advantage of Software Defined Networking (SDN) and networking virtualization technologies.

2.1.2 T-NOVA

T-NOVA [8] was an integrated project co-funded by the European Commission, where Altice Labs was also a participating partner, with the goal of promoting the Network Function Virtualization (NFV) concepts by introducing a novel enabling framework allowing network operators to deploy Virtual Network Functions (VNFs) both for their own needs as well as to their customers as value-added services.

Virtual network appliances (gateways, proxies, firewalls, transcoders, analyzers etc.) would then be provided on-demand as-a-Service, so the network operators would not need to acquire, install and maintain specialized hardware at the customers' premises.

The project devised a management and orchestration platform, providing automated provisioning, configuration, monitoring and optimization of Network Functions-as-a-Service (NFaaS) over virtualized infrastructures, leveraging and enhancing cloud management architectures for elastic provisioning and allocation of resources assigned to the hosting of network functions.

2.1.3 SLICENET

SLICENET [9] is one of the recently started and ongoing 5G-PPP [5] projects under the Horizon 2020 initiative, aspiring to be one of the most important innovations in communications of the decade due to its role in maximizing network resource sharing, optimizing flexibility to meet diverging requirements from diverse vertical businesses, and upgrading operational capabilities to offer configurable warranties in Quality of Service (QoS) and/or Quality of Experience (QoE).

The project is use-case driven, experimenting with three use-cases:

- **Smart Grid Self-Healing**: increase automation in power distribution with self-healing solutions towards a smarter grid;
- **eHealth Smart / Connected Ambulance**: using an ambulance as a connection hub (or mobile edge) for the emergency medical equipment and wearable devices to provide advancements on the emergency services;
- **Smart City**: remote water metering and intelligent public lighting system in a city in Romania, to assess the various technical and operational Key Performance Indicators (KPIs) against their initial *status quo*.

Aiming to maximize the potential of 5G infrastructures and their services, it will resort to advanced software networking and cognitive network management, enabling infrastructure sharing across multiple operator domains in SDN/NFV networks, targeting truly end-to-end (E2E) slicing through a highly innovative slice provisioning, control, management and orchestration framework, oriented to the verticals' QoE.

2.1.4 5G-NORMA

The 5G NORMA [10] project is yet another one of the 5G-PPP [5] projects under the Horizon 2020 framework, aimed to develop a novel mobile network architecture to provide the required adaptability to efficiently handle fluctuations in traffic demand resulting from heterogeneous and dynamically changing services.

The project explores concepts from Software-Defined Networking (SDN), Network Function Virtualization (NFV) and multi-tenancy to leverage the adaptability and efficiency of the network, enabling an inherent and dynamic sharing and distribution of network resources between operators in order to increase their revenue, leading to enhanced and flexible 5G base stations, software-based centralized controllers and software-based Radio Access Networks (RAN).

2.2 Research Papers

2.2.1 Autonomics and SDN for Self-Organizing Networks

This work [11] studies the relationship between Autonomic Network Management (ANM) and Software-Defined Networking (SDN) on Long Term Evolution (LTE) Self-Organizing Networks (SONs) environments, considering that the ANM and SDN paradigms share a few common goals with complementary levels of abstractions and expectations.

The researchers considered that the work previously done on the Unified Management Framework (UMF) explored by the UniverSelf project [12], which focused on higher level self-functionality, often assumed as given a fictional adaptation layer between the autonomics and the managed infrastructure, while the SDN architectures provided an uniform control substrate for the programmatic management of network resources.

However, while flow-based controls in core networks and data centers became popular, similar widely-adopted abstractions had yet to be defined for radio access and SONs, and the research work aimed to create a novel abstraction layer designed to realize SON programmability. A controller prototype designated Autonomic SDN (AutoSDN) was then devised to be integrated with UMF for self-optimization on LTE-Advanced heterogeneous networks, enabling SON functions to be provided by 3rd parties and to be hot-plugged into the network.

2.2.2 A Virtual SDN-Enabled LTE EPC Architecture: A Case Study for S-/P-Gateways Functions

This paper [13] focuses on mobile core network nodes such as the MME, HSS and Serving-/PDN-Gateway as standardized for the LTE Evolved Packet Core (EPC). The authors stated that moving all EPC network nodes completely into a data center to handle the data traffic via SDN-enabled switches could be one straightforward solution for a virtualized EPC architecture, but that would keep the conventional monolithic architecture unchanged.

Thus, they suggested that a possible split in the EPC functionality between a centralized data center and operator's transport network elements could be needed to provide the desired flexibility, performance and TCO reduction

The work comprised an analysis on EPC nodes in order to classify their functions according to their impact on data-plane and control-plane processing, proposing a mapping for these functions on four alternative deployment frameworks based on SDN and OpenFlow (OF), investigating the OF implementation's capability to realize basic core operations such as QoS, data classification, tunneling and charging.

Their analysis showed that functions that involved high data packet processing had more potential to be kept on the data-plane network element, i.e. realized by an OpenFlow Switch, and they argued for an enhanced OF network element NE+ containing additional network functions next to the basic OF protocol.

2.2.3 Self-Healing Mechanisms for Software Defined Networks

The goal of this paper [14] is to provide SDN with fault management capabilities by using autonomic principles like self-healing mechanisms, proposing a generic self-healing approach that relies on Bayesian Networks [15] for the diagnosis block, and it is applied to a centralized SDN infrastructure to demonstrate its functioning in the presence of faults.

The authors claim that even though programmable networks brought by SDNs are perceived by the operators as a cornerstone to reduce the time to deploy new services, to augment the flexibility and to adapt network resources to customer needs at runtime, and despite the vulnerabilities identified due to the centralization of the intelligence on SDN, its research is more centered on forwarding traffic and reconfiguration issues, not considering to a great extent the fault management aspects of the control plane.

Note: 'Softwarized 5G networks resiliency with self-healing' [16] is another research paper from the same authors approaching the same theme.

2.2.4 5G on the Horizon: Key Challenges for the Radio-Access Network

This research work [17] discusses what key challenges the Radio Access Networks (RANs) will face from the numerous devices and networks that will be interconnected, as well as from the traffic demand that will constantly rise as we move towards the 5$^{\text{th}}$ generation (5G) of wireless/mobile broadband.

Also, as mixed usage of cells of diverse sizes and access points with different characteristics and technologies in an operating environment are necessary, heterogeneity will also be a feature that is expected to characterize the emerging wireless world.

With wireless networks posing specific requirements that needed to be fulfilled, the authors considered that approaches for introducing intelligence would need to be investigated by the research community in order to provide energy- and cost-efficient solutions, at which a certain application/service/quality provision would be achieved.

They then investigated the introduction of intelligence in heterogeneous network deployments and the cloud radio-access network (RAN), elaborating on emerging enabling technologies for applying intelligence focused on the concepts of Software-Defined Networking (SDN) and Network Function Virtualization (NFV), providing an overview for delivering intelligence towards the 5G of wireless/mobile broadband by taking into account the complex context of operation and essential requirements such as QoE, energy efficiency, cost efficiency, and resource efficiency.

2.2.5 Toward Elastic Distributed SDN/NFV Controller for 5G Mobile Cloud Management Systems

This paper [18] again draws attention to the expected rise in mobile data traffic and the expectations in the future 5G mobile network architecture to offer capacities to accommodate and to meet further stringent latency and reliability requirements to support diverse high data rate applications and services.

The authors state that the emerging Mobile Cloud Computing (MCC), as a key paradigm, promises to increase the capability of mobile devices through provisioning of computational resources on demand, enabling resource-constrained mobile devices to offload their processing and storage requirements to the cloud infrastructure.

The authors also state that the Software-Defined Networking (SDN) paradigm allows the decoupling of the control and data planes of traditional networks, providing programmability and flexibility, allowing the network to dynamically adapt to change traffic patterns and user demands.

However, while the SDN implementations are gaining momentum, the control plane is still suffering from scalability and performance concerns for a very large network. On their paper, they addressed the scalability and performance issues in the context of 5G mobile networks by introducing a novel SDN/OpenFlow-based architecture and control plane framework tailored for MCC-based systems, and more specifically for FMC-based systems (an emerging concept that allows seamless migration of services according to the corresponding users mobility), where mobile nodes and network services are subject to constraints of movements and migrations.

Their approach consisted in a distribution of the SDN/OpenFlow control plane on a two-level hierarchical architecture (contrary to the centralized approach with a single SDN controller), comprising a first level with a Global FMC Controller (G-FMCC), and a second level with several Local FMC Controllers (L-FMCCs)

With their control plane framework and Network Function Virtualization (NFV) concept, the L-FMCCs were deployed on-demand, where and when needed, depending on the global system load, and the results obtained via analysis showed that their solution ensured more efficient management of control plane, performance maintaining, and network resources preservation.

2.2.6 Network Store: Exploring Slicing in Future 5G Networks

This paper [19] provides a revolutionary vision of 5G networks, in which SDN technologies are used for the programmability of the wireless network, and where a NFV-ready network store is provided to Mobile Network Operators (MNOs), Enterprises, and Over-The-Top (OTT) third parties.

The authors propose a network that serves as a digital distribution platform of programmable Virtualized Network Functions (VNFs) that enables 5G application use-cases, similar to what happens with currently existing application stores, such as Apple's App Store for iOS and Google's Play Store for Android, where applications are delivered to user specific software platforms.

Their vision is to provide a digital marketplace (the 5G Network Store), gathering 5G enabling Network Applications and Network Functions, written to run on top of commodity cloud infrastructures, connected to Remote Radio Heads (RRH), with the store being the same to the network provider as the application store is currently to a software platform.

2.2.7 Leveraging SDN to provide an in-network QoE measurement framework

As online video streaming using HTTP Adaptive Streaming (HAS) is becoming the most popular content delivery mechanism for media services, network and content providers would like to ensure a high degree of video Quality of Experience (QoE) for their end-users. However, traditional network-level metrics do not necessarily reflect the end-users' true perception of delivered content.

This paper [20] introduces an In-network QoE Measurement Framework (IQMF) that provides QoE monitoring for HAS streams as a service to leverage Software-Defined Networking SDN for its control plane functionality to streamline non-intrusive quality monitoring and to offer a closed control loop for QoE-aware service management.

IQMF adopts two specifically designed QoE metrics to capture the user experience of HAS streams with respect to video fidelity and switching impact. The authors used a pan-European

SDN testbed to demonstrate how IQMF can be used as a foundation for in-network QoE measurement and service optimisation.

2.3 Commercial Products

2.3.1 Cisco Systems: Tetration

Cisco Tetration [21] is a product categorized as a Network Performance Management solution that uses behavior-based application insight and machine learning algorithms, allowing customers to build dynamic segmentation policy models and automate policy enforcement.

It collects rich network telemetry data from hardware and software sensors, providing advanced analytics and visibility with an infrastructure agnostic approach supporting both on-premise and public cloud workloads.

The platform is designed to support any data center infrastructure at scale and to analyze millions of events per second, providing actionable insight near realtime, and is capable of retaining billions of records long term without loss of granularity.

Deployment is achieved by installing software sensors on virtual machines or bare-metal servers (SDN/NFV integration), and customers with existing data center infrastructures can do it so either with Cisco proprietary or third party solutions.

2.3.2 ZTE: Network Performance Management

ZTE's Network Performance Management [22] is a commercial solution devised to ensure a stable network operation, with improved performance indicators, reduced operation and maintenance costs. It supports both wired and wireless networks, providing good customer experience through monitoring performance indicators, troubleshooting faults, improving network performance, managing capacity, and processing customer complaints.

Adhering to the concept of customized service, the product provides complete professional service solutions to improve customers' operation efficiency with reduced costs. Network performance management includes wireless network optimization, fixed network access optimization, bearer network optimization, value-added network optimization, network security solution, and significant event guarantee.

2.3.3 MYCOM OSI: PrOptima

MYCOM OSI's PrOptima [23] is another commercial product, a network performance management solution with converged end-to-end (E2E) support for mobile, IP, virtualized and fixed networks, offering:

- Out-of-the-box support for multiple domains, technologies, network equipment vendors and managed service suppliers;
- Ability to scale-up and process very large volumes of performance data in near realtime using next generation object storage for peta-byte scale;
- Flexible configuration to allow advanced reporting and analysis capabilities;
- Supports performance of hybrid NFV networks and IoT data traffic;

- Agile methodologies and latest technologies for cloud-based deployments;
- Northbound HTTP REST APIs to create an agile, open and collaborative ecosystem environment essential for modern information security operations;
- Enhanced web-based geographic information system to visualize the most critical KPIs in customizable views.

2.4 Discussion

In this chapter an overview was provided over some commercial products, research projects and research papers focusing on the upcoming 5G networks, Self-Organized Networks (SON), Software-Defined Networking (SDN) and Network Function Virtualization (NFV) topics.

The research work, either ongoing or previously done on the past recent years, reveals that the SON, SDN and NFV paradigms will play crucial roles, and are regarded as the foundations of all the innovation that will transform the mobile industry.

While we have seen innovation over the past decades on the devices we use to connect or access the network, applications, services, storage solutions and so forth, the underlying networks that connect all of these things has remained virtually unchanged.

The SDN and NFV paradigms allow the separation of the control and data planes, providing more efficient orchestration and automation of the network services, optimizing the network services themselves by decoupling the network functions from the hardware, so they can run in software to accelerate the service provisioning realized by the service providers.

This decoupling of the software from the hardware is the reason why SDN and NFV are considered so important and vital to build the future networks, as they enable the innovation to create new services to generate revenue and reduce the CapEx/OpEx of the organizations.

The presented research projects and papers (among others) are a natural consequence and proof of that, with both companies and scientific communities embracing the research of these paradigms as they deem it necessary for the improvement of their applications and to tackle the problems they do not yet resolve.

Some commercial products already taking advantage of these technologies were also presented, which shows us that companies are starting to bear fruits from the research work done on these fields so far. Moreover, we can also look at it as a foresight on what is about to come, where there will be a proliferation of products exploiting the results of the research work done in SDN and NFV.

Although SDN/NFV will play an important role, these technologies alone cannot solve all the problems, and they need to be further studied, improved and complemented/combined with other technologies, as revealed by the presented research papers and projects.

Autonomics is also regarded important, as there will be an increased complexity of the underlying networks with the introduction of these technologies. The use of autonomic capabilities together with SDN/NFV is what thrives the self-organizing features of the future networks, allowing further optimization of the resources and services through automated network management.

Another key concept that has been generating some hype lately is the Network Slicing, which is a specific form of virtualization that envisions multiple logical networks running on top of a shared physical network infrastructure. It is expected from this technology to open lucrative new business opportunities for mobile operators, as they will be able to split its physical network resources into multiple logical slices and lease these slices out to interested parties.

Altice Labs, as a leading researching center on telecommunication technologies, is not unaware of the advancements on this area and the advantages they will bring. Its Altaia performance management platform, although advanced and feature rich, needs to evolve like any other solution that aspires to survive the demands of a constantly changing market and technology advancements, thus the involvement on the SELFNET project.

The work presented on this dissertation, and the research project it is part of, is aligned with this vision, aiming to gather the required knowledge on the SDN/NFV paradigms which will provide an insight into possible solutions to tackle the problem of the static nature of non virtualized and non programmable resources on which the Altaia performance management platform is currently sitting on.

CHAPTER 3

Technologies

This chapter provides an overview on some technologies that were considered to be used on the project and they are divided by key areas, each area presenting technologies that relate to each other followed by a brief discussion.

Service coordination technologies will be discussed in section 3.1, message bus technologies will be discussed in section 3.2, monitoring tools will be discussed in section 3.3, aggregation technologies will be discussed in section 3.4 and finally storage (database) technologies will be discussed in section 3.5.

3.1 Service Coordination

3.1.1 Apache ZooKeeper

ZooKeeper is an open-source and out-of-the-box distributed, scalable, and high-performance coordination service for distributed applications under the umbrella of the Apache Software Foundation. It is essentially a distributed hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for distributed systems.

It allows distributed processes to coordinate with each other through a shared hierarchical namespace which is organized similarly to a standard file system, in which the name space consists of data registers (znodes) that are similar to files and directories, as seen in figure 3.1.

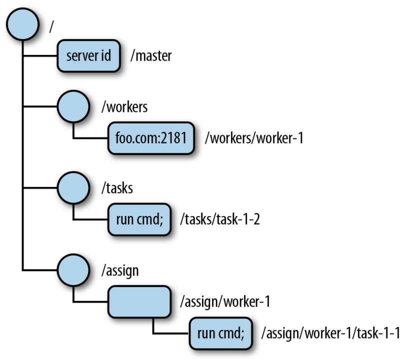


Figure 3.1: Zookeeper Data Tree [24]

While being a coordination service for distributed systems, it is a distributed application on its own and follows a simple client-server model where clients are nodes that make use of the service, and servers are nodes that provide the service. Together, a collection of ZooKeeper servers running in *quorum* mode [25] forms an *ensemble* [25] (i.e., a cluster) where the data tree is replicated across all servers.

The clients can interact with any server, which is assigned to them by the leader of the *ensemble* (shown in figure3.2), providing high reliability and availability. If one of the nodes fail, a new one will be assigned by the leader, and a new leader will elected if the current one fails.

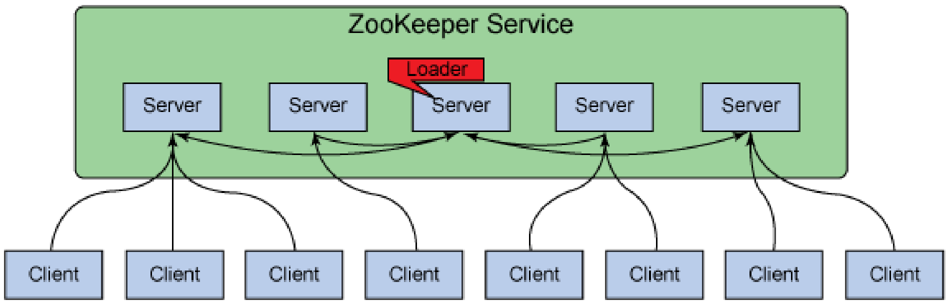


Figure 3.2: Zookeeper Ensemble [26]

All in all, Zookeeper provides an abstraction layer to be used by distributed services, allowing the developers to focus on the development of the applications without having to worry about the coordination service itself.

3.1.2 HashiCorp Consul

Consul [27] is a tool for discovering and configuring services in an infrastructure. It is similar to ZooKeeper as it also follows a server-client model with a cluster of servers and it provides the following key features:

- **Service Discovery**: some clients of Consul can provide a service (e.g., an API or a database) while other clients can use Consul to discover those provided services. The applications can easily find the services they depend upon using either DNS or HTTP;

- **Health Checking**: the clients can also provide health check services, either associated with a given service (e.g., a service response code) or with the local node (e.g. memory usage). This information can be used to route traffic away from unhealthy hosts or by an operator to monitor cluster health;

- **KV Store**: the applications can make use of Consul's hierarchical key/value store for any number of purposes. For instance, dynamic configuration, feature flagging, coordination, leader election, and so on;

- **Multi Datacenter**: it supports multiple datacenters out-of-the box, creating an abstraction layer that supports coordination between multiple regions

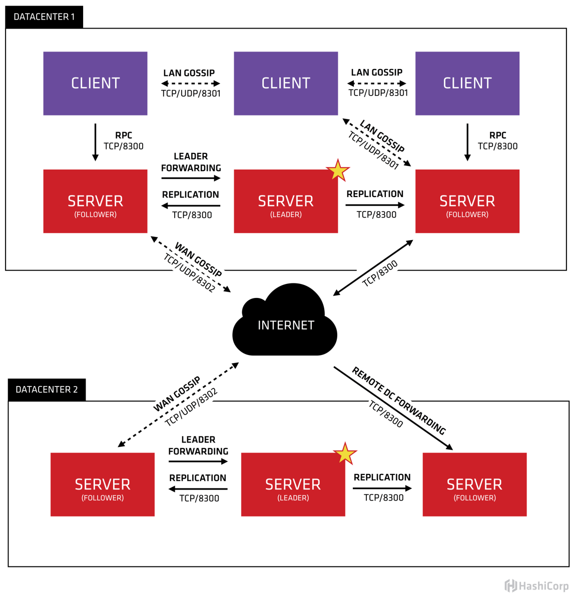


Figure 3.3: Consul High Level Architecture [28]

Figure 3.3 represents a typical Consul deployment architecture, representing two data centers with several Consul servers maintaining the data sent by the clients replicated between them, even across two data centers separated by an Internet/WAN connection. Like in the Zookeeper case, there is a leader that must be elected and likewise a discovery service.

3.1.3 Discussion

The Aggregation Framework of SELFNET, which will be discussed in section 4.2, being a very complex structure will need a service coordination tool. For this purpose two solutions where analyzed, the Apache Zookeeper and the HashiCorp Consul. While Consul is designed to work with higher level protocols, such as HTTP and DNS, Zookeeper on the other hand is designed to work on a lower level.

Since the Aggregation Framework will work with low level interactions with several components, it was an easy (but rational) choice to pick Zookeeper as the service coordination tool. Furthermore, many other chosen technologies that will be discussed ahead also use this service coordination tool, reinforcing its choice.

3.2 Message Bus

3.2.1 Apache Kafka

Kafka [29] is a distributed messaging system providing fast, highly scalable and redundant messaging through a pub-sub model. Its distributed design allows a large number of permanent or ad-hoc consumers, providing high availability and resiliency to node failures and supports automatic recovery. These characteristics make Kafka an ideal fit for communication and integration between components of large scale data systems in real world.

All Kafka messages are organized into topics, where consumers pull messages (records) off of a Kafka topic, while producers push messages into a Kafka topic. Also, being a distributed system, it runs in a cluster and each node of the cluster is called a *broker*.

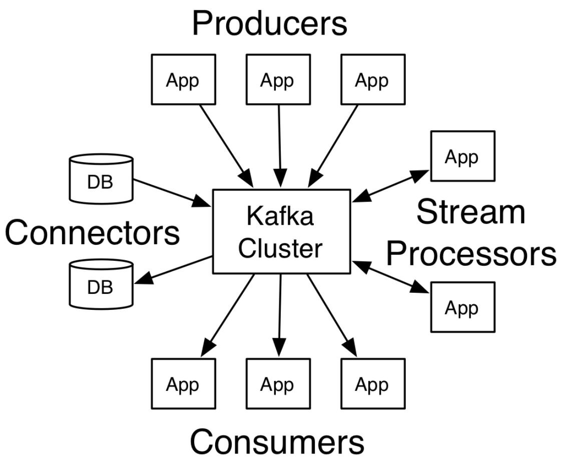


Figure 3.4: Kafka High Level Architecture [29]

It has four core APIs as illustrated in figure 3.4:

- **Producer API**: allows an application to publish a stream of records to one or more topics;
- **Consumer API**: allows an application to subscribe to one or more topics and process the stream of records produced to them;
- **Streams API**: allows an application to act as a stream processor, effectively transforming the input streams to output streams, consuming an input stream from one or more topics and producing an output stream to one or more output topics;
- **Connector API**: allows building and running reusable producers or consumers that connect topics to existing applications or data systems (e.g., a connector to a relational database might capture every change to a table).

3.2.2 RabbitMQ

RabbitMQ [30] is a widely deployed and well known general purpose message broker with pub-sub communication styles patterns that support several messaging protocols, directly and through the use of plugins, originally developed to support the Advanced Message Queuing Protocol (AMQP) [31]. It supports the following key features:

- **Reliability**: offers a variety of features to let developers trade off performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability;
- **Flexible Routing**: features several built-in exchange types for typical routing logic and messages are routed through exchanges before arriving at queues;

- **Clustering**: its servers can be clustered together, forming a single logical broker;
- **Highly Available Queues**: queues can be mirrored across several machines in a cluster, ensuring the safety of the messages, even in the event of hardware failure;
- **Tracing**: offers tracing support to help finding out what is going on if the messaging system is misbehaving;
- **Federation**: offers a federation model for servers that need to be more loosely and unreliably connected than in a cluster;
- **Many Clients**: supports a large number of development platforms as there are RabbitMQ clients for almost any language.

It uses a smart broker / dumb consumer model (the opposite to Kafka that employs a dumb broker and uses smart consumers), focused on consistent delivery of messages to consumers that consume at a roughly similar pace as the broker keeps track of consumer state.

3.2.3 Discussion

The Aggregation Framework will have a large amount of data flowing inwards and towards other framework of SELFNET and, with that in mind, it is essential that the chosen message bus will be able to handle such a high volume of data while maintaining a healthy performance.

The analyzed and considered technologies were the Apache Kafka and RabbitMQ. Between these two, Kafka is the most performant one when it comes to message throughput (shown in figure 3.5), with RabbitMQ being the one that offers more features when it comes to easiness of development and management as it is a more traditional message broker.

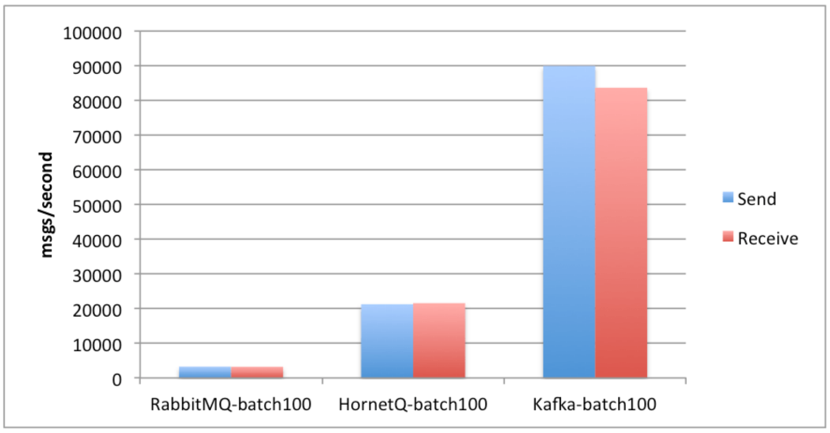


Figure 3.5: Kafka vs RabbitMQ - Message Throughput [32]

Since performance is the key to this choice, Kafka was the chosen technology. Moreover, Zookeeper is a requirement of Kafka which reinforces the choice of both these technologies.

3.3 Monitoring Tools

3.3.1 OpenStack Monasca

Monasca is an OpenStack project that provides an open-source multi-tenant, highly scalable, performant, fault-tolerant Monitoring-as-a-Service (at scale) solution, hence the name Monasca (MONaSca).

The metrics can be published to the Monasca API, stored and queried. Alarms can be created and notifications can be sent when alarms transition state.

It builds an extensible platform for advanced monitoring services that can be used by both operators and tenants to gain operational insight and visibility, ensuring availability and stability.

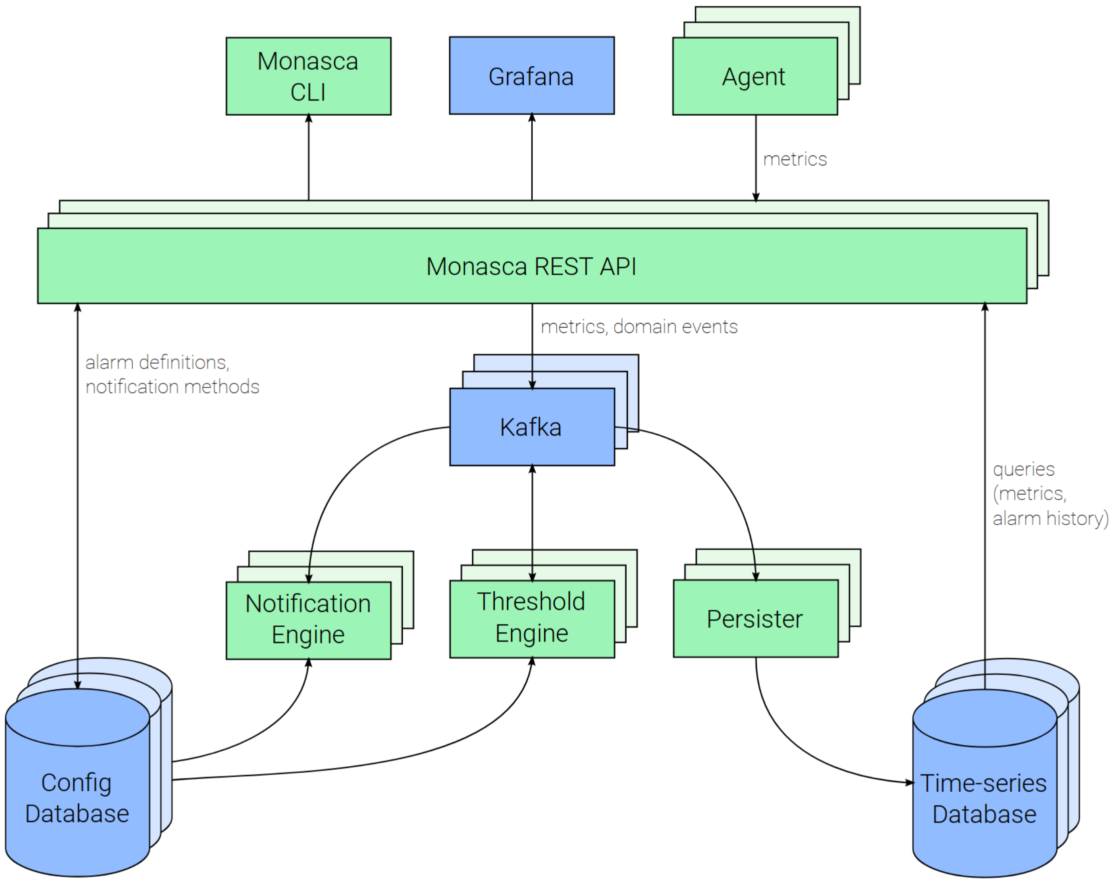


Figure 3.6: Monasca's High Level Architecture [33]

The architecture is show in figure 3.6 where only the core components are represented, leaving out some components that not only are not relevant to this discussion, but also were not used in the SELFNET project. It supports several components of-the-shelf:

- **Apache Kafka**: used as the message bus and central to all internal communications between the remaining components;

- **MySQL**: used to store all kinds of configurations of the service, namely the alarm definitions and the notification methods (Config Database);
- **InfluxDB**: where all the metrics and triggered alarms are stored;
- **Apache Storm**: used for the Threshold Engine, where the latter is a somewhat complex topology [34];
- **Apache Zookeeper**: required as a dependency of Kafka and Storm;
- **Grafana**: used to create plots/graphics for the metrics and alarms;
- **Keystone**: not represented in the architecture but is used as the identity service for authentication, authorization and multi-tenancy.

The remaining components serve the following purposes:

- **Persister**: as the name implies, it is the component responsible for retrieving the metrics and alarms from the message bus and storing them on the time-series database (InfluxDB);
- **Notification Engine**: responsible for creating notifications for the triggered alarms;
- **Monasca CLI**: a shell command line interface used to interact with the service;
- **Agent**: usually deployed in virtual machines (VM) to gather metrics on them.

All interactions with Monasca are done through its REST API, which is the external component responsible for handling all sorts of operations, like creating or querying metrics, alarms, alarm-definitions, notifications and so forth. The Monasca service will be further discussed in section 6.1.

3.3.2 OpenStack Ceilometer

Ceilometer [35] is a data collection service that collects events and metering data by monitoring notifications sent from OpenStack services. It publishes collected data to various targets including data stores and message queues.

It is a component of the OpenStack Telemetry [36] project and its data can be used to provide customer billing, resource tracking, and alarming capabilities across all OpenStack core components.

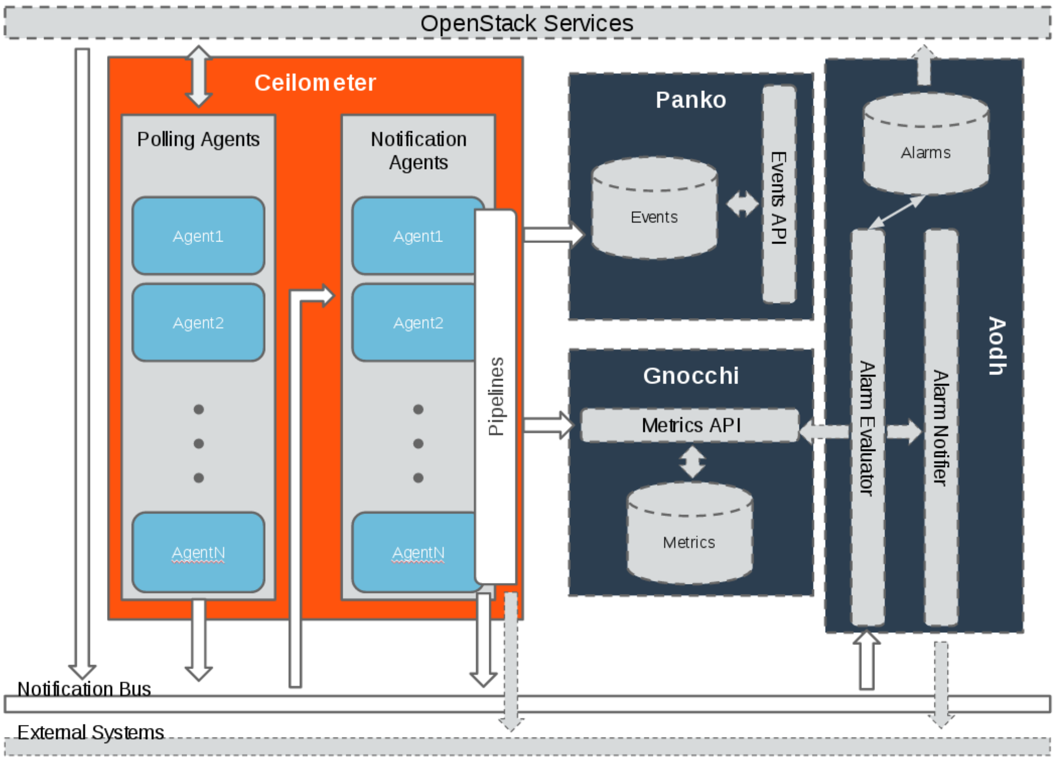


Figure 3.7: Ceilometer's High Level Architecture [37]

The overall summary of Ceilometer's logical architecture, shown in figure 3.7, is as follows:

- **Agents**:
  - **Notification Agent**: takes messages generated on the notification bus and transforms them into Ceilometer samples or events (the preferred method);
  - **Polling Agent**: polls some API or other tool to collect information at a regular interval (less preferred method due to the load it can impose on the API services).
- **Gnocchi**: developed to capture measurement data in a time series format to optimize storage and querying;
- **Panko**: the event storage project designed to capture document-oriented data such as logs and system event actions;
- **Aodh**: the alarming service which can send alerts when user defined rules are broken.

Accessing the data can be done through REST APIs and the data gathered from the polling and notifications agents contains a wealth of data that can be combined with historical or temporal context in order to derive even more data. Ceilometer offers various transformers which can be used to manipulate data in the pipeline.

3.3.3 Discussion

In this section two monitoring solutions were presented, the OpenStack Ceilometer and Monasca. While one could be lead to conclude that just only one of them would be chosen, that is not the case. Both of them where chosen and the reasons behind it are as follows.

The SELFNET consortium decided that the data to be collected from the physical components would be performed either by the LibreNMS monitoring tool (discussed in section 5.2.4) and by a sensor called Flow Monitoring Agent (discussed in section 5.2.1), developed by one of the partners of SELFNET.

Since the infrastructure virtualization solution used in the project is the OpenStack, Ceilometer is a natural choice as a monitoring tool to collect data in such a virtualized environment. After all Ceilometer is part of the OpenStack's ecosystem and it is tightly integrated on it.

The remaining question was then how to monitor the data produced by the Aggregation Framework, and this is where Monasca comes in. Monasca is a mature solution, also under the umbrella of OpenStack, that featured three particular components on its architecture that covered the requirements: its time-series database (perfect for the storage the aggregated metrics), its threshold engine (covering the alarming service) and its notification engine (covering the alarm notification requirements).

3.4 Aggregation Tools

3.4.1 Apache Storm

Apache Storm [38][39] is a free and open source distributed realtime computation system designed to process large amounts of data streams, doing in realtime what Apache Hadoop [40] does for batch processing.

- **Fast**: it can process up to 1 million tuples/records per second per node;
- **Scalable**: being a distributed platform, it is possible to add more computation nodes (single machines executing Storm applications) in order to increase the processing capacity of the applications;
- **Fault Tolerant**: worker processes are automatically restarted by Storm whenever they die, and if running in a Storm cluster they can even be restarted in another node;
- **Guarantees Data Processing**: messages entering a Storm process are guaranteed to be processed at least once, with the ability to replay lost tuples/records in the event of failures;
- **Easy to Operate**: it is simple to deploy and requires little maintenance once deployed;
- **Programming Language Agnostic**: the applications (topologies) can be written in any programming language, so long they are able to read/write from standard input/output streams, even though Storm runs on Java Virtual Machine (JVM).

A Storm cluster follows a master-slave model where Zookeeper is used to coordinate the master and slave processes. The Nimbus node is the master, responsible for distributing the application code across various Supervisor nodes (the worker nodes) and there is only a single active Nimbus node in a cluster (passive fallback Nimbus nodes can be added to the cluster), while the Supervisor nodes are responsible for creating, starting and stopping the multiple worker processes assigned to them (figure 3.8).

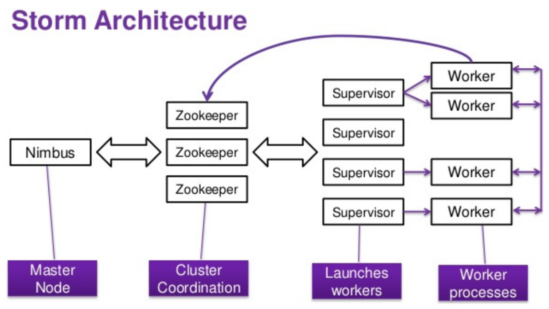


Figure 3.8: Apache Storm Architecture [41]

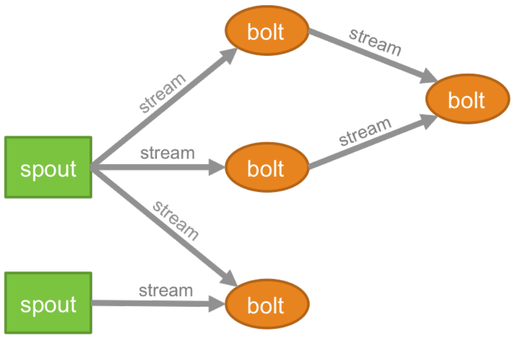


Figure 3.9: Apache Storm Topology [42]

A Storm topology (figure 3.9) is an abstraction that defines a graph of computation (a directed acyclic graph) where the nodes (spouts and bolts) are connected together by streams:

- **Tuple**: a collection of key value pairs (a named list of values) where each value can be a primitive or custom type;
- **Stream**: an unbounded sequence of tuples that can be processed in parallel;
- **Spout**: the input source of tuples (entry points in a topology), responsible for reading data from an external source (message queues, databases, file systems, etc.), converting the data into streams of tuples and emitting them to bolts;
- **Bolt**: contains the actual processing logic, processing the tuples of the input streams (coming from spouts or other bolts) and producing the output streams by filtering, aggregating, joining or other methods of transformation, and can also emit streams for further processing downstream by other bolts or export data for persistent storage.

3.4.2 Trifacta Wrangler

Trifacta Wrangler [43] is a proprietary data wrangling and analysis solution, designed to transform and aggregate data with the aid of machine learning algorithms and automatic data profiling.

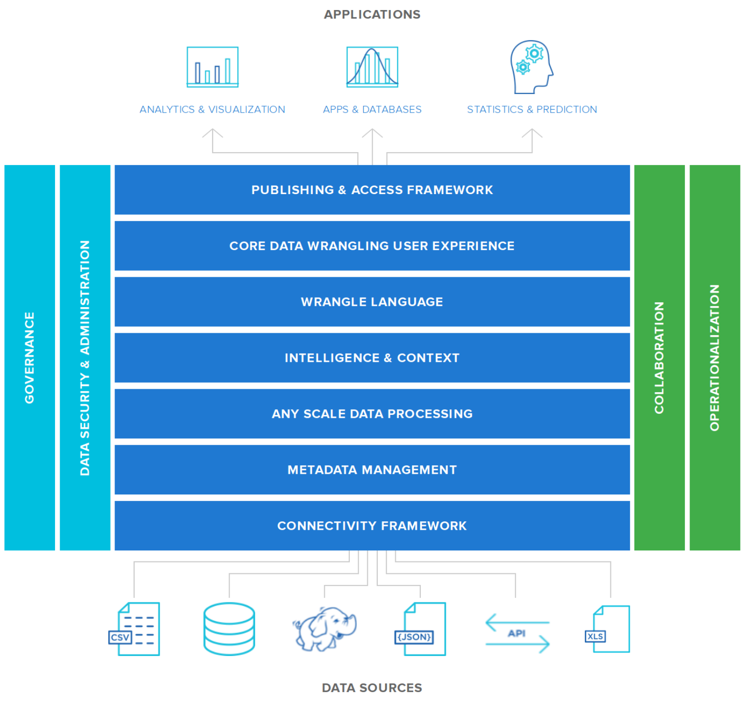


Figure 3.10: Trifacta Wrangler Architecture [44]

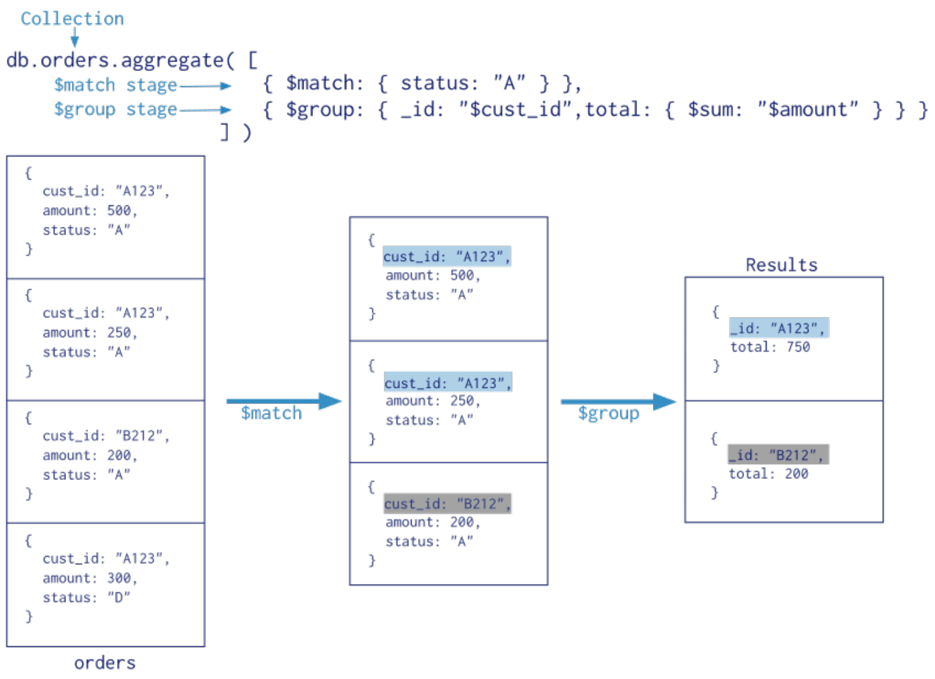
Its general architecture is shown in figure 3.10 and some of its features are described below:

- **Connectivity Framework**: an API framework that includes connections to various sources such as Apache Hadoop, Files in several formats (CSV, Text, JSON, XML, etc), relational databases, Cloud services and so on, supporting governance and security features;
- **Metadata Management**: support for enriching data with geographic, demographic, census and other common types of reference data;
- **Any Scale Data Processing**: on-the-fly data transformation in the application with its Intelligent Execution Engine or with external services such as Apache Spark, Google DataFlow or Photon, its in-memory engine;
- **Intelligence & Context**: data registered into the platform is analyzed with machine learning algorithms, making suggestions on how data can be wrangled;
- **Wrangle Language**: provides an abstraction of the data wrangling logic created in the application from the underlying data processing;
- **Publishing & Access Framework**: the wrangled data can be published to a variety of analytical tools, databases, file systems and compression formats;
- **Operationalization**: scheduling and monitoring of workflows with the possibility of setting data recipes into repeatable pipelines in periodic schedules.

3.4.3 MongoDB Aggregation Framework

MongoDB features an Aggregation Framework [45] that allows data processing through a set of supported operations (sum, average, min, max, and so on) [46], grouping values from multiple documents together and producing single results.

The concept of the framework is based on data processing pipelines, where the aggregations are results of the documents that are transformed through stages [47] of those pipelines by means of supported operations.



```
db.orders.aggregate( [
      { $match: { status: "A" } },
      { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
   ] )
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{
  _id: "A123",
  total: 750
}
{
  _id: "B212",
  total: 200
}
```

Figure 3.11: MongoDB Aggregation Example [45]

A simple example is shown in figure 3.11 where a match is applied to a collection of documents, filtering them by their IDs and aggregating them in a single result, producing a sum of values.

3.4.4 Discussion

SELFNET requires that the metrics collected from its sensors can be aggregated in order to leverage their information. Furthermore these aggregations must be done in short periods of time, and in some cases in realtime.

The MongoDB Aggregation Framework would require that the raw metrics would be stored in MongoDB which is not the case (see section 3.5), imposing an extra level on the process of aggregation of metrics that would cripple the performance. Moreover, this framework has severe limitations when it comes to memory usage and the size of produced aggregated results [48], which makes it unfit for the purpose.

Trifacta Wrangler, although a rich solution with considerable and interesting set of features, it is in first place proprietary (consequently close-sourced) and it seems to be most oriented for batch processing and manual interaction by data Analysts, falling off the scope of dynamicity required by the SELFNET project.

That leaves us with Apache Storm, which is open-source and whose features match the requirements of the project when it comes to realtime aggregation of metrics. Moreover, it is required for the Monasca Threshold Engine, one of the chosen technologies, re-enforcing this choice as a whole.

Another consideration that must be done pertains the batch aggregation of metrics. Altice Labs' Altaia technology was used for this purpose (the Aggregation Engine that will be mentioned in section 4.2), and given that the internal workings of this component are out of the scope of this work, the technologies it uses will not be discussed on this chapter.

3.5 Storage

3.5.1 InfluxDB

InfluxDB [49] is a high-performance data storage written specifically for time series data, designed for any use case that involves large amounts of timestamped data, like application metrics, DevOps monitoring, IoT sensor data, and real-time analytics.

It allows for high throughput ingest, compression and real-time querying, and good at conserving space when configured to keep data for a defined length of time, automatically expiring and deleting any unwanted data from the system.

These are some of the features that it currently supports:

- Custom high performance datastore written specifically for time series data. The Time-Structured Merge Tree (TSM) engine allows for high ingest speed and data compression;
- Written entirely in Go [50]. It compiles into a single binary with no external dependencies;
- Simple, high performing write and query HTTP/S APIs;
- Expressive SQL-like query language tailored to easily query aggregated data;
- Tags allow series to be indexed for fast and efficient queries;
- Retention policies efficiently auto-expire stale data;
- Continuous queries automatically compute aggregate data to make frequent queries more efficient.

The query language for interacting with the data is InfluxQL and its SQL-like, specifically created to feel familiar to those coming from other SQL environments, while also providing features specific to storing and analyzing time series data.

Dealing with real time data can lead to huge amounts of data over a long period of time, and that can create storage concerns. To deal with this particular problem, InfluxDB supports data downsampling to keep the high precision raw data for only a limited time, and storing the lower precision, summarized data for much longer or forever.

3.5.2 MongoDB

MongoDB [51] is NoSQL, open-source, document oriented database that provides high performance, high availability, and automatic scaling. The records on MongoDB are documents, which are a data structure composed of key/value pairs.

The documents are stored on collections which are roughly the equivalent to tables on the traditional relational databases, but unlike the latter, MongoDB does not enforce a schema, allowing a collection to have documents with different sets of key/value pairs that may hold different types do data (polymorphism).

Some of the most known features of MongoDB are:

- **High Performance**: support for embedded data models reducing I/O activity on the database system, and indexes to support very fast queries that can include keys from embedded documents and arrays;
- **Rich Query Language**: a rich query language that supports CRUD operations, data aggregation, text search and geospatial queries;
- **High Availability**: it has a replication facility called replica set, which is a group of MongoDB servers that maintain the same data set, that provides automatic failover and data redundancy to increase the data availability;
- **Horizontal Scalability**: sharding, which allows distribution of data across a cluster of machines.

One note that should be made about MongoDB is that it only supports Atomicity, Consistency, Isolation and Durability (ACID) transactions at the document level, consequently it does not support multi-document transactions.

3.5.3 Neo4j

Neo4j [52] is an open-source graph database implemented using Java. A graph database is a database that, as the name suggests, models the data in the form of a graph. The nodes of the graph represent entities while the edges represent relationships between the nodes.

Neo4j stores relationships and connections as first-class entities, unlike the relational databases that store highly structured data with several records storing the same type of data without storing relationships between them.

Some advantages and features of Neo4j are:

- **Cypher Query Language**: Neo4j provides a declarative query language using an ascii-art syntax that represents the graph visually. The commands of this language are very easy to learn;
- **No joins**: it does not require complex joins to retrieve connected/related data as it is very easy to retrieve the adjacent nodes or relationships;
- **ACID properties**: supports full ACID rules which is an advantage to other NoSQL databases, namely the aforementioned MongoDB;
- **Built-in Web Application**: it provides a built-in web application, the Neo4j Browser, that can be used to create and query the graph data easily;

- **Drivers**: it features several drivers to support many programming languages (Java, Python, Scala, etc.), and it even supports interaction with other databases such as MongoDB, Cassandra and so on.
- **Flexibale schema**: Neo4j is essentially schema-less, there is no need to follow a fixed schema and one can add or remove properties as per requirement, but it also provides schema constraints;
- **High availability**: supports high availability for large enterprise applications with transactional guarantees;
- **Easy retrieval**: with Neo4j, it is not only possible to represent, but also easily retrieve (traverse/navigate) connected data faster when compared to other databases.

3.5.4 Apache Cassandra

Apache Cassandra is a distributed database for managing large amounts of structured data across many commodity servers, and is a NoSQL Column family implementation supporting the Big Table data model using the architectural aspects introduced by Amazon Dynamo. Some of the strong points of Cassandra are:

- Highly scalable and highly available with no single point of failure;
- NoSQL column family implementation;
- Very high write throughput and good read throughput;
- SQL-like query language (since 0.8) and support search through secondary indexes;
- Tunable consistency and support for replication;
- Flexible schema.

And below are some of the features of Cassandra:

- **Elastic scalability**: it allows to add more hardware to accommodate more customers and more data as per requirement;
- **Always on architecture**: has no single point of failure and it is continuously available for business-critical applications that cannot afford a failure;
- **Fast linear-scale performance**: it is linearly scalable, i.e., it increases the throughput as it is increased the number of nodes in the cluster. Therefore it maintains a quick response time;
- **Flexible data storage**: accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to the data structures according to the need;
- **Easy data distribution**: it provides the flexibility to distribute data where it is needed by replicating data across multiple data centers;

- **Transaction support**: supports properties like Atomicity, Consistency, Isolation, and Durability (ACID);
- **Fast writes**: designed to run on cheap commodity hardware. It performs very fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

3.5.5 Discussion

When it comes to storage technologies, the ability to store the raw and aggregated metrics of the Aggregation Framework in the most effective way (given the nature of this type of data) is the most important key on this topic.

Metrics are naturally related to and dependent on the time at they were collected and aggregated. Equally natural is the choice of a time-series database as a solution to store them, as traditional relational databases are not really fit to this purpose.

Two databases (InfluxDB and Gnocchi, the latter not discussed here) were initially considered as possible solutions, but in the end InfluxDB was chosen to store aggregated metrics because it provided clear and complete documentation, and it was part of the Monasca service, which also presented added benefits. As to the raw metrics, the SELFNET consortium had already chosen Apache Cassandra as the persistence solution for this type of data and InfluxDB was not used here.

But besides the storage of the aggregated metrics, there were other types of data that need to be considered. There were some services and/or components of the Aggregation Framework that had data that needed to be persisted, namely the Aggregation Configuration Manager (ACM) and the Monitoring Catalog (MC) which will be discussed in section 4.2, whose data storage has other requirements.

MongoDB was chosen has the database to be used by the ACM because it only needed to maintain simple key/value pairs of data related to the aggregation rules applied to the Aggregation Framework.

However, MongoDB only supports atomic operations at the document level, making it unfit to be used by the Monitoring Catalog. This is so because it would require that the catalog would be stored in a single document in order to guarantee its consistency, and it would not make much sense to retrieve, modify and save the full catalog every time a simple operation or modification needed to be done.

On the other hand, Neo4j would allow a full representation of the catalog as a graph, with all the advantages of the ACID properties of this database, supporting transactions which ensured the consistency of the data in more complex operations. For this reason, Neo4j was chosen to be used as the Monitoring Catalog's database.

3.6 Summary

This chapter presented several key technologies that were considered to be used on the project, each key area briefly discussing some advantages and disadvantages and stating which technologies were chosen in the end.

CHAPTER 4

Architecture

This chapter will cover SELFNET's logical architecture (sections 4.1 and 4.2) taking a top-down approach, starting with a high level overview of the system's architecture, and then breaking it down into more detailed views in order to gain insight into its compositional sub-systems, down to the point where individual components of the system are shown, focusing on those that are the subject of this thesis.

With this in mind the presented views of the logical architecture will be broken down into three logical level as follows:

- **Level 0** [**L0**]: the top most conceptual view of the SELFNET's architecture;
- **Level 1** [**L1**]: the intermediate conceptual view of the logical architecture, focused on the Monitoring & Analyzer sub-layer of SELFNET;
- **Level 2** [**L3**]: the most detailed view of the logical architecture breaking it down to individual components, showing an implementable perspective.

This breakdown of the SELFNET's architecture is essential and provides context to the reader. The first and higher level of the architecture will present a general perspective of SELFNET's architecture showing its major layers and components, their purposes and how they relate to each other.

The intermediate level will make a transition, going a step further and presenting a high level view of one of the layers, the Monitoring & Analyzer sub-layer, where the components that are within the scope of this thesis are located.

Afterwards, another step further will be taken and the most detailed high level view of the Monitoring & Analyzer sub-layer will be given, starting by breaking down the inner frameworks of the sub-layer and up to the point where the individual components within them will be shown.

At last but not the least, since the SELFNET project is use-case driven, the impact of its uses-cases on the architecture will be discussed, with greater focus on the Monitoring & Analyzer sub-layer and on one of the three existing use-cases (the most mature one), the Self-Protection Use Case (SP-UC).

4.1 SELFNET Architecture

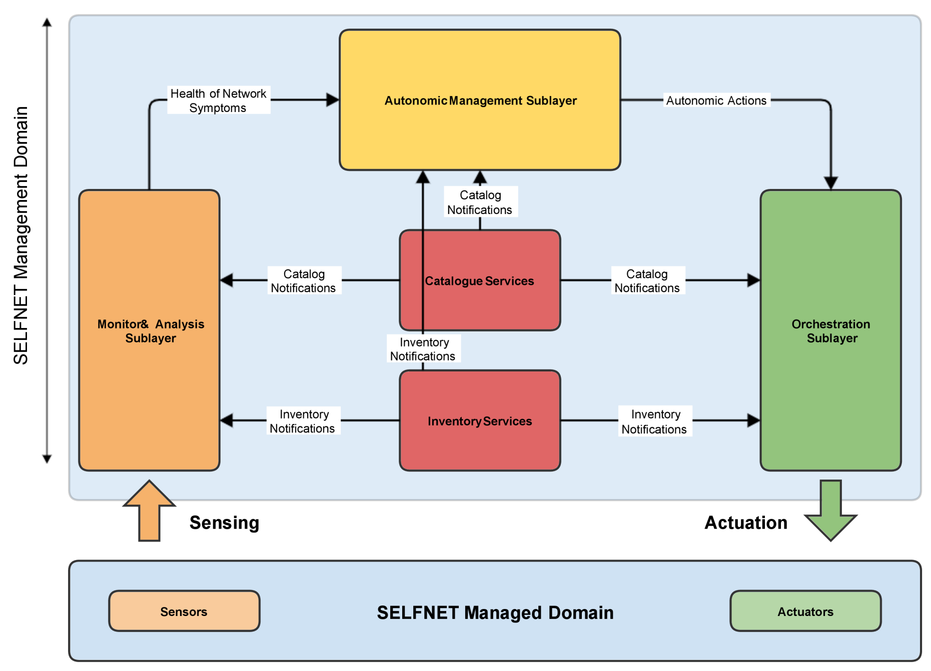


Figure 4.1: SELFNET Logical Architecture - Level 0

The first high level logical architecture (L0) of SELFNET is represented in figure 4.1, in which there are three major sub-layers: the Monitoring & Analysis, the Autonomic Management and the Orchestration. There are also the Catalog and Inventory services, both traversal to all the three sub-layers. The purposes of these layers and services can be summarized the following way:

- **Monitoring & Analysis**: a 5G network is composed of SDNs, NFVs, and many legacy network components that will provide considerable amounts of data through sensors deployed on that same network, and the purpose/responsibility of this layer is to gather that data, aggregate and analyze it, in order to provide Health of Network (HoN) symptoms with the aid of prediction algorithms that could point out potential problems;
- **Autonomic Management**: the purpose of this layer is to take the HoN symptoms provided by the Monitoring & Analysis, perform diagnosis on them in order to determine the root causes, and take proactive measures to prevent the underlying end-to-end network services from being disrupted;
- **Orchestration**: as the name implies, this layer is responsible for orchestrating the underlying network services (SDN & NFV Orchestration), enforcing the actions previously determined by the Autonomic layer in order to coordinate the virtualized network;
- **Catalog and Inventory**: these are central services responsible for keeping all the information related to the on-boarding and instantiation of resources that are common to the sub-layers:

– **Catalog Service**: it is a repository that holds all the information related to the internal policies available on the sub-layers, like which type of sensors are available, how can they be configured and deployed, what type of data they provide or, for instance, which rules of aggregation of metrics are in place and which counters of which sensors do they use to produce those aggregated metrics;
– **Inventory Service**: it is a repository that holds all the information of the instantiated resources and services, like already deployed sensors, NFVs, SDNs and so on.

Now that we have already examined the first high level (L0) logical architecture of SELFNET, the next step is to examine the high level (L1) logical architecture of the Monitor & Analyzer sub-layer, illustrated in figure 4.2:

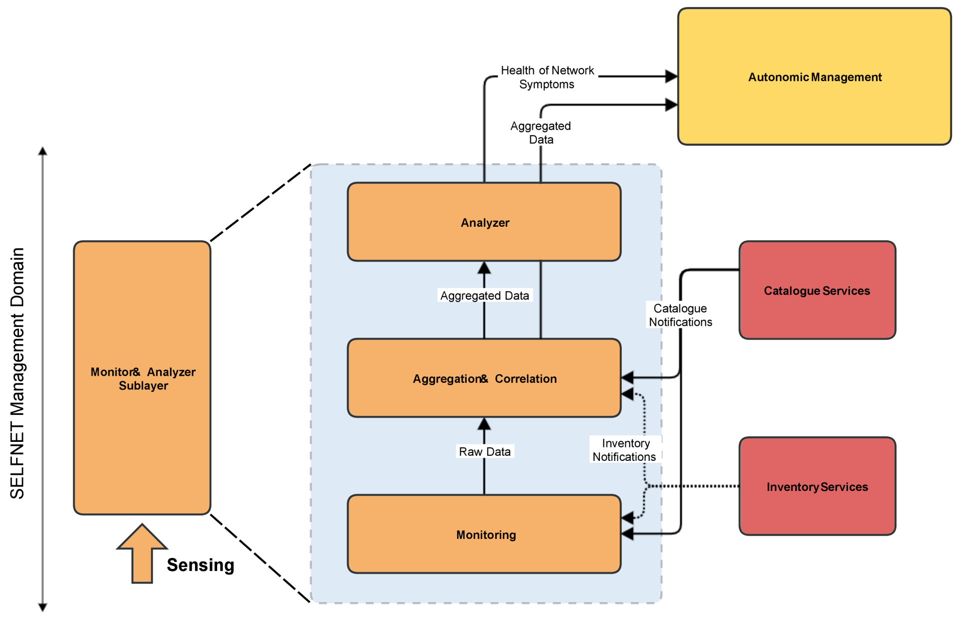


Figure 4.2: SELFNET Logical Architecture - Level 1

The main purpose of the analysis of this high level architecture is to show how the Monitor & Analyzer sub-layer is organized internally. With that in mind, the Inventory Services component will not be further examined as it is out of the scope of this work, and for the time being it is suffice to say that Monitoring Catalog, which will be further detailed in in section 4.2.1.1, is an inner component of the Catalog Service.

Taking a bottom up approach and starting with the Monitoring framework, its purpose and/or responsibility is to collect all the data provided by the sensors deployed in the network and store it in a database for later use by the Aggregation & Correlation framework.

Once that data is stored and available to the Aggregation & Correlation framework, it will be (as the name indicates) aggregated and/or correlated in order to provide the necessary information for the Analyzer framework.

Although not clearly presented in the figure, the Aggregation & Correlation framework also has threshold and notification mechanisms, and the Autonomic framework can also directly feed up on data provided by the Aggregation & Correlation framework in some cases, for instance, alarm notifications related to threshold policies.

The Analyzer framework's purpose is to provide HoN symptoms to the Autonomic framework in order for it to determine probable causes of underlying problems on the network and take proactive measures to resolve them (autonomic actions).

Next we will take another further step and examine the high level architecture of the Aggregation framework in section 4.2, where its inner components will be detailed.

4.2 Aggregation Framework Architecture

Figure 4.3 gives a first level of decomposition of the Aggregation framework before stepping into an even more detailed architecture that will be presented later on in figure 4.4. The Analyzer and Monitoring frameworks are also present to give some context, but the Analyzer framework will not be further detailed as it is out of the scope of this work; the sole purpose of showing the Monitoring framework is to help to better understand how the sensing data reaches the Aggregation framework.

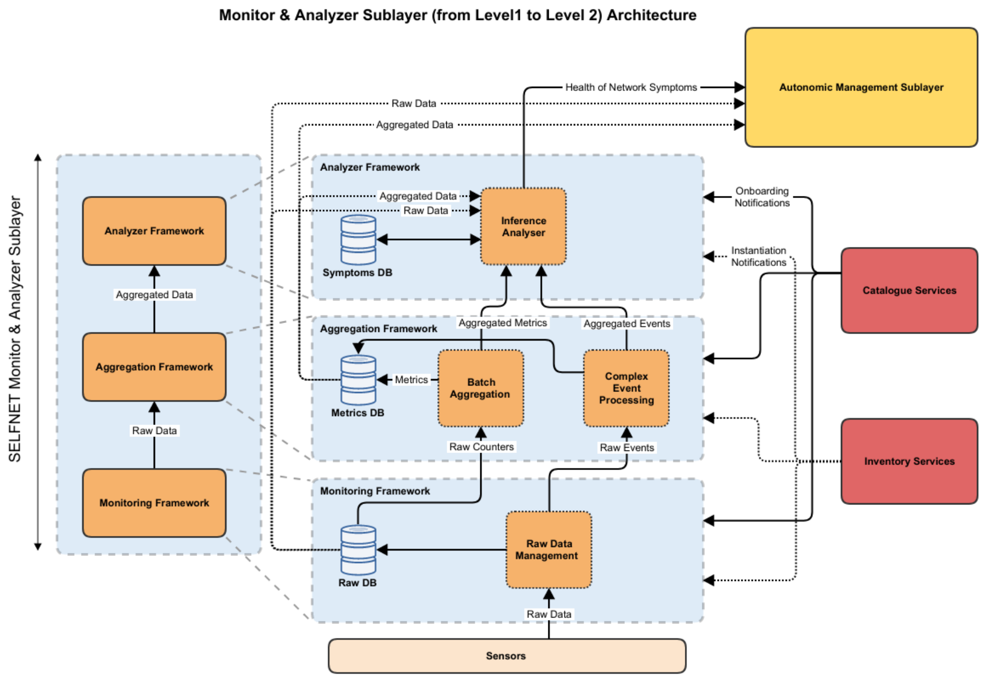


Figure 4.3: SELFNET Logical Architecture - Level 2

Considering the Monitoring framework and taking into account what was referred about this framework in section 4.1, we can now see that the component responsible for collecting and persisting the sensing data is the Raw Data Management. The sensing data is stored in the database that goes by the name of Raw Database (Raw DB). The data stored in the Raw DB can then be used by the Aggregation and the Analyzer of the Autonomic frameworks.

Stepping now into the Aggregation framework, it contains three major components, each one of them with specific roles: the Metrics Database, the Batch Aggregation Framework

(BAF) and the Complex Event Processing Framework (CEPF).

The Metrics DB is a time-series database where all the aggregated data is persisted, and the CEPF provides real-time or near real-time (in a five seconds window, at most) aggregated data. CEPF is out of the scope of this work and will not be further analyzed.

The BAF is the component responsible for fetching the raw sensing data from the Metrics DB and produce aggregated metrics (non real-time, e.g. every 30 seconds or every 1 hour).

Figure 4.4 depicts the architecture now with more detail.

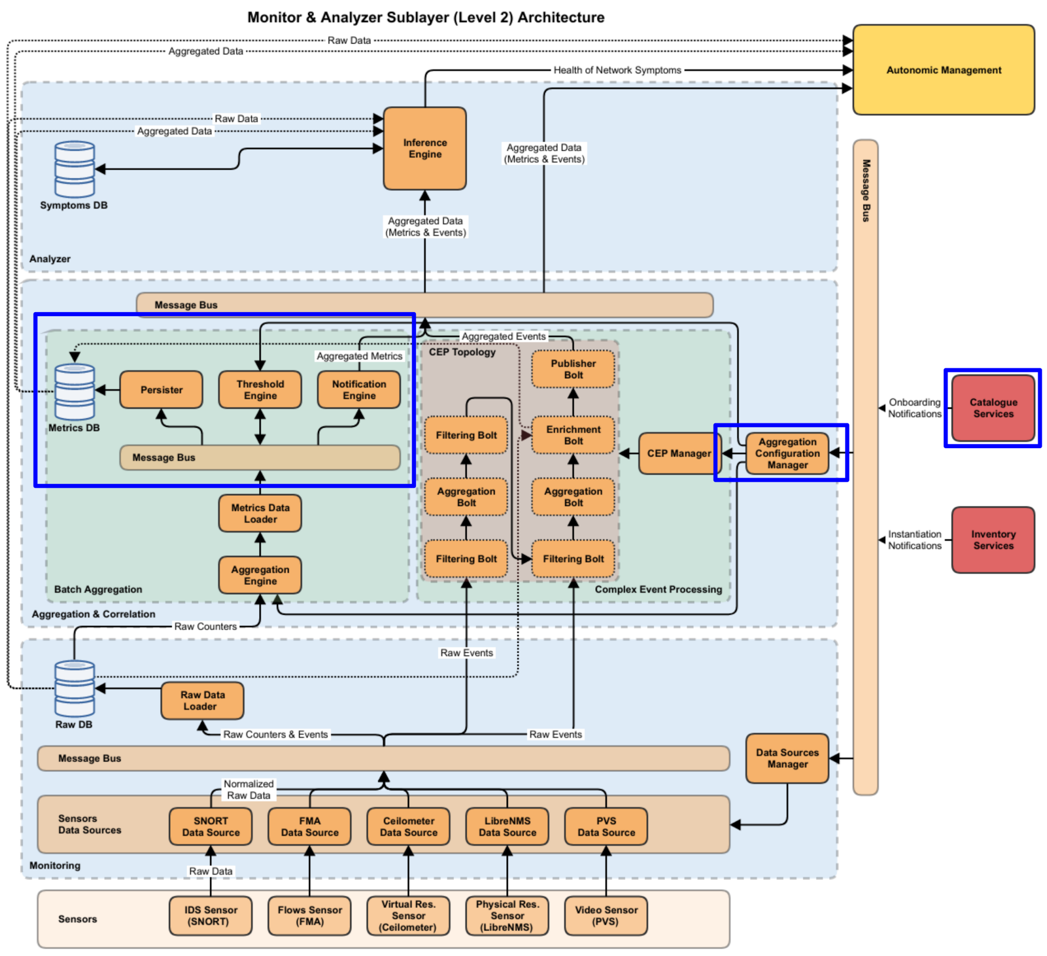


Figure 4.4: SELFNET Logical Architecture - Level 2 (detailed)

The components highlighted in the blue rectangles are the focus of this thesis. Most precisely, and starting from the right to the left side, there is the Monitoring Catalog (MC), which is part of the Catalog Services (other catalogs are out of the scope), the Aggregation Configuration Manager (ACM) and Monasca which was first introduced in section 3.3.1 and that comprises all the components inside the left most blue rectangle that are part of the BAF.

One last component of the BAF must be mentioned, the Aggregation Engine (AE), which is responsible for, as the name suggests, the aggregation of metrics. It fetches raw metrics from the Raw DB and aggregates them based in on-boarded rules from the Monitoring Catalog and stores them on the Metrics DB.

4.2.1 Catalog Services

4.2.1.1 Monitoring Catalog

This component is responsible for the on-boarding of aggregation and threshold rules that will be put in place on the BAF and CEPF components, as well as for the information on all types of sensors that are available: what kind of sensing data they provide and how the rules can make use them to produce aggregated metrics (real-time or not). The details of the implementation of this component will be later discussed in section 6.2 where its internal structure will be dissected, but for now it is enough to say that when a rule is on-boarded, the Aggregation Configuration Manager is notified about that through the message bus.

4.2.2 Aggregation Framework

4.2.2.1 Aggregation Configuration Manager

The enforcement of the rules on-boarded on the Monitoring Catalog is done by this component. In figure 4.4 it is possible to see that the ACM reaches out three specific components of the Aggregation framework: the Aggregation Engine, the Threshold Engine and the CEP Manager. All the real-time aggregation rules will be enforced on the CEP Manager, the non real-time ones on the Aggregation Engine and, of course, the threshold rules will be enforced on the Threshold Engine. In section 6.3 the implementation details will be discussed.

4.2.2.2 Monasca

Looking at figure 4.4 we can now see three new specific components together with the Metrics DB: the Persister, the Threshold Engine and the Notification Engine. They can be easily related to figure 3.6 presented in section 3.3.1, hence it is clear now that these components are provided by the Monasca service.

The specific details of the implementation will be discussed in section 6.1; this section states the purpose of these three new components:

- **Persister**: its role is to retrieve the aggregated metrics published by the Aggregation Engine on the internal message bus and store them in the Metrics DB; alarms published by the Threshold Engine will also be persisted on the Metrics DB;
- **Threshold Engine**: its role is to monitor the aggregated metrics published by the Aggregations Engine on the internal message bus and check if any of them cross any of the configured thresholds; if a threshold is crossed, the Threshold Engine publishes alarms on the internal message bus that will be picked up both by the Persister and the Notification Engine;
- **Notification Engine**: its role is to monitor the internal message bus for any published alarms, process that information and publish alarm notifications on the external message bus so that the Analyzer and Autonomic frameworks can be aware of the situation.

4.3 Use Cases and their Impact on the Architecture

It was already stated in section 1.2 that the SELFNET project is use-case driven, having three specific use-cases. We will now discuss the details on each one of them and how they impact the architecture of SELFNET.

4.3.1 Self-Healing

At the time being, this is the least mature use-case of the project, and as a consequence, it will not be so detailed as the other two. The main goal of the Self-Healing Use Case (SH-UC) is to deal with physical failures of the network. For instance, power outages or physical connection disruptions that may affect the network services.

In order to comply with the requirements of this use-case, sensing data coming from physical sensors will be needed, which might give the necessary information about energy fluctuations that could point out when a physical failure may occur.

Although it is important to act as soon as possible in such scenario, it is not critical that it happens in real-time. It is only necessary to fix the problem in a somewhat short period. For instance, important network services could be migrated from one Data Center Point-of-Presence (DC-PoP) to another, and all that is required is to forecast a power failure before it actually happens, and in such case, real-time sensing data is not as important as possible patterns based on previously gathered data (long and short term speaking).

4.3.2 Self-Optimization

The goal of the Self-Optimization Use Case (SO-UC) is to optimize the Quality-of-Experience (QoE) in video streaming with 4K resolutions in network links that are susceptible to be highly congested. In figure 4.5 we can find the representation of the control loop of the SO-UC that will be explained below.

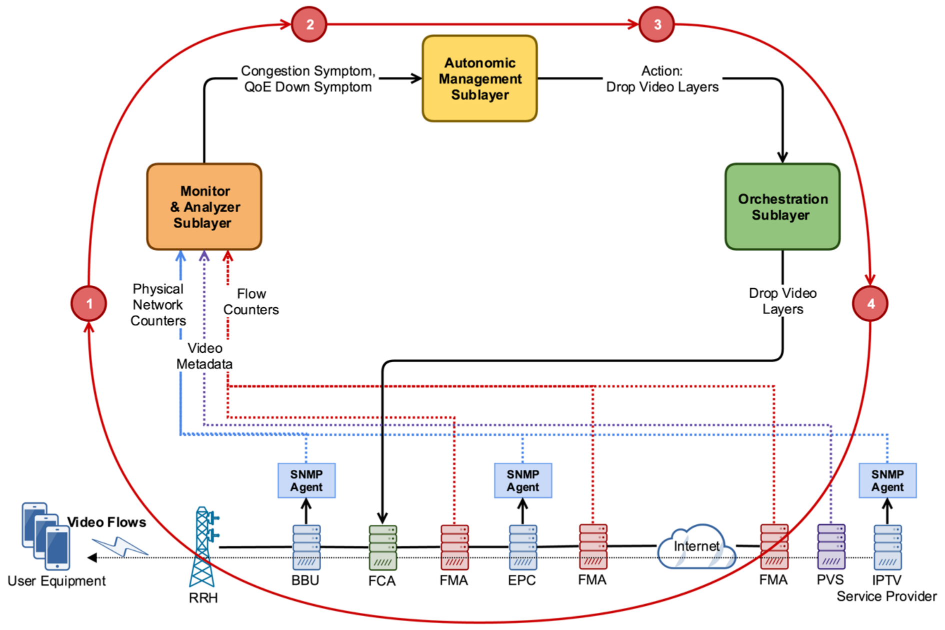


Figure 4.5: Self-Optimization Use Case

The first step is done by the Monitoring & Analyzer sub-layer, by collecting sensing information related to the network flows and counters of the physical network equipment. This sensing information is then persisted in the Raw DB of the Monitoring framework, aggregated by the Aggregation framework in order to determine a Congestion Index (CI) that will be processed by the Analyzer framework to determine the QoE of the end-user that is watching the video stream.

The CI is determined with sensing data provided by a sensor specifically designed by one of the partners of the SELFNET project that is called Flow Monitoring Agent (FMA) (it will be introduced in section 5.2.1) and common already existing physical sensors, for instance with the use of the well known Simple Network Management Protocol (SNMP) (see section 5.2.4).

This use-case stimulates primarily the real-time components of the Aggregation framework as video streaming is real-time by nature and, in case of congestion, the measures needed to mitigate the problem (dropping video layers in this case) must be quickly put in place.

4.3.3 Self-Protection

The primary goal of the Self-Protection Use Case (SP-UC) is to identify cyber-attacks (botnets) and take proactive measures to isolate the attacker (command & control) and the affected users/devices (the botnet zombies). There are two control loops in this use-case (actually there are three but only the first two will be covered here), and figure 4.6 depicts the representation of the 1st loop of the SP-UC that will be explained below.

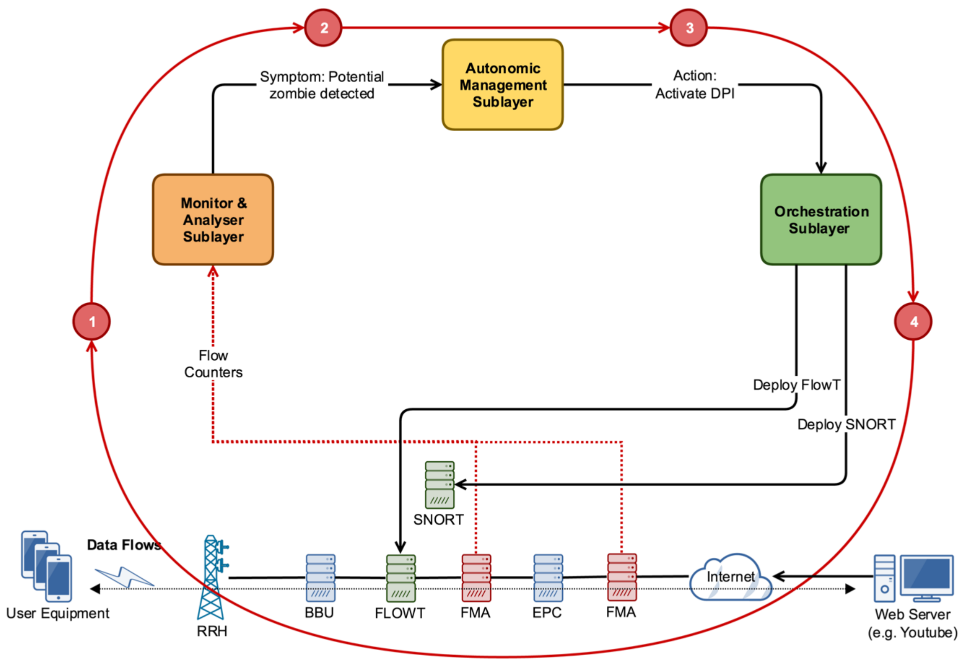


Figure 4.6: Self-Protection Use Case - Loop 1

In this loop the Monitoring & Analyzer sub-layer will monitor the network communication flows through the Flow Monitoring Agent (FMA), which is a sensor specifically developed to gather raw counters on these flows (see section 5.2.1). The data collected by this sensor must then be persisted, aggregated and analyzed by the Monitoring & Analyzer sub-layer. These tasks that will be respectively handled by the Monitoring, Aggregation and Analyzer frameworks of this sub-layer.

In order to identify suspicious network communications, a considerable amount of sensing data must be collected by the FMA, and the Aggregation framework must be able to produce aggregated metrics about the network flows, establishing or indicating patterns that can point out the disruptive communications. One way to accomplish that is by aggregating the network flows by their source and destination endpoints/hosts, for instance using their IP addresses and ports, in a non-realtime fashion and with a pre-defined and relatively short aggregation time period (e.g. 30 seconds).

These aggregated metrics must also be persisted by the Aggregation framework (the Metrics DB), and when communications between the same endpoints/hosts start revealing themselves suspicious, for instance the communication frequencies and intervals within the aggregation time period (the Threshold Engine), the Analyzer framework must be notified (the Notification Engine).

This approach with identification of patterns, based on the aggregated metrics followed by notifications, eliminates the need of having the Analyzer framework constantly querying the Aggregation layer, or at least tries to avoids this issue. Also, the Aggregation framework must have a way to expose its raw metrics, aggregated metrics and notifications to the external components, namely the Analyzer and Autonomic frameworks, as well a means for these components to dynamically interact with it. This is where the Message Bus and Monitoring Catalog come in, allowing the Analyzer and Autonomic frameworks to take advantage of the Aggregation framework features, to effectively create the required aggregated metrics, thresholds and notifications.

Once the Analyzer framework detects a suspicious communication pattern (potential zombie detected), it then reports it back to the Autonomic Management sub-layer, so that on its hand it can evaluate the symptoms and decide on how to react next.

In this particular use-case the Autonomic framework will issue a request to the Orchestration sub-layer to activate a Deep Packet Inspection (DPI) sensor, SNORT in this case (see section 5.2.2), which is a Virtual Network Function (VNF), with the objective to confirm if the suspicious network communications are in fact part of a cyber-attack (a botnet in this case), or otherwise dismiss them as false-positives.

As a consequence, the Orchestration sub-layer will also deploy a flow mirroring SDN application (the FLOWT), in order for the DPI to be able to analyze the network flows. Once the deployment of both of them are performed by the Orchestration sub-layer, the 2$^{\text{nd}}$ loop of the SP-UC (shown in figure 4.7 below) will take place.

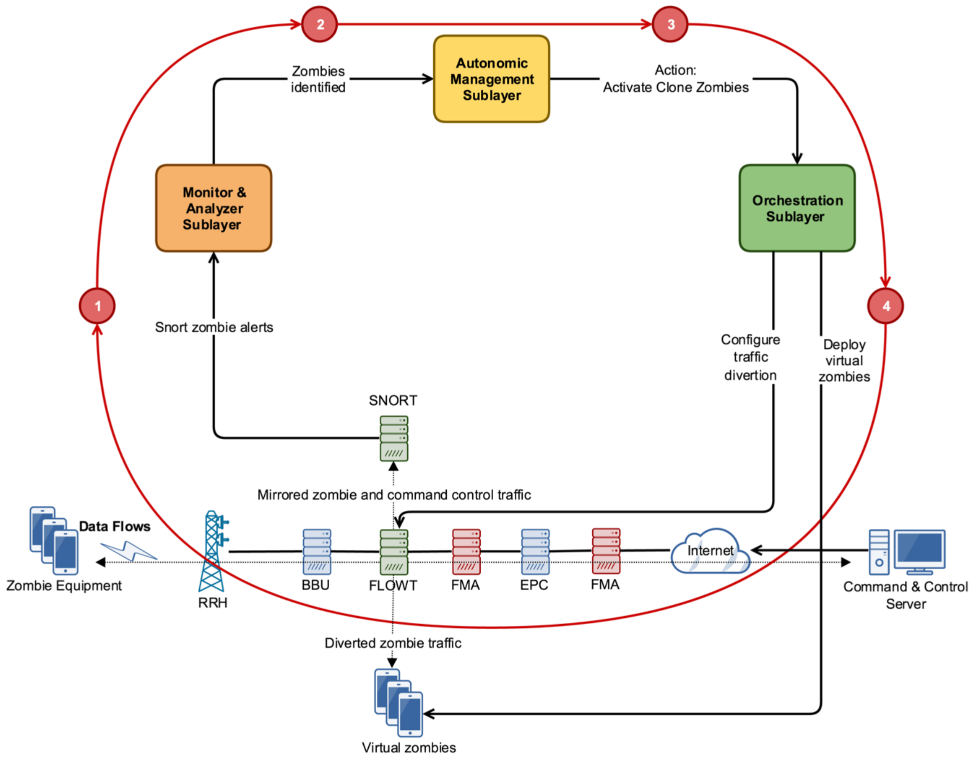


Figure 4.7: Self-Protection Use Case - Loop 2

When the suspicious network communications are indeed confirmed to be part of a cyber-attack by the DPI sensor, it will then start issuing event alerts towards the Monitoring & Analyzer sub-layer. These alerts must be correlated in realtime by the Complex Event Processing Framework (CEPF) of the Aggregation framework.

This process will have the purpose of identifying the compromised hosts belonging to the Botnet, specified in cyber-security nomenclature as *zombies*, as well the host that controls all the zombies, also known as the Command & Control (CnC).

Similarly to the 1st loop, the Aggregation framework will issue alarm notifications on the produced (realtime) aggregated metrics for the upper external framework to pick up.

The Analyzer framework will once again report back the confirmed infected hosts to the Autonomic Management sub-layer, which on its hand, will issue a request to the Orchestration sub-layer to deploy virtual zombies in a Honey-Net [53][54] to receive diverted traffic from the CnC.

4.4 Summary

This chapter started by introducing SELFNET's High Level architecture with a top-down approach until the most detailed level of the architecture was exposed, up to a point where the Aggregation Framework was exposed and discussed in detail. Finally SELFNET's use-cases were presented and explained to reveal their impact on the architecture of the project.

CHAPTER 5

Interfaces, Data Sources and Data Models

This chapter discusses important topics before getting into the details of the implementation of three components that are the scope of this thesis in the next chapter (the Monasca, the Aggregation Configuration Manager and the Monitoring Catalog). This description is important to better understand what data is produced and consumed by these components, as well as how it is modeled and/or represented.

We will start by taking a look at the Aggregation Framework APIs (section 5.1) to understand their purpose and what data they provide, followed by a summary of the several data sources (section 5.2) which are responsible for feeding the Aggregation framework with sensing data, and lastly an analysis will be provided on the two data models (section 5.3) that are responsible for defining and normalize the data provided by the several APIs and sensors.

5.1 Aggregation Framework APIs

The communications between the Aggregation framework and other surrounding external components of the SELFNET architecture are done on its boundaries through specific interfaces.

This section will describe them; they can be divided in three specific boundaries with the following logical arrangement: *Northbound* - where the external components can feed on data produced by the Aggregation framework; *Eastbound* - where the Aggregation framework receives its aggregation and thresholding rules from; *Southbound* - where the Aggregation framework feeds on sensing data.

5.1.1 Northbound APIs

The Northbound APIs comprise all the interfaces that expose the data processed by the Aggregation framework to the upper layers/frameworks of the SELFNET, namely the Analyzer and Autonomic frameworks. There are three of them, the Aggregated Metrics Database API, the Raw Database API and the Alarm Notifications API that will be discussed in the following sub-sections.

5.1.1.1 Aggregated Metrics Database API

The Aggregated Metrics Database API is, in its essence, the Monasca API [55] which provides a restful JSON interface for interacting with and managing monitoring related resources. Its main resources can be summarized as follows:

- **Metrics**: Provides for storage and retrieval of metrics;
- **Measurements**: Operations for querying measurements of metrics;
- **Statistics**: Operations for evaluating statistics of metrics;
- **Notification Methods**: Represents a method, such as email, which can be associated with an alarm definition via an action: when an alarm is triggered, notification methods associated with the alarm definition are triggered; in the particular case of SELFNET only one method is used, and it has been developed with the specific needs of this project in mind (further discussed in section 6.1.7);
- **Alarm Definitions**: Provides CRUD operations for alarm definitions;
- **Alarms**: Provides CRUD operations for alarms, and querying the alarm state history.

However, when it comes to the northbound interactions with the upper layers of the SELFNET architecture, particularly the Analyzer and the Autonomic frameworks, only the read operations of the API are to be used. The specific operations are listed on table 5.1.

Metrics	List Metrics	[56]
	List Names (of Metrics)	[57]
	List Dimension Values	[58]
	List Dimension Names	[59]
	List Measurements (of Metrics)	[60]
	List Statistics (of Metrics)	[61]
Alarms	List Alarms	[62]
	List Alarms State History	[63]
	List Alarm State History	[64]

Table 5.1: Monasca API - Northbound (read-only) operations

The remaining operations of this API are only used within the Aggregation framework, and some not used at all. The most important ones are listed in table 5.2 for future reference, which will become useful for the discussion of the implementation of both the Monasca and the Aggregation Configuration Manager (ACM) in sections 6.1 and 6.3.

Metrics	Create Metric	[65]
Notification Methods	Create Notification Method	[66]
	List Notification Methods	[67]
Alarm Definitions	Create Alarm Definition	[68]
	List Alarm Definitions	[69]
	Delete Alarm Definition	[70]

Table 5.2: Monasca API - Internal (read/write) operations

5.1.1.2 Raw Database API

The Raw Database API exposes the raw data persisted in the Raw DB externally to the Analyzer and the Autonomic frameworks, as well as internally within the Aggregation framework, specifically to the Aggregation Engine (AE).

The term API is used loosely in this case since there is no REST, CLI interfaces or a message bus to retrieve data. Instead, accessing the values contained within the database is made via the Cassandra Query Language (CQL), which is similar to the well known Structured Query Language (SQL).

The structure of the data stored in this database is out of the scope of this work and it will not be further detailed. The sole purpose of this section is to clarify that the data stored in the Raw DB, although it is mainly used by the AE, it is also exposed externally (unprocessed in any way).

5.1.1.3 Alarm Notifications API

The component responsible for the alarm notifications is the Monasca Notification Engine, which monitors the message bus for any incoming alarms issued by the Monasca Threshold Engine. The Notification Engine (natively) supports several types of notifications such as *Email*, *Webooks* and *Pagerduty*, which were originally developed and integrated in the Notification Engine as plugins.

These notification types are essentially designed to notify human operators; however, they do not address the needs of the SELFNET project, as the goal is to have as much automated tasks as possible whenever and wherever possible and justifiable.

With that in mind, a new plugin was developed to take the raw alarm notifications issued by Notification Engine and republish them in the message bus respecting the RADM (this model will be discussed in section 5.3.1).

The term API is loosely used here, as the interactions with the Analyzer and Autonomic framework with these notifications are a one-way communication, i.e. the upper frameworks are only able to read the alarm notification messages that are published to the message bus. Furthermore, since these notifications are a result of alarms issued by the Threshold Engine, they are directly associated with thresholds that will be defined through the Monitoring Catalog API that will be discussed in section 5.1.2.1.

The structure of the alarm notifications have several fields/attributes, many of them that will be discussed in section 5.3.1. Therefore, only one particular attribute and its inner contents, the *dataDefinition*, will be discussed here as its structure is unique to the alarm notifications. The contents and inner structure of the *dataDefinition* are as follows:

- **severity**: the numerical representation of the severity level, an integer ranging from 0 (the least severe) to 3 (the most severe), also mapped to the Monasca raw (string) severity levels (LOW, MEDIUM, HIGH and CRITICAL);
- **newState**: the current state of the alarm with three possible values: OK, ALARM and UNDETERMINED;
- **oldState**: the previous state of the alarm with the same possible values as the *newState*;
- **alarmDescription**: a brief description of the alarm, if set in the Monasca alarm definition (threshold);
- **alarmTimestamp**: the alarm triggering time in unix/epoch milliseconds;
- **metadata**: other data (key/value pairs) providing extra information about the alarm;
- **metrics**: a list containing the metrics that crossed the threshold, hence producing the alarm; each metric on the list contains the following data:
  - **id**: the ID (internal to Monasca) of the aggregated metric, if applicable;
  - **name**: the name of the aggregated metric;
  - **dimensions**: the dimensions of the aggregated metric.
- **subAlarms**: a list containing the sub-alarms that together triggered the alarm (a threshold expression may be composed by sub-expressions, each one generating a sub-alarm); each sub-alarm contains the following data:
  - **function**: the mathematical or statistical operation that is used in the sub-expression that generated the sub-alarm (e.g., MIN, MAX, COUNT, SUM, AVG, etc.);
  - **deterministic**: a boolean field; if true it means that only the *OK* and *ALARM* transition states (deterministic states) were taken into account, otherwise the *UNDETERMINED* state was also included (non-deterministic);
  - **period**: the period (in seconds) considered to accept the threshold as effectively crossed;
  - **periods**: the number of periods (number of times) considered to accept the threshold as effectively crossed;
  - **threshold**: the threshold (numeric) value;
  - **operator**: the relational operator used in the sub-expression (e.g. >=, <=, >, <, and son on);
  - **metricDefinition**: the aggregated metric used in the sub-expression which include the following fields:
    - **id**: the ID (internal to Monasca) of the aggregated metric, if applicable;
    - **name**: the name of the aggregated metrics;

    * **dimensions**: the dimensions of the aggregated metric used as filters, consisting in key/value pairs that represent the name of the dimension and its value (if applicable);
  - **currentValues**: a list
  - **subAlarmState**: the sub-alarm state with the same possible values of the *newState* and *oldState* fields.

To better understand the structure of the Alarm Notification API, a real example can be consulted in Appendix C.1.2 as a practical case.

5.1.2 Eastbound

The eastbound communications of the Aggregation framework are restricted to the Monitoring Catalog (MC), more specifically between the MC and the Aggregation Configuration Manager (ACM). Whenever a change in the MC occurs, the ACM is notified through the message bus and then it retrieves the pertaining information from the MC's restful API. The implementation of the MC will be a subject to discuss on chapter 6. This section discusses the information related to its interface.

5.1.2.1 Monitoring Catalog API

As referred before, the MC has a restful API. The interface is composed of several endpoints supporting CRUD operations and the responses given in JSON format. The responses of the read operations follow the format specified by the Monitoring Catalog Data Model (MCDM) that will be discussed in section 5.3.2, as well the data sent in the create and update operations that must follow this model. The endpoints and their respective operations are summarized on table 5.3.

Endpoint	**Supported Operations**			
	Create	**Read**	**Update**	**Delete**
/catalog		●		
/catalog/monitoring		●		
/catalog/thresholding		●		
/catalog/monitoring/sensors	●	●		
/catalog/monitoring/sensors/{sensor_id_or_name}		●	●	●
/catalog/monitoring/sensors/{sensor_id_or_name}/ resource_types	●	●		
/catalog/monitoring/sensors/{sensor_id_or_name}/ resource_types/{rtype_id_or_name}		●	●	●
/catalog/monitoring/sensors/{sensor_id_or_name}/ resource_types/{rtype_id_or_name}/data_types	●	●		

Endpoint	**Supported Operations**			
	Create	**Read**	**Update**	**Delete**
/catalog/monitoring/sensors/{sensor_id_or_name}/resource_types/{rtype_id_or_name}/data_types/{dtype_id_or_name}		●	●	●
/catalog/monitoring/rules_groups	●	●		
/catalog/monitoring/rules_groups/{rules_group_id_or_name}		●	●	●
/catalog/monitoring/rules_groups/{rules_group_id_or_name}/dimensions	●	●		
/catalog/monitoring/rules_groups/{rules_group_id_or_name}/dimensions/{dimension_id_or_name}		●	●	●
/catalog/monitoring/rules_groups/{rules_group_id_or_name}/rules	●	●		
/catalog/monitoring/rules_groups/{rules_group_id_or_name}/rules/{rule_id_or_name}		●	●	●
/catalog/thresholding/thresholds_groups	●	●		
/catalog/thresholding/thresholds_groups/{thresholds_group_id_or_name}		●	●	●
/catalog/thresholding/thresholds_groups/{thresholds_group_id_or_name}/thresholds	●	●		
/catalog/thresholding/thresholds_groups/{thresholds_group_id_or_name}/thresholds/{threshold_id_or_name}		●	●	●

Table 5.3: Monitoring Catalog - REST Endpoints and Operations

A practical example of the content of the full catalog, corresponding to the */catalog* endpoint, can be found on Appendix C.2 (Monitoring Catalog Data Model Example).

5.1.3 Southbound

There is only a single southbound interface which is used to input sensor reported data into the Aggregation Layer. It is comprised of multiple topics of a Kafka message bus that follows a publish/subscribe communication model, effectively separating the different types of sensing data from the several data sources that will be discussed in section 5.2.

The sensing data published to the Kafka topics follow a data model in order to normalize its structure, so it can be more easily interpreted and processed. The Raw Aggregation Data Model (RADM), which will be discussed in section 5.3.1, is this reference model.

5.2 Data Sources (Sensors)

The Aggregation framework feeds on data that comes from the Monitoring framework and, in order to better understand how this information is collected as well as its different types and purposes, in this section we will discuss the several sensors or data sources that feed the Aggregation framework with sensing data.

5.2.1 Flow Monitoring Agent

The Flow Monitoring Agent (FMA) is a software sensor specifically designed by a partner of the SELFNET project, with the responsibility of monitoring network flows, and that can be categorized (according to its original author) as a Physical Network Function (PNF).

It reports both flow counters (statistics) and meta-data (events) to two distinct Kafka topics (the *de facto* messaging bus of the project) and it has a great focus on Ultra High Definition (UHD) video flows, a feature that is of particular relevance to the SO-UC described in section 4.3.2.

5.2.2 Snort

Snort [71] is an open-source, rule-based, Intrusion Prevention System (IPS) and Intrusion Detection System (IDS) [72] with the ability to perform real-time traffic analysis and packet logging on IP networks.

Combining the benefits of signature, protocol, and anomaly-based inspection, Snort is the most widely deployed IPS/IDS technology worldwide [73]. Its Deep Packet Inspection (DPI) and anomaly-based inspection methods deliver flexible protection from malware attacks, providing SELFNET the required capabilities to address the SP-UC discussed in section 4.3.3, performing analysis on the network communications to detect possible botnets and their respective infected hosts (zombies).

Unlike the FMA, which is a sensor specifically developed for the SELFNET project, Snort does not natively report its data in the RADM format and, in order to address this issue, the Monitoring framework effectively translates the data produced by Snort to this required format, publishing the normalized data in its respective Kafka topic, reported as an event, so it can be correctly interpreted and used by the Aggregation framework.

5.2.3 Ceilometer

Ceilometer was already discussed in section 3.3.2 and is also used as a data source in SELFNET. It provides sensing data (counters and events) from the virtualization infrastructure (i.e. OpenStack) allowing the Aggregation framework to also process information related to all the virtual resources in use, for instance Virtual Machines (VM) CPU and RAM usage, or even from the hypervisor and/or cloud infrastructure underlying services, like virtual routers and switches incoming/outgoing traffic.

Just like in Snort's case, Ceilometer (unmodified) does not publish its data according to the RADM, but unlike Snort's case, this data is not transformed by the Monitoring framework but rather through a modification done on the publishing mechanism of Ceilometer, effectively giving it means of directly publishing the data to its assigned Kafka topic, already normalized in accordance with the RADM format.

5.2.4 LibreNMS

LibreNMS [74] is an open-source, auto-discovering network monitoring system with support for a wide range of network hardware and operating systems including Cisco, Linux, Windows, Juniper, VMware, HP, Dell and many more.

Its features include automatic discovery of the network, supporting several well known and widely used protocols such as CDP, LLDP, OSPF, BGP, SNMP and ARP.

Suffice to say that, like other already discussed sensing data sources, the data gathered and published by LibreNMS needs to be transformed in order for it to conform to the RADM specifications, sensing data which is then published on a Kafka topic already normalized and once again giving the Aggregation framework means to easily process it.

5.3 Data Models

5.3.1 Raw and Aggregation Data Model

In section 5.2 we discussed the several data sources that collect sensing data from the SELFNET infrastructure, most of them reporting their data in their own specific structures, making it difficult for the Aggregation framework to process that data in a simple and systematic way.

To mitigate this problem the Raw and Aggregation Data Model (RADM) was conceived to normalize that data, organizing and simplifying the different reported data structures to a single common one that could be more easily digested by the Aggregation framework.

Although this model was initially created to normalize the data reported by the data sources (sensors), later on was also adopted to represent the structure of the alarm notifications published by the Aggregation framework, effectively normalizing all the data consumed or produced by it, with the exception of the Raw DB.

With that in mind, the RADM categorizes the incoming/outgoing data to/from the Aggregation framework in three distinct data types:

- **Counters**: all the (incoming) statistical data that can be used to produce aggregated metrics, for instance the data reported by the FMA which monitors network flows, reporting statistical fields like *totalOctets*, *totalpktCount* and so on, periodically or when a change in a flow occurs;

- **Events**: all the (incoming) data related to changes or occurrences in the infrastructure detected by the sensors, for instance a very high CPU usage on a VM reported by Ceilometer or a botnet zombie detected by Snort;

- **Alarm (Notifications)**: all the (outgoing) data related to alarm notifications produced/published by the Aggregation framework when a certain configured threshold is crossed.

The structure of the RADM is composed by a set of generic attributes/fields that are used to hold the information reported by the different data sources, which are listed in table 5.4, briefly describing their purposes in order to give the reader an insight on the common representation of the normalized data that conforms to this model.

Field	Description
timestamp	The data source reporting time in unix/epoch milliseconds.
dataType	The type of data that is being reported, having one of three possible values: *statistics*, *event* or *alarm*.
resourceType	The type of resource being reported, which depends on the resource that is being monitored by that data source. For instance *flow*, *vm*, *snort_dpi_event* and so on (see section 5.2).
reporterID	The unique identifier of the reporting data source which is a full distinguished name composed by key/value pairs from the inner attributes of the *reporterDescription* field.
resourceID	The unique identifier of the resource being reported by the data source which is a full distinguished name composed by key/value pairs from the inner attributes of the *resourceDescription* field.
reporterDescription	Contains a set of attributes (key/value pairs) that represent the description of the data source reporting the data.
resourceDescription	Contains a set of attributes (key/value pairs) that represent the description of the resource being reported by the data source.
dataDefinition	Contains a set of attributes (key/value pairs) that represent the data itself (*statistics*, *event* or *alarm* data) that is being reported by the data source.

Table 5.4: RADM Fields

In order to better understand how the data is modelled by the RADM, a couple of examples can be found in Appendix C.1 (Raw Aggregation Data Model Examples).

5.3.2 Monitoring Catalog Data Model

The Monitoring Catalog Data Model (MCDM) is the model that characterizes the structure of the Monitoring Catalog (MC), and is designed to define how the output of the MC should be viewed by and given to the SELFNET components that interact with it.

It is composed of two major structures, the *monitoring* and the *thresholding* sections that can be summarized as follows:

- **monitoring**: this section of the MC holds all the information related to the available data source types (sensor types) and aggregation rules that shall be enforced in the Aggregation framework, hence it is sub-divided in two other sections, the *sensors* section and the *rules_groups* section;
- **thresholding**: it holds all the information related to the threshold rules that shall be enforced in the Aggregations framework, and it only has one sub-section, the *thresholds_-groups*.

Its most basic representation in JSON, which would be an empty catalog, can be represented this way:

```
{
    "monitoring": {
        "sensors": [],
        "rules_groups": []
    },
    "thresholding": {
        "thresholds_groups": []
    }
}
```

We will now detail the contents of the *sensors*, *rules_groups* and *thresholds_groups* sections (all of them lists), starting with the *sensors* one:

- **name**: the name of the sensor (e.g., FMA);
- **description**: an optional brief description of the sensor;
- **resource_types**: a list of resource types supported by the sensor with each element containing the following attributes:
  - **name**: the name of the resource type (e.g., FLOW);
  - **data_types**: a list of data types supported by the sensor with each element containing the following attributes:
    - **name**: the name (or type) of the data type according to the RADM - *statistics*, *event* or *alarm*;
    - **identifiers**: a list containing the IDs reported by the sensor according to the RADM; in this case they are always same across all sensors - the *resourceID* and the *reporterID* fields;
    - **reporter_description**: a list containing the attributes that describe the reporter (e.g., *reporterHostname*, *reporterIP*, etc.);
    - **resource_description**: a list containing attributes that describe the resources monitored by the sensor (e.g., *flowHash*, *encapsulationLayer*, etc.)
    - **data_definition**: a list containing attributes that represent what the sensor can effectively monitor and report .(e.g. *currentPktPerPeriod*, *totalOctets*, etc.)

Next the *rules_groups* section of the MCDM will be described. This structure is further divided in two major sections, the *dimensions* and the *rules* sections, with the latter having some attributes dependent on the first one:

- **name**: the name of the aggregation rule;
- **sensor_ref_list**: a list containing the names of the sensors that are used by the rules defined in this rules group; the names of the sensors must be valid, i.e they must exist in the sensors list of the catalog;
- **period**: the period (in seconds) of the monitored data that will be used for the aggregations rules;
- **dimensions**: a list containing mappings of sensor resources to names that will represent the dimensions of the aggregated metrics on the time-series database of SELFNET:

  - **source_ref**: the sensor resource represented in dot notation, indicating the path that needs to be traversed in the *sensors* section of the catalog in order to reach it (e.g. *FMA.FLOW.event.srcIP*);
  - **name**: the name of the dimension that will represent the *source_ref* in the aggregated metric stored in the time-series database (e.g. *SourceIP*);

- **rules**: a list containing the aggregation rules that will be enforced in the Aggregation framework by this rules group:

  - **name**: the name of the aggregation rule;
  - **group_by**: a list containing resources of one or more sensors, again in dot notation, that should be used to group counters or events to produce the aggregated metric; for instance *FMA.FLOW.event.srcIP* used as the single entry of this list would produce aggregated metrics of network communications grouped by its source IP;
  - **filters**: a list containing expressions to exclude certain counters or events from the aggregated metric, using dot notation expressions of the sensors resources, operators and values; for instance *"FMA.FLOW.event.flowLayer == 0"* would exclude all the counters or events from the aggregation where the *flowLayer* value is zero;
  - **formula**: the expression used to effectively produce the aggregated metric; in its most basic form it is composed by mathematical or statistic operations (MIN, MAX, SUM, COUNT, AVG, LAST, etc) applied to the sensors data (again represented in dot notation) or even to already existing aggregation rules within the same *rules_group* which can be viewed as sub-expressions; basic mathematical operations (addition, subtraction, division and multiplication) can be used together with the sub-expressions to compose more complex expressions, for instance *"AVG(Sensor.FMA.FLOW.statistics.currentOuterOctets) + SUM(Rule.avg_pkt_count)"*;
  - **metadata**: other data providing extra information about the aggregation rule.

The last section of the MCDM, *thresholds_groups*, will now be described. This structure is also a list whose elements are groups of threshold rules that belong to a particular logical problem that needs to be addressed (e.g. organized by use-cases). Since the thresholds need to be applied on aggregated metrics already being produced, an attribute dependency on the *rules_groups* section is expected, as we will see:

- **name**: the name of the threshold group;

- **thresholds**: a list containing the thresholds of this group, each threshold obeying the following structure;

  - **name**: the name of the threshold rule;
  - **description**: an optional brief description of the threshold;
  - **severity**: the numerical representation of the severity level, an integer ranging from 0 (the least severe) to 3 (the most severe);
  - **state_notifications**: a list containing the transitional states of the threshold for which a notification should be emitted, with three possible states: alarm - when the

values cross above the specified threshold, ok - when the values get back below the specified threshold, undetermined - when it is not possible to determine the values due to the absence of sufficient data, e.g. a sensor not reporting its sensing data;

– **match_by**: a list that contains the dimensions of the aggregated metrics that should be taken into account when applying the threshold expression; it is similar to the *group_by* attribute of the rules, but while the latter is used to effectively create the dimensions of the rule, this one is used to match those dimensions; it is also in dot notation format and it must match the dimensions defined in the *rules_group* (e.g. *SelfProtection-Batch.SourceIP*);

– **expression**: the expression that defines the threshold; in its most basic form it is also composed by mathematical or statistic operations (MIN, MAX, SUM, COUNT, AVG, LAST, etc) and logical operators (>=, <=, >, <, etc) applied to aggregated metrics defined by the rules that exist on the *rules_groups* section of the catalog, which can be viewed as sub-expressions; the sub-expressions can be joined together by using relational operators (AND/OR) to create more complex expressions, for instance *"AVG(SelfProtection-Batch.avg_pkt_count, deterministic, 180) >= 2 AND AVG(SelfProtection-Batch.comunication_frequency, deterministic, 180) >= 6"*; taking the previous threshold expression example, it is also possible to define which state transitions should be taken into account (ok, alarm, undetermined) with use of the keyword *determined*; if present will only consider the *alarm* and *ok* transitions and if absent it will also consider the *undetermined* state; it is also possible to define the period (in seconds) that should be considered before triggering the alarm (the number 180 in the example); although not present in the given example, it is alto possible to define the number of times the threshold should be crossed before triggering the alarm, by placing the keyword *times* followed by an integer number, after the value of the threshold on the sub-expressions (e.g. *"AVG(SelfProtection-Batch.avg_pkt_count, deterministic, 180) >= 2 times 3"* would only trigger an alarm if the threshold is crossed three times);

– **metadata**: other data providing extra information about the threshold rule.

This ends the description of the MCDM, which is a fairly basic description and by itself without any example, as simple as it could be, may be insufficient to properly understand it. With that in mind, an example can be found on Appendix C.2 (Monitoring Catalog Data Model Example) to aid the previously given description of the model.

Furthermore, since the threshold engine of the SELFNET project is in fact the threshold engine of Monasca, to complement what has been said on the *expression* attribute of the *thresholds_groups*, further documentation [75] is available online.

5.4 Summary

This chapter exposed some important APIs, data sources, and data models used on the project, starting with the Aggregation Framework APIs which were divided by boundaries to better frame the several APIs context. Next the existing data sources, i.e. the sensors, were discussed explaining what types of data they provide. Finally, the two data models that were used as guides for the different data types where exposed and detailed.

CHAPTER 6

Implementation

In this chapter the components that are the subject of this thesis will be addressed. It will start with an analysis on Monasca, describing in detail its micro-services that were used and integrated on the project, as well their purposes, including some details of the Notification Engine plugin that was developed purposely to address the requirements of the project (section 6.1).

It will also be discussed how the Monitoring Catalog and the Aggregation Configuration Manager were implemented as well as their respective architectures and used technologies (sections 6.2 and 6.3). A brief summary will also be presented at the end of the chapter (section 6.4)

6.1 Monasca

Monasca was first introduced in section 3.3.1, and later on its role on the SELFNET architecture was mentioned in section 4.2, explaining which components of Monasca were part of the BAF. We will now take a further step and examine its internal architecture, seen in figure 6.1 (next page), and check how everything is implemented.

The architecture of Monasca follows a micro-services architecture pattern, services that will be described in the upcoming sections.

Although it was previously developed in *Java*, currently the default programming language of Monasca is *Python*. Most of the components have already been converted to *Python*; their former counterparts developed in *Java*, although still available for use, are considered deprecated and marked for removal in the future. The exception to this rule is the Monasca Threshold Engine, which being an Apache Storm topology, it is still developed in *Java*.

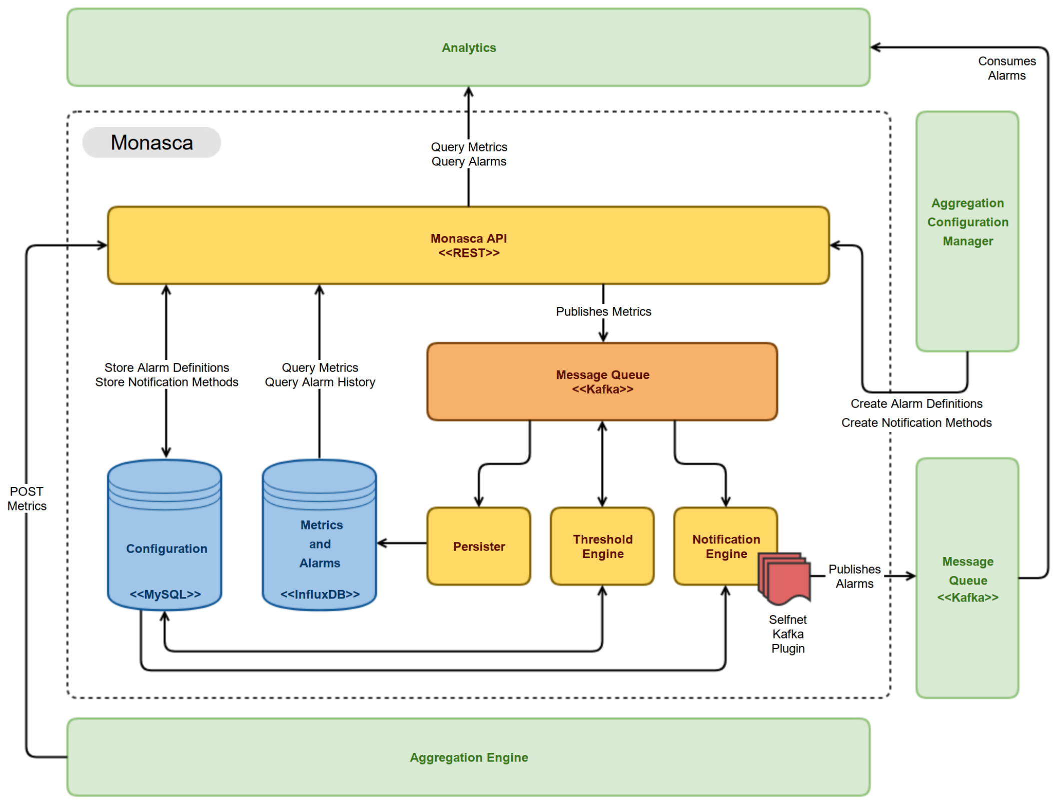


Figure 6.1: Monasca Architecture

6.1.1 Message Bus

The message bus is a central component of the Monasca architecture, and the technology chosen by their developers was the Apache Kafka. All the metrics sent through the Monasca API are first published here, so they can be read both by the Persister, which as the name indicates is the component responsible for persisting the metrics (and alarms) on a database, and by the Threshold Engine which also needs to consume the metrics in order to process them and determine if they have exceeded defined thresholds.

The Threshold Engine also publishes alarms to this message bus, that can be consumed both by the Persister to be stored on a database, and by the Notification Engine to determine if alarm notifications must be sent.

6.1.2 Storage

Monasca uses two storage/database technologies. MySQL is used as the configuration database, storing *alarm-definitions* and *notification-methods* in particular. When the Monasca API is used to create new *alarm-definitions* (thresholds) and *notification-methods*, this is where they are stored. The Threshold Engine also uses this database to write/read *alarm-states*, and the Notification Engine on its hand uses it to read *alarm-states*.

InfluxDB is used store all the metrics and alarms produced by Monasca. The write operations are only performed by the Persister while the read operations (queries) can be be directly performed by the Monasca API.

6.1.3 Monasca API

The Monasca API [76] is a RESTful API based on the Falcon Web Framework [77], which is a reliable, high-performance Python web framework for building large-scale, responsive application backends and micro-services that work with any WSGI server.

Programmatically speaking, the API is divided in several Python classes [78], separating and organizing the API in logical groups that together compose the Monasca API and are responsible for handling the HTTP operations (POST/GET/PUT/PATCH/DELETE). These are the aforementioned classes and their supported operations:

- **Alarm Definitions API**:
  - **alarm-definition-create**: create an alarm definition;
  - **alarm-definition-delete**: delete the alarm definition;
  - **alarm-definition-list**: list alarm definitions (of a tenant);
  - **alarm-definition-patch**: patch the alarm definition;
  - **alarm-definition-show**: describe the alarm definition;
  - **alarm-definition-update**: update the alarm definition;
- **Alarms API**:
  - **alarm-count**: count alarms;
  - **alarm-delete**: delete the alarm;
  - **alarm-history**: alarm state transition history;
  - **alarm-history-list**: list alarms state history;
  - **alarm-list**: list alarms for this tenant;
  - **alarm-patch**: patch the alarm state;
  - **alarm-show**: describe the alarm;
  - **alarm-update**: update the alarm state;
- **Metrics API**:
  - **measurement-list**: list measurements for the specified metric;
  - **metric-create**: create metric;
  - **metric-create-raw**: create metric from raw JSON body;
  - **metric-list**: list metrics for this tenant;
  - **metric-name-list**: list names of metrics;
  - **metric-statistics**: list measurement statistics for the specified metric;
  - **dimension-name-list**: list names of metric dimensions:
  - **dimension-value-list**: list names of metric dimensions;
- **Notifications API**:
  - **notification-create**: create notification;

- **notification-delete**: delete notification;
- **notification-list**: list notifications (of a tenant);
- **notification-patch**: patch notification;
- **notification-show**: describe the notification;
- **notification-update**: update notification;

- **Notification Types API**:
  - **notification-type-list**: list notification types supported by Monasca.

Please notice that the operations of the *Notification API* do not create (list, delete, etc.) notifications *per se*. In other words, the operations of this API are the means (methods) to configure the Notification Engine, that when associated with alarm-definitions (thresholds), will make it publish the specified notifications.

6.1.4 Monasca Persister

The Monasca Persister [79] is the component responsible for consuming the metrics and alarms from the message bus and storing them in the metrics and alarms database (InfluxDB). Its basic design consists in a Kafka consumer that publishes to a LMAX Disruptor (a high performance inter-thread messaging library) [80] with output processors that use prepared batch statements to write to the database. The number of output processors/threads in the Monasca Persister can be configured, providing horizontal scalability and allowing it to process more messages.

6.1.5 Threshold Engine

The Monasca Threshold Engine [81] is the component responsible for computing thresholds on metrics and publish alarms to the message bus when they are exceeded. It is based on Apache Storm (a free and open distributed real-time computation system) and it is comprised of a single topology whose architecture is shown in figure 6.2.

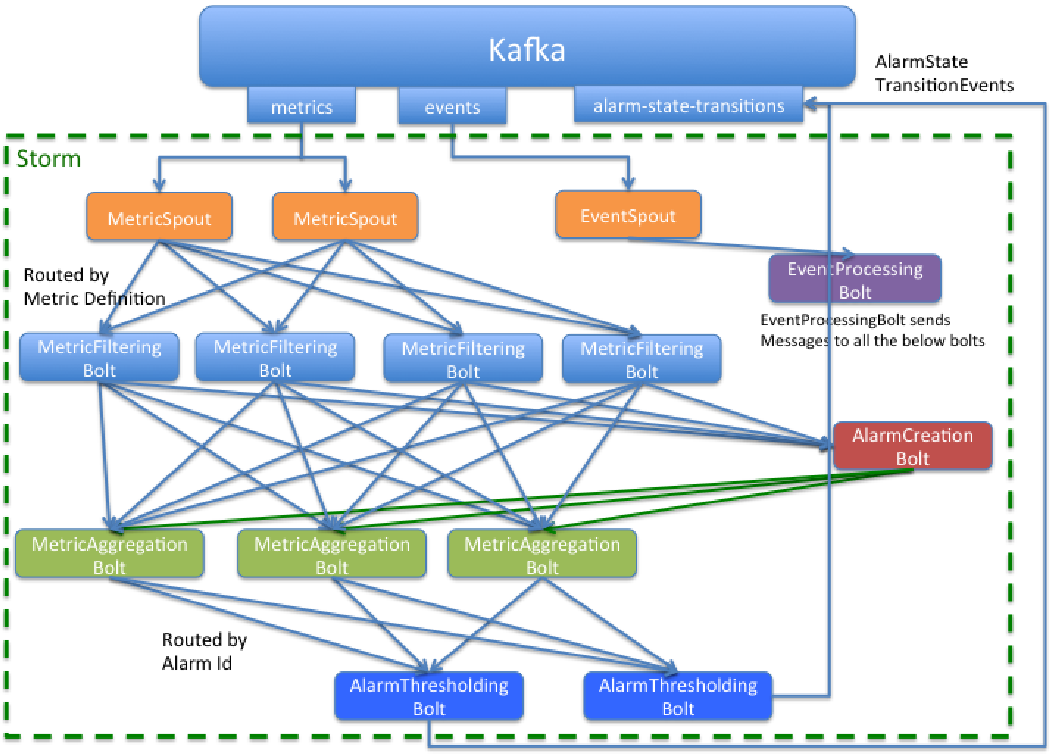


Figure 6.2: Monasca Threshold Architecture [81]

The three possible sates of the alarms are *OK*, *ALARM* and *UNDETERMINED* as already explained in section 5.1.1.3 (Alarm Notifications API). The alarms are determined by an expression that is defined as an alarm-definition (the threshold), in the Monasca nomenclature. A simple example of an expression used in an alarm-definition could be:

```
AVG(avg_pkt_count, 180) >= 10 AND AVG(communication_frequency, 180) >= 10
```

The alarm state transitions to ALARM if the (full) expression evaluates to true; it transitions to OK if it evaluates to false; and it transitions to UNDETERMINED if there are not any metrics for the two times of the measuring period. The expression is composed by sub-expressions and, for the example above, both of them would have to evaluate to true (generating two sub-alarms) in order to have an ALARM state transition.

The Threshold Engine is designed as a series of Storm Spouts that feed external data into the system as messages, and Storm Bolts that process incoming messages and optionally produce output messages for a downstream Bolt [82].

The flow of the metrics is MetricSpout to MetricFilteringBolt to MetricAggregationBolt with these spouts and bolts having the following responsabilities:

- **MetricSpout**: reads the metrics from the message bus and sends them on through Storm, routing them to a specific MetricFilteringBolt based on a routing algorithm that computes a hash code like value based on the metric definition, ensuring that a metric with the same definition is always routed to the same MetricFilteringBolt;
- **MetricFilteringBolt**: checks what alarm definitions a metric matches, if any; if it matches a new alarm-definition, this bolt first sends the metric to the AlarmCreationBolt (based on the alarm-definition ID), sending it next to the MetricAggregationBolts, once for each matching alarm definition, resorting to a combination of metric name and tenant ID to ensure that the same MetricAggregationBolt gets the metric each time;
- **AlarmCreationBolt**: looks at its incoming metrics and creates new alarms as needed, also adding the metric to an existing alarm if it fits there, and it forwards new sub-alarms to the MetricAggregationBolts when an alarm is created; this bolt processes many fewer metrics (the metrics routed through one of the MetricFilteringBolts) because few of them are associated with an alarm;
- **MetricAggregationBolt**: adds the metric information to its total for each sub-alarm, and uses the aggregated metrics to evaluate each sub-alarm once a minute; if the state changes on the sub-alarm, the state change is forwarded to the AlarmThresholdingBolts.
- **AlarmThresholdingBolt**: looks at the entire alarm expression to evaluate the state of the alarm and publishes the alarm-state transitions to the message bus;

6.1.6 Notification Engine

The Notification Engine [83] reads alarms from the message bus and publishes notifications following its configured notification methods. According to the documentation, it comprises two types of engines rather than just one: the *notification engine* and the *retry engine*.

The *notification engine* uses three Kafka topics and generates notifications following specific steps associated to Python classes:

- **Kafka Topics**:
  - **alarm_topic**: inbound alarms to the notification engine;
  - **notification_topic**: notifications successfully sent;
  - **notification_retry_topic**: unsuccessful notifications.
- **Notification Steps**:
  1. Read alarms from Kafka without auto-commit (*KafkaConsumer class*);
  2. Determine the notification type for an alarm (*AlarmProcessor class*);
  3. Send the notification (*NotificationProcessor class*);
  4. Successful notifications are added to the *notification_topic* (*NotificationEngine class*);

5. Failed notifications are added to the *notification_retry_topic* (*NotificationEngine class*);
6. Commit offset to Kafka (*KafkaConsumer class*).

The *retry engine* uses two Kafka topics and also generates notifications following specific steps associated to Python classes:

- **Kafka Topics**:
  - **notification_retry_topic**: notifications that need to be retried;
  - **notification_topic**: notifications successfully sent;
- **Notification Steps**:
  1. Reads notification JSON data from Kafka without auto-commit (*KafkaConsumer class*);
  2. Rebuild the notification that failed (*RetryEngine class*);
  3. Send the notification (*NotificationProcessor class*);
  4. Successful notifications are added to the *notification_topic* (*RetryEngine class*);
  5. Failed notifications that have not hit the retry limit are added back to the *notification_retry_topic* (*RetryEngine class*);
  6. Failed notifications that have hit the retry limit are discarded (*RetryEngine class*);
  7. Commit offset to Kafka (*KafkaConsumer class*).

6.1.7 Notification Engine Plugin

As stated in section 5.1.1.3 of chapter 5, the Notification Engine (natively) supports several types of notifications such as Email, Webooks and Pagerduty, which were originally developed and integrated in the Notification Engine as plugins [84].

Also, these notifications' types are essentially designed to notify human operators; however, they do not address the needs of the SELFNET project (notifying the Autonomic layer) and, with that in mind, a new plugin was developed to take the raw alarm notifications issued by the Notification Engine and republish them in the message bus respecting the RADM.

The native notification types of the Notification Engine are simple Python classes that extend an abstract class [85], taking the raw JSON notifications produced by the Notification Engine, processing them according to a specific format and then sending the content to their destinations, for instance an email inbox.

The SELFNET's plugin for the Notification Engine does just that. It takes the raw notifications, parses their contents and creates a new JSON structure that respects the RADM, and finally publishes that JSON structure to a Kafka topic with the name *monasca-notifications*.

Code-wise, the plugin is very simple and uses native Python libraries and a few lines of code. The exception is the *kafka-python* [86] library that was used to create a Kafka Producer, giving the plugin the ability to publish notifications to Kafka.

Once this plugin is added to the Notification Engine, it then shows up as an available *notification-type* in Monasca which can be used to create *notification-methods*, and these on their hand can be associated to *alarm-definitions*, becoming possible to send alarm notifications to the message bus to be consumed by the Analyzer and the Autonomic frameworks.

6.2 Monitoring Catalog

6.2.1 Architecture

The Monitoring Catalog (MC) is the single point of the SELFNET architecture where it is possible for the external components to create, read, update and delete aggregations and thresholding rules that are enforced on the Aggregation framework.

Its internal architecture is represented in figure 6.3, but in its most basic sense, it is comprised by a database where the catalog itself is stored, a RESTful API to interact with the service and a notification module to warn external service about the modifications or updates of the catalog.

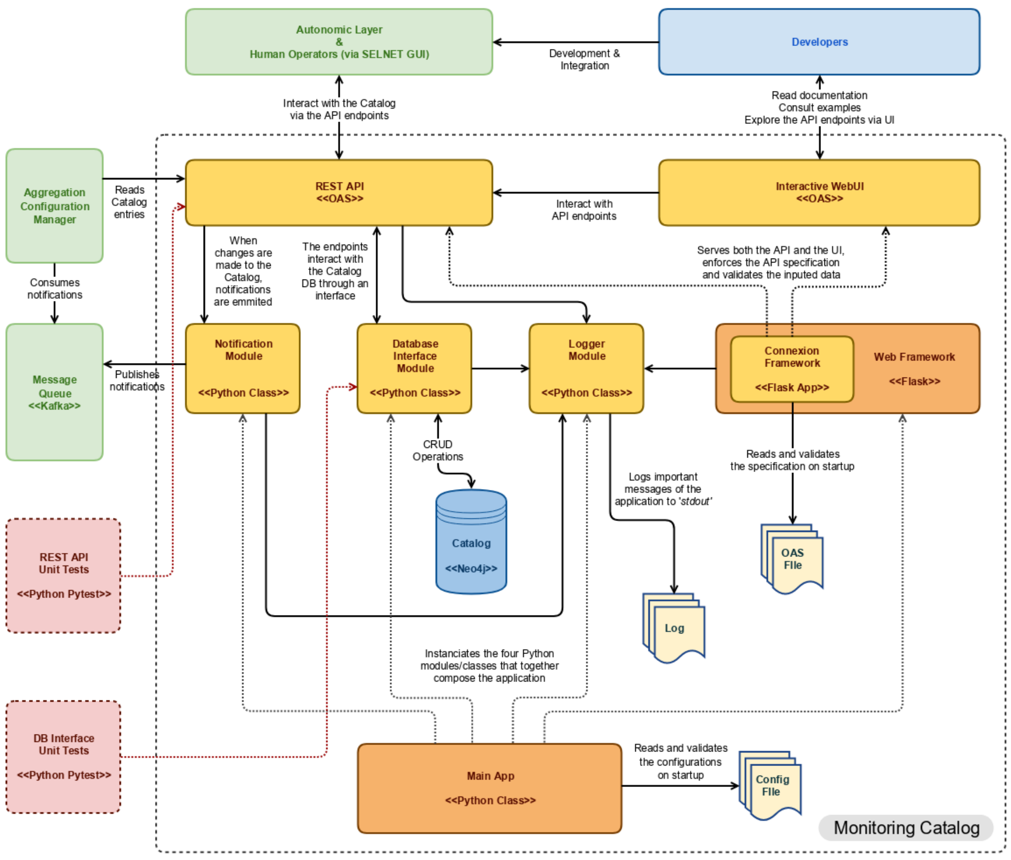

Autonomic Layer
&
Human Operators (via SELNET GUI)
Developers
Development & Integration
Interact with the Catalog via the API endpoints
Read documentation
Consult examples
Explore the API endpoints via UI
Aggregation Configuration Manager
Reads Catalog entries
REST API
<<OAS>>
Interact with API endpoints
Interactive WebUI
<<OAS>>
Consumes notifications
When changes are made to the Catalog, notifications are emmited
The endpoints interact with the Catalog DB through an interface
Serves both the API and the UI, enforces the API specification and validates the inputed data
Message Queue
<<Kafka>>
Publishes notifications
Notification Module
<<Python Class>>
Database Interface Module
<<Python Class>>
Logger Module
<<Python Class>>
Connexion Framework
<<Flask App>>
Web Framework
<<Flask>>
CRUD Operations
Catalog
<<Neo4j>>
Logs important messages of the application to 'stdout'
Reads and validates the specification on startup
OAS FIle
Log
REST API Unit Tests
<<Python Pytest>>
Instanciates the four Python modules/classes that together compose the application
DB Interface Unit Tests
<<Python Pytest>>
Main App
<<Python Class>>
Reads and validates the configurations on startup
Config FIle
Monitoring Catalog

Figure 6.3: Monitoring Catalog Architecture

- **Main App**: the class responsible for instantiating the four modules, classes or frameworks that together compose the Monitoring Catalog service - the Notification Module, the DB Interface, the Logger and the Web Framework; the service is started by running this single Python file; on the startup process a JSON configuration file is read, validated against a JSON schema and, if not misconfigured, the configurations are applied and the MC starts running;

- **Logger**: logs all the important messages of the application to the standard output, which can be easily piped with Linux commands to redirect the output (e.g. redirect to a log file in */var/log* and used together with *logrotate*); it uses the standard *logging* library module of Python [87], the logging levels are those supported by the library and the logging level can be configured in the main configuration file of the application;

- **Web Framework**: Flask [88] is used as the Web Framework, together with the Connexion Framework [89][90][91], which is a framework that handles HTTP requests based on the OpenAPI Specification (OAS) that will be discussed in section 6.2.2; the Web Framework serves both the RESTful API of the MC as well an interactive Web UI that contains rich documentation of the API endpoints, providing the developers the complete information they need to develop and integrate their services;

- **DB Interface Module**: creates an abstraction layer of the database to be used in the routing functions associated with the API endpoints;

- **Notification Module**: creates an abstractions layer, also to be used by the API routing functions, but this time around to handle the notifications that need to be published on the message bus whenever changes are done to the catalog;

- **Message Queue**: the notifications are published to a Kafka topic that is monitored by the Aggregation Configuration Manager (ACM);

- **Aggregation Configuration Manager**: consumes the notifications from the Kafka topic when they arrive, and based on them, it retrieves the newly created or modified rules from the catalog in order to enforce them on the Aggregation framework; when rules are deleted from the MC, the ACM simply removes them from the the Aggregation framework, not interacting with the MC API in this case;

- **Autonomic Layer & H. Operators**: The first approach with the onboarding of aggregation and thresholding rules is the manual one, i.e human operators will onboard these rules manually on the catalog through the SELFNET Web UI; the SELFNET system will then start aggregating metrics based on these rules up to a point that the Analyzer framework together with the Autonomic frameworks may start learning from the gathered information, leading to automated onboarding of new rules (see the SP-UC in section 4.3.3 of chapter 4);

- **REST API & Interactive WebUI**: the basic purpose of these components are more or less self-explanatory, but they will be further explored in section 6.2.2;

- **Unit Tests - REST API & DB Interface**: the Monitoring Catalog comprises a fair amount of endpoints and operations that can be done on the latter, making the manual testing and inspection to insure that every single one of them is operating as specified, prone to errors; with that in mind, two Python test modules (using Pytest [92]) were made to automate the process, either throughout the development course, as well as for debugging purposes, once the MC is deployed and running on the testbed; these testing files contain an extensive set of tests for all the supported endpoints and operations, comprising hundreds of code lines (over 900 each) and as such, only the output examples can be found in Appendix B to give a general idea of their operations and end results.

6.2.2 Catalog API

The Monitoring Catalog API is designed following the OpenAPI Specification (OAS) [93], formerly known as the Swagger specification [94], which is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful Web services created by a consortium of industry experts with an open governance structure under the Linux Foundation, the OpenAPI Initiative (OAI) [95].

As explained in the previous section, the Connexion framework is used, which allows the API specification to be written first (API first approach), and later on this framework it will handle the mappings of the endpoints to the routing functions, guaranteeing that it will work as specified and even before code is written.

Apart from automatically handling the endpoints mappings, it also has the following features:

- Validates requests and endpoint parameters automatically, based on the specification;
- Provides a WebUI so that the users of the API can have live documentation and even call the API's endpoints through it;
- Handles OAuth 2 token-based authentication;
- Supports API versioning;
- Supports automatic serialization of payloads; if the specification defines that an endpoint returns JSON, Connexion will automatically serialize the return value and set the right content type in the HTTP header.

The supported endpoints and operation were already listed in table 5.3 in chapter 5, and an excerpt of the YAML specification file can be found in Appendix A. The complete set of endpoints supported by the API are also represented by a graph in figure 6.4.

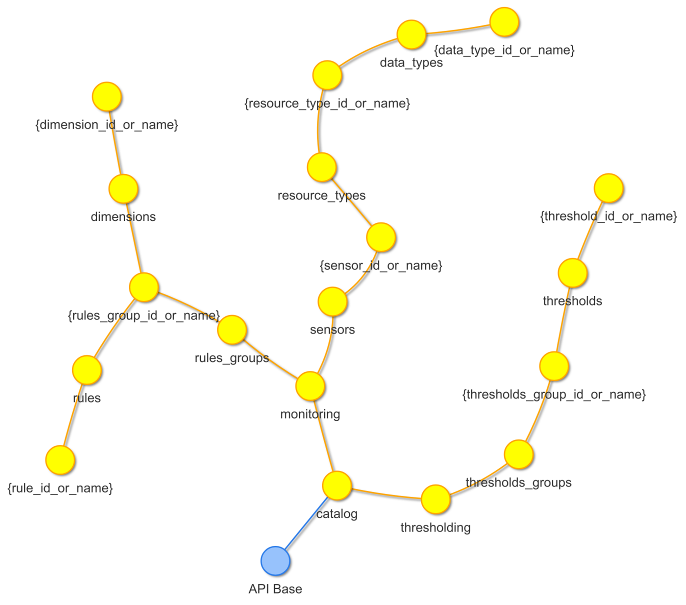
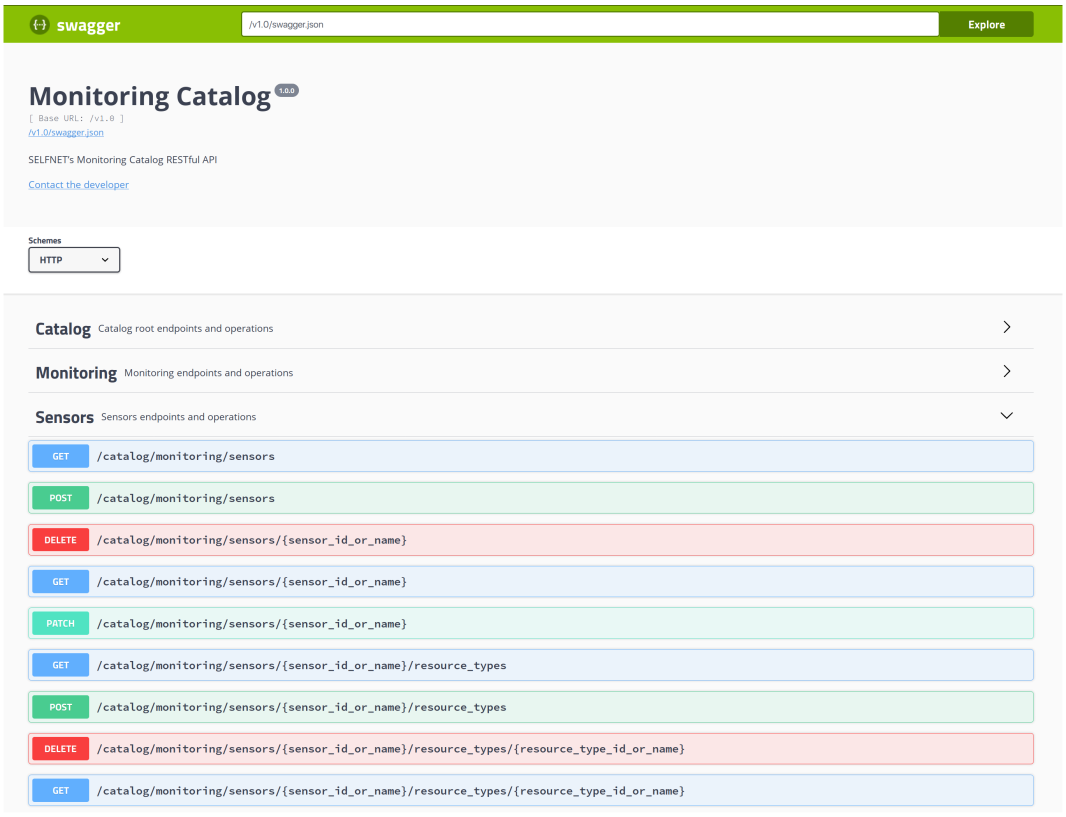


Figure 6.4: Graph of the Monitoring Catalog API's Endpoints



Monitoring Catalog 1.0.0

[Base URL: /v1.0]
/v1.0/swagger.json

SELFNET's Monitoring Catalog RESTful API

Contact the developer

Schemes
HTTP

Catalog Catalog root endpoints and operations

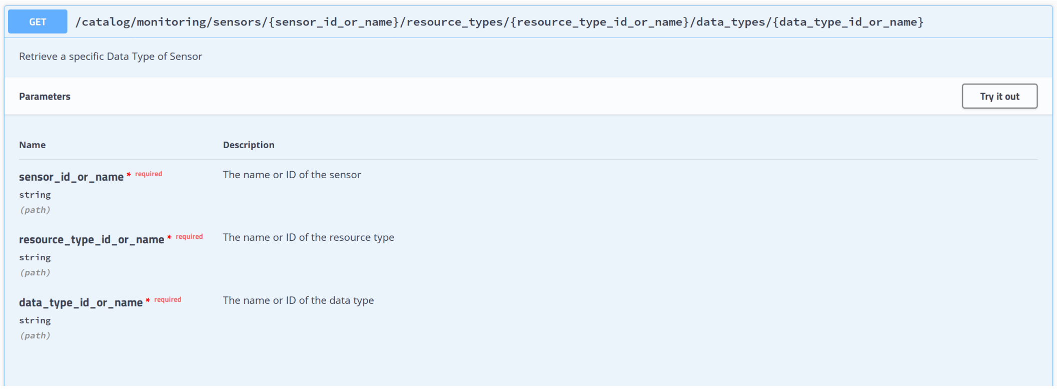
Monitoring Monitoring endpoints and operations

Sensors Sensors endpoints and operations

```
GET     /catalog/monitoring/sensors
POST    /catalog/monitoring/sensors
DELETE  /catalog/monitoring/sensors/{sensor_id_or_name}
GET     /catalog/monitoring/sensors/{sensor_id_or_name}
PATCH   /catalog/monitoring/sensors/{sensor_id_or_name}
GET     /catalog/monitoring/sensors/{sensor_id_or_name}/resource_types
POST    /catalog/monitoring/sensors/{sensor_id_or_name}/resource_types
DELETE  /catalog/monitoring/sensors/{sensor_id_or_name}/resource_types/{resource_type_id_or_name}
GET     /catalog/monitoring/sensors/{sensor_id_or_name}/resource_types/{resource_type_id_or_name}
```

Figure 6.5: Monitoring Catalog WebUI - General View

Figure 6.5 shows the MC API's WebUI that has been mentioned before, where only a small set of the endpoints and operations are shown, as the complete set cannot be fully displayed and analyzed here. Each section of the documentation can be expanded, with the Sensors one showing its content just to give the general idea of the organization of the documentation.



```
GET  /catalog/monitoring/sensors/{sensor_id_or_name}/resource_types/{resource_type_id_or_name}/data_types/{data_type_id_or_name}
```

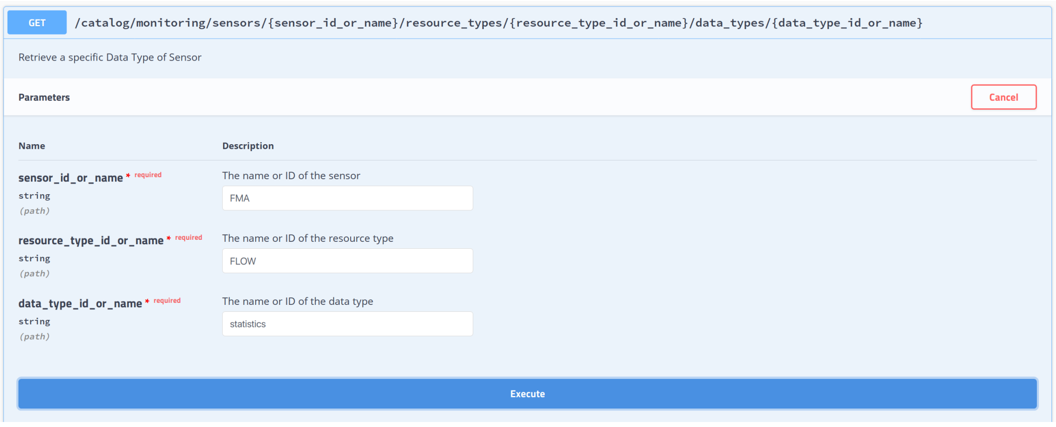
Retrieve a specific Data Type of Sensor

Parameters

Name	Description
sensor_id_or_name * required string *(path)*	The name or ID of the sensor
resource_type_id_or_name * required string *(path)*	The name or ID of the resource type
data_type_id_or_name * required string *(path)*	The name or ID of the data type

Figure 6.6: Monitoring Catalog WebUI - Sensor's Data Type GET Endpoint (Part 1)

Figure 6.6 shows the (partial) view of one of the endpoints' documentation if further expanded, in this case the endpoint that allows the retrieval of information of one of the Sensor's Data Type. The interface is interactive and if the 'Try it out' button is used, it will allow the user to test the endpoint filling up the required fields with the necessary information as shown in figure 6.7.



```
/catalog/monitoring/sensors/{sensor_id_or_name}/resource_types/{resource_type_id_or_name}/data_types/{data_type_id_or_name}
```

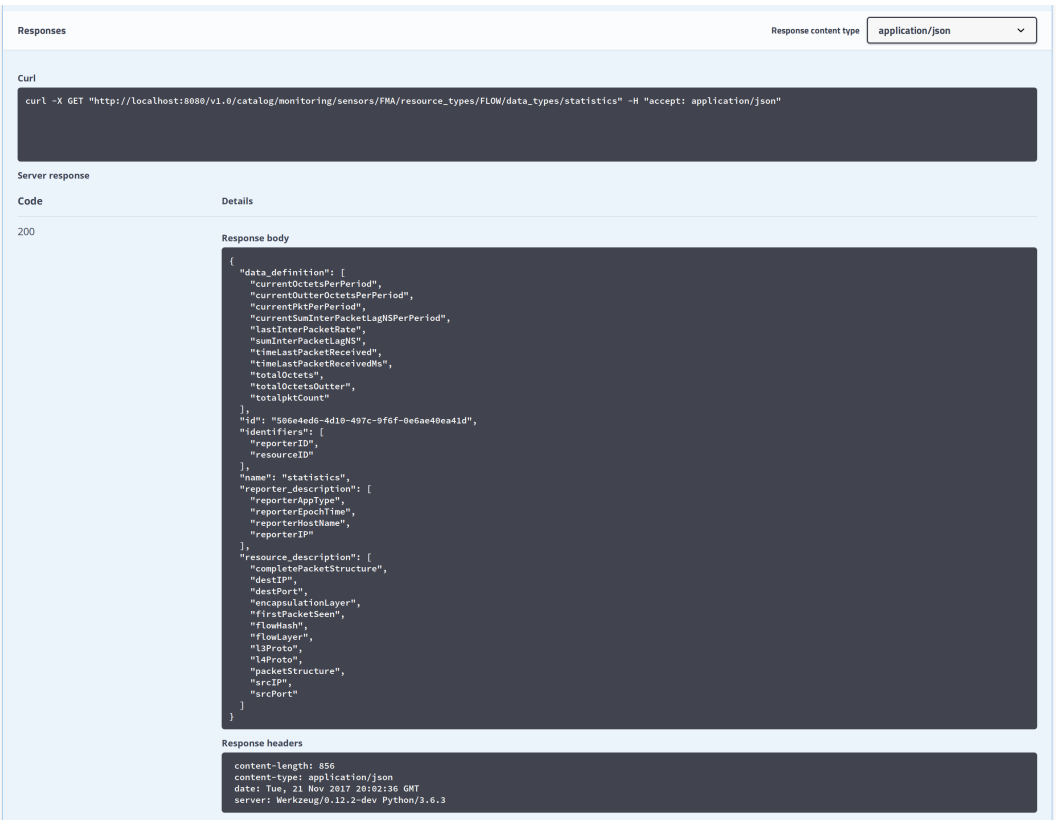
Retrieve a specific Data Type of Sensor

Parameters

Name	Description
sensor_id_or_name * required string *(path)*	The name or ID of the sensor FMA
resource_type_id_or_name * required string *(path)*	The name or ID of the resource type FLOW
data_type_id_or_name * required string *(path)*	The name or ID of the data type statistics

Figure 6.7: Monitoring Catalog WebUI - Sensor's Data Type GET Endpoint (Part 2)

Once the execution takes place, the result will be returned as shown in figure 6.8, and if the information can be retrieved based on the input data of course.



Curl

```
curl -X GET "http://localhost:8080/v1.0/catalog/monitoring/sensors/FMA/resource_types/FLOW/data_types/statistics" -H "accept: application/json"
```

Code	Details
200	Response body

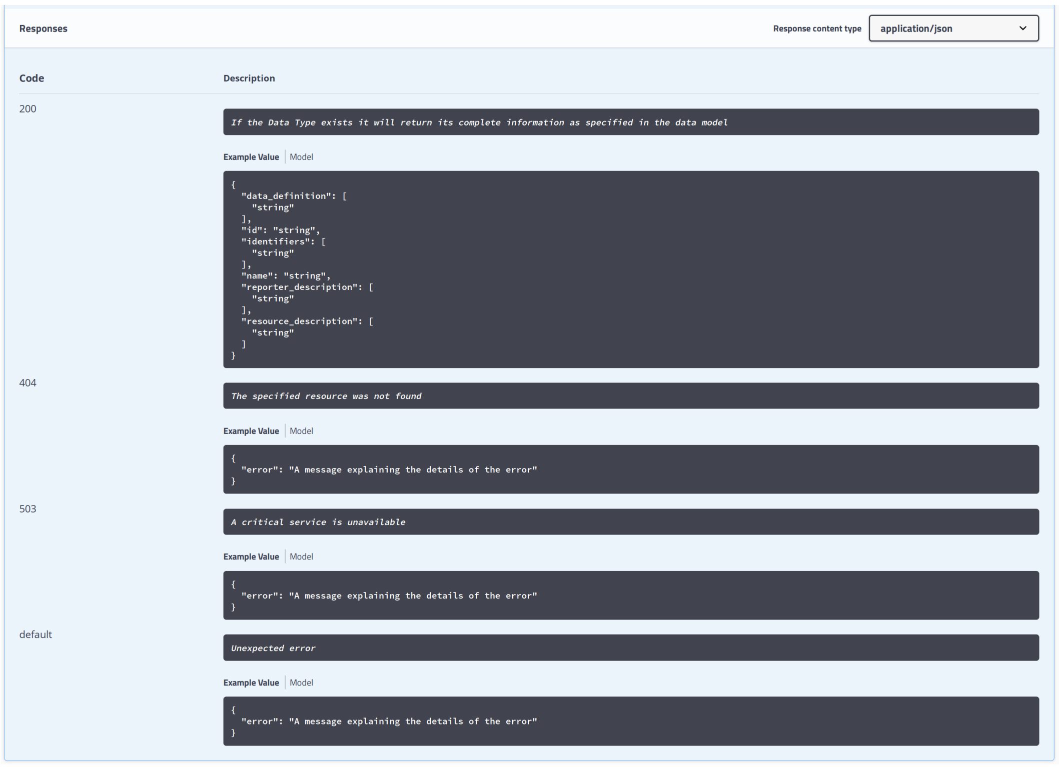
```
{
  "data_definition": [
    "currentOctetsPerPeriod",
    "currentOutterOctetsPerPeriod",
    "currentPktPerPeriod",
    "currentSumInterPacketLagNSPerPeriod",
    "lastInterPacketRate",
    "sumInterPacketLagNS",
    "timeLastPacketReceived",
    "timeLastPacketReceivedMs",
    "totalOctets",
    "totalOctetsOutter",
    "totalpktCount"
  ],
  "id": "506e4ed6-4d10-497c-9f6f-0e6ae40ea41d",
  "identifiers": [
    "reporterID",
    "resourceID"
  ],
  "name": "statistics",
  "reporter_description": [
    "reporterAppType",
    "reporterEpochTime",
    "reporterHostName",
    "reporterIP"
  ],
  "resource_description": [
    "completePacketStructure",
    "destIP",
    "destPort",
    "encapsulationLayer",
    "firstPacketSeen",
    "flowHash",
    "flowLayer",
    "l3Proto",
    "l4Proto",
    "packetStructure",
    "srcIP",
    "srcPort"
  ]
}
```

Response headers

```
content-length: 856
content-type: application/json
date: Tue, 21 Nov 2017 20:02:36 GMT
server: Werkzeug/0.12.2-dev Python/3.6.3
```

Figure 6.8: Monitoring Catalog WebUI - Sensor's Data Type GET Endpoint (Part 3)

At the end of the documentation of the endpoint, the possible responses and their respective examples are shown, so the user can understand which HTTP codes to expect and how to take measures to deal with them as well their contents, as shown in figure 6.9.


Responses

Response content type: application/json

Code	Description
200	*If the Data Type exists it will return its complete information as specified in the data model*
404	*The specified resource was not found*
503	*A critical service is unavailable*
default	*Unexpected error*

200 — Example Value | Model

```
{
  "data_definition": [
    "string"
  ],
  "id": "string",
  "identifiers": [
    "string"
  ],
  "name": "string",
  "reporter_description": [
    "string"
  ],
  "resource_description": [
    "string"
  ]
}
```

404 — Example Value | Model

```
{
  "error": "A message explaining the details of the error"
}
```

503 — Example Value | Model

```
{
  "error": "A message explaining the details of the error"
}
```

default — Example Value | Model

```
{
  "error": "A message explaining the details of the error"
}
```

Figure 6.9: Monitoring Catalog WebUI - Sensor's Data Type GET Endpoint (Part 4)

The previous figures and brief descriptions of the Monitoring Calatog's WebUI should be enough to understand how its content is organized. The remaining endpoints documentation and usage follow the logic of the one that was used as an example. Although the HTTP GET operation was the one used in the example for its simplicity, rest assured the remaining supported operations (POST, PACTH and DELETE) follow the same logic.

However, some differences should be noticed. The PATCH and POST operations, beside the ID's/Names of specific resources that should be given in the endpoint's URL path, they also require information to be sent in the HTTP body. This makes sense and it should be intuitive, since the POST operations aim to create resources, implying that the information of the new resource that will be created must be sent. For instance, to add a new Data Type to a sensor, the Sensors' ID or name and the Sensor's Resource Type ID or name need to be present in the endpoint's URL path and the information of Data Type in the body, with the response body's JSON code shown in figure 6.8, as a perfect example of what should it be. The PATCH operation aims to modify a resource, and in this case, the difference to the POST body is that only the fields that need to be modified/added should be sent in the HTTP body.

It should be noticeable by now, but it is worth mentioning that all the responses of the API are in JSON format, as well the HTTP body content that must sent in the POST and PATCH operations. All the data inputted in the endpoints must comply with the specification, otherwise the API will throw the required HTTP errors with codes and messages specified in the design.

It is also worth mentioning that the JSON code sent in the HTTP body is parsed and validated automatically by the Connexion framework as per the specification, taking out this burden from the developer, although in some particular cases additional code must be added to complement what cannot be validated by the specification alone. The WebUI is also taken care by this framework, with all its content fully and automatically generated based solely on the specification file.

6.2.3 Storage

Neo4j is used as the database to store the Monitoring Catalog. It is a Graph Database that is able to represent the catalog. Before moving ahead, it should be mentioned that it was agreed in internal meetings of the Altice Labs team participating in the SELFNET project, that the catalog would not required versioning, i.e there is only a single version of the catalog present in the database. In fact, the catalog can be considered the database itself.

The graph database models the data in the form of a graph, which is somewhat obvious. Compared to the traditional relational databases, the analogy represented in 6.1 can be made:

RBDMS	**Graph Database**
Tables	Graphs
Rows	Nodes
Columns and data	Properties and their values
Constraints	Relationships
Joins	Traversal

Table 6.1: RDBM vs Graph DB

With this analogy in mind, it will be easier to understand the upcoming dissection of the catalog representation on the database. Also, the examples that will follow will take the Monitoring Catalog Data Model (MCDM) example available in Appendix C.2 as a reference.

We will start with the root nodes of the catalog shown in figure 6.10.

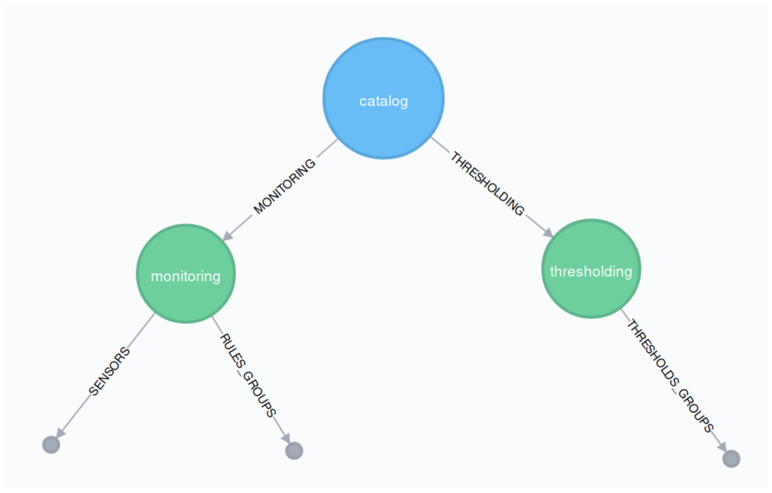


Figure 6.10: Monitoring Catalog DB - Catalog's Root

The whole content of the MCDM example is the catalog, and it is subdivided in two major sections, the *thresholding* section that contains the *thresholds_groups* subsection and the *monitoring* section which holds both the *sensors* and the *rules_group* subsections. Figure 6.10 represents this basic structure of the catalog which is for all means, an empty catalog.

The mapping between the MCDM example and the graph representation is very straightforward, where the nodes have the same name as their counterparts in the JSON format of the example. The relations between the nodes (the edges) are named before the nodes they are connected to, in order to be easier to remember them and to facilitate the queries on the database.

Both the nodes and the relationship can have properties and values of basics types (strings, numbers, and lists). For the particular case of the Monitoring Catalog, none of the relationships have properties associated as they were not required. The nodes however do, as they hold the information of the catalog that can be represented in these basic types (e.g. the *id*, *name*, *description* and so on). More complex information of the catalog must be represented with further nodes and relationships that can use the basic types to represent the remaining information.

Now it is considered in figure 6.11 the representation of the sensors in the database.

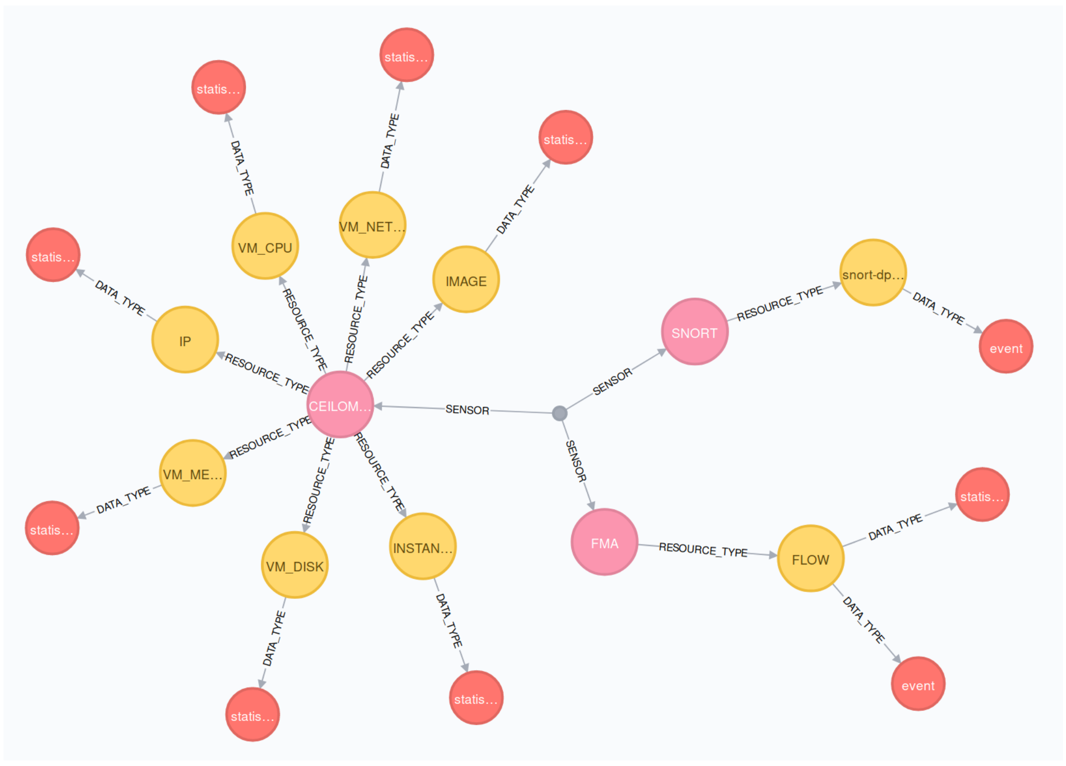
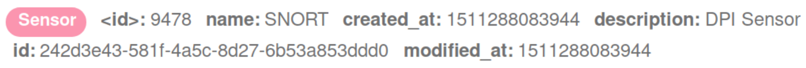


Figure 6.11: Monitoring Catalog DB - Sensors

Considering the *sensors* field in the MCDM example, it is observed that it is a list of objects, which is not of a basic type. The same happens with the *resource_type* fields of each sensor and with the *data_type* field of each resource type. This means that they must be represented with further nodes and relationships, as seen in figure 6.11. The sensors are represented in pink, the resource types in yellow and the data types in red.

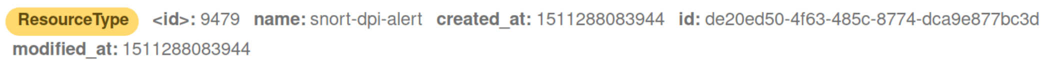
The relationships naming follow the same convention explained before, and they are responsible for stipulating which data types belong to which resource types, and which resource types belong to which sensors. At last but no the least, the three sensors are connected to a single node (the small gray one) representing the list of sensors shown in the MCDM example.

That leaves us with the basic types that can be represented directly on each node. The following figures will show the properties and values of the three nodes that together represent the SNORT sensor as an example.



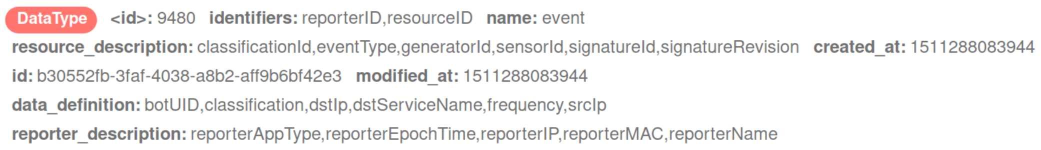
Sensor **<id>:** 9478 **name:** SNORT **created_at:** 1511288083944 **description:** DPI Sensor
id: 242d3e43-581f-4a5c-8d27-6b53a853ddd0 **modified_at:** 1511288083944

Figure 6.12: Monitoring Catalog DB - SNORT Sensor Properties



ResourceType **<id>:** 9479 **name:** snort-dpi-alert **created_at:** 1511288083944 **id:** de20ed50-4f63-485c-8774-dca9e877bc3d
modified_at: 1511288083944

Figure 6.13: Monitoring Catalog DB - SNORT Resource Type Properties



DataType **<id>:** 9480 **identifiers:** reporterID,resourceID **name:** event
resource_description: classificationId,eventType,generatorId,sensorId,signatureId,signatureRevision **created_at:** 1511288083944
id: b30552fb-3faf-4038-a8b2-aff9b6bf42e3 **modified_at:** 1511288083944
data_definition: botUID,classification,dstIp,dstServiceName,frequency,srcIp
reporter_description: reporterAppType,reporterEpochTime,reporterIP,reporterMAC,reporterName

Figure 6.14: Monitoring Catalog DB - SNORT Data Type Properties

As per the MCDM, the sensor must have a name and a description which are represented as properties of the node as seen in figure 6.12, the resource types of the sensors must have a name, also represented as a property of the node and show in figure 6.13, and the data types of the resource types must have a name, identifiers, a resource description, a data definition and a reporter description, also shown in figure 6.14.

It is also observed that each node has two different types of IDs. The *<id>* is an internal property of Neo4j that is automatically given to each node, which is reusable and consequently cannot be used reliably. Has many of the objects of the MCDM required an ID to be used, for this purpose each node of the catalog on the database also has an extra ID, the *id* property which is an UUID generated when the node is added to the database, and this one is immutable and unique.

Next the *rules_groups* section of the MCDM will be shown in figure 6.11, which per the example given, it only comprises one rules group.

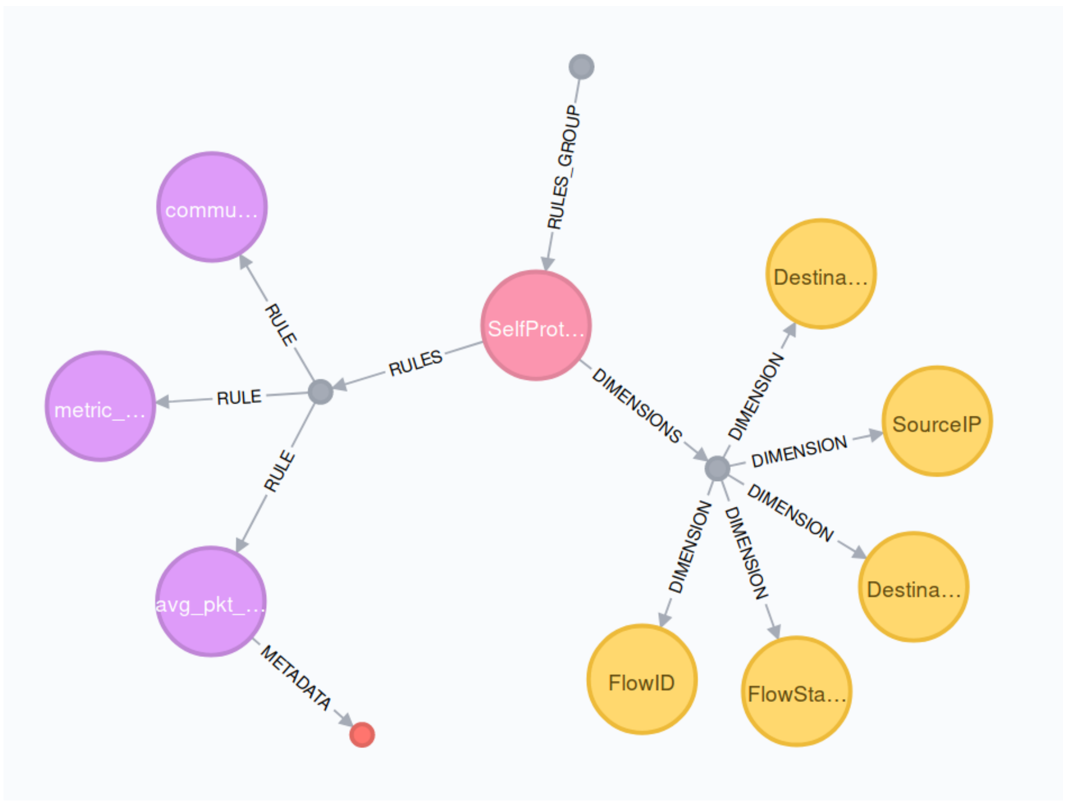


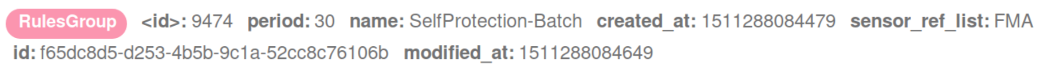
Figure 6.15: Monitoring Catalog DB - Rules Groups

This section of the MCDM is also a list of objects (multiple rules groups) and also, as specified by the model, each rules group should contain two particular lists, the *dimensions* and the *rules* lists, and the rules themselves can contain a *metadata* field composed by primitive key/values pairs of extra information. All the above cannot be represented by basic type properties, thus the representation contains multiple nodes and relationships.

The analogy here is the same one explained for the sensors example. The pink nodes represent the rules groups, the yellow nodes the dimensions, the purple nodes the aggregation rules, and the red node the metadata associated with the aggregations' rules.

The top most small gray node is used to connect together all the rules groups (the list of rules groups, and there is only one in this case), and the other two connect together the rules and the dimensions (the list of rules and the list of dimensions). These two last nodes are not required to represent the data, but they keep things organized and more importantly, they facilitate the queries on the database.

Just like it happened with the sensors example, the only thing missing is the properties of the nodes. Those will be shown in figures 6.16, 6.17, 6.18 and 6.19 using the only rules group node, the only metadata node, and one of the rules and dimensions nodes as examples.



RulesGroup **<id>:** 9474 **period:** 30 **name:** SelfProtection-Batch **created_at:** 1511288084479 **sensor_ref_list:** FMA **id:** f65dc8d5-d253-4b5b-9c1a-52cc8c76106b **modified_at:** 1511288084649

Figure 6.16: Monitoring Catalog DB - Rules Group Properties

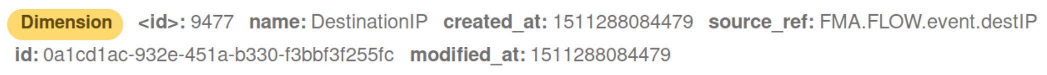

Dimension <id>: 9477 **name:** DestinationIP **created_at:** 1511288084479 **source_ref:** FMA.FLOW.event.destIP
id: 0a1cd1ac-932e-451a-b330-f3bbf3f255fc **modified_at:** 1511288084479

Figure 6.17: Monitoring Catalog DB - Dimension Properties

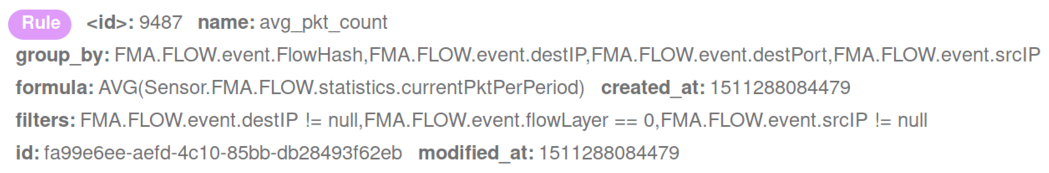

Rule <id>: 9487 **name:** avg_pkt_count
group_by: FMA.FLOW.event.FlowHash,FMA.FLOW.event.destIP,FMA.FLOW.event.destPort,FMA.FLOW.event.srcIP
formula: AVG(Sensor.FMA.FLOW.statistics.currentPktPerPeriod) **created_at:** 1511288084479
filters: FMA.FLOW.event.destIP != null,FMA.FLOW.event.flowLayer == 0,FMA.FLOW.event.srcIP != null
id: fa99e6ee-aefd-4c10-85bb-db28493f62eb **modified_at:** 1511288084479

Figure 6.18: Monitoring Catalog DB - Rule Properties

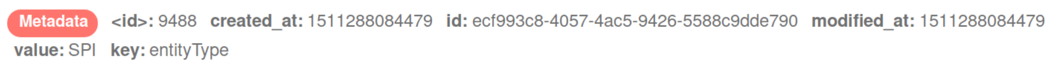

Metadata <id>: 9488 **created_at:** 1511288084479 **id:** ecf993c8-4057-4ac5-9426-5588c9dde790 **modified_at:** 1511288084479
value: SPI **key:** entityType

Figure 6.19: Monitoring Catalog DB - Rule Metadata Properties

As per the MCDM, the rules group must have a *name* and a *source_ref_list* which are represented as properties of the node as seen in figure 6.16, the dimensions of the rules group must have a *name* and a *source_ref*, also represented as properties of the node and shown in figure 6.17, the rules of the rules group must have a *name*, a *group_by*, a *formula*, *filters*, all of them represented as properties of the node as shown in figure 6.18, and it also can have *metadata* represented by nodes related with the latter where the key/value pairs are represented with properties as shown in figure 6.19.

Finally, the only remaining section of the MCDM is the *threshold_groups*. The example given in the appendix only contains one of these groups, which is represented in the database as shown in figure 6.20.

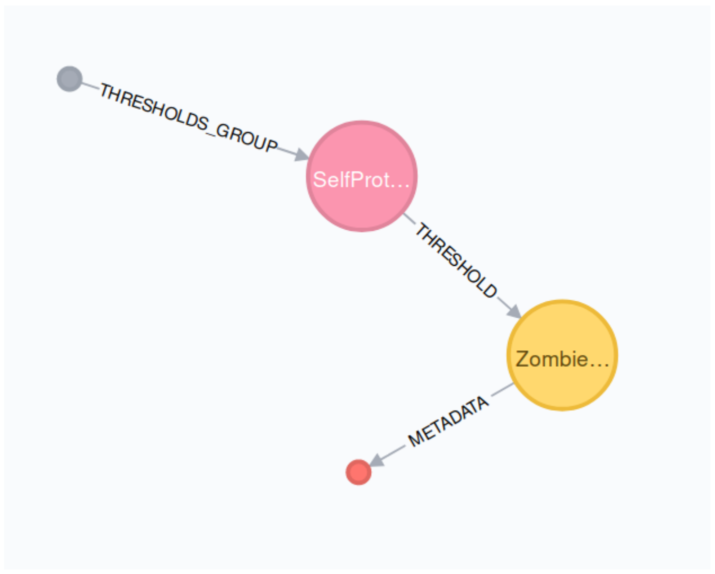


Figure 6.20: Monitoring Catalog DB - Threshold Groups

As in the other main sections of the MCDM, this one is also a list of objects (multiple threshold groups), and as specified by the model, each threshold group may contain several thresholds, which are objects themselves. Apart from the metadata, the inner fields of each threshold in the list can be easily represented by basic type properties.

The pink nodes represent the thresholds groups, the yellow nodes the thresholds, and the red nodes the metadata associated with the thresholds. The convention used for the naming of the relations is the same one explained in the previous examples.

The small gray node is used to connect together all the threshold groups (the list of threshold groups). In this case there is only one threshold group and also only one threshold rule. Once again, the only thing missing is the properties of the nodes, which will be shown in figures 6.21, 6.22 and 6.23.

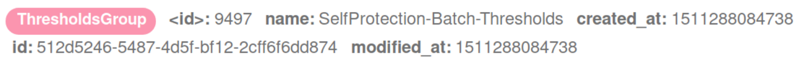

ThresholdsGroup <id>: 9497 name: SelfProtection-Batch-Thresholds created_at: 1511288084738
id: 512d5246-5487-4d5f-bf12-2cff6f6dd874 modified_at: 1511288084738

Figure 6.21: Monitoring Catalog DB - Thresholds Group Properties

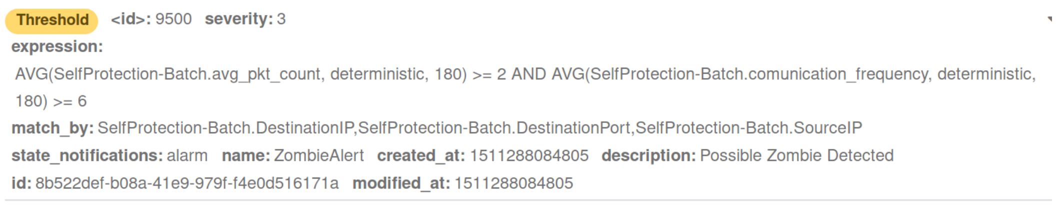

Threshold <id>: 9500 severity: 3
expression:
AVG(SelfProtection-Batch.avg_pkt_count, deterministic, 180) >= 2 AND AVG(SelfProtection-Batch.comunication_frequency, deterministic, 180) >= 6
match_by: SelfProtection-Batch.DestinationIP,SelfProtection-Batch.DestinationPort,SelfProtection-Batch.SourceIP
state_notifications: alarm name: ZombieAlert created_at: 1511288084805 description: Possible Zombie Detected
id: 8b522def-b08a-41e9-979f-f4e0d516171a modified_at: 1511288084805

Figure 6.22: Monitoring Catalog DB - Threshold Properties

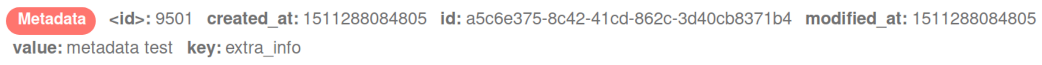

Metadata <id>: 9501 created_at: 1511288084805 id: a5c6e375-8c42-41cd-862c-3d40cb8371b4 modified_at: 1511288084805
value: metadata test key: extra_info

Figure 6.23: Monitoring Catalog DB - Threshold Metadata Properties

The MCDM determines that each threshold group should have a *name*, which is represented as a property of the node as seen in figure 6.21. The thresholds must have a *name*, a *description*, a *severity* level, the *state_notifications*, the *match_by* dimensions and an *expression*, all of them represented as properties of the node as shown in figure 6.22, and it also can have *metadata* represented by nodes related with the latter where the key/value pairs are represented with properties as shown in figure 6.23.

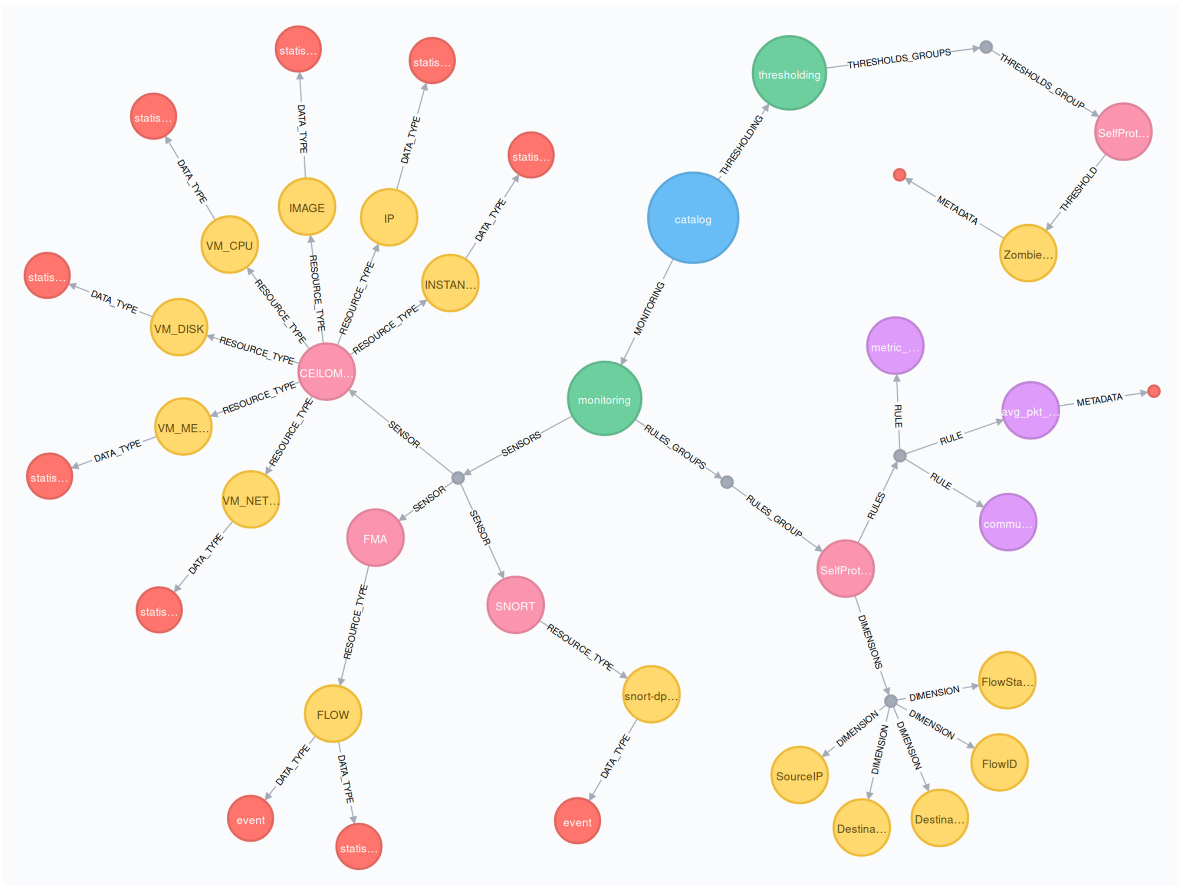


Figure 6.24: Monitoring Catalog DB - Full Graph Representation

After having described how the MCDM example provided in Appendix C.2 is represented in the database section by section, figure 6.24 shows how everything is connected together, where the full content of the example is represented in the database as the complete catalog.

6.2.4 Catalog Notifications

As stated in section 6.2.1, the Monitoring Catalog contains a notification module that creates an abstraction layer to handle the notifications that need to be published on the message bus whenever changes are done to the catalog. This mechanism must exist in order to make the Aggregation Configuration Manager aware of these changes, so it can enforce them on the Aggregation framework.

The MCDM defines three major groups in the catalog: the *sensors* "group", the *rules_-groups* and the *thresholds_groups*. Only the changes made in the last two are relevant in this context, as the *sensors* one is only used internally on the catalog as a reference for the formulas/expressions used in the thresholding or aggregation rules.

The messages are published to a Kafka topic (mcat-notifications) in JSON format containing five basic fields:

- **timestamp**: the UTC time when the change on the catalog was made;

- **operation**: the type of operation made on the catalog - *CREATE*, *UPDATE* or *DELETE*;

- **type**: the catalog's group type (*rules_group* or *thresholds_group*) to which the modification belongs to;

- **type_id**: the UUID of the group's entry on the catalog where the modification was made;

- **period**: if it is a *rules_group* type, its period of aggregation must also be sent, otherwise it will be empty (*null*).

```
{
  "timestamp": "2017-12-03T21:48:07.345+0000",
  "operation": "CREATE",
  "type": "thresholds_group",
  "type_id":
   ↪   "d231e6be-5aba-44fe-a357-6b80c747b786",
  "period": null
}
```

```
{
  "timestamp": "2017-12-03T21:48:08.501+0000",
  "operation": "UPDATE",
  "type": "rules_group",
  "type_id":
   ↪   "465aab26-78d1-46a2-8d1a-34795cb6c779",
  "period": 30
}
```

Table 6.2: Monitoring Catalog - Notification Examples

For better clarity, two examples are shown in table 6.2. The content of these messages comprises all the required information for the Aggregation Configuration Manager starting process of the rules enforcement on the Aggregation framework, as we will see in the next section.

6.3 Aggregation Configuration Manager

6.3.1 Architecture

The Aggregation Configuration Manager (ACM) is the component responsible for enforcing the on-boarded rules on the Monitoring Catalog throughout the Aggregation framework. Specifically, the batch aggregation rules are enforced on the Aggregation Engine, the stream (realtime) aggregation rules are enforced on the CEP Engine, and the threshold rules are enforced on Monasca, where the aggregated metrics are stored and where the Threshold Engine is located.

Its internal architecture is represented in figure 6.25, and it follows a master/slave or master/worker model where Zookeeper is used to coordinate the services of the Aggregation Configuration Manager.

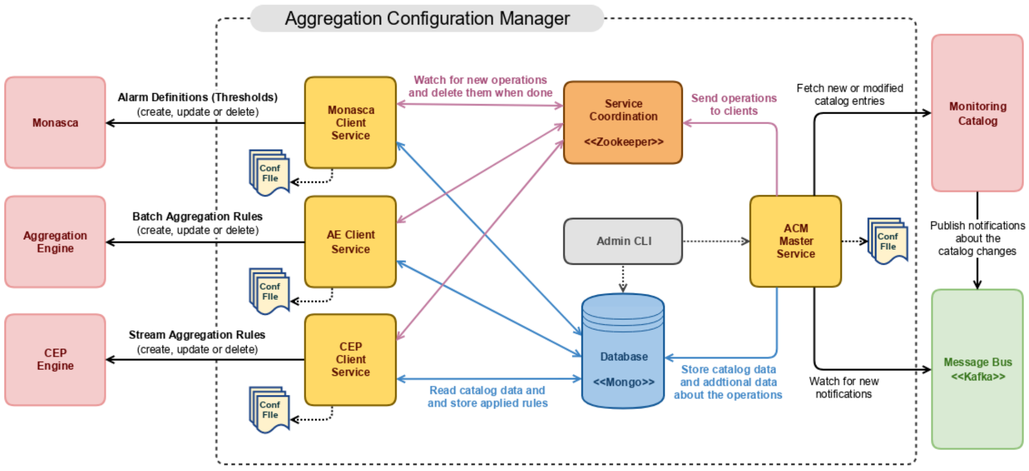

Aggregation Configuration Manager
Monasca
Aggregation Engine
CEP Engine
Alarm Definitions (Thresholds) (create, update or delete)
Batch Aggregation Rules (create, update or delete)
Stream Aggregation Rules (create, update or delete)
Monasca Client Service
AE Client Service
CEP Client Service
Conf File
Watch for new operations and delete them when done
Service Coordination <<Zookeeper>>
Send operations to clients
Admin CLI
Database <<Mongo>>
Read catalog data and and store applied rules
Store catalog data and addtional data about the operations
ACM Master Service
Fetch new or modified catalog entries
Watch for new notifications
Monitoring Catalog
Publish notifications about the catalog changes
Message Bus <<Kafka>>

Figure 6.25: Aggregation Configuration Manager Architecture

6.3.2 Service Coordination

Zookeeper is used to coordinate the tasks given by the ACM's master service to the service clients. The tasks in this context are directly associated to the operations that were made on the Monitoring Catalog (*CREATE*, *UPDATE* or *DELETE*), and figure 6.26 illustrates the structure of the *znodes* that was used for this purpose.

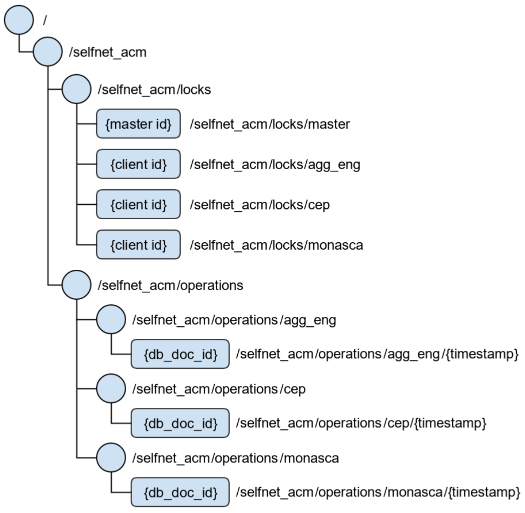

/
/selfnet_acm
/selfnet_acm/locks
{master id} /selfnet_acm/locks/master
{client id} /selfnet_acm/locks/agg_eng
{client id} /selfnet_acm/locks/cep
{client id} /selfnet_acm/locks/monasca
/selfnet_acm/operations
/selfnet_acm/operations/agg_eng
{db_doc_id} /selfnet_acm/operations/agg_eng/{timestamp}
/selfnet_acm/operations/cep
{db_doc_id} /selfnet_acm/operations/cep/{timestamp}
/selfnet_acm/operations/monasca
{db_doc_id} /selfnet_acm/operations/monasca/{timestamp}

Figure 6.26: Aggregation Configuration Manager - Zookeeper Data Tree

The complete structure is kept inside a root znode, the */selfnet_acm*. This znode is further arranged in two inner znodes, the */selfnet_acm/locks* and the */selfnet_acm/operations*.

The */selfnet_acm/locks* znodes are used by the master and client services to signal that they are up and running, connected to Zookeeper and ready to do their jobs. They are ephemeral znodes, meaning that whenever those services are terminated or stop responding, Zookeeper will remove them. Only one instance of each service is allowed to be running at a time and this is ensured by these znodes, with possible backup services of each one ready to take the place of the primary ones should any of them go offline.

The */selfnet_acm/operations* znodes is where the master service will place the tasks of each client service. Each client service has its own znode, and the content of these znodes (the value) are the document IDs on the database where they should retrieve the information necessary for them to further proceed with their tasks. Each one of these client services znodes have a *timestamp* used as their names (the key), allowing the clients to process their tasks in the same order as they were made in the Monitoring Catalog.

6.3.3 Database

MongoDB was used as a database, with three distinct collections of documents, one for each client service. The *agg_eng* collection for the Aggregation Engine client, the *cep* collection for the CEP Engine client and the *monasca* collection for the Monasca client.

Despite each client having its own collection of documents, the documents stored on the three collections are always of two types and they follow the same structure as explained ahead:

- **Operation Documents**: these documents are created by the master service upon receiving a notification message from the Monitoring Catalog, storing the necessary information for the clients to perform their tasks; once the clients have fulfilled the task, their are removed from the database:
  - **_id**: the internal document ID on the database;
  - **type**: in this case the value is always 'OPERATION_DATA' to help the client distinguish between the two different document types if and when necessary;
  - **timestamp**: the same timestamp used in the notification message sent from the Monitoring Catalog to the message bus;
  - **operation**: the same operation specified in the notification message sent from the Monitoring Catalog to the message bus, i.e CREATE, UPDATE or DELETE;
  - **catalog_endpoint**: the catalog's endpoint where the information was retrieved, which is determined by the Master service based on the values of *type* and *type_id* fields of the message sent from the Monitoring Catalog to the message bus;
  - **catalog_data**: the catalog's JSON data retrieved from the Monitoring Catalog when the operations are CREATE or UPDATE, empty (null) otherwise, i.e on a DELETE operation.
- **Status Documents**:
  - **_id**: the internal document ID on the database;
  - **type**: in this case the value is always 'STATUS_DATA' to help the client distinguish between the two different document types if and when necessary;
  - **created_at**: the same timestamp used in the notification message sent from the Monitoring Catalog to the message bus and when the operation is of type 'CREATE' (copied from the *timestamp* field of the operation document);

– **modified_at**: the same timestamp used in the notification message sent from the Monitoring Catalog to the message bus, and when the operation is of type 'UPDATE' (copied from the *timestamp* field of the operation document);
– **status**: 'OK' if the task was successfully processed 'ERROR' otherwise;
– **catalog_endpoint**: the catalog's endpoint where the information was retrieved, copied from the *catalog_endpoint* field of the operation document; it is used on the update and delete operations to find the correct document;
– **catalog_data**: a copy of the *catalog_data* field of the operation document if it is a create operation; if it is an update operation, the content will be updated based on the content of the same field of the operation document;
– **client_data**: the data sent in response by the end service where the rules of the catalog are enforced, i.e. the response sent by Monasca, the Aggregation Engine or the CEP Engine.

6.3.4 Master Service

The Master Service is responsible for watching any incoming notifications sent by the Monitoring Catalog, and instructing the Client Services to apply the changes made on the catalog to their respective end clients or services.

During its start up, the Master Service reads from its configuration file all the information required for it to interact with the four services it communicates with directly: the Zookeeper, the Mongo Database, Kafka and of course the Monitoring Catalog.

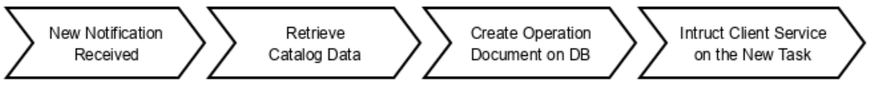

New Notification Received
Retrieve Catalog Data
Create Operation Document on DB
Intruct Client Service on the New Task

Figure 6.27: Aggregation Configuration Manager - Master Service Steps

Figure 6.27 summarizes the steps taken by the Master Service upon receiving a notification from the Monitoring Catalog. For each message received, the Master Service will look to its data fields (see section 6.2.4) to determine in first place the catalog's data type and ID, so it can retrieve the correspondent entry data from the Monitoring Catalog if necessary, i.e if it is a CREATE or UPDATE operation.

Also based on the catalog's data type and the period stated on the message, it will determine on which collection of the database it should put the operation document, as well as the analogous znode of Zookeeper to instruct the respect service client.

For instance, if the message states that the type is *thresholds_group*, it will create an operation document on the *monasca* collection of the database, containing the timestamp and operation retrieved from the notification message, the catalog's endpoint it used to retrieve the respective catalog data, the catalog data itself and with the document type of OPERATION_DATA.

For this same example, after the document is created on the database, it will create the znode */selfnet_acm/operations/monasca/{timestamp}* with the ID of the document as its value, which will, in a single step, notify the Monasca Client that it has a new task and which document it should retrieve in order to gather the necessary information to proceed with its task.

If the message stated that the type was *rules_group* instead, the process would be similar but with a catch. The period specified in the notification message is required in this case in order to determine if the task should be handled by the CEP Engine client or the Aggregation Engine client.

This happens for two reasons. The first one is that, on internal meetings of Altice Labs, it was decided that the ACM should be the component responsible for determining if an aggregation rule (or a group of them) should be considered a batch aggregation rule or a realtime aggregation rule base on the period of aggregation (e.g. if the aggregation period is less than 5 seconds it is considered realtime).

The second one is that, although for the CREATE and UPDATE operations the period could be retrieved directly from the catalog, that would be impossible on a DELETE operation as the rule (or group of) would not exist on the catalog, thus the period must be retained before the rule is actually deleted from the catalog and be sent on the notification message.

6.3.5 Client Services

Although there are three distinct types of Client Services, they all follow the same steps to handle their tasks. Where they differ is how they transpose the rules from the Monitoring Catalog to their respective end client components.

The Monasca Client Service will translate the threshold rules from the catalog to a structure that can be interpreted by the Monasca's API, and likewise the AE Client Service and the CEP Client Service will translate the aggregation rules from the catalog to structures that can be interpreted by the Aggregation Engine's API and the CEP Engine's API respectively.

Just like it happens in the Master Service case, during their start up the Client Services will read from their configuration files all the information required for them to interact with the three services they communicate with directly, the Zookeeper and the Mongo Database which are common to all three of them, and the Monasca, Aggregation Engine and CEP Engine depending on the Client Service.

The client services have a watcher on their respective znodes, which monitors the child znodes created there (the tasks). As soon as they detect the creation of a new child znode, they begin the task process following the steps illustrated in figure 6.28.

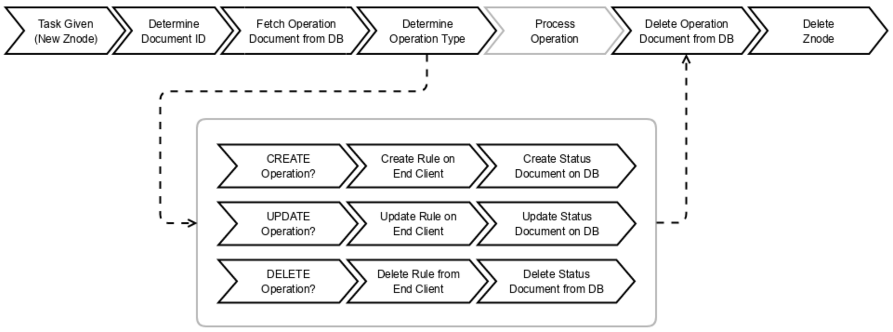

Task Given (New Znode)
Determine Document ID
Fetch Operation Document from DB
Determine Operation Type
Process Operation
Delete Operation Document from DB
Delete Znode
CREATE Operation?
Create Rule on End Client
Create Status Document on DB
UPDATE Operation?
Update Rule on End Client
Update Status Document on DB
DELETE Operation?
Delete Rule from End Client
Delete Status Document from DB

Figure 6.28: Aggregation Configuration Manager - Client Services Steps

The very first step the client does is retrieving the znode value, which is the operation document ID, so it can fetch the operation document from the database. Once it holds the operation document, it can then determine the type of operation as well as the rule or group of rules from the catalog data that it needs to enforce or remove from its end service.

If it is a CREATE operation, the client will then enforce the rule(s) on its end service, and afterwards, it will create the status document on the database which will contain the result status (OK or ERROR), the catalog endpoint from where the catalog data was retrieved by the Master Service, the catalog data itself which was used to enforce the rule(s) on the end service, the client response data and the timestamps of creation and modification which have the same value in this case.

If it is an UPDATE operation, the client will also retrieve the existing status document (this is where the *catalog_endpoint* field comes in), so it can compare the catalog data used previously with the new catalog data stored in the operation document, effectively determining the differences between them and enforcing the changes in the end service. Afterwards, the client will update the status document by changing the modification timestamp, the status of the operation, replacing the catalog data with the new one and of course also updating the client response data.

If it is a DELETE operation, the Client Service will retrieve the status document from the database in order to have access to the catalog data that was used to enforce the rule(s) on the end service. It will use that information to remove the rules from the end service, and delete the status document from the database afterwards.

Independently of the operation type, after the Client Service has handled the enforcement of the rule(s) and the database's status document, it will finally remove the operation document from the database and delete the child znode from Zookeeper, finishing then the task and process the next one or keep watching for new incoming tasks.

6.4 Summary

The implementation chapter exposed how the Monitoring Catalog and the Aggregation Configuration Manager components were developed, what technologies were used to accomplish their requirements, what features they support and which external components they interact with.

Monasca was not necessarily implemented but rather used to fulfill some requirements of the project, and the used micro-services were exposed and explained in detail as well the development of the notifications plugin, that one implemented purposely to cover one of the requirements of the project that was not natively supported by Monasca.

CHAPTER 7

Application and Results

In this chapter the SELFNET testbed will be introduced, as well as how the components that are under the scope of this thesis were deployed (section 7.1). Next the practical application of the Self-Protection Use Case (SP-UC) and its respective results will be analyzed (section 7.2). Some load tests will be presented along with a discussion on the results (section 7.3), and finally some conclusions will be taken at the end of this chapter (section 7.4).

7.1 Testbed and Integration/Deployment

The SELFNET project comprises a large amount of different services, some of them developed by Altice Labs, and many others developed by other partners of the project. As such, a testbed was devised and put at disposal of all the project's partners in order to deploy and integrate all the required and developed logical components, to test the project's use cases and to prepare the periodic demonstrations that would be presented to European reviewing boards of the project.

SELFNET is a fair complex project and the testbed's complexity goes along with it, making it unrealistic to get into its specific architecture details within the scope of this thesis. With that in mind, a very high level architecture of the testbed is presented in figure 7.1 where the main layers and resources of the architecture are shown.

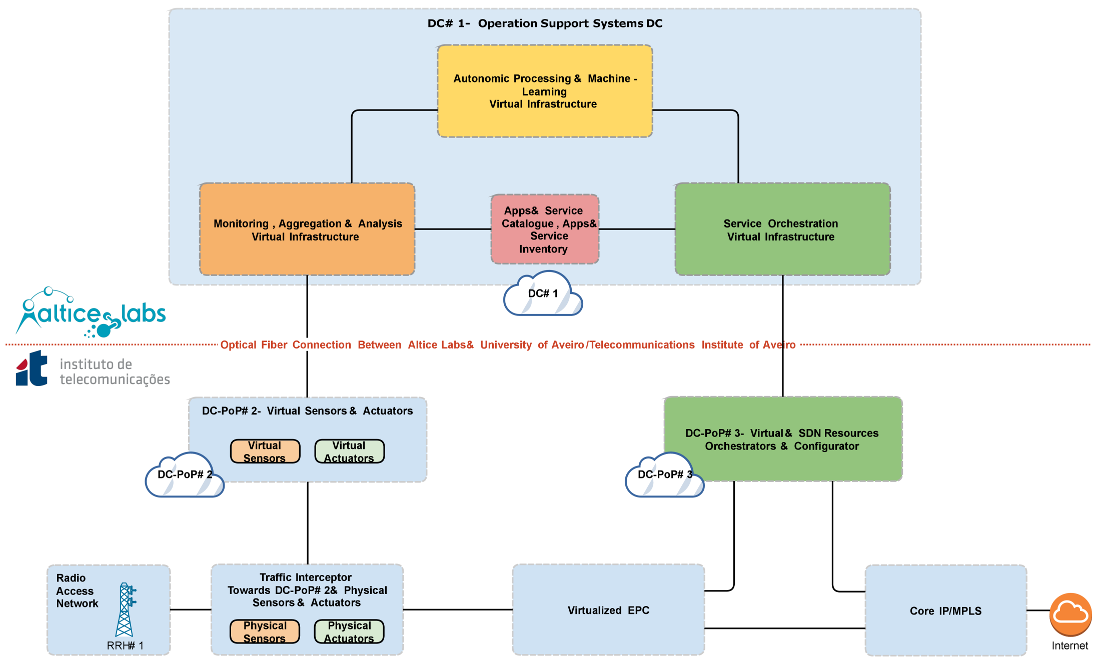

DC# 1- Operation Support Systems DC
Autonomic Processing & Machine - Learning Virtual Infrastructure
Monitoring , Aggregation & Analysis Virtual Infrastructure
Apps& Service Catalogue , Apps& Service Inventory
Service Orchestration Virtual Infrastructure
DC# 1
altice labs
instituto de telecomunicações
Optical Fiber Connection Between Altice Labs& University of Aveiro/Telecommunications Institute of Aveiro
DC-PoP# 2- Virtual Sensors & Actuators
Virtual Sensors
Virtual Actuators
DC-PoP# 2
DC-PoP# 3- Virtual & SDN Resources Orchestrators & Configurator
DC-PoP# 3
Radio Access Network
RRH# 1
Traffic Interceptor Towards DC-PoP# 2& Physical Sensors & Actuators
Physical Sensors
Physical Actuators
Virtualized EPC
Core IP/MPLS
Internet

Figure 7.1: SELFNET Testbed - High Level Infrastructure Architecture

The testbed itself is provided by Altice Labs and Instituto de Telecomunicações of Aveiro (short IT), where part of the resources are located within the premises of the first one and the remaining on the latter. The two premises are connected together by an optical fibre link, allowing Altice Labs' datacenter to communicate with the IT's infrastructure.

This solution provides a flexible network comprised of several physical (bare-metal) and virtual servers, network connections, sensors, traffic interceptors, actuators and so on, as well as an Internet connectivity and a Long Term Evolution (LTE) radio network which is connected to a virtualized Evolved Packet Core (EPC) [96].

7.1.1 Integration/Deployment

Given that the three services within the scope of this thesis are the Monasca, the Monitoring Catalog and the Aggregation Configuration Manager, these will be the ones covered in this section.

The integration of these three services on the testbed was done by deploying them in two Virtual Machines (VMs) within the Altice Labs premises. Monasca, with all the components of its micro-service architecture and taking into account its heavy duty purpose on the Aggregation framework, was deployed on one of the VMs while the ACM and the MC were deployed on the other VM.

Each VM has four CPU cores, 16GB of RAM and 32GB of hard disk space, both with Ubuntu Linux 16.04 LTS installed as the operating system. Docker [97] containers were used for the deployment of the services within each VM to facilitate both the development cycle and the deployment either on the testbed's VMs as well by the partners who would require or wish to do so on their own testbeds.

- **Monasca** - The integration and deployment of this service has been completed in May

2017 and it has since been operating 24/7 without any major problems. Using figure 6.1 (Chapter 6 - Implementation) as a reference, the Monasca micro-services are deployed in a total of eleven containers distributed as follows:

  - **InfluxDB**: 1 container;
  - **MySQL**: 1 container;
  - **Kafka**: 1 container;
  - **Monasca API**: 1 container;
  - **Persister**: 1 container;
  - **Notification Engine**: 1 container;
  - **Threshold Engine**: 3 containers (Apache Storm Supervisor, Storm Nimbus and Storm UI);
  - **Zookeeper**: 1 container (required for Kafka and the Threshold Engine);
  - **Grafana**: 1 container (extra - time-series analysis and monitoring WebUI).

- **Monitoring Catalog and Aggregation Configuration Manager** - The integration and deployment of these services is ongoing at the moment. Using figures 6.3 and 6.25 (Chapter 6 - Implementation) as references, the micro-services of these components are deployed in a total of nine containers distributed as follows:

  - **MongDB**: 1 container (ACM);
  - **Neo4j**: 1 container (MC);
  - **Kafka**: 1 container (MC and ACM);
  - **Zookeeper**: 1 container (required for Kafka and the ACM);
  - **MC API**: 1 container;
  - **ACM Master Service**: 1 container;
  - **ACM Monasca Client**: 1 container;
  - **ACM AE Client**: 1 container;
  - **ACM CEP Client**: 1 container.

The Docker containers are managed on the VMs with the help of Docker Compose [98], which is a tool for defining and running multi-container Docker applications. The containers are connected together using Docker's container networking [99], exposing only the required ports to the host and the outside external services.

7.2 Self-Protection Use Case in Practice

7.2.1 Scenario

As already stated previously, the SELFNET project is use case driven and it comprises three use cases that were discussed in chapter 4 (section 4.3). Since the Self-Protection Use-Case (SP-UC) is the one that at the moment is the most refined, matured, already integrated and used in the testbed many times, and also used in demonstrations for the reviewing boards of the project, it will be the one used here as a practical case.

The main goal of the SP-UC is to detect the bots of a botnet (zombies) that successfully managed to infiltrate the network by infecting some hosts, and take action to sever their communications to the botnet's Command & Control CnC in order to neutralize the botnet. The overall process is divided in three loops with several steps, as described below:

- **Loop 1** - Preliminary identification of suspect communication patterns that may indicate that a host or group of hosts may be infected with zombies of a botnet. To achieve this, the following steps are taken:
  - two batch aggregation rules are (manually) onboarded on the Monitoring Catalog (MC) which will be enforced in the Aggregation Engine (AE) by the Aggregation Configuration Manager ACM;
  - a threshold rule, which will be applied over these two batch aggregation rules, must be also (manually) onboarded on the MC to be enforced on Monasca, which when crossed, will produce alarms and consequent notifications for the Autonomic framework;
  - the Autonomic framework produces a symptom based on the alarm notification and aggregated metrics, and instruct the Orchestration framework to deploy a Virtual Network Function (VNF) to mirror the suspicious traffic for Deep Packet Inspection (DPI);
  - in parallel with the previous step, the Orchestration framework is also instructed to deploy a DPI VNF (SNORT) to identify the possible zombies of the botnet.
- **Loop 2** - Confirmation (or dismissal) of the suspect communication patterns and diversion of the botnet's traffic. To achieve this, the following steps are taken:
  - at this phase of the process the DPI is already collecting information and generating events that need to be aggregated in realtime, and for this purpose, a stream aggregation rule is (dynamically) onboarded on the MC which will be enforced on the CEP Engine by the ACM;
  - the stream aggregated metrics produced by the CEP Engine will confirm (or dismiss) if the communications indeed fall into the botnet pattern and will start generating alerts for the Autonomic framework;
  - the Autonomic framework produces a new symptom and instructs the Orchestration framework to deploy a HoneyNet VNF to isolate the communications from the zombies to the botnet, by diverting the traffic to this HoneyNet;
  - at the same time the Autonomic framework initiates a learning phase using a Machine Learning process with Neural Networks [100], with the goal of filtering out false positives and acting more quickly upon new zombies that may appear afterwards, which when over will (dynamically) onboard a new batch aggregation rule and a new threshold rule on the MC, starting then the 3rd loop.
- **Loop 3** - Severing the botnet's communications to neutralize it. To achieve this, the following steps are taken:
  - once the new batch aggregation rule and the new threshold rule are in place, the following communications from new zombies will generate new alarms for the Autonomic framework (different from the previous alarms of Loop 1);

– upon receiving these new alarms, the Autonomic framework will generate a new symptom and instruct the Orchestration framework to simply drop the related communications, severing or disrupting any communication attempts from the botnet's CnC to its zombies, i.e. the botnet is from this moment onwards effectively neutralized;

7.2.2 Results

In this section we will take a look at the results obtained in the SP-UC, where some charts related to the three loops described in the previous section will be used as support. All the data represented in these charts comes from real sensing data performed in the testbed, where all the components and services from the SELFNET's architecture necessary to exercise this use case are deployed, including the botnet. Also, the data present on the charts is stored in the Influx time-series database of Monasca, and the charts were plotted using Grafana [101], which is an open-source platform for time-series analytics and monitoring.

7.2.2.1 SP-UC Loop 1

Figures 7.2 and 7.3 represent the batch aggregation metrics enforced on the AE that are required for this loop of the use case, *Average Packet Count* and the *Communication Frequency* respectively. The aggregation is done on sensing data collected by the Flow Monitoring Agent (FMA) on all the traffic within the testbed, and the metrics are tied together with a unique combination of *SourceIP*, *DestIP* and *DestPort*.

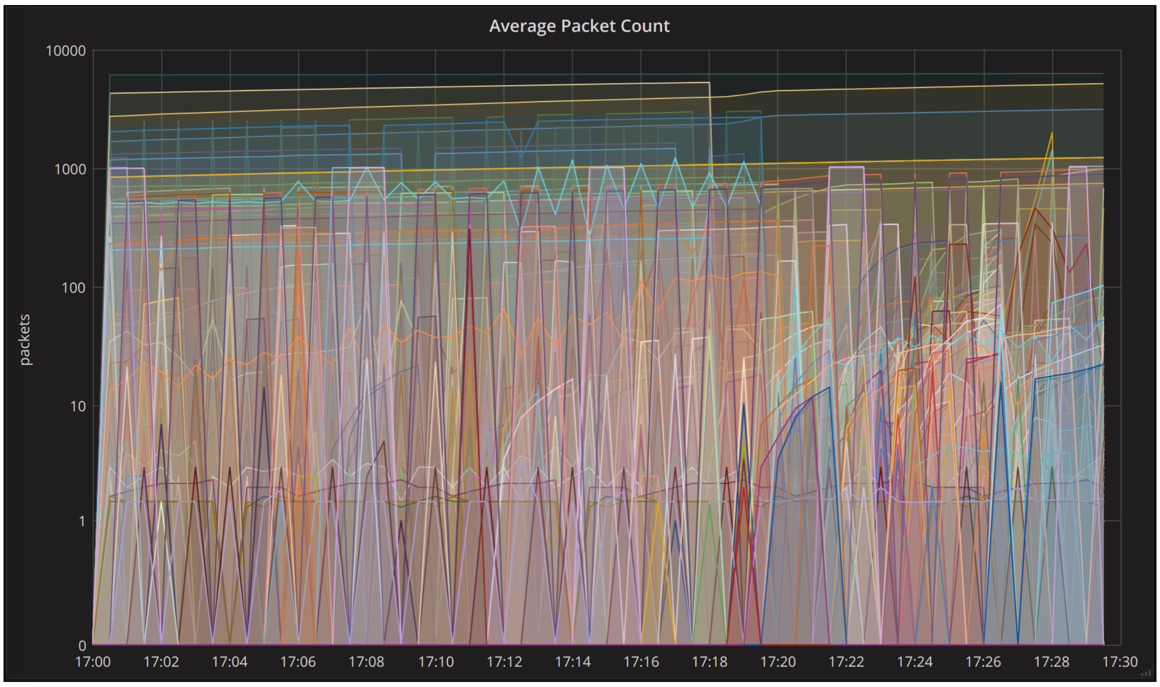


Figure 7.2: Batch Aggregation - Average Packet Count

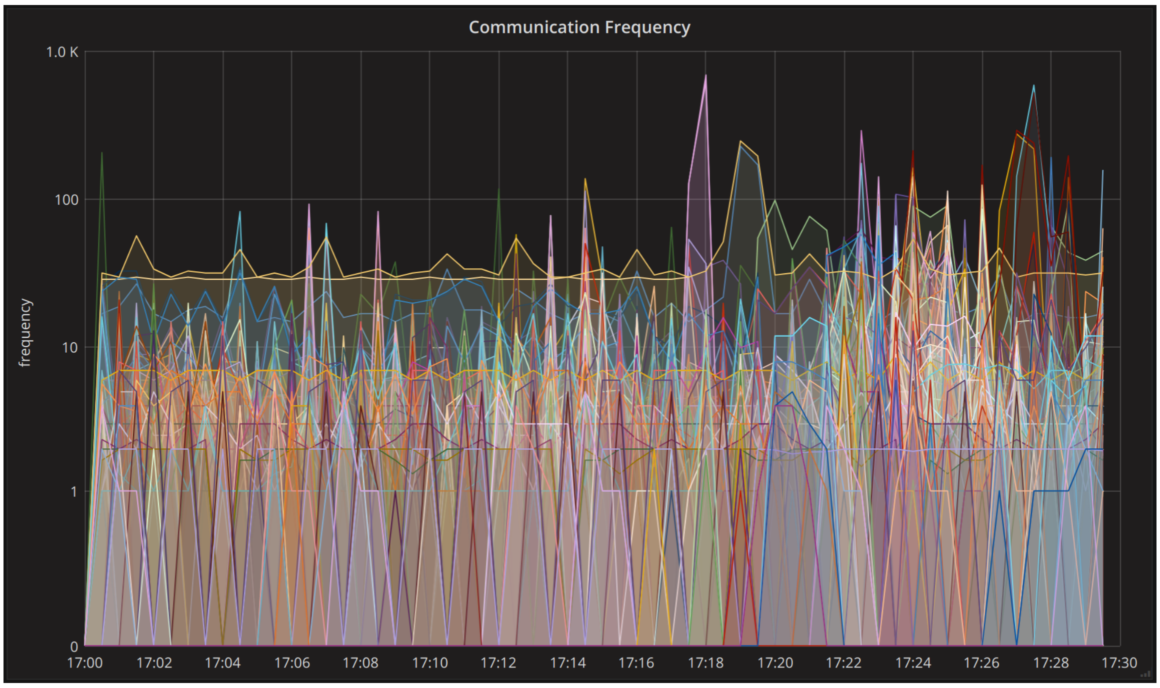
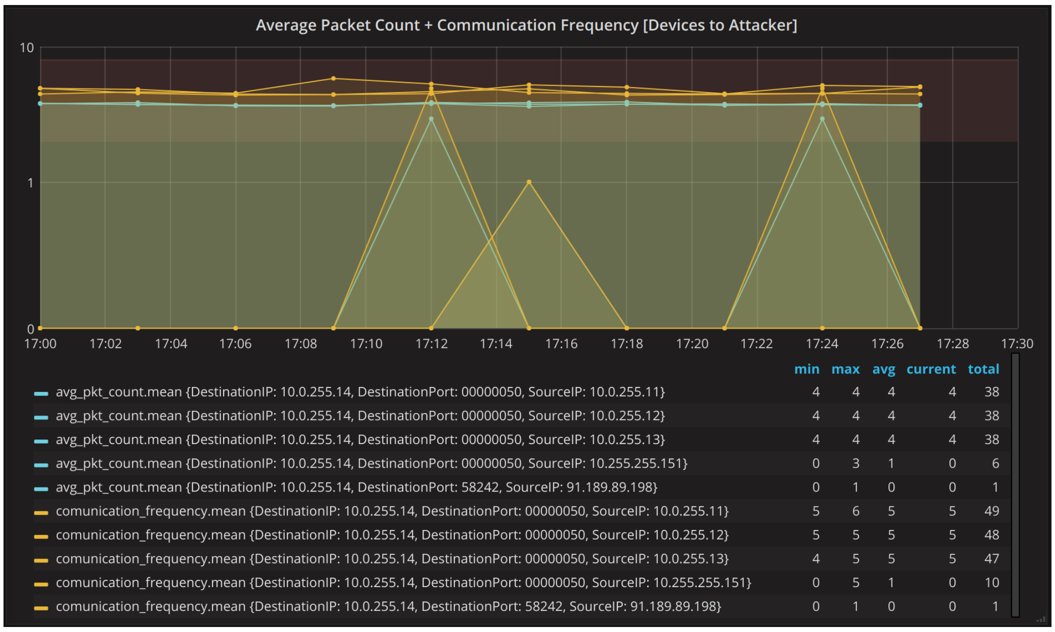


Figure 7.3: Batch Aggregation - Communication Frequency

While no useful information can be retrieved from the charts on figures 7.2 and 7.3 (they merely show the huge amount of aggregated metrics produced as a result of the enforced aggregation rules), figures 7.4 and 7.5 represent the same aggregated metrics but this time around they are filtered to show only the traffic of the botnet. The brown stripes on the charts represent the threshold window (values between 2 and 9) that will trigger alarms for every communication flow that falls under it (even if it is not related to the botnet, i.e false positives). It is worth mentioning that the configured threshold only takes into account sustained values within a period of 180 seconds at least, to ensure that fluctuating communications do not trigger alarms so easily, i.e reducing the number of false positives. The blue lines represent the *Average Packet Count* and the yellow ones the *Communication Frequency*.

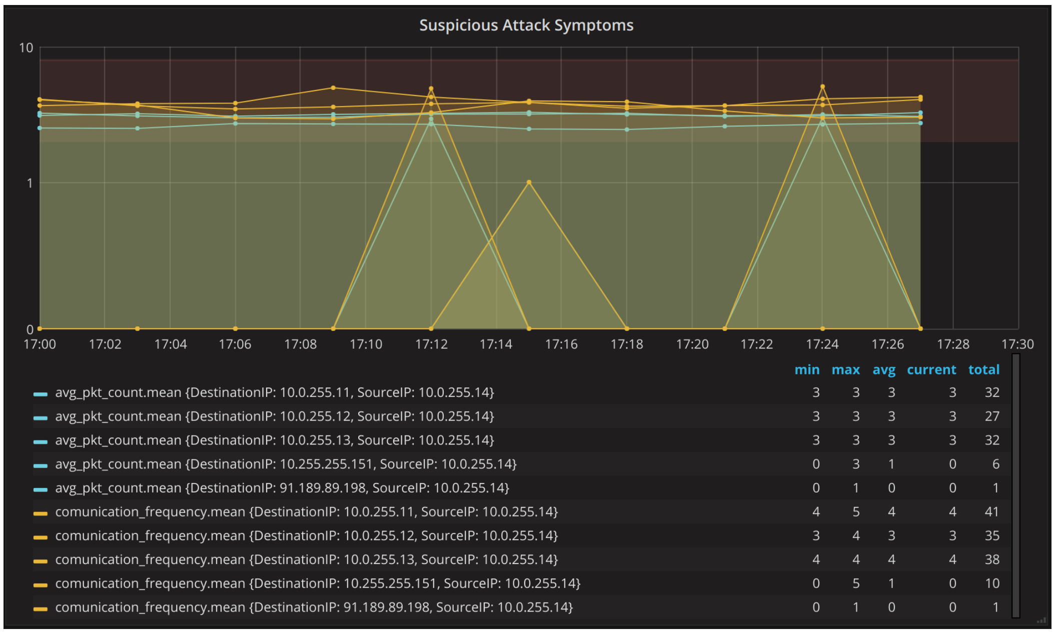
In these charts five IP addresses can be observed:

- **10.0.255.11-13**: the three zombies of the botnet deployed on the testbed;
- **10.0.255.14**: the botnet's CnC, also deployed on the testbed;
- **91.189.89.198**: it can be ignored as it is only a browser connection to the CnC's GUI, used to monitor the botnet throughout the use case exercise.



	min	max	avg	current	total
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	4	4	4	4	38
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	4	4	4	4	38
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	4	4	4	4	38
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.255.255.151}	0	3	1	0	6
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 58242, SourceIP: 91.189.89.198}	0	1	0	0	1
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	5	6	5	5	49
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	5	5	5	5	48
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	4	5	5	5	47
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.255.255.151}	0	5	1	0	10
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 58242, SourceIP: 91.189.89.198}	0	1	0	0	1

Figure 7.4: Batch Aggregation - Traffic from the Zombies to the CnC - Loop 1



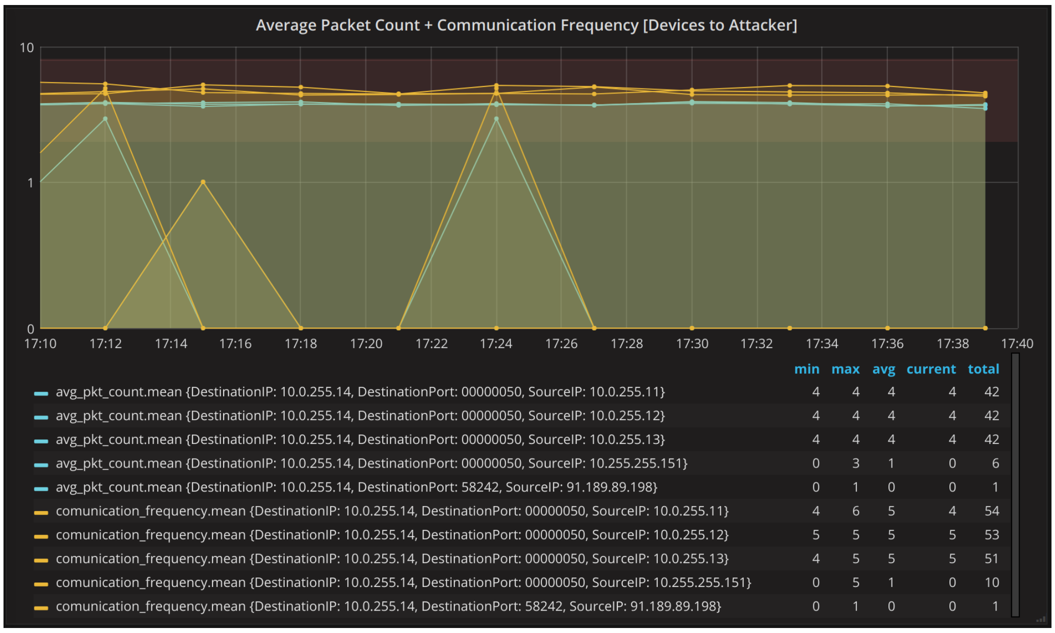
	min	max	avg	current	total
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	3	3	3	3	32
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	3	3	3	3	27
🟦 avg_pkt_count.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	3	3	3	3	32
🟦 avg_pkt_count.mean {DestinationIP: 10.255.255.151, SourceIP: 10.0.255.14}	0	3	1	0	6
🟦 avg_pkt_count.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	1	0	0	1
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	4	5	4	4	41
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	3	4	3	3	35
🟨 comunication_frequency.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	4	4	4	4	38
🟨 comunication_frequency.mean {DestinationIP: 10.255.255.151, SourceIP: 10.0.255.14}	0	5	1	0	10
🟨 comunication_frequency.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	1	0	0	1

Figure 7.5: Batch Aggregation - Traffic from the CnC to the Zombies - Loop 1

At this phase of the SP-UC, since there are communication patterns that fall under the configured threshold, the alarms are being triggered and the Autonomic framework is already being notified on them. This is the point explained in the use case scenario, where the Autonomic framework generates a symptom and instructs the Orchestration framework to deploy a VNF to mirror the suspicious traffic, and another one performs the DPI of the mirrored traffic.

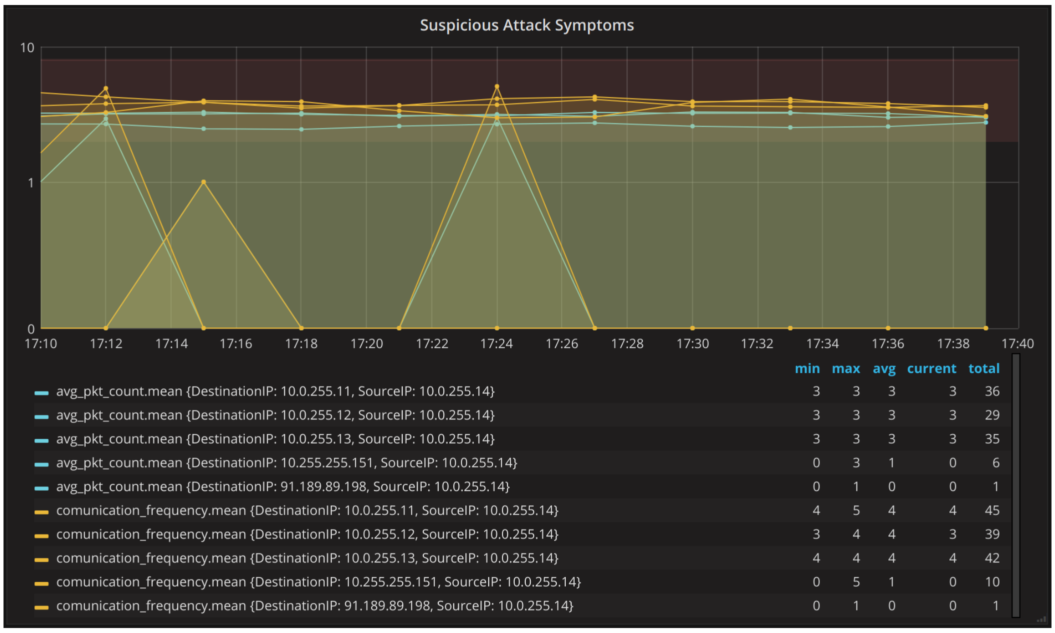
7.2.2.2 SP-UC Loop 2

Figures 7.6 and 7.7 represent the same data seen in figures 7.4 and 7.5, but further ahead in time, merely serving as a comparison reference for the two upcoming figures, showing that the botnet's communications are still flowing within the network while the DPI is ongoing.



	min	max	avg	current	total
avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	4	4	4	4	42
avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	4	4	4	4	42
avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	4	4	4	4	42
avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.255.255.151}	0	3	1	0	6
avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 58242, SourceIP: 91.189.89.198}	0	1	0	0	1
comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	4	6	5	4	54
comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	5	5	5	5	53
comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	4	5	5	5	51
comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.255.255.151}	0	5	1	0	10
comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 58242, SourceIP: 91.189.89.198}	0	1	0	0	1

Figure 7.6: Batch Aggregation - Traffic from the Zombies to the CnC - Loop 2



	min	max	avg	current	total
avg_pkt_count.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	3	3	3	3	36
avg_pkt_count.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	3	3	3	3	29
avg_pkt_count.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	3	3	3	3	35
avg_pkt_count.mean {DestinationIP: 10.255.255.151, SourceIP: 10.0.255.14}	0	3	1	0	6
avg_pkt_count.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	1	0	0	1
comunication_frequency.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	4	5	4	4	45
comunication_frequency.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	3	4	4	3	39
comunication_frequency.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	4	4	4	4	42
comunication_frequency.mean {DestinationIP: 10.255.255.151, SourceIP: 10.0.255.14}	0	5	1	0	10
comunication_frequency.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	1	0	0	1

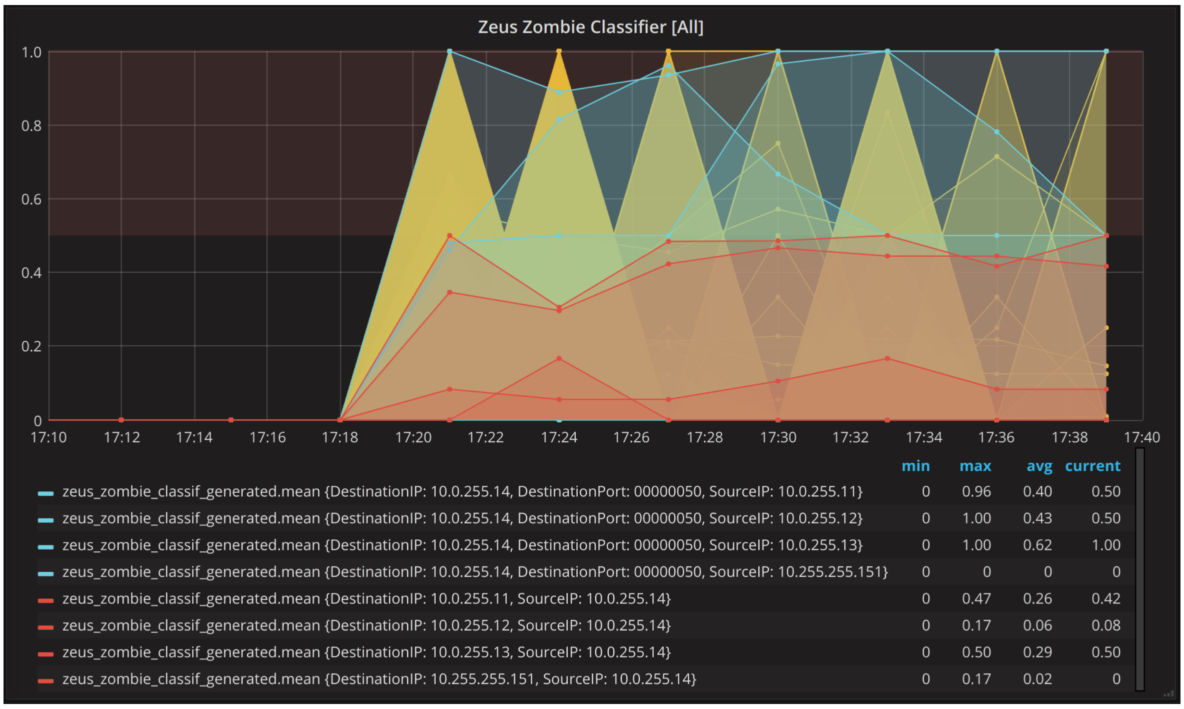
Figure 7.7: Batch Aggregation - Traffic from the CnC to the Zombies - Loop 2

As explained in the use case scenario, at this phase of the process the DPI confirms that the communications belong to a recognized botnet, which will trigger alarms sent to the Autonomic framework leading to the deployment of a HoneyNet to isolate the communications of the zombies by diverting the traffic to it.

Also, the Autonomic framework initiates its learning phase to determine a formula to better identify the communication patterns of the botnet, which will be used to generate a new aggregation rule with the goal of distinguishing the false positives from the communications that indeed belong to the botnet.

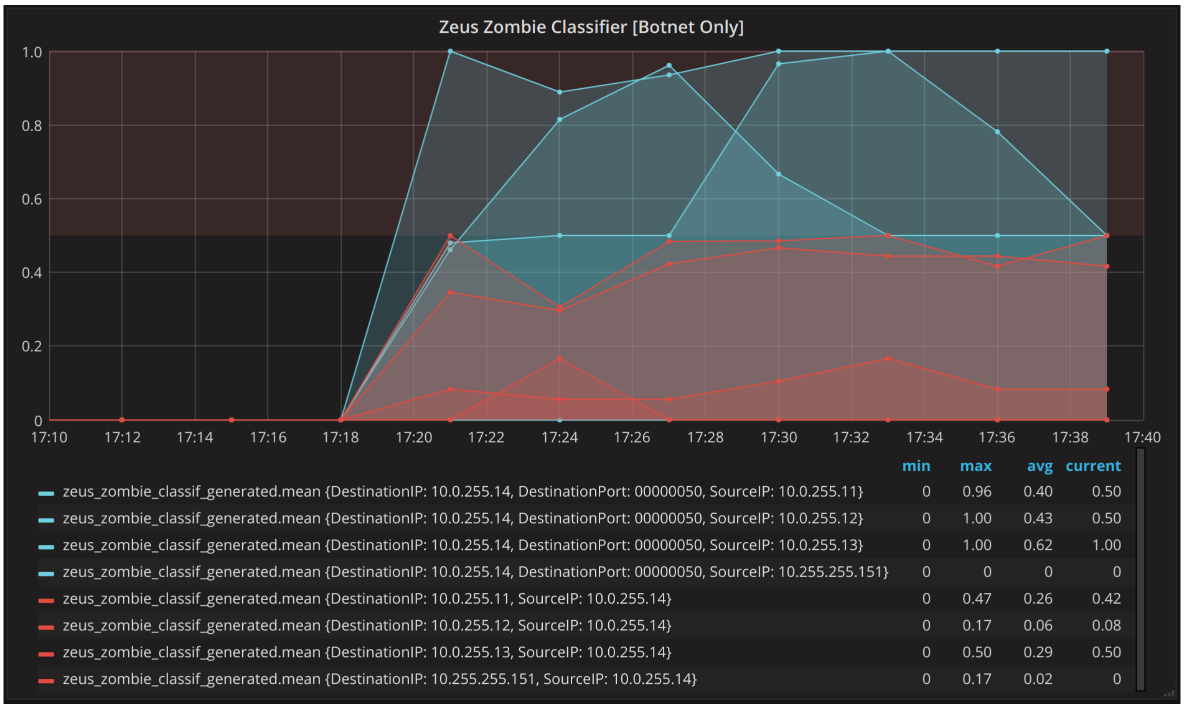
Once this learning process is finished, the Autonomic framework will then (dynamically) onboard the new batch aggregation rule with an associated threshold rule on the MC to be enforced on both the AE and Monasca.

The result of this action is shown in figure 7.8, where the false positives are represented in yellow while the communications of the botnet are represented in blue and red. The brown stripe on the chart represents the threshold zone where only the communications with values above 0.5 (a coefficient determined by the generated formula) with a rate sustained for at least 180 seconds will trigger alarms for the Autonomic framework.



	min	max	avg	current
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	0	0.96	0.40	0.50
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	0	1.00	0.43	0.50
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	0	1.00	0.62	1.00
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.255.255.151}	0	0	0	0
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	0	0.47	0.26	0.42
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	0	0.17	0.06	0.08
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	0	0.50	0.29	0.50
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.255.255.151, SourceIP: 10.0.255.14}	0	0.17	0.02	0

Figure 7.8: Batch Aggregation - Zeus Botnet Classification (Full) - Loop 2



	min	max	avg	current
zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	0	0.96	0.40	0.50
zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	0	1.00	0.43	0.50
zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	0	1.00	0.62	1.00
zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.255.255.151}	0	0	0	0
zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	0	0.47	0.26	0.42
zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	0	0.17	0.06	0.08
zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	0	0.50	0.29	0.50
zeus_zombie_classif_generated.mean {DestinationIP: 10.255.255.151, SourceIP: 10.0.255.14}	0	0.17	0.02	0

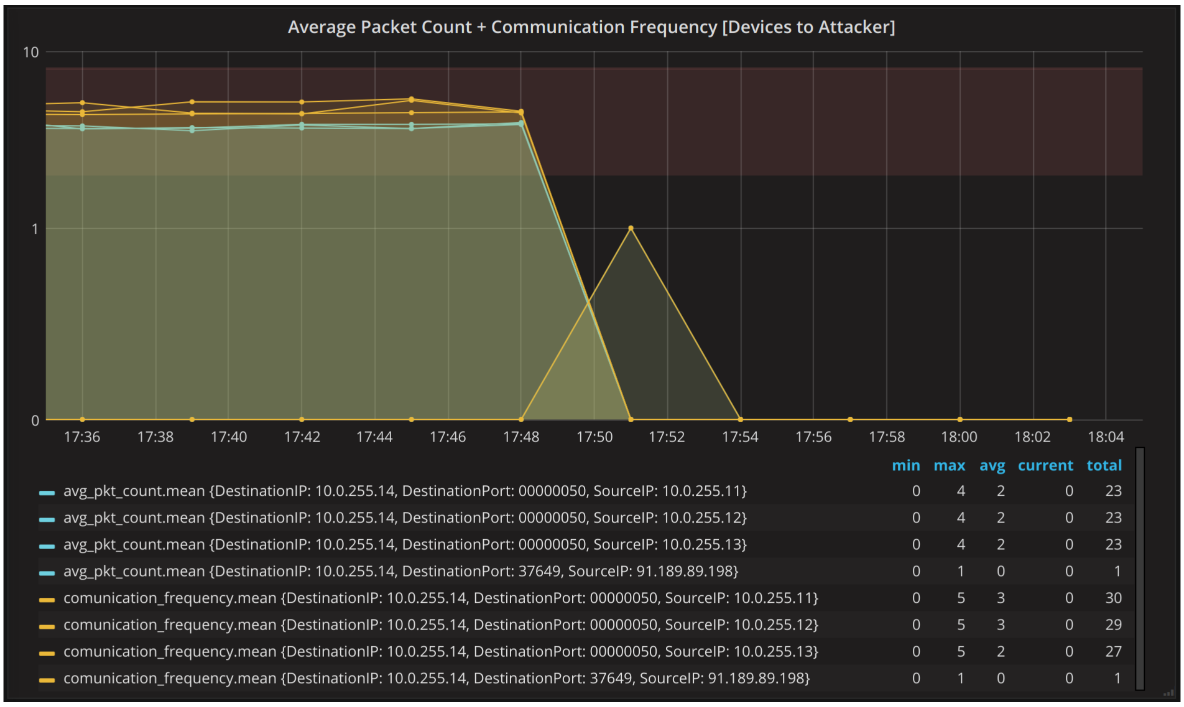
Figure 7.9: Batch Aggregation - Zeus Botnet Classification (Filtered) - Loop 2

Figure 7.9 represents the same data shown on the previous one, but this time around with the false positives filtered out. In this figure the communications from the botnet's CnC to the three zombies are clearly visible within the threshold zone, meaning that these communications will be the ones triggering alarms for the Autonomic framework, so it can instruct the Orchestration framework to drop them out.

7.2.2.3 SP-UC Loop 3

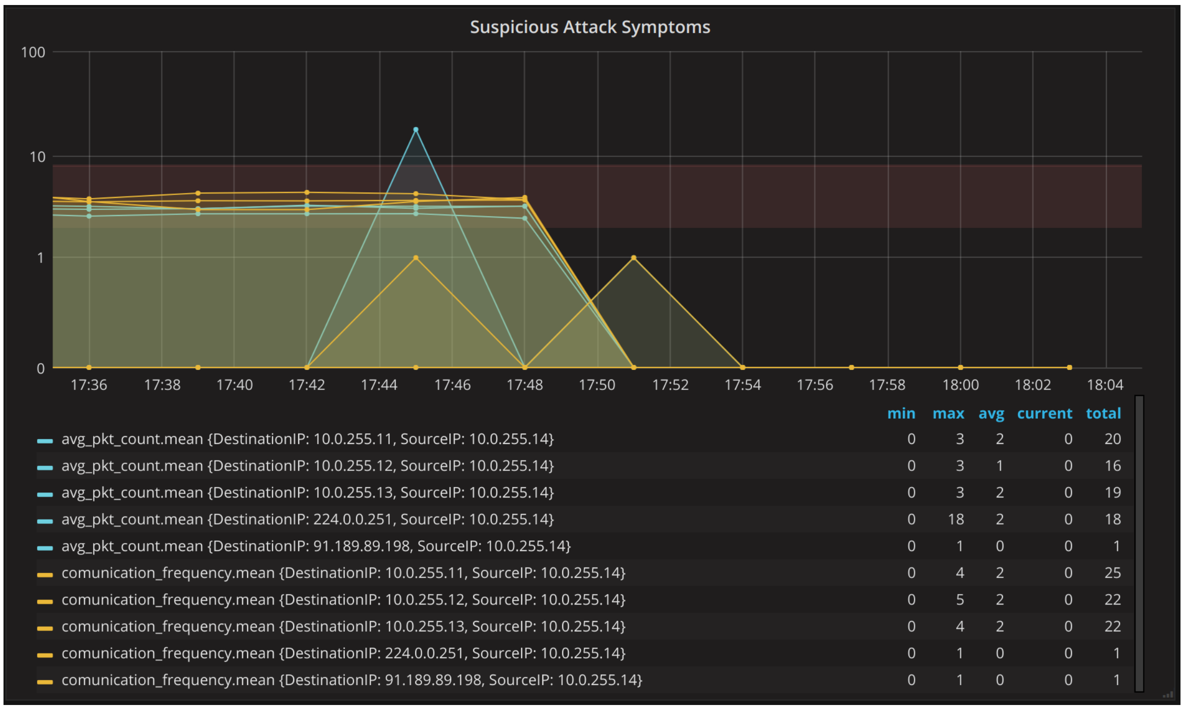
At last but not the least, we reach a phase in the process of the SP-UC where the communications of the botnet are already being perfectly identified, and alarms are being generated for them and sent to the Autonomic framework. The Orchestration framework is then instructed to drop these communications and the results can be seen in figures 7.10, 7.11, 7.12 and 7.13.

Although the communications of the botnet are being successfully disrupted, observing with close attention figure 7.12, it is noticed that the communications related to the false positives are not, which means that "legit" communications of regular traffic are not affected in any way by the SELFNET's self-protection abilities.



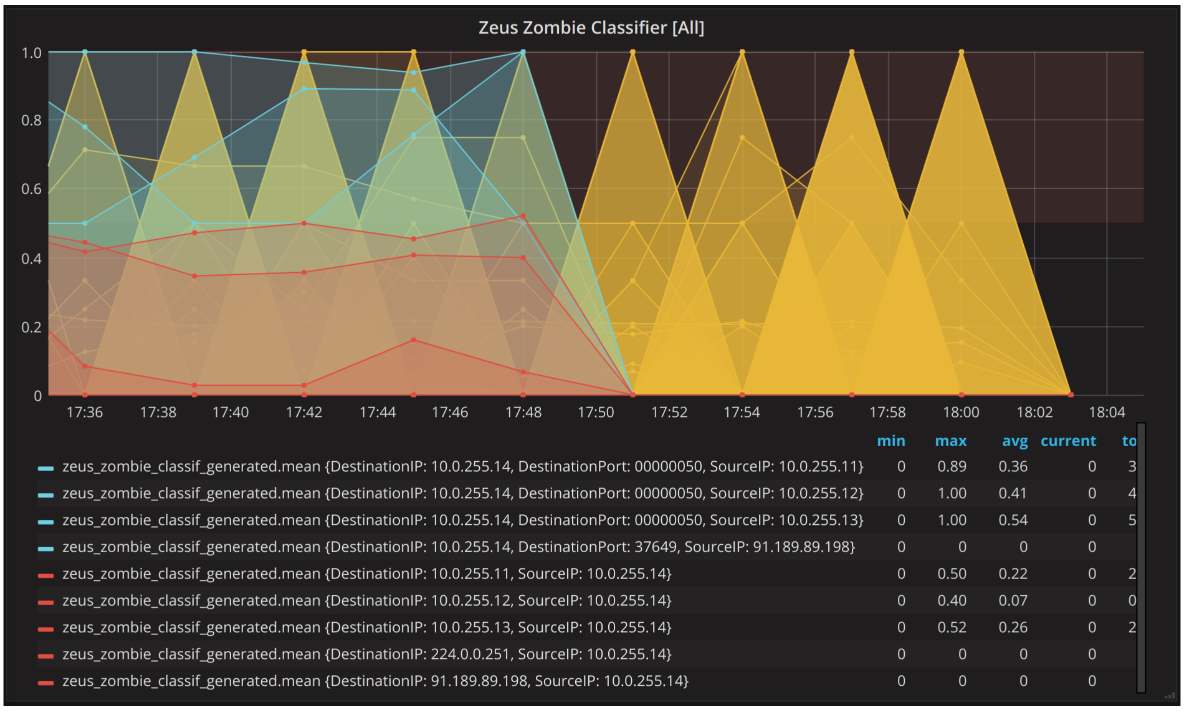
	min	max	avg	current	total
▬ avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	0	4	2	0	23
▬ avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	0	4	2	0	23
▬ avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	0	4	2	0	23
▬ avg_pkt_count.mean {DestinationIP: 10.0.255.14, DestinationPort: 37649, SourceIP: 91.189.89.198}	0	1	0	0	1
▬ comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	0	5	3	0	30
▬ comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	0	5	3	0	29
▬ comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	0	5	2	0	27
▬ comunication_frequency.mean {DestinationIP: 10.0.255.14, DestinationPort: 37649, SourceIP: 91.189.89.198}	0	1	0	0	1

Figure 7.10: Batch Aggregation - Traffic from the Zombies to the CnC - Loop 3



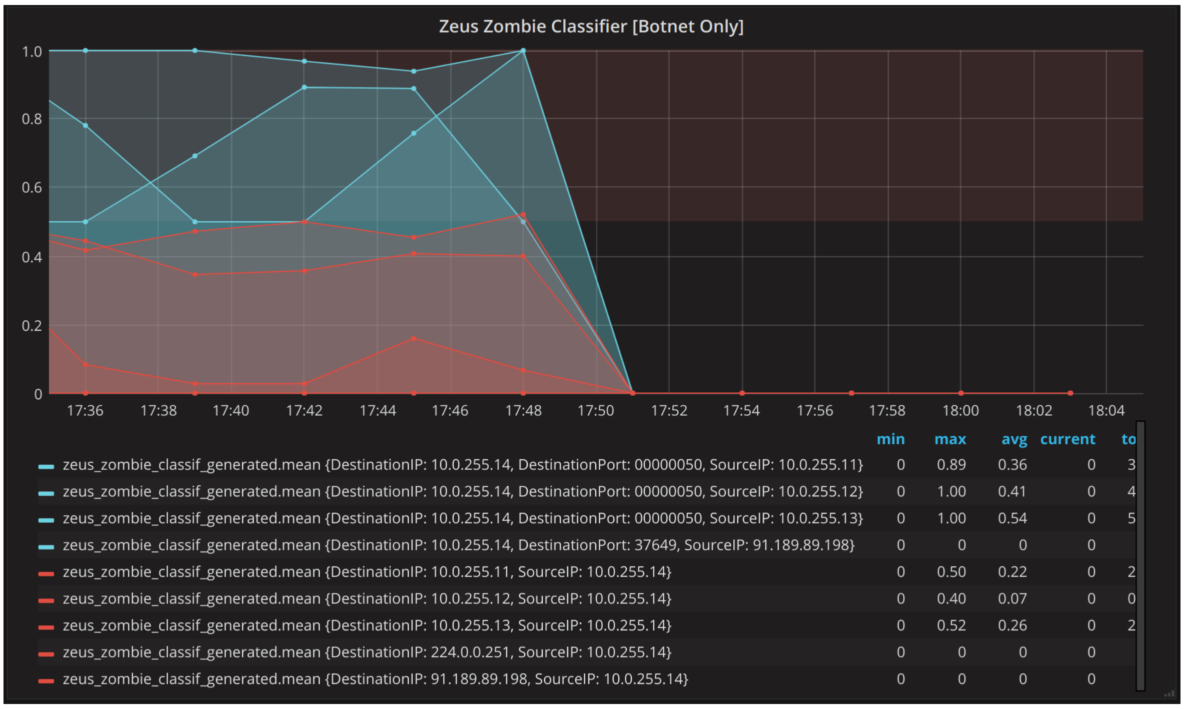
	min	max	avg	current	total
▬ avg_pkt_count.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	0	3	2	0	20
▬ avg_pkt_count.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	0	3	1	0	16
▬ avg_pkt_count.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	0	3	2	0	19
▬ avg_pkt_count.mean {DestinationIP: 224.0.0.251, SourceIP: 10.0.255.14}	0	18	2	0	18
▬ avg_pkt_count.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	1	0	0	1
▬ comunication_frequency.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	0	4	2	0	25
▬ comunication_frequency.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	0	5	2	0	22
▬ comunication_frequency.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	0	4	2	0	22
▬ comunication_frequency.mean {DestinationIP: 224.0.0.251, SourceIP: 10.0.255.14}	0	1	0	0	1
▬ comunication_frequency.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	1	0	0	1

Figure 7.11: Batch Aggregation - Traffic from the CnC to the Zombies - Loop 3



	min	max	avg	current	to
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	0	0.89	0.36	0	3
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	0	1.00	0.41	0	4
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	0	1.00	0.54	0	5
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 37649, SourceIP: 91.189.89.198}	0	0	0	0	
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	0	0.50	0.22	0	2
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	0	0.40	0.07	0	0
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	0	0.52	0.26	0	2
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 224.0.0.251, SourceIP: 10.0.255.14}	0	0	0	0	
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	0	0	0	

Figure 7.12: Batch Aggregation - Zeus Botnet Classification (Full) - Loop 3



	min	max	avg	current	to
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.11}	0	0.89	0.36	0	3
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.12}	0	1.00	0.41	0	4
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 00000050, SourceIP: 10.0.255.13}	0	1.00	0.54	0	5
🟦 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.14, DestinationPort: 37649, SourceIP: 91.189.89.198}	0	0	0	0	
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.11, SourceIP: 10.0.255.14}	0	0.50	0.22	0	2
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.12, SourceIP: 10.0.255.14}	0	0.40	0.07	0	0
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 10.0.255.13, SourceIP: 10.0.255.14}	0	0.52	0.26	0	2
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 224.0.0.251, SourceIP: 10.0.255.14}	0	0	0	0	
🟥 zeus_zombie_classif_generated.mean {DestinationIP: 91.189.89.198, SourceIP: 10.0.255.14}	0	0	0	0	

Figure 7.13: Batch Aggregation - Zeus Botnet Classification (Filtered) - Loop 3

7.3 Load Tests

In order to determine the performance of the developed and/or integrated services that are subject to the scope of this thesis, a series of load tests were devised and performed, either by using freely available tools or by developing custom ones.

Both the Monitoring Catalog and Monasca have REST APIs, and therefore the load tests were done using the RESTful Stress tool [102], an open-source tool designed to test, check and stress RESTful web services using simple client invocations on HTTP verbs (GET, POST, PUT, DELETE).

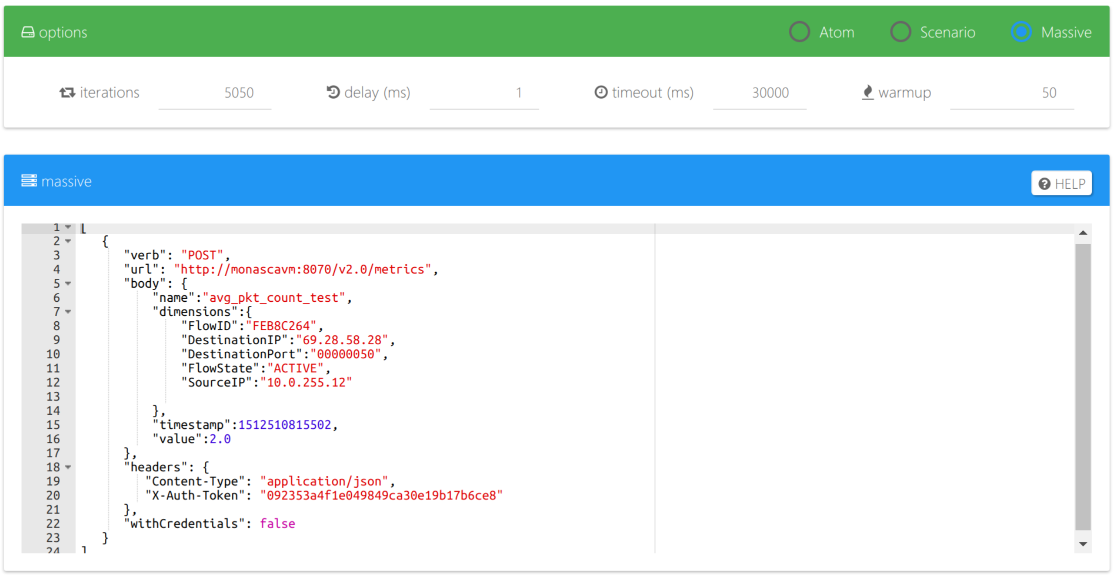
As to the Aggregation Configuration Manager, a series of timestamp variables were carefully placed within its source code to collect information about the times spent in specific operations, providing statistics when activated and through load tests stimulating this component by "hammering" the Kafka topic this component is watching on, simulating the operations done in the Monitoring Catalog.

7.3.1 Monasca

7.3.1.1 Load Test 1 - Submission of Metrics

This first load test of Monasca comprises only the performance on the submission of metrics to its time-series database. The metrics themselves are not directly stored by the API on the database, but rather published to the message bus and consequently consumed by the persister which holds the responsibility of actually storing them on the database.

Figure 7.14 shows the settings of the load test which simulates the typical metrics submission performed by the Aggregation Engine, and it is composed by 5050 iterations with 50 of them used as warm up to discard possible initial peak values.



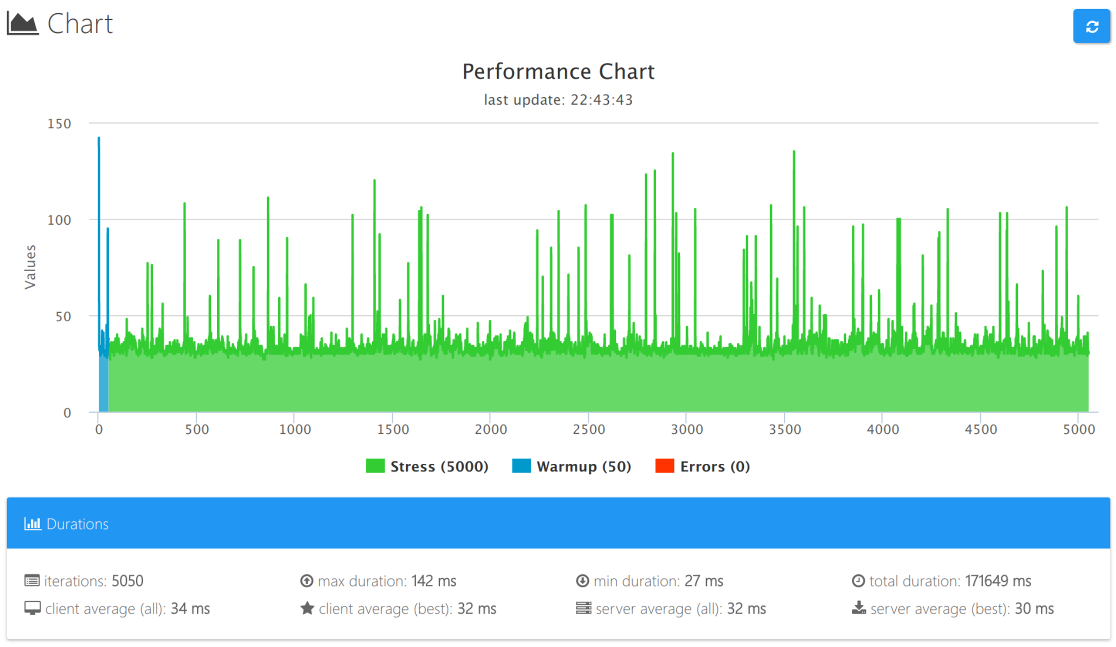
Massive [✓]

iterations 5050 | delay (ms) 1 | timeout (ms) 30000 | warmup 50

```
[
    {
        "verb": "POST",
        "url": "http://monascavm:8070/v2.0/metrics",
        "body": {
            "name":"avg_pkt_count_test",
            "dimensions":{
                "FlowID":"FEB8C264",
                "DestinationIP":"69.28.58.28",
                "DestinationPort":"00000050",
                "FlowState":"ACTIVE",
                "SourceIP":"10.0.255.12"

            },
            "timestamp":1512510815502,
            "value":2.0
        },
        "headers": {
           "Content-Type": "application/json",
           "X-Auth-Token": "092353a4f1e049849ca30e19b17b6ce8"
        },
        "withCredentials": false
    }
]
```

Figure 7.14: Monasca - Load Test 1 Settings

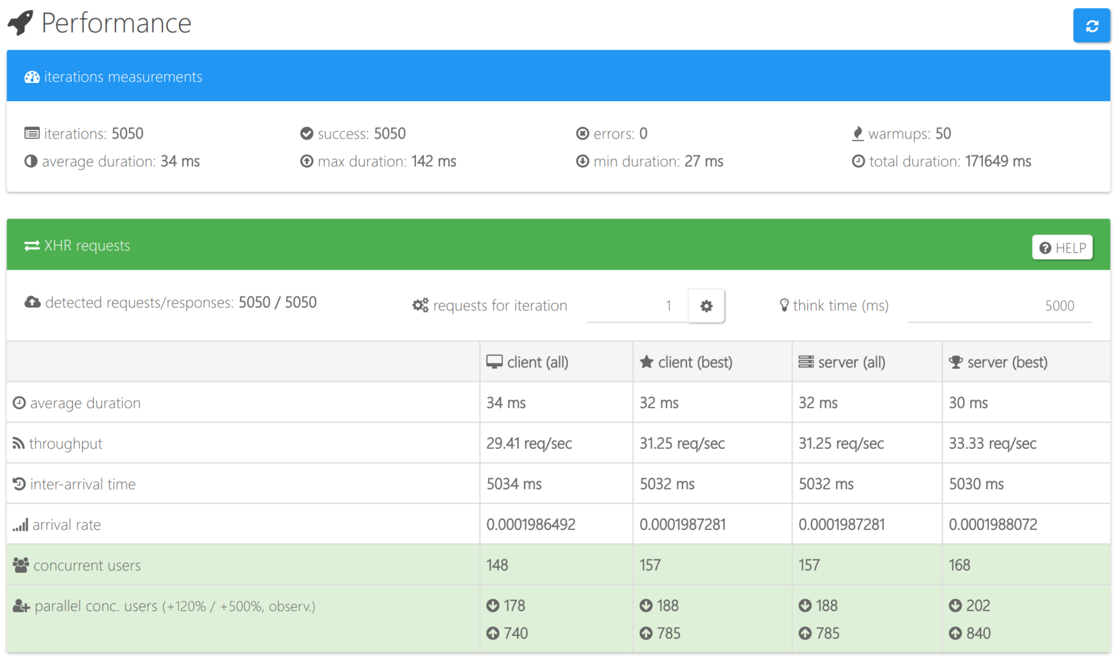
Figure 7.15 is a chart showing the results of the load test, where some peaks in the requests can be seen, but most of them fall within the average time per request as perceived by the client, i.e. 34ms.



iterations: 5050	max duration: 142 ms	min duration: 27 ms	total duration: 171649 ms
client average (all): 34 ms	client average (best): 32 ms	server average (all): 32 ms	server average (best): 30 ms

Figure 7.15: Monasca - Load Test 1 Chart

Figure 7.16 summarizes the results, where the most important ones are the average duration and the throughput as perceived the the client, which is 34ms and 29.41 req/sec respectively.



iterations: 5050	success: 5050	errors: 0	warmups: 50
average duration: 34 ms	max duration: 142 ms	min duration: 27 ms	total duration: 171649 ms

detected requests/responses: 5050 / 5050 — requests for iteration 1 — think time (ms) 5000

	client (all)	client (best)	server (all)	server (best)
average duration	34 ms	32 ms	32 ms	30 ms
throughput	29.41 req/sec	31.25 req/sec	31.25 req/sec	33.33 req/sec
inter-arrival time	5034 ms	5032 ms	5032 ms	5030 ms
arrival rate	0.0001986492	0.0001987281	0.0001987281	0.0001988072
concurrent users	148	157	157	168
parallel conc. users (+120% / +500%, observ.)	178 740	188 785	188 785	202 840

Figure 7.16: Monasca - Load Test 1 Performance

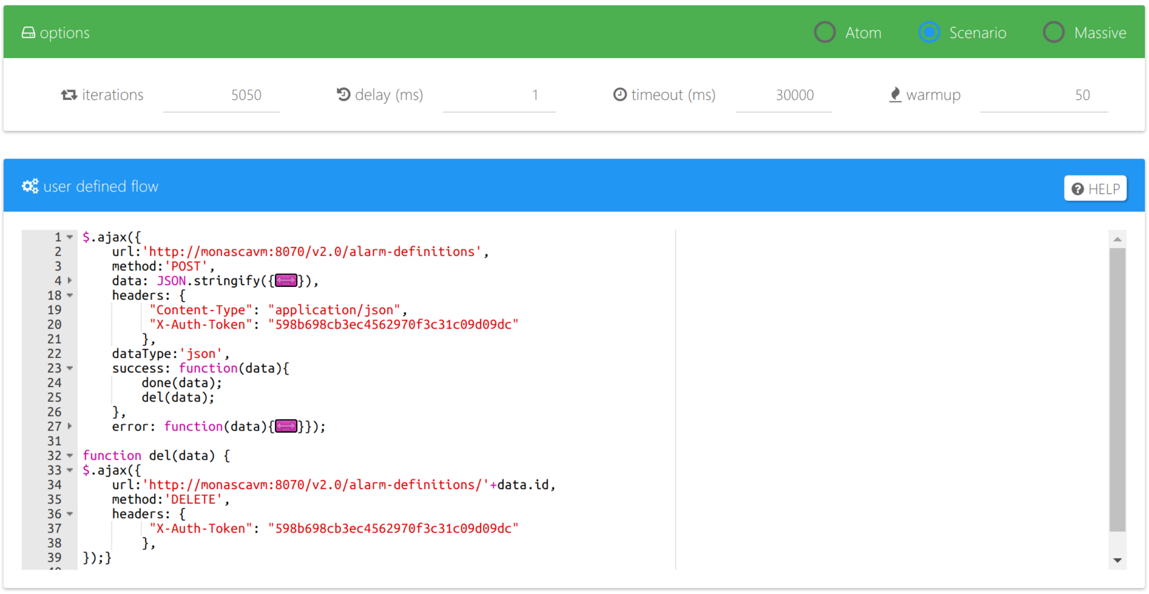
The interpretation of these results means that the Aggregation Engine, as the client of Monasca, should be able to sustain the submission of 29 aggregated metrics per second in normal usage with every submission taking 34ms on average.

Measurements taken from the Monasca Persister's logs show that, on average, the Aggregation Engine is submitting 10 aggregated metrics per second, which means that the service is perfectly capable of coping with this rate.

7.3.1.2 Load Test 2 - Submission of Thresholds

The second load test of Monasca comprises the performance on the submission of thresholds only. As it happens with the submission of metrics, the thresholds are not submitted directly to the Threshold Engine, but rather published to a Kafka topic which is monitored by it.

Figure 7.17 shows the settings of this load test used to simulate the typical threshold submission performed by the ACM's Monasca Client. It is composed by 5050 iterations with 50 of them used as warm up to discard possible initial peak values.



options — Atom [✓ Scenario] Massive

iterations	5050	delay (ms)	1	timeout (ms)	30000	warmup	50

user defined flow

HELP

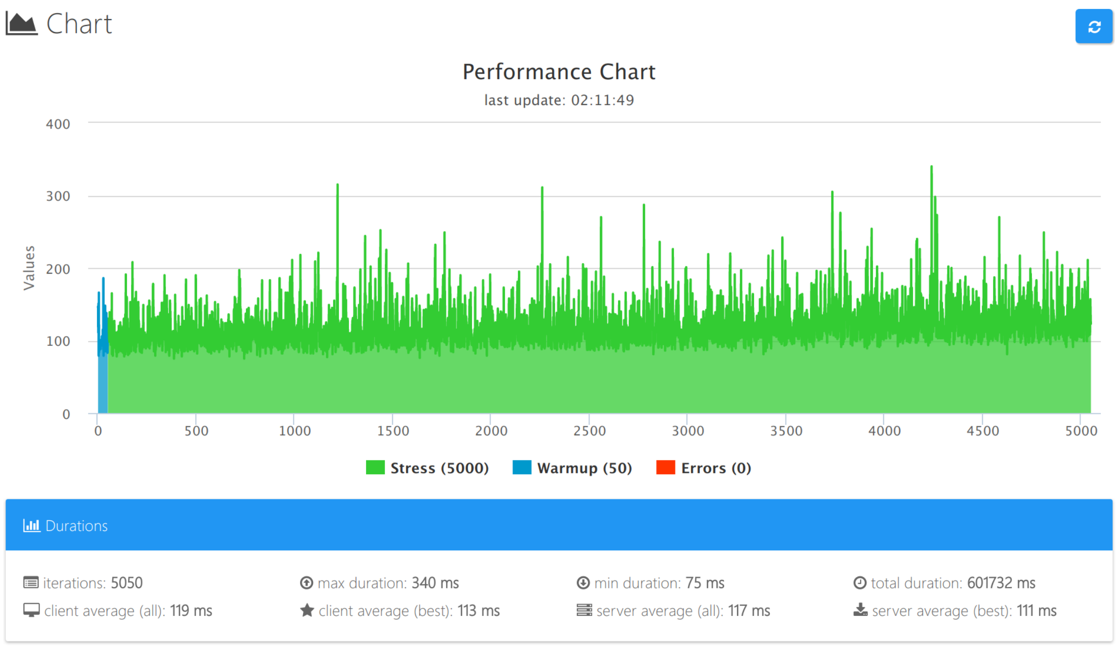
```
$.ajax({
    url:'http://monascavm:8070/v2.0/alarm-definitions',
    method:'POST',
    data: JSON.stringify({↔}),
    headers: {
         "Content-Type": "application/json",
         "X-Auth-Token": "598b698cb3ec4562970f3c31c09d09dc"
        },
    dataType:'json',
    success: function(data){
        done(data);
        del(data);
    },
    error: function(data){↔}});

function del(data) {
$.ajax({
    url:'http://monascavm:8070/v2.0/alarm-definitions/'+data.id,
    method:'DELETE',
    headers: {
         "X-Auth-Token": "598b698cb3ec4562970f3c31c09d09dc"
        },
});}
```

Figure 7.17: Monasca - Load Test 2 Settings

This load test was difficult to perform, since the submitted thresholds needed to be removed from Monasca afterwards, so they would not be left there polluting the service and consuming unnecessary resources (the Threshold Engine is one of the most resource hungry services of Monasca). Although not ideal, a simple average is performed on the time spent on the two requests (POST and DELETE) in the final stats, giving a fair approximation of the time spent on the threshold submission alone.

The threshold's JSON data is omitted in the figure for clarity of the procedure, but it can be observed the Monitoring Catalog Data Model example in Appendix C.2 to get an idea of the submitted data.

Figure 7.18 is the chart showing the results of the load test, where the reported average time per request as perceived by the client is 119ms. This value needs to be divided by two, which means that the approximate average time per request is in fact 60ms.

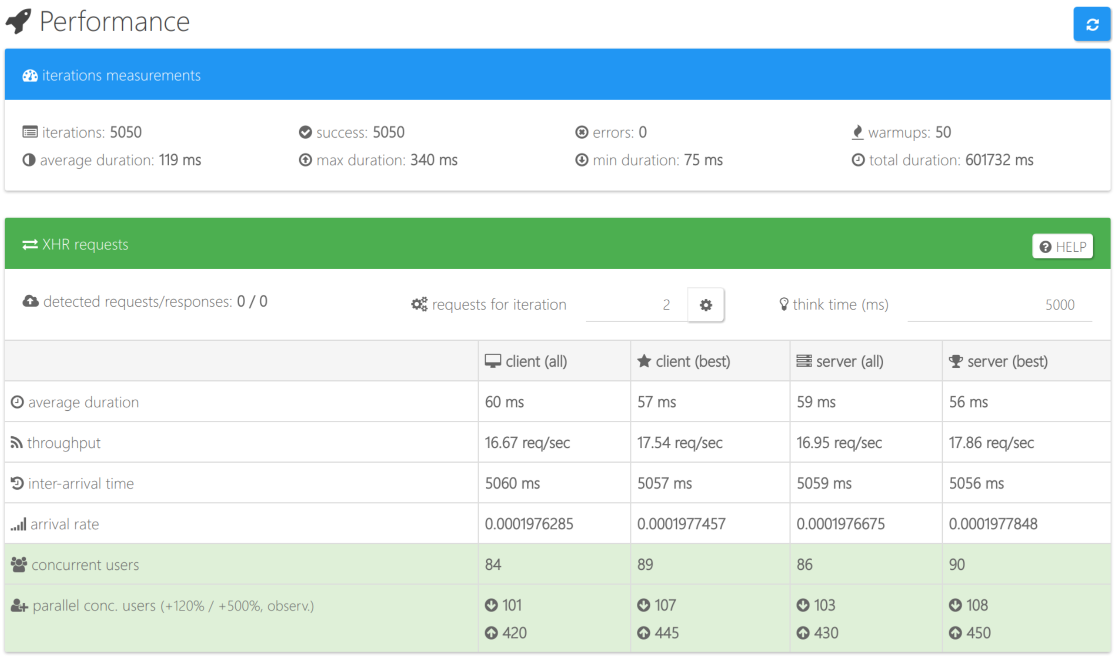


Durations

iterations: 5050	max duration: 340 ms	min duration: 75 ms	total duration: 601732 ms
client average (all): 119 ms	client average (best): 113 ms	server average (all): 117 ms	server average (best): 111 ms

Figure 7.18: Monasca - Load Test 2 Chart

Figure 7.19 once again summarizes the results, where the most important ones are the average duration and the throughput as perceived the the client, which is 60ms and 16.67 req/sec respectively.



iterations measurements

iterations: 5050	success: 5050	errors: 0	warmups: 50
average duration: 119 ms	max duration: 340 ms	min duration: 75 ms	total duration: 601732 ms

XHR requests

detected requests/responses: 0 / 0 requests for iteration 2 think time (ms) 5000

	client (all)	client (best)	server (all)	server (best)
average duration	60 ms	57 ms	59 ms	56 ms
throughput	16.67 req/sec	17.54 req/sec	16.95 req/sec	17.86 req/sec
inter-arrival time	5060 ms	5057 ms	5059 ms	5056 ms
arrival rate	0.0001976285	0.0001977457	0.0001976675	0.0001977848
concurrent users	84	89	86	90
parallel conc. users (+120% / +500%, observ.)	101 420	107 445	103 430	108 450

Figure 7.19: Monasca - Load Test 2 Performance

The interpretation of these results means that the ACM's Monasca Client, as the client of

Monasca in this case, should be able to sustain the submission of 16 thresholds per second in normal usage with every submission taking 60ms on average.

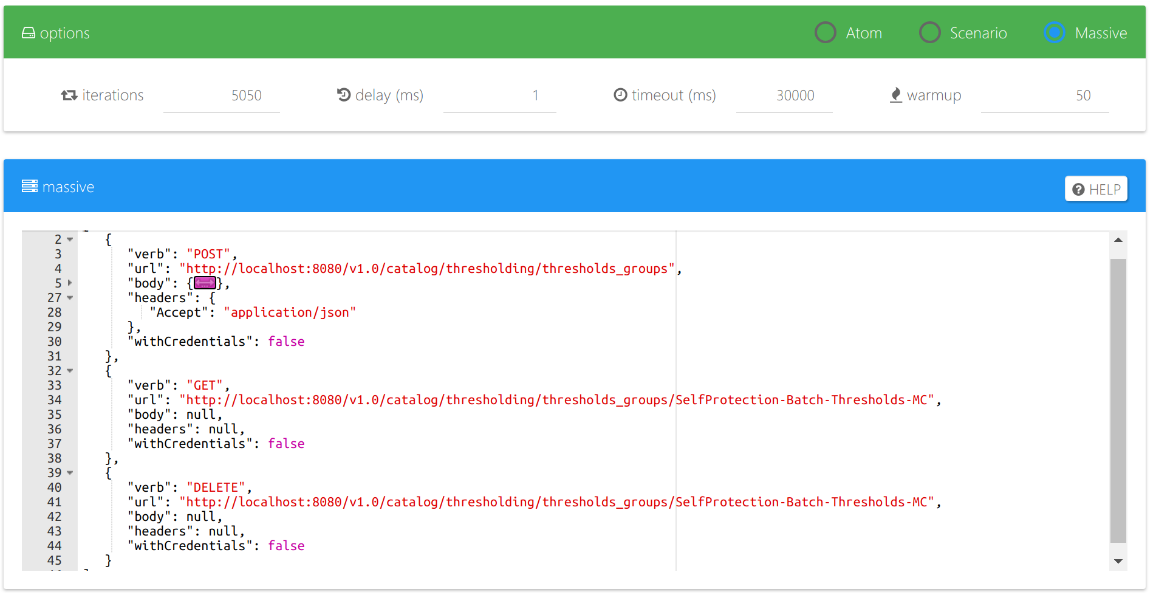
Although it presents worst values when compared to the submission of metrics, it is not expected from the SELFNET's project to even come close to this rate of threshold submission on Monasca. The thresholds are to be submitted sparingly and when strictly required, and not that heavily as it happens with submission the metrics, so these values are perfectly acceptable.

7.3.2 Monitoring Catalog

7.3.2.1 Load Test 1 - Overall Performance

The first load test of the Monitoring Catalog comprises the three operations deemed more relevant for its usage by the Autonomic framework (POST, GET and DELETE), as it is not expected from this service to use the PATCH operation so frequently as the others.

Figure 7.20 shows the settings of this load test used to simulate a typical threshold submission on the Monitoring Catalog (rules groups were also tested but the end results were approximate); it is composed by 5050 iterations with 50 of them used as warm up to discard possible initial peak values.



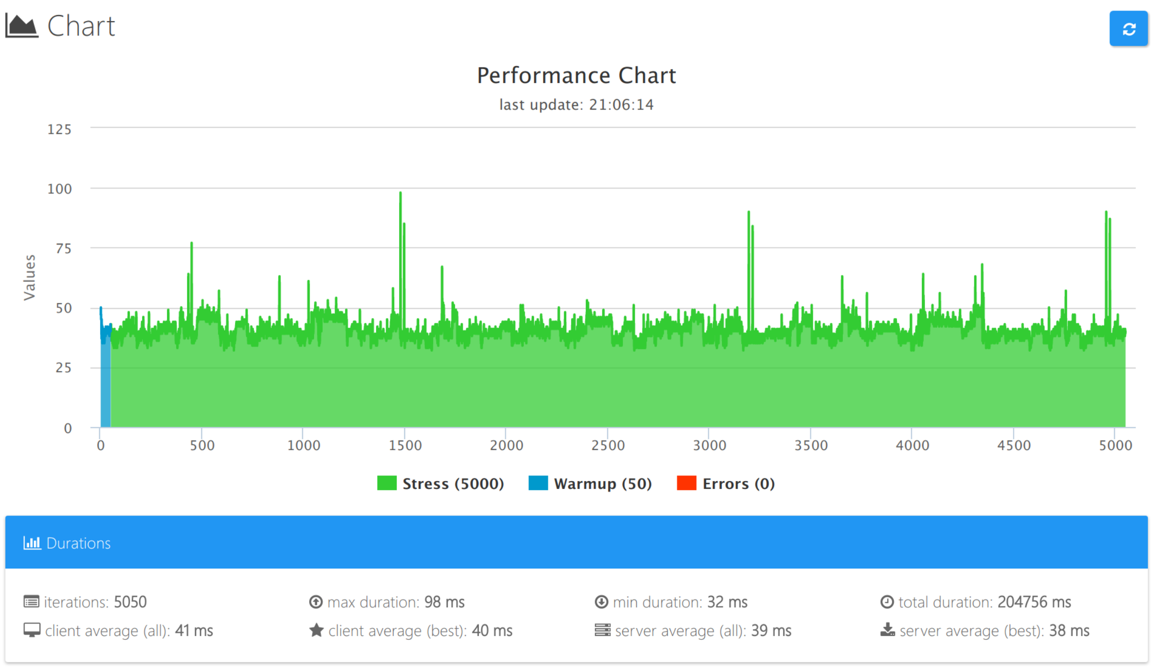
iterations 5050 · delay (ms) 1 · timeout (ms) 30000 · warmup 50

```
    {
        "verb": "POST",
        "url": "http://localhost:8080/v1.0/catalog/thresholding/thresholds_groups",
        "body": {↔},
        "headers": {
            "Accept": "application/json"
        },
        "withCredentials": false
    },
    {
        "verb": "GET",
        "url": "http://localhost:8080/v1.0/catalog/thresholding/thresholds_groups/SelfProtection-Batch-Thresholds-MC",
        "body": null,
        "headers": null,
        "withCredentials": false
    },
    {
        "verb": "DELETE",
        "url": "http://localhost:8080/v1.0/catalog/thresholding/thresholds_groups/SelfProtection-Batch-Thresholds-MC",
        "body": null,
        "headers": null,
        "withCredentials": false
    }
```

Figure 7.20: Monitoring Catalog - Load Test 1 Settings

The threshold's JSON data is omitted in the figure for clarity of the procedure, but as explained before, it can be observed the Monitoring Catalog Data Model example in Appendix C.2 to get an idea of the submitted data.

Figure 7.21 is the chart showing the results of the load test, where the reported average time per request as perceived by the client is 41ms. This test comprises three operations, and if we divide this value by three, the approximate average time per request would be 13.7ms.

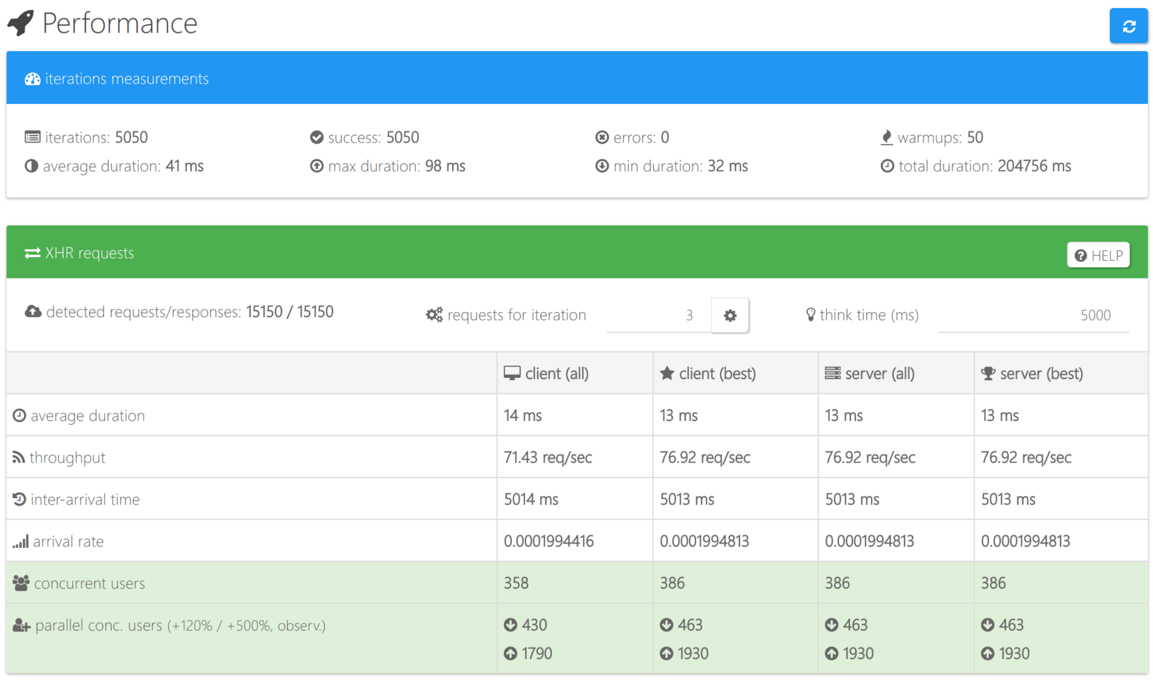


Durations

iterations: 5050	max duration: 98 ms	min duration: 32 ms	total duration: 204756 ms
client average (all): 41 ms	client average (best): 40 ms	server average (all): 39 ms	server average (best): 38 ms

Figure 7.21: Monitoring Catalog - Load Test 1 Chart

Figure 7.22 once again summarizes the results, where the most important ones are the average duration and the throughput as perceived by the client, 14ms and 71.43 req/sec respectively.



iterations measurements

iterations: 5050	success: 5050	errors: 0	warmups: 50
average duration: 41 ms	max duration: 98 ms	min duration: 32 ms	total duration: 204756 ms

XHR requests

detected requests/responses: 15150 / 15150 — requests for iteration 3 — think time (ms) 5000

	client (all)	client (best)	server (all)	server (best)
average duration	14 ms	13 ms	13 ms	13 ms
throughput	71.43 req/sec	76.92 req/sec	76.92 req/sec	76.92 req/sec
inter-arrival time	5014 ms	5013 ms	5013 ms	5013 ms
arrival rate	0.0001994416	0.0001994813	0.0001994813	0.0001994813
concurrent users	358	386	386	386
parallel conc. users (+120% / +500%, observ.)	430 1790	463 1930	463 1930	463 1930

Figure 7.22: Monitoring Catalog - Load Test 1 Performance

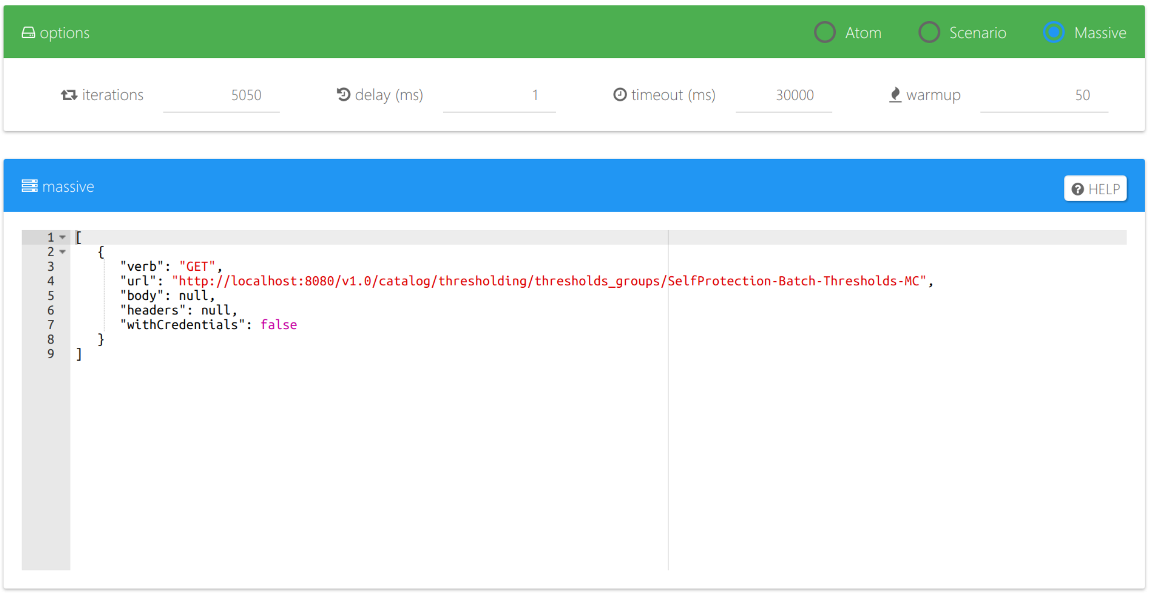
The interpretation of these results means that the Autonomic framework, as the client of Monitoring Catalog, should be able to sustain the submission of 71 threshold groups (or rules groups) per second in normal usage with every submission taking 14ms on average.

The Monitoring Catalog shows the better performance so far when compared to Monasca, but on one hand the performance is well beyond the needs of the SELFNET project as it is not expected from the Autonomic framework to impose such an heavy load on this service. On the other hand, a reason for Monasca to have a little bit subpar performance (comparably) is directly connected to the fact that all of its services are operating within a single VM subject of heavy load, while its load tests were done, which is not the case of the Monitoring Catalog.

7.3.2.2 Load Test 2 - Retrieving Thresholds Groups

The second load test of the Monitoring Catalog is similar to the first one, where the only difference is that only the read operation (GET) is tested. This is so because that is the sole operation that the Aggregation Configuration Manager will perform on this service.

Figure 7.23 shows the settings of the load test which simulates the typical request performed by the Aggregation Configuration Manager; it is composed by 5050 iterations with 50 of them used as warm up to discard possible initial peak values.

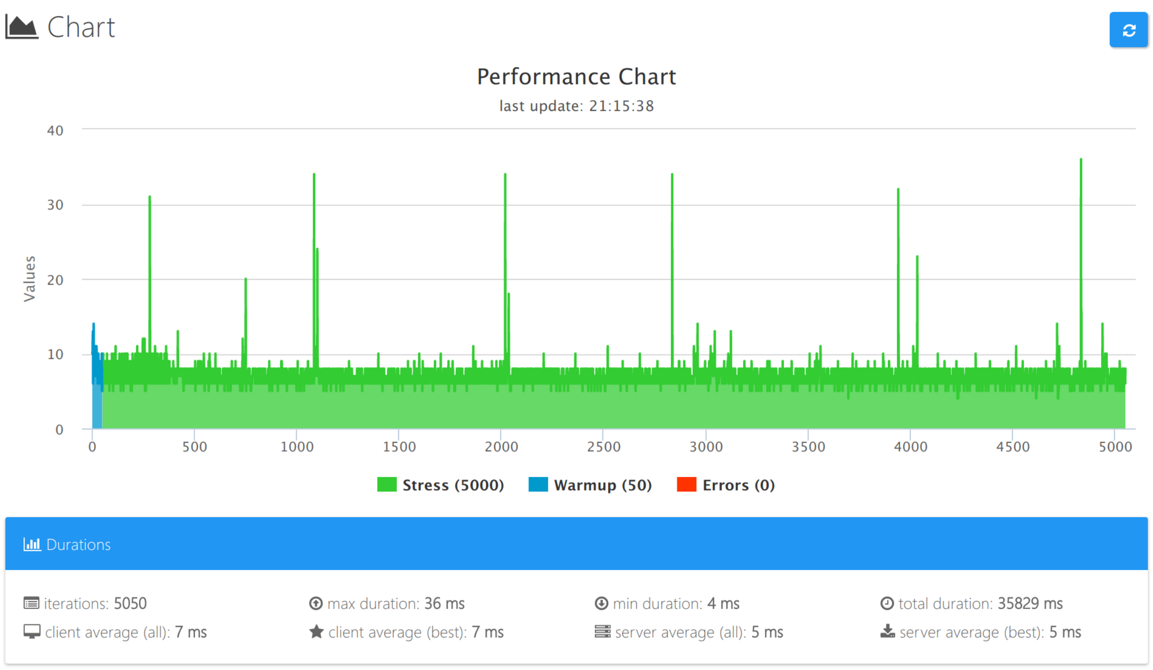


iterations 5050 · delay (ms) 1 · timeout (ms) 30000 · warmup 50

```
[
    {
        "verb": "GET",
        "url": "http://localhost:8080/v1.0/catalog/thresholding/thresholds_groups/SelfProtection-Batch-Thresholds-MC",
        "body": null,
        "headers": null,
        "withCredentials": false
    }
]
```

Figure 7.23: Monitoring Catalog - Load Test 2 Settings

Figure 7.24 is the chart showing the results of the load test, where some peaks in the requests can be seen, but most of them fall within the average time per request of 7ms as perceived by the client.

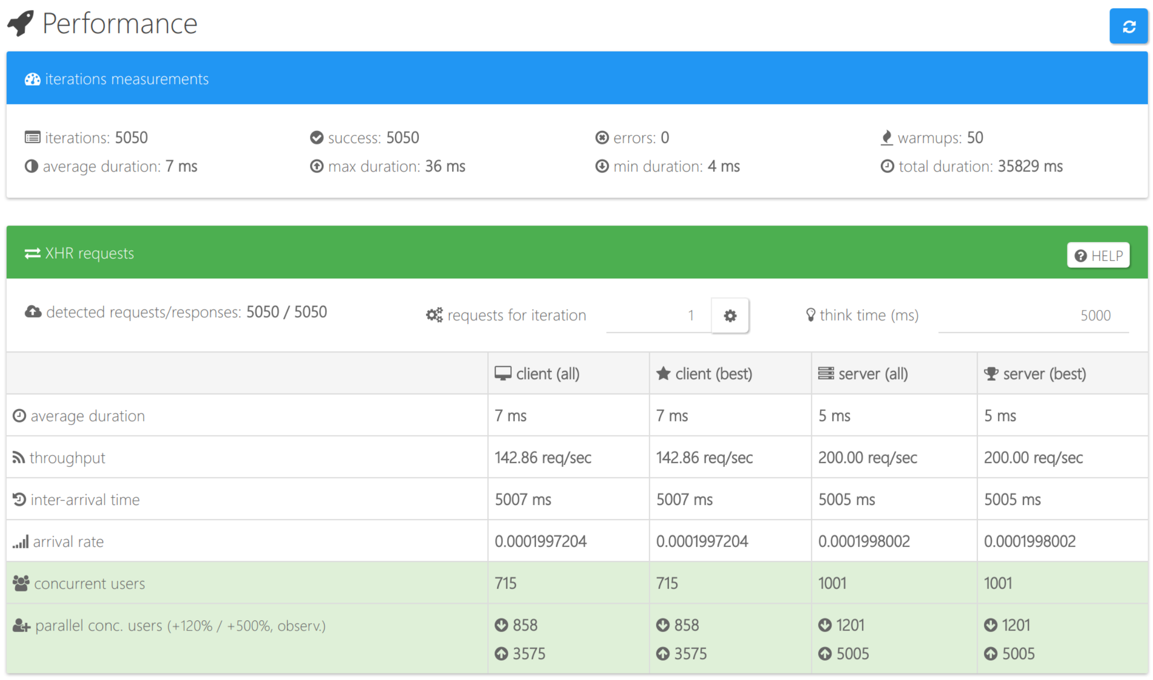


Durations

iterations: 5050 | max duration: 36 ms | min duration: 4 ms | total duration: 35829 ms
client average (all): 7 ms | client average (best): 7 ms | server average (all): 5 ms | server average (best): 5 ms

Figure 7.24: Monitoring Catalog - Load Test 2 Chart

Figure 7.25 summarizes the results where the most important ones are the average duration and the throughput as perceived the the client, 7ms and 142.86 req/sec respectively.



iterations measurements

iterations: 5050 | success: 5050 | errors: 0 | warmups: 50
average duration: 7 ms | max duration: 36 ms | min duration: 4 ms | total duration: 35829 ms

XHR requests

detected requests/responses: 5050 / 5050 | requests for iteration 1 | think time (ms) 5000

	client (all)	client (best)	server (all)	server (best)
average duration	7 ms	7 ms	5 ms	5 ms
throughput	142.86 req/sec	142.86 req/sec	200.00 req/sec	200.00 req/sec
inter-arrival time	5007 ms	5007 ms	5005 ms	5005 ms
arrival rate	0.0001997204	0.0001997204	0.0001998002	0.0001998002
concurrent users	715	715	1001	1001
parallel conc. users (+120% / +500%, observ.)	858 3575	858 3575	1201 5005	1201 5005

Figure 7.25: Monitoring Catalog - Load Test 2 Performance

These results are very good and, once again, well beyond the requirements of the project, and the practical result is that the Aggregation Configuration Manager should have no trouble at all fetching the information it requires from the Monitoring Catalog.

7.3.3 Aggregation Configuration Manager

As explained in the introduction of this section, series of timestamp variables were carefully placed within the source code of the components of the ACM to collect information about the times spent in specific operations performed either by the Master Service or/and the Client Services.

The load test consists in sending messages to the Kafka topic the Master Service is watching on, simulating the operations performed by the Monitoring Catalog by using the CREATE and DELETE operation types and the threshold groups or rules groups IDs present in the Monitoring Catalog, respecting the format of the messages as specified in table 6.2.

A total of 5000 messages were sent (1250 x 4), where half of them were operations related to threshold rules, and the other half were related to aggregation rules, where each rule was created and deleted afterwards. The iteration sequence was as follows:

1. Create a threshold rule;
2. Delete the threshold rule;
3. Create a batch aggregation rule;
4. Delete the batch aggregation rule.

It must be mentioned though that, when it comes to the ACM's Client Services, only the Monasca Client and the AE Client services were subject to this load test, because the CEP Client is undergoing a refactoring process at the moment and it is not possible to perform load tests on it.

With that in mind, table 7.1 shows the results of the load test reported by the ACM's Master Service:

Service	Avg. Time / Operation	Elapsed Time
Monitoring Catalog	4.91ms	0m 36s
MongoDB	0.44ms	0m 04s
Zookeeper	3.86ms	0m 22s
Overall	9.86ms	1m 07s

Number of Operations : 5000, Avg. Operations / sec: 75

Table 7.1: ACM Master Service - Load Test Results

The first thing to observe is that the Monitoring Catalog and Zookeeper operations comprise most of the processing time by the Master Service, although the Monitoring Catalog's takes the biggest share.

The second one is that the average time of the operations on the Monitoring Catalog is roughly 5ms, which falls under the values obtained for the second load test of the Monitoring Catalog as seen in figure 7.25.

And the third one is that the average number of operations per second that the Master Service can handle is 75. If we take in consideration the results of the first load test performed on the Monitoring Catalog (figure 7.22), it is fair to assume that the ACM's Master Service is able to cope with the performance of the Monitoring Catalog, as its average throughput values are within the same order of magnitude.

Next, table 7.2 and 7.2 show the results of the load test reported by the ACM's Monasca and Aggregation Engine Clients:

Service	Avg. Time / Operation	Elapsed Time
Monasca	86.49ms	3m 08s
MongoDB	1.87ms	0m 18s
Zookeeper	2.79ms	0m 13s
Overall	93.91ms	3m 39s

Number of Operations : 2500, Avg. Operations / sec: 11

Table 7.2: ACM Monasca Client - Load Test Results

Service	Avg. Time / Operation	Elapsed Time
Aggregation Engine	77.58ms	2m 57s
MongoDB	2.52ms	0m 16s
Zookeeper	4.95ms	0m 14s
Overall	91.57ms	3m 27s

Number of Operations : 2500, Avg. Operations / sec: 12

Table 7.3: ACM AE Client - Load Test Results

This time around, the operations on the coordination service and the database have little impact on the overall performance of these components. It is clearly seen that the operations performed on their end services, Monasca and the Aggregation Engine respectively, comprise most of the processing time.

If we compare the number of average operations per second of the Monasca Client with the results obtained from the Monasca second load test (figure 7.19), although the value 11 is (comparably) a bit lower, it is close to the throughput values obtained.

The fact is, as stated before, Monasca is currently operating 24/7 on the testbed under heavy load, which affects the results of these load tests, and the same can be said about the Aggregation Engine. A load test was not performed on the latter and shown here, as the AE performance is out of the scope of this work.

Nevertheless, the average number of operations per second obtained for the ACM's Client Services is enough to cope with the requirements of the SELFNET project. It is not expected, throughout the course of the project, to have such a high demand of requests per second on both Monasca and the Aggregation Engine.

Given that, it can be said that, although the ACM's Master Service has no trouble at all to cope with the performance and operating rate supported by the Monitoring Catalog, the ACM's Client Services are the components that may limit the performance of the ACM service as a whole.

There is no loss of data if the end services of the ACM cannot handle a very high rate of requests imposed by this service, but in the end its performance may be compromised or limited by the very own performance of the end services it is connected to.

7.4 Conclusions

The SELFNET's testbed proved to be invaluable to assess both the integration between the components of the architecture as well their performances. Although some of the components are currently revealing under-optimized performances, this is a consequence of the heavy load they are suffering together with the under allocation of computational resources they are allowed to use.

Nevertheless some conclusions could be drawn. First of all the Self-Protection was correctly addressed by all the components of the architecture, and it can be said that it was a success. It was already lively presented to a European board of reviewers of the project, and the feedback was quite positive.

Secondly the Monitoring Catalog surpassed the expected performance taking into account that it is not foreseeable in this project to have a need for such a high rate of requests per second from this component.

Finally, although the performance of the ACM's Client Services could not be properly assessed, the obtained results reveal that the overall performance of the ACM should be more than enough for the requirements of the project.

CHAPTER 8

Conclusion and Future Work

This dissertation presented a research work done on the paradigm of virtualized and programmable networks (NFV/SDN) focusing on the performance management, supervision and monitoring domains. Current performance management platforms in production have a static nature where the configuration processes of the information models are enforced manually.

With the advent of the upcoming 5G next-generation of mobile networks where Software-Defined Networking (SDN) together with Network Functions Virtualization (NFV) are going to play key roles, the network operators will no longer be able to rely solely on these manually controlled solutions as a consequence of the dynamics they impose.

The SELFNET project provided the necessary research platform and environment to develop proof of concepts for these new requirements, allowing Altice Labs to understand how its Altaia performance management platform must evolve in order to be prepared for this next generation mobile networks in a not so distant future.

The research work presented here focused particularly on two areas of the SELFNET project: the storage of aggregated metrics that were produced on the sensing data gathered by the sensors deployed on the network, as well as the thresholding capabilities required to take advantage of the collected information, and how the network platform should aggregate the sensing information itself resorting to the enforcement of aggregation and thresholding rules based on a catalog-driven architecture approach.

Starting with the storage of aggregated metrics and the thresholding capabilities of the SELFNET project, Monasca was chosen as an off-the-shelf solution. This choice is due to the fact that it was a mature open-source project under the umbrella of OpenStack (which was used as virtualization infrastructure solution) that featured three particular components on its architecture that covered the requirements of the project on this matter, i.e. a time-series database to store the aggregated metrics, a threshold engine for the alarming/thresholding service and notification engine for the alarm notification requirements. Moreover, the time and funding constraints of the project would not allow the development of a new and customized solution built from scratch.

Performance wise, Monasca proved to be reliable enough for its job, being able to cope with the rate of aggregated metrics submitted by the Aggregation Engine throughout the months it as been operating 24/7 on the testbed, as it was shown by the results presented in section 7.3.1. Its Threshold Engine has also reliably done its job, providing the necessary mathematical/statistical functions required for the project in order to create the thresholds that need to be addressed by the use cases, particularly the Self-Protection one.

However, the load test results have shown that the submission of thresholds on Monasca have a higher average duration with consequent lower throughput when compared to the submission of metrics. However, if we consider that the number of thresholds that need to be created is not as higher as the need to create metrics, it did not proved to be a limiting factor for the project.

Where it somewhat fell short was on the Notification Engine, as it did not provide the required notification methods for the project. Nevertheless, as Monasca is an open-source project, its source code is readily available to be modified and/or extended. Moreover, the notification methods already natively supported by Notification Engine were devised by its original developers as plugins or extensions, and this proved to be crucial on the project, allowing the development of a new plugin to circumvent the existing gap and effectively address the requirements of the project, allowing the Thresholds Engine to produce alarm notifications towards the Autonomic layer and respecting the devised format of the Raw Aggregation Data Model.

All in all, Monasca proved to be a wise and adequate solution for the project, but it must be pointed out that its performance was limited by the available computational resources of the VM where it was deployed. Although not addressed in the results presented in this document, monitoring done on both the VM and the individual Docker containers deployed on it, revealed that on one hand the Threshold Engine and the InfluxDB time-series database were very resource intensive services, and that although the containerization of the services was not itself a problem (quite the opposite I must say), having all those containers deployed on one same VM it was too much for it to handle.

On the other hand, the high load imposed by the Monasca services on the VM resources, as a consequence of the high throughput of metrics it was being submitted to, affected the load tests done on the Aggregation Configuration Manager. In the future a better balancing on the deployment of the Monasca services must be achieved to off-load the two computationally more intensive services, which would allow Monasca to have a better overall performance as a whole, and to better assess the performance results both on Monasca itself as well on the ACM's Monasca Client.

Moving on to the Monitoring Catalog, its main goal was to address the catalog-driven needs of the project's architecture. The devised model of information (the MCDM) effectively addressed the information content and structure that needed to be enforced on the Aggregation framework. It contemplated both the batch/stream aggregation rules to be enforced on the AE and CEP Engines and the thresholding rules to be enforced on Monasca.

It also contemplated the sensor information model needed both by the aggregation and thresholding rules that were part of the catalog it self, but also required for the Orchestration services. Although the latter is out of the scope of the presented work, the integration of this part of the Monitoring Catalog will be done in the upcoming weeks.

Performance wise, the Monitoring Catalog proved to have very good results (section 7.3.2), above the expectations and requirements of the project. The average time of the operations as well the number of request per second on the catalog revealed values that could without a doubt cope with the demands of the Autonomic layer.

Although not foreseen in a near future, even though if the obtained performance proves to not be enough for the requirements of the Autonomic layer, the nature of the Monitoring Catalog as a RESTful service would allow an horizontal scalability solution where multiple instances of the Catalog could be put in place together with a Load Balancer, which would provide a reliable solution, and with higher performance, to cope with the increased performance demands.

At last but not the least, the Aggregation Configuration Manager provided a bridging service between the Monitoring Catalog and the Aggregation framework, effectively "translating" the rules onboarded on the Monitoring Catalog and enforcing them on the components of the Aggregation framework, i.e the AE Engine, the CEP Engine and Monasca, depending on the types of rules that were onboarded. It was one of the last components to be developed and integrated on the Aggregation layer, and as a result it is the least mature service of them and the one with more room for improvement in the future.

Load tests on the ACM's CEP Client were not performed and shown on this document, as it is currently undergoing a refactoring process, with results only being provided about the interaction of the ACM with the Aggregation Engine and Monasca. Nevertheless, it is expected that the ACM's CEP Client will present results within the ones assessed on the other two clients, and the reasoning behind this is the same for the other two.

Just as the Aggregation Engine and Monasca are currently under heavy loads, which affects the assessment of proper load tests on their ACM's Client services, the same will happen with ACM's CEP Client, as it is known that the VM where it is currently deployed is also under heavy load.

Every project has its own funding and resources limitations, and SELFNET is no exception on that matter. For the time being there is nothing that can be done on this matter, and without the possibility of allocating more computational resources on the affected components, a proper assessment of the ACM's Client Services cannot be done.

However, the results on the load tests done on the ACM's Master service show that it is able to cope with the maximum rate imposed by the Monitoring Catalog, which shows that the ACM's Client Services are the components of the ACM that can cripple the overall performance of this service.

This service is only on its first iteration of development and integration on the project and the assessments, although compromised, show that it has room for improvement. Its current architecture only allows one service (master or client) to be run at a time, i.e. one Master Service and one Client Service of each type, with possible standby services to take over the role of the running ones in case of failure. One possible change on the architecture that could improve its overall performance would be the overcoming of this limitation, to allow multiple instances of the Client Services to be run simultaneously.

If this would be an effective solution it is yet to be determined, after all, so long the end point services which the Client Services must communicate with still operate on the edge of their available resources, no matter the good performance the ACM might have, it will always be compromised/limited by the performance of the end point services themselves.

With that said, all in all the three components which were the focus of this thesis generally fulfilled their purposes and the requirements of the SELFNET project, and they can cope with the demanded throughput, despite their limitations. There is room for improvement, it is a fact, but considering that they are prototypes to assert the validation of the proposed and designed architecture of the project, it is acceptable to say that they are not required to have stellar performances.

Furthermore, the involvement of Altice Labs on the SELFNET project is above all, a means to assess new technological solutions that could be integrated on its business performance management platform solution, meaning that new components would have to be developed from scratch that would incorporate the knowledge acquired throughout the involvement on this European project, and this time around with possible performance issues already being taken care of right from the inception stage. In other words, the research work done for the SELFNET project is not the end but rather just the starting point for future work to be done on Altaia.

As a closing point, Altice Labs will also be participating on another European project denominated SLICENET [9], that evolves from the base work of SELFNET. If on one hand the participation on the SELFNET project allowed Altice Labs to acquire knowledge on the aggregation and orchestration layers (there was also another team involved on the latter), on the other hand the transition to this new project will allow Altice Labs to acquire knowledge around the intelligence of the future networks. Using figure 4.1 presented on the Architecture chapter as a support, Altice labs was able to acquire knowledge on most of the presented high level components of the SELFNET project (directly or indirectly) except on the Autonomic layer, i.e. where the intelligence of the network itself is found. Altice Labs' participation on the SLICENET project will then provide the means to close this gap, where the focus will be on the development of challenging proof-of-concept solutions to infer and diagnose potential anomalies in NFV/SDN environments with the ability to react proactively and automatically with the aid of Machine Learning algorithms.

Finally, a personal note on my participation on the SELFNET project. It is my opinion that these kind of research projects are not only important for the advancements on new technologies, allowing them to thrive and drive the future, but also to connect together resourceful and skilled people with the most various backgrounds and areas of expertise, which in the end is what makes possible any advancement either technologically, scientifically, academically and even in the human society.

With that in mind, I would like to thank the people who provided me the opportunity to be part of this project, which certainly allowed me to further improve my skills and knowledge, and to grow up not only professionally as well as a person and human being.

Appendices

APPENDIX A

Monitoring Catalog Specification

A.1 OpenAPI YAML Specification

(This is only an excerpt of the full specification file which is over 1700 lines of code)

```
swagger: '2.0'
info:
  title: 'Monitoring Catalog'
  description: SELFNET's Monitoring Catalog RESTful API
  version: '1.0.0'
  contact:
    email: 'nuno-m-henriques@alticelabs.com'
schemes:
  - 'http'
basePath: '/v1.0'
produces:
  - 'application/json'
consumes:
  - 'application/json'

paths:
  '/catalog':
    get:
      description: Retrieve the full catalog
      tags:
        - 'Catalog'
      operationId: "catalog_get"
      x-swagger-router-controller: 'controllers.catalog'
      responses:
        '200':
          $ref: '#/responses/catalog_get'
        '503':
          $ref: '#/responses/service_unavailable'
        'default':
          $ref: '#/responses/internal_server_error'

  # some endpoints omitted here

  '/catalog/monitoring/sensors':
    get:
      description: Retrieve the full list of sensors
    tags:
      - 'Sensors'
    operationId: "sensors_get"
    x-swagger-router-controller: 'controllers.sensors'
    responses:
      '200':
        $ref: '#/responses/sensors_get'
      '404':
        $ref: '#/responses/not_found'
      '503':
        $ref: '#/responses/service_unavailable'
      'default':
        $ref: '#/responses/internal_server_error'
  post:
    description: Add a new sensor to the catalog
    tags:
      - 'Sensors'
    parameters:
      - $ref: '#/parameters/sensor_post'
    operationId: "sensor_post"
    x-swagger-router-controller: 'controllers.sensors'
    responses:
      '200':
        $ref: '#/responses/sensor_post'
      '400':
        $ref: '#/responses/bad_request'
      '404':
        $ref: '#/responses/not_found'
      '503':
        $ref: '#/responses/service_unavailable'
      'default':
        $ref: '#/responses/internal_server_error'

'/catalog/monitoring/sensors/{sensor_id_or_name}':
  get:
    description: Retrieve a specific sensor from the catalog
    tags:
      - 'Sensors'
    parameters:
      - $ref: '#/parameters/sensor_id_or_name'
    operationId: "sensor_get"
    x-swagger-router-controller: 'controllers.sensors'
    responses:
      '200':
        $ref: '#/responses/sensor_get'
      '404':
        $ref: '#/responses/not_found'
      '503':
        $ref: '#/responses/service_unavailable'
      'default':
        $ref: '#/responses/internal_server_error'
  delete:
    description: Remove a sensor from the catalog
    tags:
      - 'Sensors'
    parameters:
      - $ref: '#/parameters/sensor_id_or_name'
      operationId: "sensor_delete"
      x-swagger-router-controller: 'controllers.sensors'
      responses:
        '204':
          $ref: '#/responses/no_content'
        '404':
          $ref: '#/responses/not_found'
        '503':
          $ref: '#/responses/service_unavailable'
        'default':
          $ref: '#/responses/internal_server_error'
    patch:
      description: Update a specific sensor of the catalog
      tags:
        - 'Sensors'
      parameters:
        - $ref: '#/parameters/sensor_id_or_name'
        - $ref: '#/parameters/sensor_patch'
      operationId: "sensor_patch"
      x-swagger-router-controller: 'controllers.sensors'
      responses:
        '200':
          $ref: '#/responses/sensor_patch'
        '404':
          $ref: '#/responses/not_found'
        '503':
          $ref: '#/responses/service_unavailable'
        'default':
          $ref: '#/responses/internal_server_error'

  # remaining endpoints omitted

tags:
  - name: 'Catalog'
    description: Catalog root endpoints and operations
  - name: 'Monitoring'
    description: Monitoring endpoints and operations
  - name: 'Sensors'
    description: Sensors endpoints and operations
  - name: 'Rules Groups'
    description: Rules Groups endpoints and operations
  - name: 'Thresholding'
    description: Thresholding endpoints and operations


parameters:
  sensor_id_or_name:
    description: 'The name or ID of the sensor'
    name: sensor_id_or_name
    in: path
    type: string
    required: true

  # some parameters omitted here

  sensor_post:
    description: "The json of the 'Sensor'"
    name: data
    in: body
    schema:
      $ref: '#/definitions/parameter_sensor_post'
    required: true

  sensor_patch:
    description: "The json of the 'Sensor' delta"
    name: data
    in: body
    schema:
      $ref: '#/definitions/parameter_sensor_patch'

  resource_type_post:
    description: "The json of the 'Resource Type'"
    name: data
    in: body
    schema:
      $ref: '#/definitions/parameter_resource_type_post'
    required: true

  resource_type_patch:
    description: "The json of the 'Resource Type' delta"
    name: data
    in: body
    schema:
      $ref: '#/definitions/parameter_resource_type_patch'
    required: true

  data_type_post:
    description: "The json of the 'Data Type'"
    name: data
    in: body
    schema:
      $ref: '#/definitions/parameter_data_type_post'
    required: true

  data_type_patch:
    description: "The json of the 'Data Type' delta"
    name: data
    in: body
    schema:
      $ref: '#/definitions/parameter_data_type_patch'
    required: true

  # remaining parameters omitted

responses:
  bad_request:
    description: 'The submitted json data is invalid'
    schema:
      $ref: '#/definitions/response_error'

  conflict:
    description: 'The specified resource already exists'
    schema:
      $ref: '#/definitions/response_error'
```

```yaml
internal_server_error:
  description: 'Unexpected error'
  schema:
    $ref: '#/definitions/response_error'

no_content:
  description: 'The server successfully processed the request and is not returning any
   ↪  content'

not_found:
  description: 'The specified resource was not found'
  schema:
    $ref: '#/definitions/response_error'

service_unavailable:
  description: 'A critical service is unavailable'
  schema:
    $ref: '#/definitions/response_error'

catalog_get:
  description: TODO
  schema:
    $ref: '#/definitions/response_catalog_get'

# some responses omitted here

sensors_get:
  description: TODO
  schema:
    type: array
    items:
      $ref: '#/definitions/response_sensor_get'
    minItems: 0

sensor_get:
  description: TODO
  schema:
    $ref: '#/definitions/response_sensor_get'

sensor_post:
  description: "The ID of the new 'Sensor'"
  schema:
    $ref: '#/definitions/response_post'

sensor_patch:
  description: "The number of properties updated on the 'Sensor'"
  schema:
    $ref: '#/definitions/response_patch'

resource_types_get:
  description: TODO
  schema:
    type: array
    items:
      $ref: '#/definitions/response_resource_type_get'
    minItems: 0
  resource_type_get:
    description: TODO
    schema:
      $ref: '#/definitions/response_resource_type_get'

  resource_type_post:
    description: "The ID of the new 'Resource Type'"
    schema:
      $ref: '#/definitions/response_post'

  resource_type_patch:
    description: "The number of properties updated on the 'Resource Type'"
    schema:
      $ref: '#/definitions/response_patch'

  data_types_get:
    description: TODO
    schema:
      type: array
      items:
        $ref: '#/definitions/response_data_type_get'
      minItems: 0

  data_type_get:
    description: TODO
    schema:
      $ref: '#/definitions/response_data_type_get'

  data_type_post:
    description: "The ID of the new 'Data Type'"
    schema:
      $ref: '#/definitions/response_post'

  data_type_patch:
    description: "The number of properties updated on the 'Data Type'"
    schema:
      $ref: '#/definitions/response_patch'

  # remaining responses omitted

definitions:
  parameter_sensor_post:
    description: TODO
    type: object
    properties:
      name:
        type: string
      description:
        type: string
      resource_types:
        type: array
        items:
          $ref: '#/definitions/parameter_resource_type_post'
        minItems: 1
    required: ["name", "description", "resource_types"]
    additionalProperties: false
```

```
parameter_sensor_patch:
  description: TODO
  type: object
  properties:
    name:
      type: string
    description:
      type: string
    resource_types:
      type: array
      items:
        $ref: '#/definitions/parameter_resource_type_patch'
      minItems: 0
  additionalProperties: false

parameter_resource_type_post:
  description: TODO
  type: object
  properties:
    name:
      type: string
    data_types:
      type: array
      items:
        $ref: '#/definitions/parameter_data_type_post'
      minItems: 1
  required: ["name", "data_types"]
  additionalProperties: false

parameter_resource_type_patch:
  description: TODO
  type: object
  properties:
    id:
      type: string
      readOnly: true
    name:
      type: string
      readOnly: true
    data_types:
      type: array
      items:
        $ref: '#/definitions/parameter_data_type_patch'
      minItems: 1
  additionalProperties: false

parameter_data_type_post:
  description: TODO
  type: object
  properties:
    name:
      type: string
    identifiers:
      type: array
      items:
        type: string
      minItems: 2
    resource_description:
      type: array
      items:
        type: string
      minItems: 1
    data_definition:
      type: array
      items:
        type: string
      minItems: 1
    reporter_description:
      type: array
      items:
        type: string
      minItems: 1
  required: ["name",  "identifiers", "resource_description", "data_definition",
  ↪  "reporter_description"]
  additionalProperties: false

parameter_data_type_patch:
  description: TODO
  type: object
  properties:
    id:
      type: string
      readOnly: true
    name:
      type: string
      readOnly: true
    identifiers:
      type: array
      items:
        type: string
      minItems: 0
    resource_description:
      type: array
      items:
        type: string
      minItems: 0
    data_definition:
      type: array
      items:
        type: string
      minItems: 0
    reporter_description:
      type: array
      items:
        type: string
      minItems: 0
  additionalProperties: false

# some definitions omitted here

response_catalog_get:
  description: TODO
  type: object
  properties:
    monitoring:
      $ref: '#/definitions/response_monitoring_get'
    thresholding:
      $ref: '#/definitions/response_thresholding_get'
  required: ["monitoring", "thresholding"]
  additionalProperties: false

response_monitoring_get:
  description: TODO
  type: object
  properties:
    sensors:
      type: array
      items:
        $ref: '#/definitions/response_sensor_get'
      minItems: 0
    rules_groups:
      type: array
      items:
        $ref: '#/definitions/response_rules_group_get'
      minItems: 0
  required: ["sensors", "rules_groups"]

response_sensor_get:
  description: TODO
  type: object
  properties:
    id:
      type: string
    name:
      type: string
    description:
      type: string
    resource_types:
      type: array
      items:
        $ref: '#/definitions/response_resource_type_get'
      minItems: 0
  required: ["id", "name", "resource_types"]
  additionalProperties: false
  readOnly: true

response_resource_type_get:
  description: TODO
  type: object
  properties:
    id:
      type: string
    name:
      type: string
    data_types:
      type: array
      items:
        $ref: '#/definitions/response_data_type_get'
      minItems: 0
  required: ["id", "name", "data_types"]
  additionalProperties: false
  readOnly: true

response_data_type_get:
  description: TODO
  type: object
  properties:
    id:
      type: string
    name:
      type: string
    identifiers:
      type: array
      items:
        type: string
      minItems: 2
    resource_description:
      type: array
      items:
        type: string
      minItems: 1
    data_definition:
      type: array
      items:
        type: string
      minItems: 1
    reporter_description:
      type: array
      items:
        type: string
      minItems: 1
  required: ["id", "name",  "identifiers", "resource_description", "data_definition",
   ↪  "reporter_description"]
  additionalProperties: false
  readOnly: true

# some definitions omitted here

response_post:
  description: TODO
  type: object
  properties:
    id:
      type: string
      example: '839c3126-d6a0-41c5-95ac-603c15d68ffd'
  required: ['id']
  additionalProperties: false

response_patch:
  description: TODO
  type: object
  properties:
    nr_of_changes:
      type: integer
      format: int32
  required: ['nr_of_changes']
  additionalProperties: false
```

```yaml
response_error:
  description: TODO
  type: object
  properties:
    error:
      type: string
      example: 'A message explaining the details of the error'
  required: ['error']
  additionalProperties: false
```

APPENDIX B

Monitoring Catalog Unit Tests Examples

B.1 Database Interface

```
/usr/bin/python3.6 /opt/pycharm-professional/helpers/pycharm/_jb_pytest_runner.py --path /home/nunoh/
Projects/PycharmProjects/MonitoringCatalogAPI/tests/db_neo4j_test.py -- -v -x --color=yes
Testing started at 6:14 PM ...
Launching py.test with arguments -v -x --color=yes /home/nunoh/Projects/PycharmProjects/
MonitoringCatalogAPI/tests/db_neo4j_test.py in /home/nunoh/Projects/PycharmProjects/
MonitoringCatalogAPI/tests

============================== test session starts ===============================
platform linux -- Python 3.6.3, pytest-3.2.5, py-1.5.0, pluggy-0.5.3.dev -- /usr/bin/python3.6
cachedir: .cache
rootdir: /home/nunoh/Projects/PycharmProjects/MonitoringCatalogAPI/tests, inifile:
collecting ... collected 89 items
db_neo4j_test.py::test_init_catalog PASSED
db_neo4j_test.py::test_fetch_catalog_empty PASSED
db_neo4j_test.py::test_fetch_monitoring_empty PASSED
db_neo4j_test.py::test_fetch_thresholding_empty PASSED
db_neo4j_test.py::test_fetch_sensors_empty PASSED
db_neo4j_test.py::test_fetch_sensor_notfound PASSED
db_neo4j_test.py::test_fetch_rulesgroups_empty PASSED
db_neo4j_test.py::test_fetch_rulesgroup_notfound PASSED
db_neo4j_test.py::test_create_sensor_0 PASSED
db_neo4j_test.py::test_create_sensor_0_conflict PASSED
db_neo4j_test.py::test_fetch_sensor_0 PASSED
db_neo4j_test.py::test_fetch_sensors_0 PASSED
db_neo4j_test.py::test_create_sensor_1 PASSED
db_neo4j_test.py::test_create_sensor_2 PASSED
db_neo4j_test.py::test_fetch_sensor_1 PASSED
db_neo4j_test.py::test_delete_sensor_1 PASSED
db_neo4j_test.py::test_delete_sensor_1_notfound PASSED
db_neo4j_test.py::test_fetch_sensor_1_notfound PASSED
db_neo4j_test.py::test_fetch_sensor_2 PASSED
db_neo4j_test.py::test_fetch_sensors_0_2 PASSED
db_neo4j_test.py::test_recreate_sensor_1 PASSED
db_neo4j_test.py::test_refetch_sensor_1 PASSED
db_neo4j_test.py::test_fetch_sensors PASSED
db_neo4j_test.py::test_patch_sensor_fma PASSED
```

```
db_neo4j_test.py::test_fetch_sensor_fma PASSED
db_neo4j_test.py::test_repatch_sensor_fma PASSED
db_neo4j_test.py::test_fetch_resourcetypes_sensor_0 PASSED
db_neo4j_test.py::test_fetch_resourcetype_1_sensor_0 PASSED
db_neo4j_test.py::test_delete_resourcetype_0_sensor_0 PASSED
db_neo4j_test.py::test_delete_resourcetype_0_sensor_0_notfound PASSED
db_neo4j_test.py::test_fetch_resourcetype_0_sensor_0_notfound PASSED
db_neo4j_test.py::test_create_resourcetype_0_sensor_0 PASSED
db_neo4j_test.py::test_create_resourcetype_0_sensor_0_conflict PASSED
db_neo4j_test.py::test_fetch_resourcetype_0_sensor_0 PASSED
db_neo4j_test.py::test_patch_resourcetype_0_sensor_0 SKIPPED
Skipped: patchResourceType method not fully implemented yet

db_neo4j_test.py::test_fetch_datatypes_rtype_0_sensor_0 PASSED
db_neo4j_test.py::test_fetch_datatype_0_rtype_0_sensor_0 PASSED
db_neo4j_test.py::test_patch_datatype_0_rtype_0_sensor_0 SKIPPED
Skipped: patchDataType method not fully implemented yet

db_neo4j_test.py::test_delete_datatype_0_rtype_0_sensor_0 PASSED
db_neo4j_test.py::test_delete_datatype_0_rtype_0_sensor_0_notfound PASSED
db_neo4j_test.py::test_fetch_datatype_0_rtype_0_sensor_0_notfound PASSED
db_neo4j_test.py::test_create_datatype_0_rtype_0_sensor_0 PASSED
db_neo4j_test.py::test_refetch_datatype_0_rtype_0_sensor_0 PASSED
db_neo4j_test.py::test_create_rulesgroup_0 PASSED
db_neo4j_test.py::test_fetch_rulesgroup_0 PASSED
db_neo4j_test.py::test_fetch_rulesgroups PASSED
db_neo4j_test.py::test_delete_rulesgroup_0 PASSED
db_neo4j_test.py::test_fetch_rulesgroup_0_notfound PASSED
db_neo4j_test.py::test_recreate_rulesgroup_0 PASSED
db_neo4j_test.py::test_refetch_rulesgroup_0 PASSED
db_neo4j_test.py::test_refetch_rulesgroups PASSED
db_neo4j_test.py::test_patch_rulesgroup_0 SKIPPED
Skipped: patchRulesGroup method not fully implemented yet

db_neo4j_test.py::test_fetch_dimensions_rulesgroup_0 PASSED
db_neo4j_test.py::test_fetch_dimension_1_rulesgroup_0 PASSED
db_neo4j_test.py::test_delete_dimension_1_rulesgroup_0 PASSED
db_neo4j_test.py::test_delete_dimension_1_rulesgroup_0_notfound PASSED
db_neo4j_test.py::test_fetch_dimension_1_rulesgroup_0_notfound PASSED
db_neo4j_test.py::test_create_dimension_1_badrulesgroup_id_notfound PASSED
db_neo4j_test.py::test_create_dimension_1_rulesgroup_0 PASSED
db_neo4j_test.py::test_create_dimension_1_rulesgroup_0_conflict PASSED
db_neo4j_test.py::test_patch_dimension_2_rulesgroup_0 SKIPPED
Skipped: patchDimension method not fully implemented yet

db_neo4j_test.py::test_fetch_rules_rulesgroup_0 PASSED
db_neo4j_test.py::test_fetch_rule_1_rulesgroup_0 PASSED
db_neo4j_test.py::test_delete_rule_1_rulesgroup_0 PASSED
db_neo4j_test.py::test_delete_rule_1_rulesgroup_0_notfound PASSED
db_neo4j_test.py::test_fetch_rule_1_rulesgroup_0_notfound PASSED
db_neo4j_test.py::test_create_rule_1_badrulesgroup_id_notfound PASSED
db_neo4j_test.py::test_create_rule_1_rulesgroup_0 PASSED
db_neo4j_test.py::test_create_rule_1_rulesgroup_0_conflict PASSED
db_neo4j_test.py::test_patch_rule_2_rulesgroup_0 SKIPPED
Skipped: patchRule method not fully implemented yet

db_neo4j_test.py::test_create_thresholdsgroup_0 PASSED
```

```
db_neo4j_test.py::test_fetch_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_fetch_thresholdsgroups PASSED
db_neo4j_test.py::test_delete_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_fetch_thresholdsgroup_0_notfound PASSED
db_neo4j_test.py::test_recreate_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_refetch_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_refetch_thresholdsgroups PASSED
db_neo4j_test.py::test_fetch_thresholds_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_fetch_threshold_0_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_delete_threshold_0_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_delete_threshold_0_thresholdsgroup_0_notfound PASSED
db_neo4j_test.py::test_fetch_threshold_0_thresholdsgroup_0_notfound PASSED
db_neo4j_test.py::test_create_threshold_0_badthresholdsgroup_id_notfound PASSED
db_neo4j_test.py::test_create_threshold_0_thresholdsgroup_0 PASSED
db_neo4j_test.py::test_create_threshold_0_thresholdsgroup_0_conflict PASSED
db_neo4j_test.py::test_fetch_monitoring PASSED
db_neo4j_test.py::test_fetch_thresholding PASSED
db_neo4j_test.py::test_fetch_catalog PASSED

===================== 84 passed, 5 skipped in 1.25 seconds =====================
Process finished with exit code 0
```

B.2 REST API

```
/usr/bin/python3.6 /opt/pycharm-professional/helpers/pycharm/_jb_pytest_runner.py --path /home/nunoh/
Projects/PycharmProjects/MonitoringCatalogAPI/tests/mcat_api_test.py -- -v -x --color=yes
Testing started at 5:02 PM ...
Launching py.test with arguments -v -x --color=yes /home/nunoh/Projects/PycharmProjects/
MonitoringCatalogAPI/tests/mcat_api_test.py in /home/nunoh/Projects/PycharmProjects/
MonitoringCatalogAPI/tests

============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.2.5, py-1.5.0, pluggy-0.5.3.dev -- /usr/bin/python3.6
cachedir: .cache
rootdir: /home/nunoh/Projects/PycharmProjects/MonitoringCatalogAPI/tests, inifile:
collecting ... collected 88 items
mcat_api_test.py::test_fetch_catalog_empty PASSED
mcat_api_test.py::test_fetch_monitoring_empty PASSED
mcat_api_test.py::test_fetch_thresholding_empty PASSED
mcat_api_test.py::test_fetch_sensors_empty PASSED
mcat_api_test.py::test_fetch_sensor_notfound PASSED
mcat_api_test.py::test_fetch_rulesgroups_empty PASSED
mcat_api_test.py::test_fetch_rulesgroup_notfound PASSED
mcat_api_test.py::test_create_sensor_0 PASSED
mcat_api_test.py::test_create_sensor_0_conflict PASSED
mcat_api_test.py::test_fetch_sensor_0 PASSED
mcat_api_test.py::test_fetch_sensors_0 PASSED
mcat_api_test.py::test_create_sensor_1 PASSED
mcat_api_test.py::test_create_sensor_2 PASSED
mcat_api_test.py::test_fetch_sensor_1 PASSED
mcat_api_test.py::test_delete_sensor_1 PASSED
mcat_api_test.py::test_delete_sensor_1_notfound PASSED
mcat_api_test.py::test_fetch_sensor_1_notfound PASSED
mcat_api_test.py::test_fetch_sensor_2 PASSED
mcat_api_test.py::test_fetch_sensors_0_2 PASSED
mcat_api_test.py::test_recreate_sensor_1 PASSED
mcat_api_test.py::test_refetch_sensor_1 PASSED
mcat_api_test.py::test_fetch_sensors PASSED
mcat_api_test.py::test_patch_sensor_fma PASSED
mcat_api_test.py::test_fetch_sensor_fma PASSED
mcat_api_test.py::test_repatch_sensor_fma PASSED
mcat_api_test.py::test_fetch_resourcetypes_sensor_0 PASSED
mcat_api_test.py::test_fetch_resourcetype_1_sensor_0 PASSED
mcat_api_test.py::test_delete_resourcetype_0_sensor_0 PASSED
mcat_api_test.py::test_delete_resourcetype_0_sensor_0_notfound PASSED
mcat_api_test.py::test_fetch_resourcetype_0_sensor_0_notfound PASSED
mcat_api_test.py::test_create_resourcetype_0_sensor_0 PASSED
mcat_api_test.py::test_create_resourcetype_0_sensor_0_conflict PASSED
mcat_api_test.py::test_fetch_resourcetype_0_sensor_0 PASSED
mcat_api_test.py::test_patch_resourcetype_0_sensor_0 SKIPPED
Skipped: patchResourceType method not fully implemented yet

mcat_api_test.py::test_fetch_datatypes_rtype_0_sensor_0 PASSED
mcat_api_test.py::test_fetch_datatype_0_rtype_0_sensor_0 PASSED
mcat_api_test.py::test_patch_datatype_0_rtype_0_sensor_0 SKIPPED
Skipped: patchDataType method not fully implemented yet

mcat_api_test.py::test_delete_datatype_0_rtype_0_sensor_0 PASSED
mcat_api_test.py::test_delete_datatype_0_rtype_0_sensor_0_notfound PASSED
mcat_api_test.py::test_fetch_datatype_0_rtype_0_sensor_0_notfound PASSED
mcat_api_test.py::test_create_datatype_0_rtype_0_sensor_0 PASSED
mcat_api_test.py::test_refetch_datatype_0_rtype_0_sensor_0 PASSED
mcat_api_test.py::test_create_rulesgroup_0 PASSED
mcat_api_test.py::test_fetch_rulesgroup_0 PASSED
mcat_api_test.py::test_fetch_rulesgroups PASSED
mcat_api_test.py::test_delete_rulesgroup_0 PASSED
mcat_api_test.py::test_fetch_rulesgroup_0_notfound PASSED
mcat_api_test.py::test_recreate_rulesgroup_0 PASSED
mcat_api_test.py::test_refetch_rulesgroup_0 PASSED
mcat_api_test.py::test_refetch_rulesgroups PASSED
mcat_api_test.py::test_patch_rulesgroup_0 SKIPPED
Skipped: patchRulesGroup method not fully implemented yet

mcat_api_test.py::test_fetch_dimensions_rulesgroup_0 PASSED
mcat_api_test.py::test_fetch_dimension_1_rulesgroup_0 PASSED
mcat_api_test.py::test_delete_dimension_1_rulesgroup_0 PASSED
mcat_api_test.py::test_delete_dimension_1_rulesgroup_0_notfound PASSED
mcat_api_test.py::test_fetch_dimension_1_rulesgroup_0_notfound PASSED
mcat_api_test.py::test_create_dimension_1_badrulesgroup_id_notfound PASSED
mcat_api_test.py::test_create_dimension_1_rulesgroup_0 PASSED
mcat_api_test.py::test_create_dimension_1_rulesgroup_0_conflict PASSED
mcat_api_test.py::test_patch_dimension_2_rulesgroup_0 SKIPPED
Skipped: patchDimension method not fully implemented yet

mcat_api_test.py::test_fetch_rules_rulesgroup_0 PASSED
mcat_api_test.py::test_fetch_rule_1_rulesgroup_0 PASSED
mcat_api_test.py::test_delete_rule_1_rulesgroup_0 PASSED
mcat_api_test.py::test_delete_rule_1_rulesgroup_0_notfound PASSED
mcat_api_test.py::test_fetch_rule_1_rulesgroup_0_notfound PASSED
mcat_api_test.py::test_create_rule_1_badrulesgroup_id_notfound PASSED
mcat_api_test.py::test_create_rule_1_rulesgroup_0 PASSED
mcat_api_test.py::test_create_rule_1_rulesgroup_0_conflict PASSED
mcat_api_test.py::test_patch_rule_2_rulesgroup_0 SKIPPED
```

```
Skipped: patchRule method not fully implemented yet

mcat_api_test.py::test_create_thresholdsgroup_0 PASSED
mcat_api_test.py::test_fetch_thresholdsgroup_0 PASSED
mcat_api_test.py::test_fetch_thresholdsgroups PASSED
mcat_api_test.py::test_delete_thresholdsgroup_0 PASSED
mcat_api_test.py::test_fetch_thresholdsgroup_0_notfound PASSED
mcat_api_test.py::test_recreate_thresholdsgroup_0 PASSED
mcat_api_test.py::test_refetch_thresholdsgroup_0 PASSED
mcat_api_test.py::test_refetch_thresholdsgroups PASSED
mcat_api_test.py::test_fetch_thresholds_thresholdsgroup_0 PASSED
mcat_api_test.py::test_fetch_threshold_0_thresholdsgroup_0 PASSED
mcat_api_test.py::test_delete_threshold_0_thresholdsgroup_0 PASSED
mcat_api_test.py::test_delete_threshold_0_thresholdsgroup_0_notfound PASSED
mcat_api_test.py::test_fetch_threshold_0_thresholdsgroup_0_notfound PASSED
mcat_api_test.py::test_create_threshold_0_badthresholdsgroup_id_notfound PASSED
mcat_api_test.py::test_create_threshold_0_thresholdsgroup_0 PASSED
mcat_api_test.py::test_create_threshold_0_thresholdsgroup_0_conflict PASSED
mcat_api_test.py::test_fetch_monitoring PASSED
mcat_api_test.py::test_fetch_thresholding PASSED
mcat_api_test.py::test_fetch_catalog PASSED

===================== 83 passed, 5 skipped in 2.32 seconds =====================
Process finished with exit code 0
```

APPENDIX C

Data Models Examples

C.1 Raw Aggregation Data Model Examples

C.1.1 Flow Monitoring Agent

```
{
   "Data":[
      {
         "timestamp":1513012895097,
         "dataType":"statistics",
         "reporterID":"reporterHostName=pepe/reporterIP=10.4.1.2/reporterAppType=FMA",
         "resourceType":"FLOW",
         "resourceID":"flowLayer=0/flowHash=416BD408/l3Proto=2048/srcIP=10.0.255.14/
          ↪  destIP=10.0.255.11/l4Proto=6/srcPort=00000050/destPort=0000E75A/
          ↪  encapsulationLayer=0",
         "resourceDescription":{
            "flowLayer":0,
            "flowHash":"416BD408",
            "encapsulationLayer":0,
            "l3Proto":"2048",
            "srcIP":"10.0.255.14",
            "destIP":"10.0.255.11",
            "l4Proto":"6",
            "srcPort":"00000050",
            "destPort":"0000E75A",
            "packetStructure":"/mac:14/ip4:20/tcp:20",
            "completePacketStructure":"",
            "firstPacketSeen":1513012895097
         },
         "dataDefinition":{
            "totalOctets":443,
            "currentPktPerPeriod":4,
            "currentOutterOctetsPerPeriod":443,
            "totalpktCount":4,
            "currentOctetsPerPeriod":443,
            "timeLastPacketReceived":16673995869523385,
            "sumInterPacketLagNS":0,
            "totalOctetsOutter":443,
            "lastInterPacketRate":0,
            "timeLastPacketReceivedMs":1513012895097,
            "currentSumInterPacketLagNSPerPeriod":0
         },
         "reporterDescription":{
            "reporterHostName":"pepe",
            "reporterIP":"10.4.1.2",
            "reporterAppType":"FMA",
            "reporterEpochTime":1513012895097
         }
      }
   ]
}
```

C.1.2 Alarm Notification

```
{
  "reporterDescription":{
     "reporterName":"aggregation-alarm-notifier"
  },
  "reporterId":"reporterName=aggregation-alarm-notifier",
  "dataType":"alarm",
  "timestamp":1513013043897,
  "dataDefinition":{
     "severity":"CRITICAL",
     "newState":"ALARM",
     "oldState":"OK",
     "alarmDescription":"Potential Zombie Detected",
     "metrics":[
        {
           "dimensions":{
              "DestinationPort":"0000E752",
              "FlowID":"211B7062",
              "SourceIP":"10.0.255.14",
              "FlowState":"ACTIVE",
              "DestinationIP":"10.0.255.11"
           },
           "id":null,
           "name":"avg_pkt_count"
        },
        {
           "dimensions":{
              "DestinationPort":"0000E752",
              "FlowID":"211B7062",
              "SourceIP":"10.0.255.14",
              "FlowState":"ACTIVE",
              "DestinationIP":"10.0.255.11"
           },
           "id":null,
           "name":"comunication_frequency"
        }
     ],
     "alarmTimestamp":1513013043466,
     "subAlarms":[
        {
           "subAlarmExpression":{
              "function":"AVG",
              "deterministic":true,
              "period":180,
              "threshold":8.0,
              "periods":1,
              "operator":"LTE",
              "metricDefinition":{
                 "dimensions":{

                 },
                 "id":null,
                 "name":"avg_pkt_count"
              }
           },
      "currentValues":[
         3.0
      ],
      "subAlarmState":"ALARM"
   },
   {
      "subAlarmExpression":{
         "function":"AVG",
         "deterministic":true,
         "period":180,
         "threshold":8.0,
         "periods":1,
         "operator":"LTE",
         "metricDefinition":{
            "dimensions":{

            },
            "id":null,
            "name":"comunication_frequency"
         }
      },
      "currentValues":[
         5.0
      ],
      "subAlarmState":"ALARM"
   },
   {
      "subAlarmExpression":{
         "function":"AVG",
         "deterministic":true,
         "period":180,
         "threshold":2.0,
         "periods":1,
         "operator":"GTE",
         "metricDefinition":{
            "dimensions":{

            },
            "id":null,
            "name":"avg_pkt_count"
         }
      },
      "currentValues":[
         3.0
      ],
      "subAlarmState":"ALARM"
   },
   {
      "subAlarmExpression":{
         "function":"AVG",
         "deterministic":true,
         "period":180,
         "threshold":2.0,
         "periods":1,
         "operator":"GTE",
         "metricDefinition":{
            "dimensions":{

            },
            "id":null,
            "name":"comunication_frequency"
         }
      },
          "currentValues":[
             5.0
          ],
          "subAlarmState":"ALARM"
       }
    ]
 },
 "resourceType":"aggregation-alarm-notification",
 "resourceDescription":{
    "notificationId":"8b868466-c827-48c2-920b-e14f7ab7189b",
    "alarmDefinitionName":"ZombieAlert",
    "alarmId":"cc0543de-b3ee-4887-b81c-46ad016512b1",
    "alarmDefinitionId":"05cb86e4-a2c7-4d01-ab9a-7ef573d6ceb8",
    "tenantId":"9446514f9ef24b61a17e38f4b612c210"
 },
 "resourceId":"tenantId=9446514f9ef24b61a17e38f4b612c210/
  ↪   alarmDefinitionId=05cb86e4-a2c7-4d01-ab9a-7ef573d6ceb8/
  ↪   alarmId=cc0543de-b3ee-4887-b81c-46ad016512b1"
}
```

C.2 Monitoring Catalog Data Model Example

```
{
  "monitoring":
  {
    "sensors":
    [
      {
        "name":"FMA",
        "description":"Flow Management Sensor",
        "resource_types": [
          {
            "name": "FLOW",
            "data_types": [
              {
                "name": "statistics",
                "identifiers": [
                  "resourceID",
                  "reporterID"
                ],
                "resource_description": [
                  "flowLayer",
                  "flowHash",
                  "encapsulationLayer",
                  "l3Proto",
                  "packetStructure",
                  "completePacketStructure",
                  "firstPacketSeen",
                  "srcIP",
                  "destIP",
                  "l4Proto",
                  "srcPort",
                  "destPort"
                ],
                "data_definition": [
                  "totalOctets",
                  "currentPktPerPeriod",
                  "currentOutterOctetsPerPeriod",
                  "totalpktCount",
                  "currentOctetsPerPeriod",
                  "timeLastPacketReceived",
                  "sumInterPacketLagNS",
                "totalOctetsOutter",
                "lastInterPacketRate",
                "timeLastPacketReceivedMs",
                "currentSumInterPacketLagNSPerPeriod"
              ],
              "reporter_description": [
                "reporterHostName",
                "reporterIP",
                "reporterAppType",
                "reporterEpochTime"
              ]
            },
            {
              "name": "event",
              "identifiers": [
                "resourceID",
                "reporterID"
              ],
              "resource_description": [
                "flowLayer",
                "flowHash",
                "encapsulationLayer",
                "l3Proto",
                "srcIP",
                "destIP",
                "l4Proto",
                "srcPort",
                "destPort",
                "packetStructure",
                "completePacketStructure",
                "firstPacketSeen",
                "flowLabel"
              ],
              "data_definition": [
                "flowState",
                "payloadType",
                "temporalLayerID",
                "layerID"
              ],
              "reporter_description": [
                "reporterHostName",
                "reporterIP",
                "reporterAppType",
                "reporterEpochTime"
              ]
            }
          ]
        }
      ]
    }
  ],
  "rules_groups":
  [
    {
      "name": "SelfProtection-Batch",
      "sensor_ref_list":
      [
        "FMA"
      ],
      "period": 30,
      "dimensions":
      [
        {
```

```
    "name": "SourceIP",
    "source_ref": "FMA.FLOW.event.srcIP"
  },
  {
    "name": "DestinationIP",
    "source_ref": "FMA.FLOW.event.destIP"
  },
  {
    "name": "DestinationPort",
    "source_ref": "FMA.FLOW.event.destPort"
  },
  {
    "name": "FlowID",
    "source_ref": "FMA.FLOW.event.FlowHash"
  },
  {
    "name": "FlowState",
    "source_ref": "FMA.FLOW.event.FlowState"
  }
],
"rules":
[
  {
    "name": "avg_pkt_count",
    "group_by":
    [
      "FMA.FLOW.event.srcIP",
      "FMA.FLOW.event.destIP",
      "FMA.FLOW.event.destPort",
      "FMA.FLOW.event.FlowHash"
    ],
    "filters":
    [
      "FMA.FLOW.event.flowLayer == 0",
      "FMA.FLOW.event.srcIP != null",
      "FMA.FLOW.event.destIP != null"
    ],
    "formula": "AVG(Sensor.FMA.FLOW.statistics.currentPktPerPeriod)",
    "metadata":
    {
      "entityType": "SPI"
    }
  },
  {
    "name": "communication_frequency",
    "group_by":
    [
      "FMA.FLOW.event.srcIP",
      "FMA.FLOW.event.destIP",
      "FMA.FLOW.event.destPort",
      "FMA.FLOW.event.FlowHash"
    ],
    "filters":
    [
      "FMA.FLOW.event.flowLayer == 0",
      "FMA.FLOW.event.srcIP != null",
      "FMA.FLOW.event.destIP != null"
    ],
    "formula": "COUNT(Sensor.FMA.statistics.FLOW.destIP)",
    "metadata":
    {
    }
  },
          {
            "name": "metric_on_metric_example",
            "group_by":
            [
              "FMA.FLOW.event.srcIP",
              "FMA.FLOW.event.destIP",
              "FMA.FLOW.event.destPort",
              "FMA.FLOW.event.FlowHash"
            ],
            "filters":
            [
              "FMA.FLOW.event.flowLayer == 0",
              "FMA.FLOW.event.srcIP != null",
              "FMA.FLOW.event.destIP != null"
            ],
            "formula": "AVG(Sensor.FMA.FLOW.statistics.currentOuterOctets) +
             ↪  SUM(Rule.avg_pkt_count)",
            "metadata":
            {
            }
          }
        ]
      }
    ]
  },
  "thresholding":
  {
    "thresholds_groups":
    [
      {
        "name": "SelfProtection-Batch-Thresholds",
        "thresholds":
        [
          {
            "name": "ZombieAlert",
            "description": "Possible Zombie Detected",
            "severity": 3,
            "state_notifications": ["alarm"],
            "match_by":
            [
              "SelfProtection-Batch.SourceIP",
              "SelfProtection-Batch.DestinationIP",
              "SelfProtection-Batch.DestinationPort"
            ],
            "expression": "AVG(SelfProtection-Batch.avg_pkt_count, deterministic, 180) >= 2
             ↪  AND AVG(SelfProtection-Batc.comunication_frequency, deterministic, 180) >=
             ↪  6",
            "metadata":
            {
              "extra_info":"metadata test"
            }
          }
        ]
      }
    ]
  }
}
```

References

[1] Vodafone. (2017). Creating a GigabitSociety - The role of 5G, [Online]. Available: https://www.vodafone.com/content/dam/vodafone-images/public-policy/reports/pdf/gigabit-society-5g-04042017.pdf (visited on Dec. 2017).

[2] N. Yoshikane. (2017). Possible network parameters on IMT-2020/5G transport network, [Online]. Available: https://www.itu.int/en/ITU-T/Workshops-and-Seminars/20171016/Documents/Yoshikane.pdf (visited on Dec. 2017).

[3] 5G-PPP. (2015–2017). SELFNET - Framework for Self-Organized Network Management in Virtualized and Software Defined Networks, [Online]. Available: https://selfnet-5g.eu (visited on Dec. 2017).

[4] European Commission. (2017). HORIZON 2020 - The EU Framework Programme for Research and Innovation, [Online]. Available: https://ec.europa.eu/programmes/horizon2020/ (visited on Dec. 2017).

[5] 5G-PPP. (2014). The 5G Infrastructure Public Private Partnership (5G-PPP), [Online]. Available: https://5g-ppp.eu/ (visited on Dec. 2017).

[6] Altice Labs. (2016). Altaia - End-to-end assurance solution, [Online]. Available: http://www.alticelabs.com/content/products/BR_Altaia_ALB_EN.pdf (visited on Oct. 2017).

[7] UNIFY Consortium. (2013). UNIFY - Cloud and Carrier Networks, [Online]. Available: http://www.t-nova.eu/ (visited on Dec. 2017).

[8] T-NOVA Consortium. (2014). T-NOVA - Network Functions as-a-Service Over Virtualized Infrastructures, [Online]. Available: http://www.t-nova.eu/ (visited on Dec. 2017).

[9] 5G-PPP. (2017). SLICENET - End-to-End Cognitive Network Slicing and Slice Management Framework in Virtualised Multi-Domain, Multi-Tenant 5G Networks, [Online]. Available: https://slicenet.eu/ (visited on Dec. 2017).

[10] 5G NORMA Consortium. (2015). 5G NORMA - Novel Radio Multiservice adaptive network Architecture, [Online]. Available: https://5gnorma.5g-ppp.eu/ (visited on Dec. 2017).

[11] G. Poulios, K. Tsagkaris, P. Demestichas, A. Tall, Z. Altman, and C. Destre, *Autonomics and SDN for Self-Organizing Networks*, International Workshop on Self-Organizing Networks, 2014. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01067258/document (visited on Dec. 2017).

[12] UniverSelf Consortium. (2013). UniverSelf Website, [Online]. Available: http://www.univerself-project.eu/ (visited on Dec. 2017).

[13] A. Basta, W. Kellerer, and M. Hoffmann, *A Virtual SDN-Enabled LTE EPC Architecture: A Case Study for S-/P-Gateways Functions*, 2013 IEEE SDN for Future Networks and Services (SDN4FNS), 2013. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6702532/ (visited on Dec. 2017).

[14] J. Sanchez, I. G. B. Yahia, and N. Crespi, *Self-healing Mechanisms for Software Defined Networks*, 8th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2014), 2014. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01068045/document (visited on Dec. 2017).

[15] Ruggeri F., Faltin F. and Kenett R. (2007). Bayesian Networks, [Online]. Available: http://www.eng.tau.ac.il/~bengal/BN.pdf (visited on Dec. 2017).

[16] J. Sanchez, I. G. B. Yahia, N. Crespi, T. Rasheed, and D. Siracusa, *Softwarized 5G networks resiliency with self-healing*, 1st International Conference on 5G for Ubiquitous Connectivity (IEEE), 2014. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7041059/ (visited on Dec. 2017).

[17] P. Demestichas, A. Georgakopoulos, D. Karvounas, K. Tsagkaris, V. Stavroulaki, J. Lu, C. Xiong, and J. Yao, *5G on the Horizon: Key Challenges for the Radio-Access Network*, IEEE Vehicular Technology Magazine (Volume: 8, Issue: 3, Sept. 2013), 2013. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6568922/ (visited on Dec. 2017).

[18] A. Aissioui, A. Ksentini, A. M. Gueroui, and T. Taleb, *Toward Elastic Distributed SDN/NFV Controller for 5G Mobile Cloud Management Systems*, IEEE Access (Volume: 3), 2015. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7296589 (visited on Dec. 2017).

[19] N. Nikaein, E. Schiller, R. Favraud, K. Katsalis, D. Stavropoulos, I. Alyafawi, Z. Zhao, T. Braun, and T. Korakis, *Network Store: Exploring Slicing in Future 5G Networks*, MobiArch '15 Proceedings of the 10th International Workshop on Mobility in the Evolving Internet Architecture, 2015. [Online]. Available: https://dl.acm.org/citation.cfm?id=2795390 (visited on Dec. 2017).

[20] A. Farshad, P. Georgopoulos, M. Broadbent, M. Mu, and N. Race, *Leveraging SDN to provide an in-network QoE measurement framework*, Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE, 2015. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6963800/ (visited on Dec. 2017).

[21] Cisco Systems. (2017). Cisco Tetration Website, [Online]. Available: https://www.cisco.com/c/en/us/products/data-center-analytics/tetration-analytics/index.html#~stickynav=1 (visited on Dec. 2017).

[22] ZTE. (2017). ZTE Network Performance Management, [Online]. Available: http://www.zte.com.cn/global/services/categories/unicare/products/413041 (visited on Dec. 2017).

[23] MYCOM OSI. (2017). PrOptima - Network Performance Management, [Online]. Available: http://www.mycom-osi.com/products/proptima/service-and-network-performance-management (visited on Dec. 2017).

[24] F. Junqueira and B. Reed, *ZooKeeper - Distributed Process Coordination.* O'Reilly Media, Inc, Nov. 2013, ch. 2, p. 18, ISBN: 978-1-449-36130-3. [Online]. Available: https://www.amazon.com/ZooKeeper-Distributed-Coordination-Flavio-Junqueira-ebook/dp/B00GRCODKS (visited on Oct. 2017).

[25] ——, *ZooKeeper - Distributed Process Coordination.* O'Reilly Media, Inc, Nov. 2013, ch. 2, p. 24, ISBN: 978-1-449-36130-3. [Online]. Available: https://www.amazon.com/ZooKeeper-Distributed-Coordination-Flavio-Junqueira-ebook/dp/B00GRCODKS (visited on Oct. 2017).

[26] M. Grover. (2013). ZooKeeper fundamentals, deployment, and applications, [Online]. Available: https://www.ibm.com/developerworks/library/bd-zookeeper/index.html (visited on Oct. 2017).

[27] HashiCorp. (2017). Introduction to Consul, [Online]. Available: https://www.consul.io/intro/index.html (visited on Oct. 2017).

[28] ——, (2017). Consul Architecture, [Online]. Available: https://www.consul.io/docs/internals/architecture.html (visited on Oct. 2017).

[29] Apache Software Foundation. (2017). Kafka Introduction, [Online]. Available: https://kafka.apache.org/intro (visited on Oct. 2017).

[30] Pivotal. (2017). RabbitMQ, [Online]. Available: http://www.rabbitmq.com/ (visited on Oct. 2017).

[31] OASIS. (2017). AMQP Website, [Online]. Available: https://www.amqp.org/ (visited on Dec. 2017).

[32] M. Quadri. (2017). Apache Kafka vs RabbitMQ - Message Queue Comparison, [Online]. Available: `www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/` (visited on Nov. 2017).

[33] Hewlett-Packard Enterprise Development LP. (2017). Monasca Architecture, [Online]. Available: `http://monasca.io/docs/architecture.html` (visited on Oct. 2017).

[34] ——, (2017). Monasca Threshold, [Online]. Available: `https://github.com/openstack/monasca-thresh` (visited on Oct. 2017).

[35] OpenStack. (2017). Ceilometer's Documentation, [Online]. Available: `https://docs.openstack.org/ceilometer/latest/` (visited on Oct. 2017).

[36] ——, (2017). OpenStack Telemetry, [Online]. Available: `https://wiki.openstack.org/wiki/Telemetry` (visited on Oct. 2017).

[37] ——, (2017). Ceilometer's System Architecture, [Online]. Available: `https://docs.openstack.org/ceilometer/latest/contributor/architecture.html` (visited on Oct. 2017).

[38] Apache Software Foundation. (2015). Apache Storm Website, [Online]. Available: `http://storm.apache.org/` (visited on Nov. 2017).

[39] A. Jain, *Mastering Apache Storm*. Packt Publishing, Aug. 2017, ch. 1, ISBN: 978-1-78712-563-6. [Online]. Available: `https://www.amazon.com/Mastering-Apache-Storm-Real-time-streaming/dp/1787125637#reader_B074M54PXB` (visited on Nov. 2017).

[40] Apache Software Foundation. (2015). Apache Hadoop Website, [Online]. Available: `http://hadoop.apache.org/` (visited on Nov. 2017).

[41] Yahoo! (2015). Scaling Apache Storm (Hadoop Summit 2015), [Online]. Available: `https://www.slideshare.net/RobertEvans26/scaling-apache-storm-hadoop-summit-2015` (visited on Nov. 2017).

[42] Hortonworks. (2017). Processing Trucking IoT Data with Apache Storm - Building a Storm Topology, [Online]. Available: `https://hortonworks.com/hadoop-tutorial/processing-trucking-iot-data-with-apache-storm/#section_4` (visited on Nov. 2017).

[43] Trifacta. (2017). Trifacta Website - Wrangler Enterprise, [Online]. Available: `https://www.trifacta.com/products/wrangler-enterprise/` (visited on Nov. 2017).

[44] ——, (2017). Trifacta Website - Wrangler Architecture, [Online]. Available: `https://www.trifacta.com/products/architecture/` (visited on Nov. 2017).

[45] MongoDB, Inc. (2017). MongoDB Website - Aggregation, [Online]. Available: `https://docs.mongodb.com/manual/aggregation/` (visited on Nov. 2017).

[46] ——, (2017). MongoDB Website - Group Aggregation, [Online]. Available: `https://docs.mongodb.com/manual/reference/operator/aggregation/group/#pipe._S_group` (visited on Nov. 2017).

[47] ——, (2017). MongoDB Website - Pipeline Aggregation Stages, [Online]. Available: `https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/` (visited on Nov. 2017).

[48] ——, (2017). MongoDB Website - Aggregation Pipeline Limits, [Online]. Available: `https://docs.mongodb.com/manual/core/aggregation-pipeline-limits/` (visited on Nov. 2017).

[49] InfluxData. (2017). InfluxDB, [Online]. Available: `https://www.influxdata.com/time-series-platform/influxdb/` (visited on Oct. 2017).

[50] P. Dix. (2014). Why InfluxDB is written in Go, [Online]. Available: `https://blog.gopheracademy.com/birthday-bash-2014/why-influxdb-uses-go/` (visited on Dec. 2017).

[51] MongoDB, Inc. (2017). MongoDB Website, [Online]. Available: `https://www.mongodb.com/` (visited on Nov. 2017).

[52] Neo4j, Inc. (2017). Neo4j Website, [Online]. Available: `https://neo4j.com/` (visited on Nov. 2017).

[53] S. Krasser, J. Grizzard, and H. Owen. (2004). The Use of Honeynets to Increase Computer Network Security and User Awareness, [Online]. Available: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.7131&rep=rep1&type=pdf` (visited on Dec. 2017).

[54] SANS Institute. (2001). Honey Pots and Honey Nets - Security through Deception, [Online]. Available: `https://www.sans.org/reading-room/whitepapers/attacking/honey-pots-honey-nets-security-deception-41` (visited on Dec. 2017).

[55] Hewlett-Packard Enterprise Development LP. (2017). Monasca API Specification, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md` (visited on Nov. 2017).

[56] ——, (2017). Monasca API - List Metrics, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-metrics` (visited on Nov. 2017).

[57] ——, (2017). Monasca API - List Names, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-namesh` (visited on Nov. 2017).

[58] ——, (2017). Monasca API - List Dimension Values, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-dimension-values` (visited on Nov. 2017).

[59] ——, (2017). Monasca API - List Dimension Names, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-dimension-names` (visited on Nov. 2017).

[60] ——, (2017). Monasca API - List Measurements, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-measurements` (visited on Nov. 2017).

[61] ——, (2017). Monasca API - List Statistics, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-statistics` (visited on Nov. 2017).

[62] ——, (2017). Monasca API - List Alarms, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-alarms` (visited on Nov. 2017).

[63] ——, (2017). Monasca API - List Alarms State History, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-alarms-state-history` (visited on Nov. 2017).

[64] ——, (2017). Monasca API - List Alarm State History, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-alarm-state-history` (visited on Nov. 2017).

[65] ——, (2017). Monasca API - Create Metric, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#create-metric` (visited on Nov. 2017).

[66] ——, (2017). Monasca API - Create Notification Method, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#create-notification-method` (visited on Nov. 2017).

[67] ——, (2017). Monasca API - List Notification Methods, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-notification-methods` (visited on Nov. 2017).

[68] ——, (2017). Monasca API - Create Alarm Definition, [Online]. Available: `https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#create-alarm-definition` (visited on Nov. 2017).

[69] ——, (2017). Monasca API - List Alarm Definitions, [Online]. Available: https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#list-alarm-definitions (visited on Nov. 2017).

[70] ——, (2017). Monasca API - Delete Alarm Definition, [Online]. Available: https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#delete-alarm-definition (visited on Nov. 2017).

[71] Cisco System, Inc. (2017). Snort Website, [Online]. Available: https://www.snort.org (visited on Nov. 2017).

[72] K. Scarfone and P. Mell, *Guide to Intrusion Detection and Prevention Systems (IDPS)*. National Institute of Standards and Technology, Feb. 2007, ch. 2, pp. 2–1. [Online]. Available: http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=50951 (visited on Nov. 2017).

[73] Cisco System, Inc. (2017). Snort: The World's Most Widely Deployed IPS Technology, [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/security/brief_c17-733286.html (visited on Nov. 2017).

[74] LibreNMS. (2017). LibreNMS Website, [Online]. Available: https://www.librenms.org/ (visited on Nov. 2017).

[75] Hewlett-Packard Enterprise Development LP. (2017). Monasca API - Alarm Definition Expressions, [Online]. Available: https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md#alarm-definition-expressions (visited on Nov. 2017).

[76] ——, (2017). Monasca API, [Online]. Available: https://github.com/openstack/monasca-api (visited on Nov. 2017).

[77] Falcon Project Community. (2017). Falcon - About, [Online]. Available: https://falconframework.org/#sectionAbout (visited on Nov. 2017).

[78] Hewlett-Packard Enterprise Development LP. (2017). Monasca API Classes, [Online]. Available: https://github.com/openstack/monasca-api/tree/master/monasca_api/api (visited on Nov. 2017).

[79] ——, (2017). Monasca Persister, [Online]. Available: https://github.com/openstack/monasca-persister (visited on Nov. 2017).

[80] LMAX Ltd. (2016). LMAX Exchange Open Source, [Online]. Available: https://www.lmax.com/disruptor/ (visited on Nov. 2017).

[81] Hewlett-Packard Enterprise Development LP. (2017). Monitoring Threshold Engine, [Online]. Available: https://github.com/openstack/monasca-thresh (visited on Nov. 2017).

[82] Apache Software Foundation. (2015). Apache Storm Tutorial, [Online]. Available: http://storm.apache.org/releases/current/Tutorial.html (visited on Nov. 2017).

[83] Hewlett-Packard Enterprise Development LP. (2017). Monasca Notification Engine, [Online]. Available: https://github.com/openstack/monasca-notification (visited on Nov. 2017).

[84] ——, (2017). Monasca Notification Engine - Plugins Source Code, [Online]. Available: https://github.com/openstack/monasca-notification/tree/master/monasca_notification/plugins (visited on Nov. 2017).

[85] ——, (2017). Monasca Notification Engine - Plugins Abstract Class, [Online]. Available: https://github.com/openstack/monasca-notification/blob/master/monasca_notification/plugins/abstract_notifier.py (visited on Nov. 2017).

[86] D. Powers and D. Arthur. (2017). Kafka Python client, [Online]. Available: https://github.com/dpkp/kafka-python (visited on Nov. 2017).

[87] Python Software Foundation. (2017). Logging facility for Python, [Online]. Available: https://docs.python.org/3/library/logging.html (visited on Nov. 2017).

[88] A. Ronacher. (2017). Flask Website, [Online]. Available: http://flask.pocoo.org/ (visited on Nov. 2017).

[89] Zalando SE. (2016). Connexion Website, [Online]. Available: https://jobs.zalando.com/tech/blog/meet-connexion-our-rest-framework-for-python/?gh_src=4n3gxh1 (visited on Nov. 2017).

[90] ——, (2015). Connexion's Documentation, [Online]. Available: http://connexion.readthedocs.io/en/latest/ (visited on Nov. 2017).

[91] ——, (2017). Connexion Source Code, [Online]. Available: https://github.com/zalando/connexion (visited on Nov. 2017).

[92] H. Krekel. (2017). Pytest Website, [Online]. Available: https://docs.pytest.org/en/latest/ (visited on Nov. 2017).

[93] OpenAPI Initiative. (2017). OpenAPI Specification Website, [Online]. Available: https://github.com/OAI/OpenAPI-Specification (visited on Nov. 2017).

[94] SmartBear Software. (2017). Swagger Website, [Online]. Available: https://swagger.io/ (visited on Nov. 2017).

[95] OpenAPI Initiative. (2017). OpenAPI Initiative Website, [Online]. Available: https://www.openapis.org/ (visited on Nov. 2017).

[96] 3GPP. (2017). The Evolved Packet Core, [Online]. Available: http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core (visited on Dec. 2017).

[97] Docker Inc. (2017). Docker Website - What is Docker?, [Online]. Available: https://www.docker.com/what-docker (visited on Dec. 2017).

[98] ——, (2017). Docker Website - Overview of Docker Compose, [Online]. Available: https://docs.docker.com/compose/overview/ (visited on Dec. 2017).

[99] ——, (2017). Docker Website - Docker container networking, [Online]. Available: https://docs.docker.com/engine/userguide/networking/ (visited on Dec. 2017).

[100] G. Hinton, N. Srivastava, and K. Swersky. (2014). Why do we need machine learning and what are neural networks?, [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec1.pdf (visited on Dec. 2017).

[101] Grafana Labs. (2017). Grafana Website, [Online]. Available: https://grafana.com/ (visited on Dec. 2017).

[102] M. Bussini. (2017). RESTful Stress Website, [Online]. Available: https://github.com/maurobussini/restful-stress (visited on Dec. 2017).