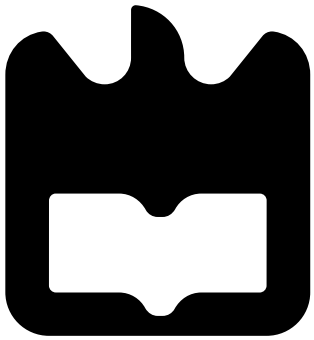
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

Rui Filipe Pedro

Gestão de Desempenho em Ambientes de Rede Virtualizados (NFV) e Programáveis (SDN)

Performance Management in Virtualized and Programmable Network Environments (NFV and SDN)



Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2017

Rui Filipe Pedro

Gestão de Desempenho em Ambientes de Rede Virtualizados (NFV) e Programáveis (SDN)

Performance Management in Virtualized and Programmable Network Environments (NFV and SDN)

"No one is dumb who is curious. The people who don't ask questions remain clueless throughout their lives."

— Neil deGrasse Tyson

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2017

Rui Filipe Pedro

Gestão de Desempenho em Ambientes de Rede Virtualizados (NFV) e Programáveis (SDN)

Performance Management in Virtualized and Programmable Network Environments (NFV and SDN)

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, Professor Associada com Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Pedro Miguel Naia Neves, Engenheiro de Telecomunicações na Altice Labs.

o júri / the jury

presidente / president

Prof. Doutor António Luís Jesus Teixeira
Professor Associado com Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)

vogais / examiners committee

Profa. Doutora Susana Sargento
Professora Associada com Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador)

Prof. Pedro Sousa
Professor Auxiliar do Departamento de Informática da Universidade do Minho (arguente)

agradecimentos / acknowledgements

Gostaria de agradecer em primeiro lugar aos meus orientadores, Professora Susana Sargento e Pedro Neves, pela grande oportunidade que me deram para desenvolver novas capacidades pessoais, académicas e profissionais, pela orientação rigorosa, pelo apoio e pelo espaço que me forneceram para decisões autónomas no que toca ao trabalho desenvolvido.

Quero também agradecer conjuntamente ao NAP, o grupo de investigação no qual me encontro, às equipas da Altice Labs, e aos membros do SELF-NET, com as quais eu tenho interagido, pelas discussões produtivas e fundamentais para a realização do meu trabalho.

Agradeço também a todos os meus amigos, pelos momentos de divertimento, de convívio, e especialmente de amizade, que me proporcionaram ao longo de todo o meu percurso académico e ao longo da minha vida.

Finalmente, e não de menos importância, um especial agradecimento a toda a minha família pelo esforo emocional e financeiro, pela educação que me deram e por todo o apoio que sempre me deram.

Palavras-chave 5G, SON, SDN, NFV, Virtualização, Monitorização, Agregação

Resumo

Mais do que tendências e domínios de conhecimento exploratórios, existe a forte convicção na indústria de que os paradigmas da virtualização das funções de rede (NFV Network Functions Virtualization) e das redes programáveis (SDN Software Defined Networking) vieram para ficar no mundo dos servios de telecomunicações. Para que possam navegar esta onda de mudança, os operadores terão que evoluir significativamente a arquitetura da sua rede, os seus mecanismos de gestão e, simultaneamente, o seu negócio. Esta Dissertação de Mestrado pretende contribuir para a evolução dos mecanismos de gestão operacional dos operadores, nomeadamente no domínio da supervisão/monitoria. Em concreto, o trabalho a desenvolver no âmbito desta Dissertação de Mestrado terá como principal objetivo a evolução da plataforma de performance management (Altaia) da Altice Labs para o novo paradigma de rede baseado nos conceitos de virtualização (NFV) e programabilidade (SDN). Importa ainda salientar que as atividades desenvolvidas no âmbito deste trabalho estaro enquadradas num projeto de I&D internacional financiado pela Comisso Europeia no âmbito do programa H2020 5G-PPP designado SELFNET (A Framework for Self-Organized Network Management in Virtualized and Software Defined Networks). A Altice Labs é um dos participantes no consórcio do SELFNET juntamente com outros 10 parceiros internacionais. O projeto tem um âmbito bastante abrangente e pretende enderear cenários de Self-Organizing Networks (SON) em contexto NFV/SDN. Uma das ferramentas essenciais para cenários SON é a deteção e predição de potenciais anomalias da rede e dos serviços. É neste contexto que a versão evoluída da plataforma Altaia será utilizada no projeto SELFNET. Esta dissertação propõem o desenvolvimento das ferramentas necessárias para modelar, persistir, e realizar processamento de dados provenientes da infraestrutura de rede em tempo real. Mais especificamente, esta dissertação desenvolveu: primeiramente o Raw and Aggregation Data Model, que unificou ambos dados brutos e agregados num único modelo, em segundo o Raw Data Loader, o componente que recebe dados da rede monitorada e os transforma de forma a serem persistidos, finalmente a Complex Event Processing Framework, uma estrutura de processamento em tempo real para processar dados usando uma abordagem dinâmica baseada em regras. Dos objectivos propostos todos foram desenvolvidos com sucesso. O Raw and Aggregation Data Model, juntamente com o Raw Data Loader, garantem que os sensors do SELFNET cumprem a especificação do modelo, unificando todos os dados de monitoria num único modelo. Além disso, a Complex Event Processing Framework foi posta em execução, carregada com regras de agregação relativas ao use case Self-Protection do SELFNET, e é capaz de providenciar, em tempo real, informação sobre botnets detetadas na rede. Do processo de desenvolvimento desta framework, surgiu um novo componente denominado de Configuration Manager, que gere dados que são partilhados usando serviços distribuídos de coordenação, usado para aplicar regras dinâmicas sobre a Complex Event Processing Framework. Este componente não foi apenas uma contribuição para o SELFNET mas também para a Altice Labs. Esta tarefa levou a que se percebesse qual é o papel das ferramentas de gestão de rede para as operadoras e para as novas gerações de rede.

Keywords 5G, SON, SDN, NFV, Virtualization, Monitoring, Aggregation

Abstract

More than tendencies and exploratory knowledge domains, there is a strong conviction in the industry that network function virtualization (NFV) and software-defined network (SDN) paradigms came here to stay in the telecommunication services world. So that operators can surf the waves of change, they will have to significantly change their network architecture, their management mechanisms, and, simultaneously , their business model. This Master Thesis intends to contribute to the operators operation management mechanisms evolution, namely in the supervision/monitoring domains. More concretely, the work to be developed in this Master Thesis, will have as its main objective the evolution of the performance management platform (Altaia) from Altice Labs to the new networking paradigm based on the NFV and SDN concepts. It is also important to outline that the activities to be developed in the scope of this work, will be aligned in an international R&D project, financed by the European Commission, covered by the H2020 5G-PPP program, that is SELFNET (A Framework for Self-Organized Network Management in Virtualized and Software Defined Networks). Altice Labs is one the participants in the SELFNET consortium, together with 10 more international partners. The project was a vast scope and intends to address Self-Organizing Network (SON) scenarios, and prediction over potential network and services prediction. It is in this context that the evolved version of Altaia will be introduced to the SELFNET project. This thesis proposes to develop the necessary tools to model, persist and perform real-time processing over network infrastructure data. More specifically, this thesis developed: firstly the Raw and Aggregation Data Model, that unified both raw and aggregated data under a single model, secondly the Raw Data Loader, a component that receives network sensed data and transforms it so it can be persisted, lastly the Complex Event Processing Framework, a real-time processing framework for processing data using a dynamic rule-based approach. Out of what it was proposed no objective was left behind, all components were successful developed. The Raw and Aggregation Data Model, together with the Raw Data Loader, enforced SELFNET sensors to follow this model, thus unifying all sensed data under a known model. Moreover, the Complex Event Processing Framework was put into to place, with aggregation rules relative to the SELFNET Self-Protection use case, and is able to provide, in real-time, information about detected botnets around the underlying network. From the development of this framework a new component emerged, a Configuration Manager that manages data to be shared using distributed coordination services, used to apply dynamic rules over the Complex Event Processing Framework. This component is not only a contribution for SELFNET but as well as for Altice Labs. In a more high-level point-of-view, this task brought a new understanding about the role of network management tools for network operators and next-generation networks.

Contents

List of Figures

List of Tables

Abbreviations

5G	Fifth generation networks
ACM	Aggregation configuration manager
AMN	Autonomic network management
AMQP	Advanced message queueing protocol
API	Application programming interface
BAF	Batch aggregation framework
BGP	Border gateway protocol
CDP	Cisco discovery protocol
CEP	Complex event processing
CEPF	Complex event processing framework
CI	Congestion index
CLI	Command line interface
CPU	Central processing unit
CQL	Cassandra query language
CRUD	Create, read, update and delete
DB	Data base
DC	Data center
DNS	Domain name system
DPI	Deep packet inspection
EPC	Evolved packet core
FMA	Flow monitoring agent
HAS	Hypertext transfer protocol adaptive streaming
HD	High definition
HDFS	Hadoop distributed file system
HTTP	Hypertext transfer protocol
IMQF	In-network quality of experience measurement framework
IMT	International mobile telecommunication system
IP	Internet protocol
IPTV	Internet protocol television
JSON	JavaScript object notation
JVM	Java virtual machine
LLDP	Link layer discovery protocol
LTE	Long term evolution
MAC	Media access control
MIT	Massachusetts Institute of Technology

NFV	Network functions virtualization
OSS	Operation support system
PNF	Physical network function
PoP	Point-of-presence
PPP	Public-private partnership
QHD	Quality high definition
QoE	Quality of experience
QoS	Quality of service
RAM	Random access memory
RAN	Radio access network
RDL	Raw data loader
REST	Representational state transfer
RPC	Remote call procedure
RTBI	Real-time business intelligence
SDM	Software-defined monitoring
SDN	Software-defined network
SDR	Software-defined radio
SH-UC	Self-healing use case
SLA	Service level agreement
SNMP	Simple network management protocol
SO-UC	Self-optimization use case
SON	Self-organizing network
SP-UC	Self-protection use case
SQL	Structured query language
SQM	Service quality management
UC	Use case
UHD	Ultra high definition
UI	User interface
UML	Unified modeling language
VCPU	Virtual central processing unit
VM	Virtual machine
VNF	Virtual network function
YAML	YAML ain't markup language

Chapter 1

Introduction

With the everlasting search for perfection that comes from the human thirst for knowledge, it is natural that so much effort is put into discovering new ways of doing what we do today, better. Computer networking is no exception of this, it follows the same path of evolution, which brings new technologies, paradigms and ways of solving problems.

As such phenomena continues to happen, we can already perceive that new next-generation networks are being idealized and researched. One proof of this is the fifth generation of mobile networks (5G), which brings novelty to the thriving world that is networking.

This document aims to research performance management in virtualized and programmable network environments (NFV and SDN, respectively), and starts by introducing its work context with Sections 1.1, 1.2 and 1.3. Having provided the environment of this thesis, Section 1.4 presents the motivation and objectives that drive this research, and lastly, Section 1.5 exposes this document outline, providing an overview of what is to come.

1.1 5G Networks

5G networks or fifth generation mobile networks, commonly abbreviated as 5G, are a next step beyond 4G/IMT-Advanced [1] network standards. 5G aims to surpass 4G capabilities by increasing data transfer rates, improving mobile broadband density, coverage, latency and signalling efficiency, supporting device-to-device communication, as well as reducing battery consumption on mobile devices and therefore enabling better Internet of Things (IoT) developments [2].

Given that, as of the date, there are not any 5G standards [1], it is a difficult task to define what exactly is 5G. However, its main goals can help us perceive that 5G will further extend the networks that we know today, giving space to new use cases [3] and user demands. An important aspect to realize from this evolution that is the 5G pursuit, is the flexibility it will bring to mobile networking by introducing network function virtualization (NFV) and software defined network (SDN) concepts, as well as the ability to build complex, heterogeneous and self-organized networks controlled by intelligent management systems. Ultimately, these advancements will most likely introduce new networking paradigms to the ecosystem.

1.2 ALTAIA

ALTAIA is the Altice Labs product for performance and QoS management for telecommunications networks and systems. It gives a telecommunications operator information about its underlying infrastructure to be used for problems detection and resolution as well as historical data analysis.

It is a performance management platform that supports a vast functionality set that allows the operator to perform measurements over 2G, 3G and 4G networks (e.g. LTE) and their network services (e.g. IPTV, High-Speed Internet). ALTAIA collects, processes and analyses data regarding non-virtualized (non-NFV) and non-programmable (non-SDN) resources and services. Moreover, it is a system that calculates an average of 100 million indicators per hour, being able to reach peak values of 1000 million indicators per hour. It is also deployed in a fully redundant architecture with disaster recovery on two different sites.

1.3 SELFNET

SELFNET is a European project [4] supported by the European Commission under the Horizon 2020 Program, that aims to design and implement an autonomic network management framework to achieve self-organizing capabilities in managing network infrastructures by automatically detecting and mitigating a range of common network problems that are currently being addressed in a manual fashion by network providers.

The self-organizing network (SON) capabilities aim to significantly reduce operational costs (OPEX) and improve user experience. SELFNET explores a smart integration of several state-of-the-art techonologies 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 capable of assisting network operators in key management tasks: automated network monitoring with automatic NFV applications deployment in order to facilitate system-wide awareness of Health of Network metrics to have a 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.

In order to build and validate such framework, SELFNET defines three different use cases designed to address the following major network management problems: Self-Healing, Self-Optimization and Self-Protection. These use cases are described in Section 1.3.1 and their architectural impact is further detailed in Section 4.3.

1.3.1 Use Cases

1.3.1.1 Self-Healing

The Self-Healing use case goal is to detect and predict common network failures in a 5G environment, exposing infrastructure and/or operation vulnerabilities, hardware and/or software failures, or power supply outages, in order to recover and heal the network reactively or preventively.

With this in mind, to reach this goal a Self-Healing analyzer will be used to infer Health of Network (HoN) metrics combined with Self-Healing diagnosis intelligence to predict potential

problems. Furthermore, it enables a decision making intelligence to infer proactive healing actions to apply on the network.

That said, these processes will bring innovations in two main areas: intelligent management capabilities to improve the QoE/Quality of Service (QoS) of 5G systems, and in infrastructure metrics and service level agreement (SLA) indicators to infer HoN metrics and implement context-aware decisions in 5G control plane.

1.3.1.2 Self-Optimization

This use case sets its goal in providing autonomic behaviours to automatically respond to actual and predicted QoE degradation, together with end-to-end proactive energy management for an optimized resource deployment across 5G networks.

That being the case, SELFNET monitoring and analysis tools will be used in order to observe or predict large video traffic loads, so that packet marking and intelligent encoding schemes, and self-adjusting traffic management mechanisms can be put in place to reduce video loss and delay.

Bringing these mechanisms will introduce innovations: new sensors, actuators and decision making logic to realize QoE-based video streaming and novel energy monitoring sensors to develop a global view on energy usage across the network.

1.3.1.3 Self-Protection

The Self-Protection use case aims to detect and mitigate cyber-attacks and restore security to 5G network traffic.

By deploying virtual network functions (VNF) such as: virtual traffic monitor/deep packet inspection (DPI), virtual threat management system, virtual honeynets, virtual intrusion protection system. These VNFs can be chained in different parts of the network (e.g. mobile access point, point-of-presence (PoP), in the network core).

Following this virtualized way of introducing dynamic functionalities to the network, the use case presents innovation in: new ways of deploying multi-tenant security services distributed across edge and core networks, and new business chances for network and service providers with the security as a service concept.

1.4 Motivation and Objectives

With the advancements on the next generation mobile networks (i.e. 5G) that bring into play new network environments by introducing network functions virtualization and software defined networks functionalities to the existing networks, it is natural that the existing software that allows operators to manage their networks and the services they provide must also evolve to support this new environment.

As such, one motivation behind this thesis is to research and create a proof of concept that will allow Altice Labs to understand how ALTAIA must be evolved in order to encompass a 5G network scenario. Furthermore, another import motivation factor is the SELFNET project which will provide a research platform that will allow to validate the proof of concept to be developed. This can only happen due to the overlapping of the subjects addressed by ALTAIA and SELFNET, since SELFNET aims to create an autonomic management framework for 5G networks, it also aims to provide a performance management system for 5G networks, this also being the next evolutionary step for ALTAIA.

The performance management system advancement task in a 5G scenario comprises data gathering, aggregation, correlation and analysis originated from virtualized and programmable network elements (NFV and SDN, respectively). This data can be processed in two different ways: batch (non-real-time processing) and streaming (real-time processing). In both scenarios, indicators will be produced in order to expose the services and network "health" (i.e. Health of Network Symptoms). These symptoms will be consumed by machine-learning algorithms for three scenarios: self-optimization, self-healing and self-protection.

The main objective of this work is to build a real-time performance management system that allows dynamic configurations in order to be able to ingest new gathered data from virtualized networks, due to the dynamic nature of virtualized networks, and to produce new indicators originated from autonomic nature, i.e. machine learning algorithms that will request new metrics to be calculated in order to learn better ways to model network symptoms and how to address them. Moreover, this main objective can be structured as follows:

1. Data gathering from virtualized and programmable services and network elements (NFV and SDN, respectively);
2. Streaming processing of the gathered data (real-time or near-real-time processing times) based on dynamic rules;
3. Provide indicators about the virtualized and non-virtualized network and services status per use case:
    - **Self-Protection:** bot-nets detection by aggregating deep packet inspection functions reports on individual bots detected;
    - **Self-Optimization:** provide QoE and QoS indicators by calculating congestion indexes and available bandwidth based on virtualized and non-virtualized network equipment counters, and network flows data.

1.5 Thesis Outline

Despite the existence of a table of contents, which objective is to provide an organized view over the content to be exposed, it is important to also explain how sections are connected and why this document structure is organized in such way, the reasoning is as follows:

- **Chapter 2 Related Work -** holds related work references and high level descriptions of other similar projects or frameworks and how they contribute to the 5G Network ecosystem;
- **Chapter 3 Technologies -** presents the state-of-the-art research regarding software technologies related to this thesis scope. It also includes the reasoning behind the choice of said technologies;
- **Chapter 4 Architecture -** covers SELFNET's high level architecture, both logical and functional, focusing on this thesis core work: Raw Data Loader and the CEP Framework;
- **Chapter 5 Interfaces, Data Sources and Data Models -** documents the existing interfaces that allow interaction between the presented components, as well as the relevant data sources (sensors and monitoring tools), that will be referred throughout this document, and the data models associated;
- **Chapter 6 Implementation -** exposes the implementation of said components in a detailed fashion;
- **Chapter 7 Application and Results -** summarizes the results and practical applications obtained from the development process;
- **Chapter 8 Conclusion and Future Work -** establishes a critic vision over the achieved results and sets goals for future work.

Chapter 2

Related Work

Several companies and research centers also have their eyes set on the future of 5G, some with the objective of developing new products, others with the aim to contribute to 5G standards. That being said, this Section exposes some of the related work regarding 5G and this thesis theme in order to shed some light upon the surrounding and evolving network ecosystem. Moreover, this chapter divides the related work research in three sections: research projects, products, and research papers. At the end, it provides a discussion area to understand the relevance of this research.

2.1 Research projects

2.1.1 T-NOVA

T-NOVA [5] was an Integrated Project co-funded by the European Commission, with the aim of introducing a framework, to network operators, that is able to offer virtual network appliances (firewalls, gateways, proxies, etc) to their clients on-demand and as-a-Service. The aim of this project was to reduce the hardware necessity from customers, and thus introducing a cloud orchestration platform for the automated provision, configuration, monitoring and optimization of Network Functions-as-a-Service (NFaaS) over virtualized network infrastructures.

2.1.2 UNIFY

UNIFY [6] was a project co-funded by the European Commission, that attempted to address the lack of flexibility caused by the rigid network control. In order to tackle this problem, their idea was to pursue full network and service virtualization and attain flexible services and operational efficiency. They aimed to research and develop an orchestrator system that was able to put in place, verify and observe end-to-end services in both home and enterprise networks.

2.1.3 CROWD

CROWD [7] was a project co-funded by the European Commission, which focused on connectivity management for energy optimized wireless dense networks. The increasing wireless users also increases the demand for more infrastructure elements that allow this growth

to happen. However, it implies that more hardware is put in place to handle this density change, potentially leading to a scenario of wireless chaos and huge energy wastes. In order to provide a solution for this matter, CROWD objective was to arrive to a new connectivity management way by using novel heuristic algorithms that exploited regional information to offer more energy efficient operations, enhancements to the MAC layer for 802.11 and LTE, and new dynamic back-haul reconfiguration strategies to achieve energy consumption levels proportional to traffic demands.

2.1.4 5G-NORMA

5G-NORMA [8] is one of the 5G-PPP projects under the Horizon 2020 initiative that aims to develop a novel mobile network architecture to address adaptability in network resources usage when dealing with traffic demand fluctuations originated from heterogeneous and dynamic network services. Exploring software-defined networking (SDN) and network functions virtualization (NFV) will allow the 5G-NORMA project to tackle multi-tenancy and multi-service scenarios on mobile networks, in order to develop enhanced 5G base stations, software-based centralized controllers and software-based radio access network (RAN) elements.

2.2 Products

2.2.1 PrOptima

PrOptima™[9] is a MYCOM OSI telecom company product, designed to address end-to-end network performance management for mobile, IP, virtualized and fixed networks. It offers: out-of-the-box support for multiple domains, technologies, network equipment vendors, and service suppliers; scaling ability to process large volumes of performance data in near real-time using next generation object storage; advanced reporting and analysis capabilities with flexible configurations; ready to support performance of hybrid NFV based networks as well as internet of things (IoT) data traffic; amongst others.

2.2.2 ZTE Network Performance Management

Following the development path of network providers, ZTE offers a network performance management service [10] for fixed and wireless networks, ensuring stable network operations, improving performance indicators, reducing operations and maintenance costs, and providing good customer experience through smart network management that encompasses network monitoring to be provided with performance indicators to be used for troubleshooting faults, and improving network performance.

2.2.3 ALTAIA

ALTAIA is a service performance and quality management system that is part of Altice Labs NOSSIS suite of OSS systems [11]. Its central role in management architecture involves, in a first phase, the real-time capture and analysis of network performance indicators. This online, integrated and consolidated overview of the status of the service, designated in the eTom (Business Process Framework) and SQM (Service Quality Management) architecture, forms one of the main pillars of the operations management “puzzle”.

Using both these indicators and inventory data, delivered through its integration with inventory systems, ALTAIA's next step is to monitor and oversee the service levels agreed with the clients (SLA - Service Level Agreement). It detects and generates alarms for degradations which could potentially degenerate into breaches of contractual thresholds.

To support the distributed collection and calculation of KPIs, ALTAIA uses an Altice Labs pluggable asynchronous framework that provides clustering capabilities and asynchronous communication between modules. ALTAIA is able to manage, schedule and distribute data collection tasks across a cluster of agents responsible for their execution.

2.3 Research papers

2.3.1 Key Challenges for the Radio-Access Network

The work in [12] discusses the challenges envisioned for the radio-access network (RAN) that will take place with the advancement of radio technologies in 5G. It is important to notice that it was published in September 2013 and it still offers a quite complete high level overview of the upcoming 5G wireless world. It mentions that the evolution of radio-access technologies can be seen as seven different evolution directions:

1. improving service provision and cost efficiency, as seen in the past with the transitions from second generation up to the forth generation of mobile communications, primarily targeting resource usage improvements (e.g. spectrum usage);
2. decreasing the size of radio cells being deployed with the aim of improving capacity, reducing resources cost and better spectrum usage;
3. introducing composite wireless infrastructures that interconnect cellular systems with wireless local area networks (WLANs), to improve application provisioning by offering application through the most appropriate wireless networks;
4. establishing heterogeneous network deployments based on one cellular standard (i.e. under their assumption, the 4G/LTE-advanced standard), which make use of different types of infrastructure elements that can be configured to improve cellular coverage, efficiency while providing a better service;
5. applying flexible spectrum management that makes use of different radio cells sizes, which offer different wireless ranges, to further improve the radio spectrum usage;
6. allowing machine-to-machine (M2M) communications to create dynamic network constructs, that features end user devices intercommunication, and change the management mechanism in order to support this scenario;
7. exploiting cloud-RAN and mobile cloud concepts that bring a shared use of storage and computing resources, avoiding multiple deployment of identical components, thus reducing costs and pushing a new ways of managing resources.

To address these challenges, the introduction of intelligence based systems is paramount in order to be able to fully grasp and manage the complexity of the heterogeneous network and cloud-RANs brought by 5G. This intelligence is needed to manage the network functionalities in their optimal configurations, something that requires a system capable of sensing its own network, recognize non optimal situations, decide upon an action needed to be taken and, finally actuate.

The work envisions that the main enabling and emerging technologies are software defined networks (SDNs) and network function virtualization (NFV), two technologies that require network operators to introduce virtual infrastructures comprised of high-volume hardware

servers connected using high-volume network switches organized by automated orchestration processes, capable of supporting virtualized network functions controlled by SDN controllers.

Although radio-access technologies continue to evolve and enable more heterogeneous and complex networks, it seems that an evolution of the management principles is needed, something that confers an important role to intelligent management systems.

2.3.2 Video Quality in 5G Networks

Context-aware QoE management in the SDN control plane is the main theme that drives the work in [13]. It aims to address the challenges in delivering high-demanding multimedia applications in 5G networks by proposing a new QoE and context-aware system.

To tackle the expected increase of devices that demand Ultra High Definition (UHD) video content distribution over 5G mobile networks, the proposed system intends to make use of the SDN environment, that is expected to rise with 5G, together with traffic flows to improve QoS and QoE by using SDN-controllers that use scalable video codecs to adjust video quality dynamically.

One of the main key aspects of the system is the usage of the last generation of video codecs, the H.265 codec that possesses a scalable extension which allows an H.265 coded video stream, that itself is comprised of video layers of different qualities (e.g. high definition (HD), quality high definition (QHD), UHD, etc), to drop layers in order to adapt to the available network bandwidth.

The other key aspect, and a very important one, is the way this drop process is triggered. By making use of flow and network path statistics and topology discovery data collected from SDN switches, the network can be monitored in order to estimate the QoE over video streams, thus being able to identify cases where QoE is degraded and an action may be needed to solve the issue by: changing the video stream path on the network or even dropping video layers to continue to offer the video stream, although in a lower quality.

This work reports that many of the individual components required to implement the SDN based system have already been developed and tested, and that they are working on implementing an SDN video quality orchestrator that handles video quality management based on the gathered network metrics.

2.3.3 What Will 5G Be?

Asking the high level question of what 5G will be, the work in [14] claims that it will not be an incremental advance over 4G. While previous generations of cellular technology represented major improvements over preceding ones, they never established backwards compatibility. For this matter, 5G is expected to bring a paradigm shift that further evolves carrier networks to have higher frequencies and bandwidths, extreme network density, on both base stations and user devices, and an enormous number of antennas. However, and in contrast with the lack of backward compatibility that previous generations demonstrated, 5G will have to also integrate with existing technologies (e.g. air interfaces integration with LTE and WiFi), providing a universal radio access and seamless user experience. To encompass this, 5G will need to be able to establish intelligent and flexible core networks, so that efficient resources management can be achieved in this scenario.

2.3.4 Leveraging SDN to Provide an In-network QoE Measurement Network

With the proliferation of media services and online video streaming, HTTP Adaptive Streaming (HAS) is becoming the most popular content delivery mechanism [15]. As network and content providers offer this kind of services, they also become interested in ensuring high quality of experience (QoE) levels for their end-users. However, the existing network-level metrics are not enough to provide insight over the end-users' perception of the delivered content. To address this, the authors of this work introduce an in-network QoE measurement framework (IQMF) that provides QoE monitoring for HAS streams as a service. In order to achieve a non-intrusive quality monitoring system and to close the QoE-aware service management control loop, the software-defined networking (SDN) concept is included so that the control plane functionality can be exploited. From this research crucial service quality metrics could be measured, thus providing quality data to be used, alongside with SDNs, to properly control QoE and to ensure that high quality video content is delivered to the end-user.

2.3.5 Toward Software-Defined Cellular Networks

The work in [16] focuses on cellular networks negative aspects, their inflexible and expensive equipment, complex control plane protocols, and vendor specific configuration interfaces. Their aim is to discuss how software-defined networking (SDN) can simplify the design and management of cellular networks. It is important to notice that scalability challenges are likely to be presented when dealing with: supporting many subscribers, subscribers frequent mobility, the fine-grained measurement and control of network elements, and the real-time adaptation of network loads. To address these key issues, the authors of the work propose extensions to controller platforms, switches and base stations that controller applications are able to enforce subscriber based policies, to perform real-time and high precision traffic control at the switches level, to perform deep packet inspection, and to remotely manage base stations resource sharing. From this discussion, it emerged an SDN architecture for cellular networks, thus taking a step into software defined cellular networks.

2.3.6 DevoFlow: Scaling Flow Management for High-Performance Networks

Software-defined networking simplifies network and traffic management in enterprise and data centers. However, the work in [17] claims that the OpenFlow protocol, while a great concept, imposes excessive overheads. Despite it enabling flow level control over network switches and making network flows global visible in the network, these features are not free to take as-is. It implies implementing switches that support OpenFlow, as well as involving OpenFlow controllers for flow setups and statistical data collection, thus introducing communication overheads. To further investigate these claims, the authors of this work analysed said overheads and concluded that OpenFlow current design did not meet high-performance networks requirements. In order to address this, they proposed an extension to the OpenFlow protocol called DevoFlow, which slightly separates the coupling between control and global flow visibility, thus being able to discard some of the overheads identified.

2.3.7 Software Defined Monitoring (SDM) for 5G Mobile Backhaul Networks

In 5G and future networks, software-defined networking will play a significant role as their enabler, since it introduces the ability to design dynamic, manageable, cost-effective and adaptable network architectures. The work in [18] exposes that the networking monitoring functionalities will be transferred into a software entity that is able to work together with configurable hardware accelerators, defining the software-defined monitoring (SDM) scheme, in order to attain the necessary dynamism to monitor next-generation networks. The work proposes a novel SDM architecture for the future mobile backhaul networks. Since SDM encompasses SDN, the proposed architecture provides dynamic network management capabilities using programmable interfaces, centralized network control and virtualized abstractions. Given this, the SDM framework is open to address the various challenges introduced by the separation of the control and data planes. As an outcome of the research, major limitations were found in the current monitoring techniques, the lack of interoperability on vendor specific network hardware, the high dependence on physical resources, amongst others. Despite these limitations, the SDM framework showed that it is possible to simplify network management operations as well as to support dynamic networks monitoring.

2.3.8 SDN Meets SDR in Self-Organizing Networks: Fitting the Pieces of Network Management

The work in [19] explores the LTE self-organizing networks (SONs) which already automate several management mechanism regarding network configuration, planning and optimization. These features require network programmability at the control and data planes, making these SONs a target for implementing software-defined networking (SDN) and software-defined radio (SDR). Following this line of thought, merging SON, SDN, and SDR, the paper proposes a SON-based management framework, capable of dynamically configuring both data and control planes.

2.3.9 Enabling Software-Defined Networking for Mesh Networks in Smart Environments

The work in [20] has as its main focus, wireless mesh networks (WMNs) and states that they serve as a key for enabling technology for smart initiatives, such as smart power grids, having in mind that it might be possible to provide a self-organized wireless communication highway, capable of monitoring health and performance indicators, and enabling efficient trouble shooting notifications. While ideally providing these features to WMNs, it is important to notice that current routing protocols are still limited, making it hard to implement SON capabilities. However, the work considers that by introducing software-defined networking (SDN) and by using a three-stage routing strategies in WMNs, that it is possible to achieve SDN enabled wireless mesh networks when using centralized controllers.

2.3.10 Autonomics and SDN for Self-Organizing Networks

To understand the relationship between autonomic network management (ANM) and software-defined networking, the work in [21] studies their interaction under Long Term Evolution (LTE) Self-Organizing Networks (SONs). ANM and SDN have few common goals

and have shown to be complementary to each other in terms of network abstractions and expectations. AMN focuses on delivering self-functionalities by providing an adaptation layer between autonomic and the managed infrastructure, while SDN architecture allows programmatic management control over network resources. However, despite the fact that flow based control mechanisms are becoming more popular and being used in core networks and data centers, the same notion of network flow has yet to be defined for radio access and SON. To address the lack of flow definition over SON, this work proposes an Autonomic SDN controller that integrates with the Unified Management Framework introduced by the UNIVERSELF project, to self-optimize an LTE-Advanced heterogeneous network, thus exercising the notion of network flow over SON.

2.4 Discussion

With three main research categories, projects, products, and papers, which provide insight mainly over software-defined networking (SDN), network function virtualization (NFV), self-organizing network (SON), and autonomic network management areas, it is now possible to arrive to a conclusion on what path should be taken to achieve performance management in NFV and SDN environments.

Taking into account that the arriving 5G environment will bring new features: faster internet speeds, higher bandwidth, lower communication latencies, efficient spectrum management, support for denser wireless networks, and others; it is possible to understand that changes must be introduced to the current technologies that are in place today, something that can be perceived by looking into the necessity already identified by network operator products (as seen in Sections 2.2.1, 2.2.1 and 2.2.2). Furthermore, these necessities are currently being investigated by research projects, such as, T-NOVA which addresses NFVs as a service for operators, 5G-NORMA which is expanding the mobile network by using SDN and NFV to support multi-tenancy and multi-service. It seems that, increasing the feature pool of next-generation mobile networks also increases the complexity of their deployment and management, and for this matter a possible solution, that is still being researched, is the usage of SDN controllers to separate the data and control planes, allowing a simpler way of managing traffic, something that was covered by most presented papers in Section 2.3. However, introducing SDN to mobile networks is only one part of the scenario foreseen by 5G. The paradigm shift that is expected to happen is to combine SDN, NFV, SON and integrate them with autonomic systems: the work in 2.3.10 provides some insight on this aspect. In order to further optimize resources usage in complex network environment where the task of monitoring them is increasingly difficult, it seems natural that us, humans, start being replaced by autonomic functionalities when it comes to network management, further enhancing the self-organizing network meaning.

All things considered, the research environment regarding next-generation networks seems to be embracing the new paradigm of having SDN, NFV, SON and autonomic systems in place to further improve the network environment of today. Given this, it can be safely stated that this thesis theme is aligned with the future of networking.

Chapter 3

Technologies

This chapter provides an overview of the available technologies that were studied to develop the proposed architecture. Section 3.1 describes the available service coordination technologies, while Section 3.2 explores the considered message bus systems, then Section 3.3 covers aggregation tools that might be useful for the implementation process, Section 3.4 refers to research on storage technologies specially aimed for time-series data, and lastly Section 3.5 presents the studied monitoring tools.

3.1 Service Coordination

3.1.1 Apache Zookeeper

Apache Zookeeper [22] is a Java cross-platform open source project under the Apache foundation. In its essence, ZooKeeper is a hierarchical distributed key-value database that can be accessed similarly to a traditional file system, allowing the coordination of distributed processes through a shared hierarchical name space of data registers (znodes). Targeting distributed systems and contrary to normal file systems, ZooKeeper provides ordered access to the znodes for its clients with highly available and high throughput while maintaining a low latency.

As mentioned, ZooKeeper works in a cluster where multiple nodes serve multiple clients, as depicted in Figure 3.1. To provide high reliability the clients can interact with any server, assigned by the leader. In case of node failure, a new one will be assigned by the leader, and in case of leader failure a new one is elected from the remaining nodes.

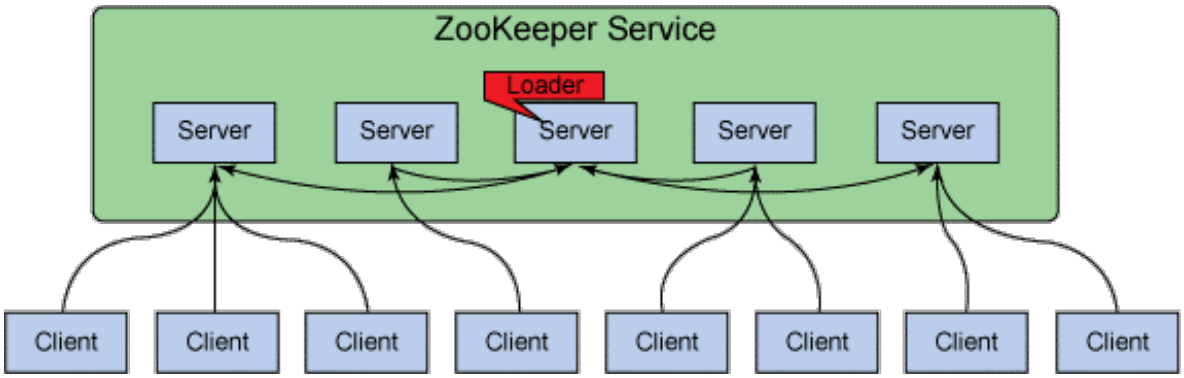


Figure 3.1: Zookeeper high-level architecture [23]

It is a distributed coordination service aimed for distributed applications. Distributed applications are applications that, instead of running on a single machine, are executed in a distributed environment over a network. Distributed applications allow a clear separation between functionalities provided enabling both:

- **Reliability:** whenever a node that is part of the network goes down, another one can resume the ongoing task;
- **Scalability:** meaning that more resources can be added when the applications load justifies so.

Despite its advantages, ensuring coordination is one of the main problems that arise with distributed applications, namely the need to guarantee that every action of each component is coordinated. To overcome the coordination problem and simplify the deployment and management of a distributed application, Zookeeper offers a service that implements higher level services for synchronization, configuration maintenance, and groups and naming [23]. Zookeeper is designed to be simple, replicated, ordered and fast. It uses a shared namespace following a similar approach to a file system, as seen in Figure 3.2, but instead of storage files and directories, Zookeeper uses memory to store its data.

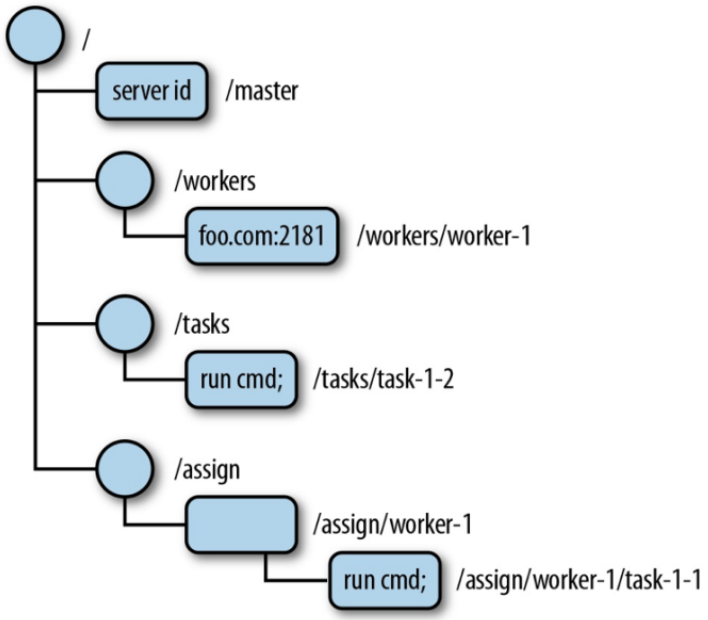


Figure 3.2: Zookeeper data tree [25]

To use Zookeeper, an application needs to use a Zookeeper client. This client is then responsible to communicate with Zookeeper servers. Zookeeper can work in single mode, where only one server is instantiated, or in a quorum mode, where multiple servers are instantiated and can respond to client requests [24].

Zookeeper has a key role in the SELFNET project because it allows developers to focus on the innovation and relevant applications functionalities instead of cluster coordination for distributed applications. Most of the aggregation modules are distributed modules working in parallel and they use Zookeeper for coordination and configuration. It will be used for the leader election on cluster oriented services and as in-memory storage for configuration between nodes in the same cluster. The configurations from Aggregation Configuration Manager (ACM) will be distributed to the Complex Event Processing (CEP) nodes using Zookeeper. On top of that it is also used to store Kafka offsets and progress information as well as the Kafka cluster leader election process.

3.1.2 HashiCorp Consul

Consul [26] is a service discovering and configuring tool that allows the creation of dynamic clusters by deploying a distributed server/client communication architecture. Its main key features are:

- **Service Discovery:** Consult clients are able to provide a service (e.g. an API or a MySQL database), enabling other clients to discover these services through DNS or HTTP;
- **Health Checking:** clients of Consult can provide health checks functionalities, that can be associated to a service, which can then be used to monitor cluster health and route traffic away from less healthy clients;
- **Key Value Data-store:** a data-store exposed by a simple HTTP API that allows applications to use a multi-purpose key value store (e.g. leader election, service coordination, dynamic configuration, amongst others);
- **Multi Datacenter support:** out-of-the-box multiple datacenter support, abstracting multi-region systems coordination.

Its deployment consists of Consul agents located at each node which then communicates with one or more Consul servers. All data is stored and replicated at the Consul servers, which themselves elect a leader. Service discovery is achieved through the existing connectivity between the servers and these agents, which are able to forward queries to servers automatically.

3.2 Message Bus

3.2.1 Apache Kafka

Apache Kafka [27] aims to provide a scalable, high-throughput, low-latency unified platform for handling real-time data feeds. It is a distributed streaming platform that targets real-time applications requiring reliable and effective pipelines to get data between systems, usually getting compared to message queues or enterprise messaging systems. Apache Kafka is a Java/Scala cross-platform open-source project under the Apache foundation.

As a distributed application, Kafka runs as a cluster on one or more servers and stores the data in topics in the shape of records composed by a key, a value, and a timestamp. It allows applications to publish and/or subscribe to streams of records, as seen in Figure 3.3.

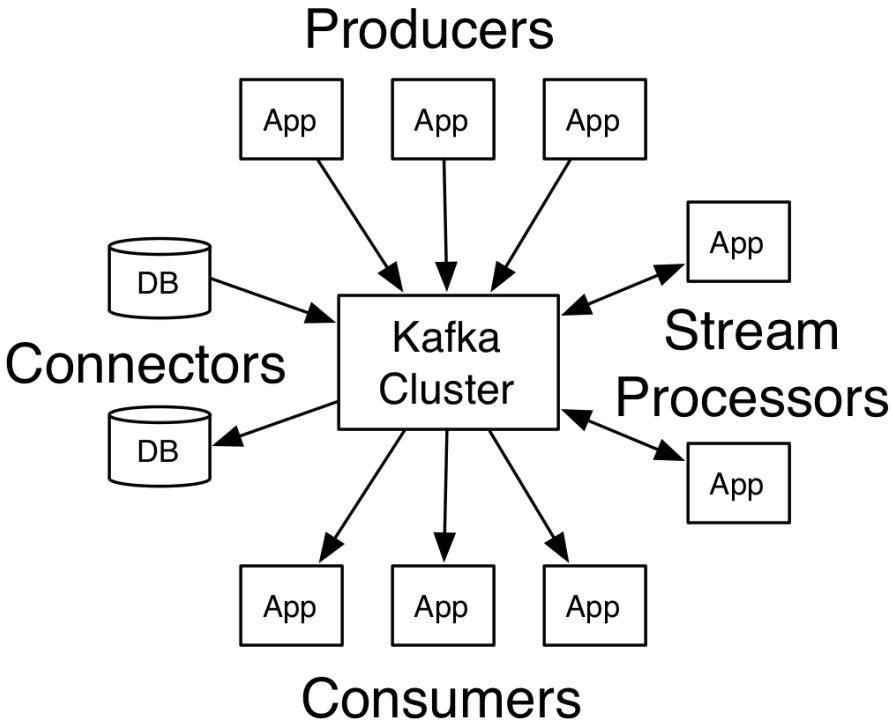


Figure 3.3: Kafka high-level architecture [29]

It allows clients to publish into streams, subscribe to streams, store streams of records in a fault-tolerant way, or process streams of records as they occur via its APIs.

The Producer API allows an application to publish a stream of records to one or more Kafka topics.

The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced.

The Streams API allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

Topics in Kafka are always multi-subscriber, so a topic can have zero, one, or many consumers that subscribed to the data written to it, but when analyzing how Kafka client subscription to topics works, there are a few important concepts to regard:

1. **Client Groups:** A single application can have multiple client threads connecting to the same topic in order to increase throughput and allow parallel processing. Thus, these clients share the same client group. All the clients in a group will connect to the same topic(s) but to different sections of the topic, thus avoiding replication of messages between clients in the same group. One message should only be consumed by only one client in a group. Clients in different groups may consume the same messages as each group can consume all the messages.

2. **Partitioning:** A topic is split into multiple sections called partitions, as seen in Figure 3.4. A client can connect to any number of partitions, but two clients in the same Client Group cannot connect to the same partition. Partitions can be split into different devices (i.e. hard drives) to improve I/O performance.
3. **Replication:** In a cluster, topic partitions can be duplicated on different nodes or devices to allow redundancy and fail recovery. For instance, even if a node or device fails, with proper partitioning, it is possible to obtain the data from another node in the cluster. For a topic with a replication factor N, Kafka will tolerate up to N-1 server failures without losing any records.
4. **Offsets:** Each client group keeps an offset for each partition in a topic, thus allowing a client to choose where to start consuming records (at the start, the end or a previously committed offset).

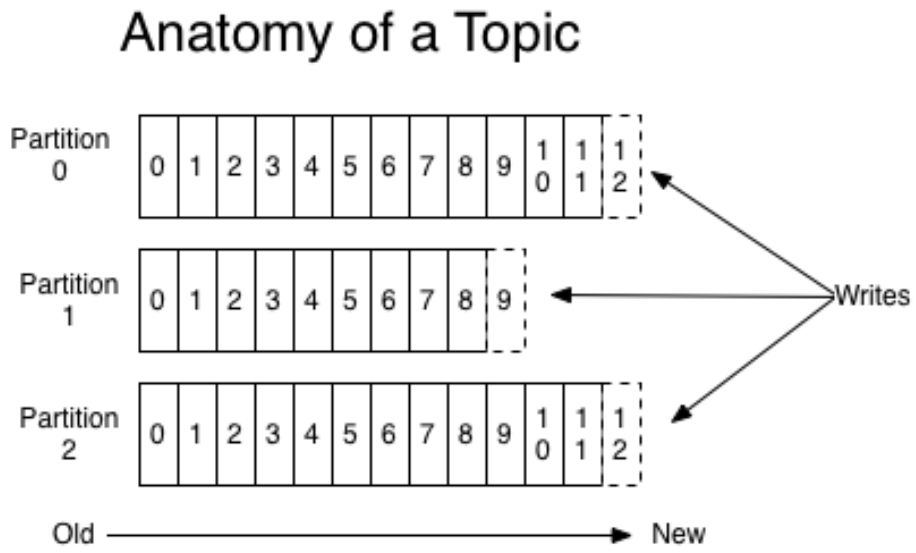


Figure 3.4: Kafka topic partitions [29]

When comparing to other messaging systems, Kafka pushes many of the typical responsibilities to the clients and the architecture using Kafka. Record Consumption is sequential on a partition and the client applications are responsible to know where they want to consume, to re-position the consumers after failure, and to know if they want to consume said records. Kafka does not allow search or record filtering on the server side. Thus, Kafka can achieve massively success [28] in the scalability of its publish/subscribe architecture.

Kafka is also known for its versatility allowing applications to use it as messaging systems, website activity tracking, metric aggregation and operational data monitoring, log aggregation, stream processing, event sourcing, commit log, etc. For SELFNET, Kafka is a valuable assistant and is expected to act as a messaging system between modules and work-packages; as a message aggregator, collecting messages from multiple and different sensors, etc.; and as a streaming processing pipeline allowing metrics and events to be sequentially processed.

3.2.2 Pivotal RabbitMQ

RabbitMQ [53] is a general purpose message broker that supports various standard messaging protocols, such as AMQP, to offer a solid and mature message bus with routing capabilities. RabbitMQ is known for being easy to use and for supporting a large number of development platforms (e.g. Python, Java, Ruby, Go, C, Node.js, amongst others), making it a flexible message broker that implements:

- message queues by using exchange points;
- multiple consumer support for each queue;
- message distribution over all available consumers;
- message redelivery upon failure situations;
- delivery order in queues with a single consumer.

It uses a smart broker versus dumb consumer model that has consistent messages delivery to consumers while keeping track of their state. However, keeping the consumer dumb and having the smart broker that keeps track of consumer states and control over reliability features on queues, makes its performance to only reach 310 messages per second (sent and received), in a single-node deployment, and reaching 1600 messages per second (sent and received) when deployed with eight nodes (depicted in Figure 3.5).

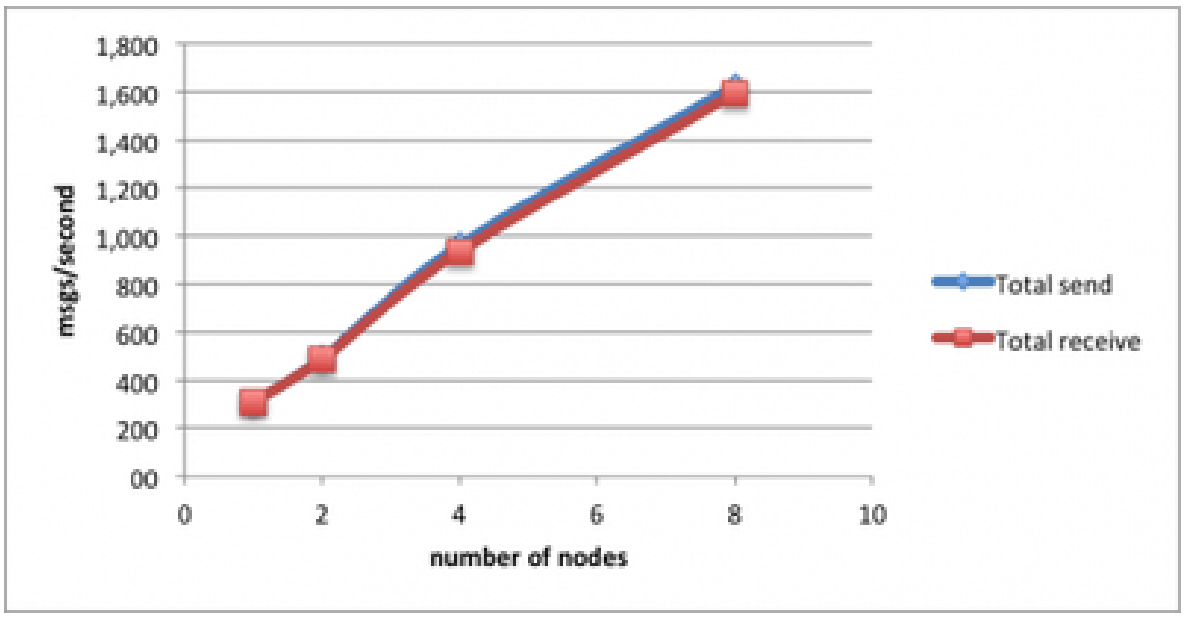


Figure 3.5: RabbitMQ Performance test [54]

Despite these facts, RabbitMQ features make the message broker popular since it offers a reliable way of communication that supports consumers and producers from different environments.

3.3 Aggregation Tools

3.3.1 Trifacta Wrangler

Wrangler is a Trifacta product [46] that offers a platform for data processing, comprised of various layers (not fully represented here):

- **Connectivity Framework:** provides data collection from several different sources: Hadoop sources, cloud services, files (CSV, TXT, JSON, XML, etc) and relational databases;
- **Any Scale Data Processing:** the data transformation processes that use their Intelligent Execution Engine, that takes user define transformation steps and automatically finds the best-fit processing framework (Apache Spark, Google DataFlow, our Photon, Trifacta's in-memory engine) based on data scale;
- **Intelligence & Context:** the framework area that learns from the registered data and how it is used, in order to suggest on how data might be transformed into useful information;
- **Core Data Wrangling User Experience:** is the module that leverages data visualization, machine learning and human-computer interaction techniques to better present and explore the processed, or to be processed, data;
- **Publishing & Access Framework:** is the northbound limit of Wrangler, an API that allows access to all data registered in Wrangler as well as, a variety of analytics, data catalogue and data governance applications.

Wrangler is available, as of July 2017, in various versions [47], having a free version [48] with several severe constraints (see [47]), thus offering a crippled software that does not meet the requirements presented in Section 4.3.

3.3.2 MongoDB Aggregation Framework

As part of what MongoDB offers as whole, there is its Aggregation Framework [42], a simpler alternative to their map-reduce feature. Currently, as of July 2017, the Aggregation Framework comes bundled in MongoDB version 3.4. This framework can be viewed as a set of data processing pipelines used for data aggregation based of previously defined models, reducing the amount of data already stored in their database, MongoDB.

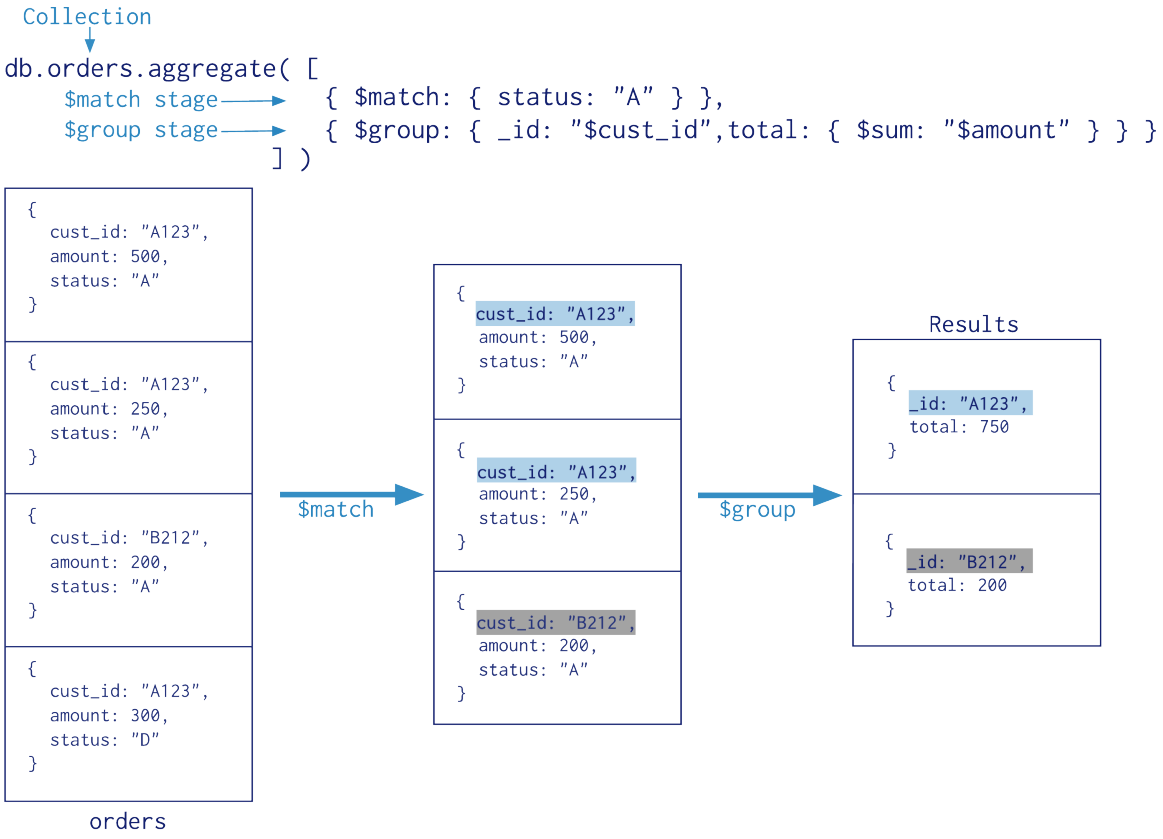


Figure 3.6: MongoDB Aggregation Framework Pipeline

These pipelines allow for data to be filtered and grouped together before being subjected to aggregation functions [43] as seen in Figure 3.6. Despite this, aggregations are somewhat limited, since they cannot refer to documents from other collections, aggregation results can only produce documents up to 16 megabytes of size [44] and pipeline stages have a 100 megabytes limit of RAM [45]. To address larger datasets configurations need to be applied in order to allow disk usage, creating temporary files and slowing down the data processing. In conclusion, MongoDB Aggregation Framework is a simple utility tool set that addresses data stored within MongoDB to produce simple aggregation results.

3.3.3 Apache Storm

Storm is an open source distributed system to process data streams in real time. Storm can receive data from multiple inputs and provides mechanisms to process and transform data. The real-time processing enabled by Storm ensures that data is processed as received, without the need of batch gathering [30]. Besides stream processing, Storm can also be used to offer continuous computation, distributed Remote Procedure Call (RPC) and real time analytics, offering the following features [31]:

- **Fast:** Storm can handle over a million tuples per second;
- **Horizontal Scalability:** it is possible to increase the number of computer nodes that offer more processing capabilities to Storm;

- **Fault Tolerant:** whenever a Storm worker dies, Storm is responsible to restart the worker on the same computer node or in another one;
- **Guaranteed Data Processing:** Storm has tools to not only guarantee that each message received is processed, but also has mechanisms that allow to process each message only once, despite its parallelism;
- **Programming language agnostic:** Storm is executed in a Java Virtual Machine (JVM). It is worth to mention, however, that applications written to run in Storm can use any programming language, since it is capable of reading the input streams and write output streams.

A Storm cluster is composed of three different components: Nimbus, Zookeeper Cluster and Supervisor Nodes. The Nimbus is responsible to coordinate a Storm cluster by distributing the code and the work across other components (only one Nimbus exists in a Storm ecosystem). Zookeeper is used to coordinate the distributed applications. Zookeeper can have many nodes allowing Nimbus and the supervisors to communicate through its channels. Supervisors consist in worker nodes, responsible to create, start and stop worker processes [31].

Storm uses three main components as its data model: Streams, Spouts and Bolts. Together, the collection of instances of these components create a Storm Topology. Further explaining each component, a stream is a collection of tuples that can be processed in parallel, a spout consists in a source of streams outside of a Storm topology responsible to pass them to the Storm topology, a bolt is where the streams are processed involving tasks like filtering, aggregations, joins and more. The outcome of a bolt can be used as input for other bolts [32]. Storm complex topologies can be created based on how spouts and bolts are combined, Figure 3.7.

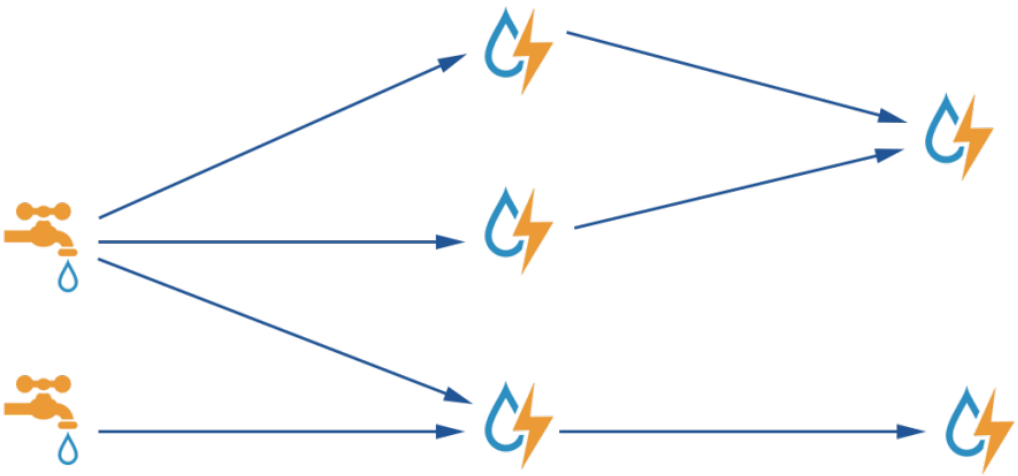


Figure 3.7: Storm topology example

Taking into consideration Apache Storms characteristics and proven scalability, it is the perfect candidate for SELFNET support of Complex Event Processing (CEP). It allows the real-time processing of a large number of events and metrics produced by sensors while scaling (only requiring addition of new computing resources without redesigning the architecture or

modules) to cope with network growth. It will also be used as the engine in the Cassandra Threshold Engine for similar reasons.

3.4 Storage

3.4.1 InfluxData InfluxDB

InfluxDB [35] is a time series database which specializes in storing metrics and events in large numbers. It is optimized for fast retrieval of time series datasets for monitoring and real-time analytics. It was first released on 2013-11; as of May 2017, its most recent stable version is version 1.2.4, released on 2017-05-08. The free open-source version is distributed under MIT license. However, some features, including deployment on multiple nodes, are only available in the commercial enterprise edition.

InfluxDB manages simple data types such as 64-bit integers, double precision floating point values and also strings in key-value stores. Time-series specific compression algorithms are employed to maximize disk space utilization.

As datasets consisting of raw metrics grow rapidly in time, InfluxDB provides a system of configurable policies controlling automatic deletion and down-sampling of older data. Benchmarks show that, when used for the particular purpose of storing time series, InfluxDB can perform even better than Cassandra, which can be seen in Figure 3.8.

Influx features a great synergy with the Monasca project (see Section 3.5.2) as it is one the storage choices for that framework.

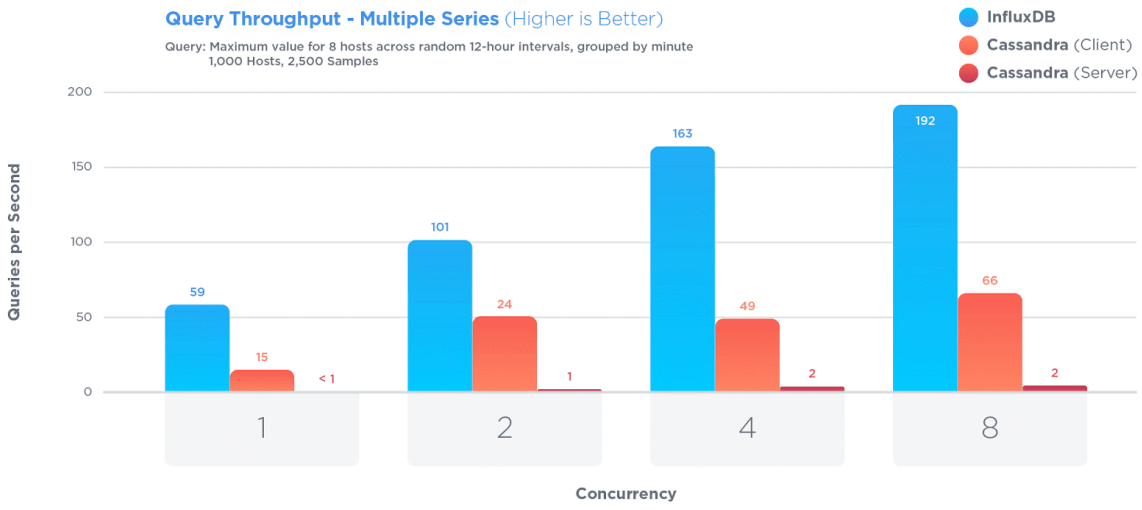


Figure 3.8: Comparison of InfluxDB with Cassandra on time series data [36]

As InfluxDB is designed as a native time series database, it is ideal for monitoring metrics, sensor data and real-time analysis since all these are data points collected at regular intervals over a period of time. Monasca supports this database, making this technology a natural choice as the metrics and alarms database for the Aggregation Framework.

3.4.2 Openstack Gnocchi

Gnocchi [37] is the project name of a TDBaaS (Time Series Database as a Service) project started under the Ceilometer program umbrella. It is a multi-tenant time series, metrics and resources database. It provides an HTTP REST interface to create and manipulate the data. It is designed to store metrics at a very large scale while providing access to metrics and resources information and history.

From the beginning of the Ceilometer project, a large part of the goal was to store time series data that were collected. In the early stages of the project, it was not really clear what and how these time series were going to be handled, manipulated and queried, so the data model used by Ceilometer was very flexible. That ended up being really powerful and handy, but the resulting performance has been terrible, to a point where storing a large amount of metrics on several weeks is really hard to achieve without having the data storage backend collapsing.

Having such a flexible data model and query system is very important, but in the end users are doing the same request over and over, and the use cases that need to be addressed are a subset of that data model. On the other hand, some queries and use cases are not solved by the current data model, either because they are not easy to be expressed or because they are just too slow to run.

Lately, during the Icehouse Design Summit in Hong-Kong [37], developers and users showed interest in having Ceilometer doing metric data aggregation, in order to keep data in a more long running fashion. No work has been done during the Icehouse cycle on that, probably due to the lack of manpower around the idea, even if the idea and motivation was validated by the core team back then.

Considering the amount of data and metrics Ceilometer generates and has to store, a new strategy and a rethinking of the problem was needed, so Gnocchi is a try on that.

3.4.3 Apache Cassandra

Apache Cassandra is a NoSQL database which aims to provide fast access to big datasets organized in tables while maintaining high availability. Cassandra was originally developed at Facebook and later open-sourced in 2008. Cassandra is now a free open-source software distributed under the Apache License 2.0 [33]. As of April 2017, the most recent stable version of Cassandra is version 3.10, released on 2017-02-03.

Cassandra's architecture enables deployment on clusters spanning tens of thousands of nodes in multiple datacenters running decentralized and asynchronously, with no single point of failure. To achieve high data throughput, Cassandra does not support the full relational database model; in particular, Cassandra does not support joins. Instead, data denormalization is encouraged. It is also a column oriented NoSQL database. As such, it relies on schemas to specify the type of data found in table columns. This contrasts the approach of document oriented NoSQL databases such as MongoDB, which make no assumptions on the structure of the data they store.

Benchmarks show that among currently available NoSQL databases, Cassandra is one of the best performing alternatives. Scalability is excellent, data throughput scales linearly for both reads and writes with the number of machines added to a cluster, as can be observed in Figure 3.9. Installing new nodes and replacing failed nodes does not require time consuming reconfiguration procedures to be applied.

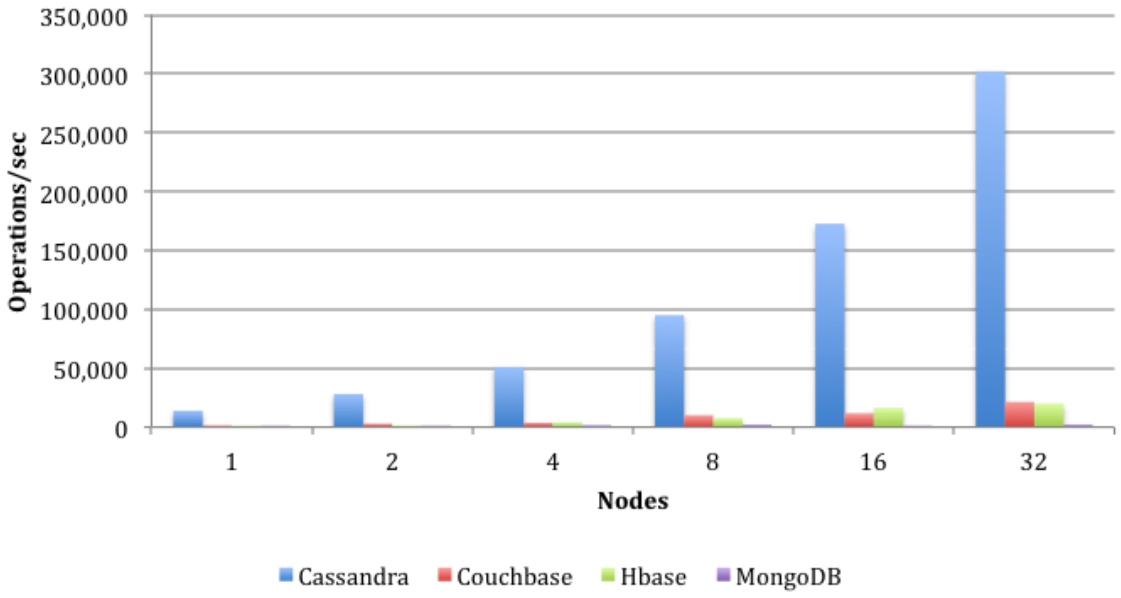


Figure 3.9: Scalability of Cassandra [34]

In addition to performance - and given the nature of the data provided by the sensors in SELFNET, where descriptors and counters vary according to what is being measured - Cassandra provides a very useful Map-type column. It allows data from different sources to be stored in the same table, without the need to create or change tables in order to accommodate for new sensors or upgrades to old ones. This simplifies database administration and lowers the need for human intervention.

3.5 Monitoring Tools

3.5.1 QoSient Argus

Argus is an open-source project network Audit Record Generation and Utilization System [49], developed by QoSient and released as stable in version 3.0.8.2, as of July 2017.

It has the objective of developing all aspects of large scale network situational awareness derived from network activity audits. It is a next-generation network flow technology, processing packets, either on the wire or in captures, into advanced network flow data.

It is composed of the Argus sensor, an advanced comprehensive network flow data generator, used to process live packet data or capture files, and then to generate detailed network flow status reports for all captured flows. Argus can also be used to provide network activity reports for network transactions, empowering security, operations and performance management tasks.

3.5.2 Openstack Monasca

Monasca [40] is an OpenStack project that provides an open-source multi-tenant, highly scalable, performant, fault-tolerant Monitoring-as-a-Service solution (MONaaS).

Metrics can be published to the Monasca API, stored and queried. Alarms can be created and notifications can be sent when the alarms transition states.

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.

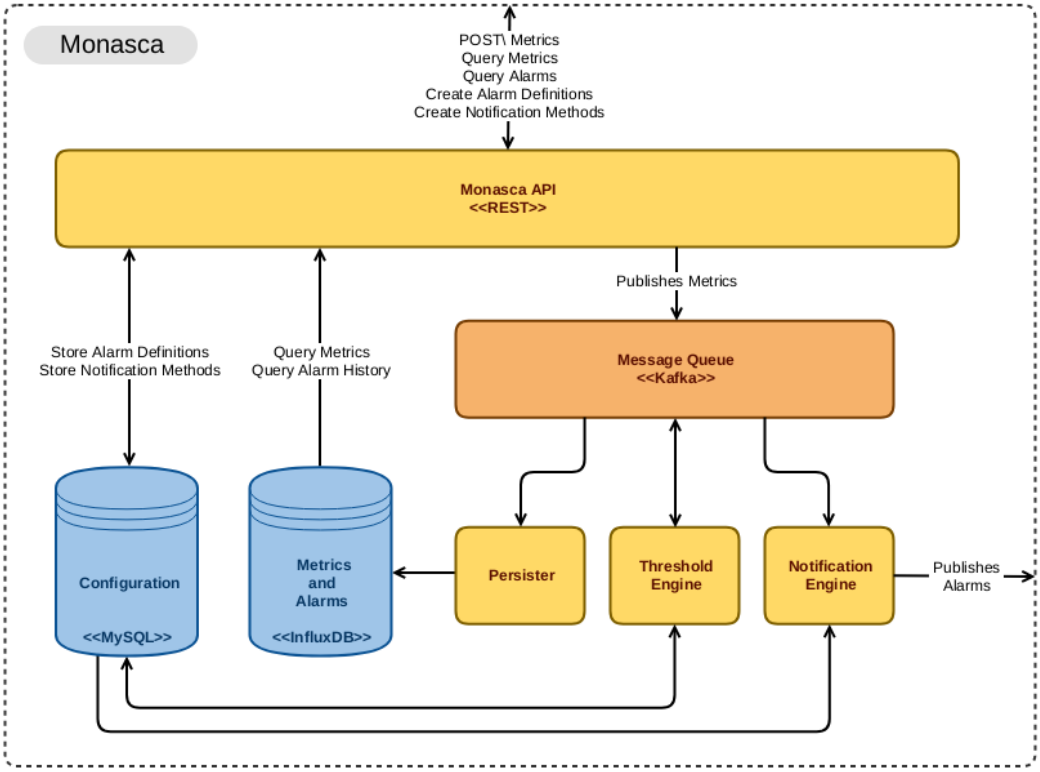


Figure 3.10: Monasca Architecture

Several off the shelf open-source components are used, including:

- **Keystone [55]:** identity service for authentication, authorization and multi-tenancy;
- **Apache Kafka [27]:** central component in Monasca and provides the infrastructure for all internal communications between its components;
- **Apache Storm [32]:** used in the Monasca Threshold Engine;
- **Apache Zookeeper [22]:** used by Kafka and Storm;
- **InfluxDB [35]:** used for storing metrics and alarm history;
- **MySQL [56]:** used to store alarm definitions and notification methods.

Figure 3.10 illustrates Monascas architecture, showing only the relevant components that were used in the Aggregation Framework. All interactions with the service is done through the Monasca API, which on its hand will interact with both databases (InfluxDB and MySQL) and with the message bus (Apache Kafka).

The configuration database (MySQL), as its name suggests, will store all the configurations of the service, namely the Alarm Definitions and the Notification Methods.

The metrics and alarms database (InfluxDB), once again as the name suggests, is where all metrics and triggered alarms will be stored.

All the metrics posted to the Monasca API will be published to the message bus, and the Monasca Persister will fetch and persist them in the corresponding database.

The Threshold Engine will also monitor the message bus for those metrics and will trigger alarms if needed, based on the Alarm Definitions present on the configuration database. If an alarm is triggered and published to the message bus, the Notification Engine is the component responsible for monitoring them and dispatch notifications if required to do so, that is, if it is configured in the corresponding Alarm Definition.

There are other components of the architecture, such as Transform Engine and the Anomaly and Prediction Engine, which were not used in the Aggregation Framework; hence they were not represented in the figure. Nevertheless, the full architecture diagram is available online [40] to make a comparison.

The rationale behind the adoption of Monasca is mainly due to the time series database (InfluxDB), as well the Threshold Engine, although these are not the only reasons. Being an open-source monitoring-as-a-service solution under the OpenStack "Big Tent" and with many major companies also involved in developing and deploying it [41], it gives us the confidence of having a solid solution that serves our needs while releasing us from the burden of having to develop a similar architecture, giving us the freedom to concentrate on the development of other features of the Aggregation Layer.

3.5.3 Openstack Ceilometer

Ceilometer [38] is part of Telemetry project and it was developed to facilitate the meter gathering of virtual resources from OpenStack deployments. Ceilometer collects relevant information of different OpenStack projects like Nova (Compute), Neutron, and so on [38]. Ceilometer also allows the creation of personalized alarms, providing resource tracking and creation of new plugins from external sources [39]. Ceilometer architecture is shown in Figure 3.11.

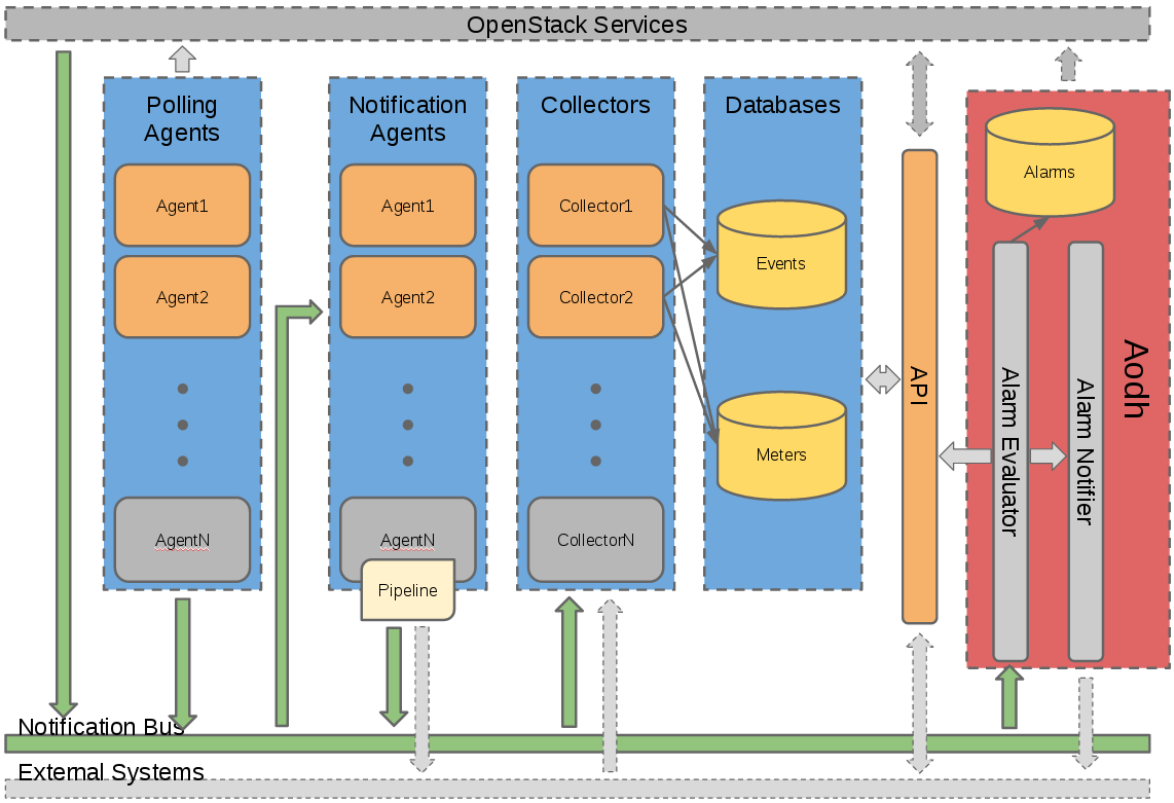


Figure 3.11: Ceilometer Architecture [38]

The components of Ceilometer Architecture are described below:

- Agents: The data gathering is done by two types of agents: Polling and notifications agents:
  1. Polling Agents poll OpenStack services;
  2. Notification agents listen notifications on message queue.

  The main difference between polling and notification agents is the method to collect data (Push or poll strategy). On one hand, polling agents retrieve information from different resources such as Compute, Network, Cinder or Glance Projects, in a given period of time. On the other hand, notification agent listens the notification bus in order to consume messages or events from the queue.
- Collector is in charge of gathering and recording events and metering data. It is an optional component.
- Ceilometer API allows to query and view data recorded by the collector.

The agent daemons could be running on the central (cloud controller) and compute node (Nova) and the collector and notification agent runs on the cloud management node. Once the information is gathered, this could be transformed into another format (aggregate) or published in different destinations. Then, the meter data is stored in a specific database like MongoDB or Gnocchi project [37]. This process is illustrated in Figure 3.12.

Figure 3.12: Ceilometer Workflow [38]

In general terms, Ceilometer provides an architecture based on plugins that allows easy scalability, extensibility and meter customization by means of the creation of new agents. These agents will gather metrics not considered before or meters from external sources. Additionally, this information can be accessed through the REST API provided by Ceilometer.

3.5.4 Apache Chukwa

Chukwa [52] is an open source data collection system for monitoring large distributed systems. More precisely, it aims to be used for log collection and analysis while being designed for big data, robustness and scalability, and providing a tool kit log analysis. Chukwa is built using Hadoop, combining the MapReduce framework and the Hadoop Distributed File System (HDFS), thus inheriting its scalability and robustness. It was first released in 8 November 2009 and last updated at 8 October 2016 (version 0.8.0); as of July 2017. However, the project still looks as if it still is in its early ages, displaying unclear documentation files, making it unfit to be chosen as a monitoring tool.

3.6 Discussion

This section has the objective of communicating the chosen technologies, their purpose and the reasoning behind each choice. For this matter, the chapter was organized in technology category sections, facilitating their discussion:

- **Service Coordination:** is to be used to coordinate different components within the Aggregation Framework (to be presented in Section 4.2.2) and so, two technologies were investigated: Zookeeper and Consul. Consul provides a way to coordinate services in a high level way by using HTTP and DNS protocols, something that is actually viewed as a negative aspect, since inside of the Aggregation Framework a more simple and lower level interaction is expected to happen between components. Based on this rationale and since Zookeeper provides such functionality, by directly implementing it on the components, it was chosen as the service coordination tool;
- **Message Bus:** is to used to input and output data to the Aggregation Framework. The technologies considered for this matter were: RabbitMQ and Kafka. RabbitMQ offers a more traditional message broker with solid robustness. However, when compared to Kafka, it comes short in terms of message throughput, as seen in Figure 3.13 with Kafka showing a 2.5x speed-up relative to RabbitMQ.

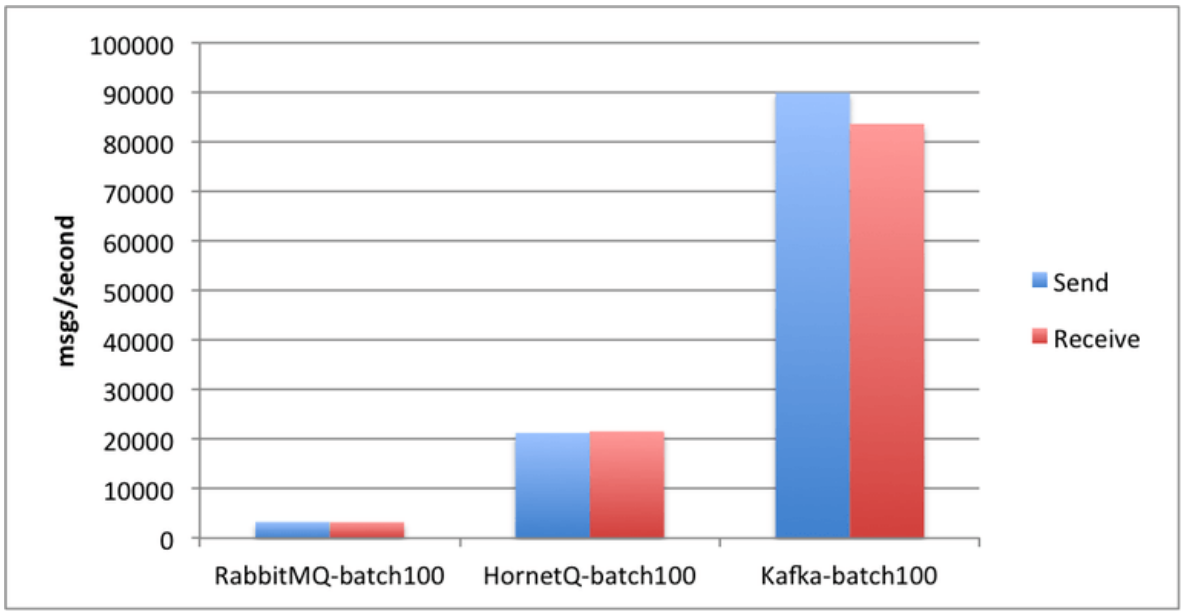


Figure 3.13: Message Bus performance comparison: RabbitMQ versus Kafka [54]

In a framework where performance is paramount, the technologies must also follow such standard, thus making Kafka a better solution than RabbitMQ. It must also be referred that Kafka has Zookeeper as a dependent service, facilitating its choice;

- **Storage:** the requirements for choosing a storage technology lie upon the need to persist aggregated data within the Aggregation Framework. The nature of the aggregated data is intricately the same of time-series data. While various database technologies were surveyed (InfluxDB, Gnocchi and Cassandra), the ones that fitted this requirement were InfluxDB and Gnocchi. Please note that Cassandra was already chosen by the SELFNET consortium for raw data persistence. Both InfluxDB and Gnocchi provided the needed functionalities of storing time-series data with multi-dimension support; however, Gnocchi showed that it was really tied to Openstack Ceilometer, forcing their model for data storage, and that the provided documentation was lacking detail, something important in order to make an educated decision. Since the same does not happen with InfluxDB, which provides clear and complete documentation, it was chosen as the database to store aggregated data;
- **Aggregation Tools:** the main feature required by the Aggregation Framework and this thesis objectives was the ability to perform real-time aggregations. Several technologies were investigated: Trifacta Wrangler, MongoDB Aggregation Framework and Apache Storm. Although MongoDB has an aggregation framework, it does not qualify has a valid tool since its aggregation operations are done in an offline manner, after data is stored in MongoDB. This leaves Wrangler and Storm as the remaining technologies to analyse. Wrangler provides a huge tool set for aggregation purposes and, despite not being an open-source technology, it offers a free version of their tool. However, since it is a close-source tool, it does not enable its extension in order to support new features that emerge from the dynamicity that SELFNET requires of the Aggregation Framework real-time component. Since Storm is open-source and is a technology to be used to

implement other frameworks or tools, it fits SELFNET requirements. Moreover, it also provides distributed clustering features with performance in mind, further aligning itself with SELFNET ideals. Given this, Apache Storm was chosen as the aggregation technology;

- **Monitoring Tools:** while other technology categories (e.g. storage, service coordination, etc) elected only one tool from their area, the monitoring tools category is open to electing more than one tool in order to provide a complete monitoring solution. The ability to collect data from the virtualized environment is the main aspect to look for. Gathering data from the physical components was already decided, by the SELFNET consortium, that would be performed either physical sensors and LibreNMS [51] monitoring tool. Because of this, QoSient Argus, which provides a way of collecting network flow information, is left out since there is already a physical sensor (Flow Monitoring Agent, presented in Section 5.2.1.1), developed by a SELFNET partner, that provides the same functionalities. While Apache Chukwa only provides log monitoring, Openstack Ceilometer overlaps and expands the features that Chukwa presents, by supporting a wide range of areas to monitor (e.g. network, storage usage, computational resources, amongst others) from the virtualized environment that is Openstack, a tool that is already used by SELFNET to virtualize its infrastructure. However, in order to monitor the produced aggregation data by the Aggregation Framework, Openstack Monasca offers the capabilities needed to store data with a time-series nature, while providing an easy to use API to expose data. It also integrates, by default, with InfluxDB. Give all of this, Openstack Ceilometer and Openstack Monasca were chosen as the monitoring tools to use.

In the end, and as a whole, the technologies chosen were: Apache Zookeeper for Service Coordination, Apache Kafka for Message Bus, InfluxData InfluxDB for Storage, Openstack Monasca with InfluxData InfluxDB and Openstack Ceilometer for Monitoring Tools, and at last, Apache Storm for Aggregation Tools. As one might notice, this choice of technologies sets an environment mostly comprised of Openstack and Apache tools, providing a rather stable tool set for developing new solutions.

Chapter 4

Architecture

A custom approach to a system architecture allows a better control over the importance of each component when exposing the architecture, something that does not happen when exclusively using models like the Unified Modelling Language (UML) or Architecture View Model (4+1 view model).

Thereby, this section will cover the SELFNET's logical architecture from a high level perspective down to the singular components perspective, i.e. the components which are the focus of this thesis, providing a way to backtrack a low level component up to its place in a higher level view. For this matter, the content presented here will follow a use case driven architecture design that establishes three logical levels:

- **Level 0 (L0):** high level conceptual perspective;
- **Level 1 (L1):** intermediate level conceptual perspective;
- **Level 2 (L2):** implementable perspective.

Please note that this approach only explores the logical side of the architecture, the functional part will be detailed in Section 6.

This chapter will be organized as follows: firstly, the SELFNET architecture will be presented to provide a high level context of the work in this thesis (Section 4.1); second, a zoom-in is provided into the detailed aspects regarding the Aggregation Sublayer (Section 4.2), and last it is presented the impact that the SELFNET use cases have over the Aggregation architecture (Section 4.3).

4.1 SELFNET Architecture

Given that the scope of this thesis is deeply engrained within SELFNET, it is paramount to present its architecture in order to provide context about this work's environment, so that we can further zoom in to the relevant components.

Figure 4.1 represents the SELFNET high level architecture, also referred to as level 0 architecture. It includes the Monitor & Analyser, Autonomic and Orchestration Sublayers, as well as the transversal Catalogue and Inventory Services.

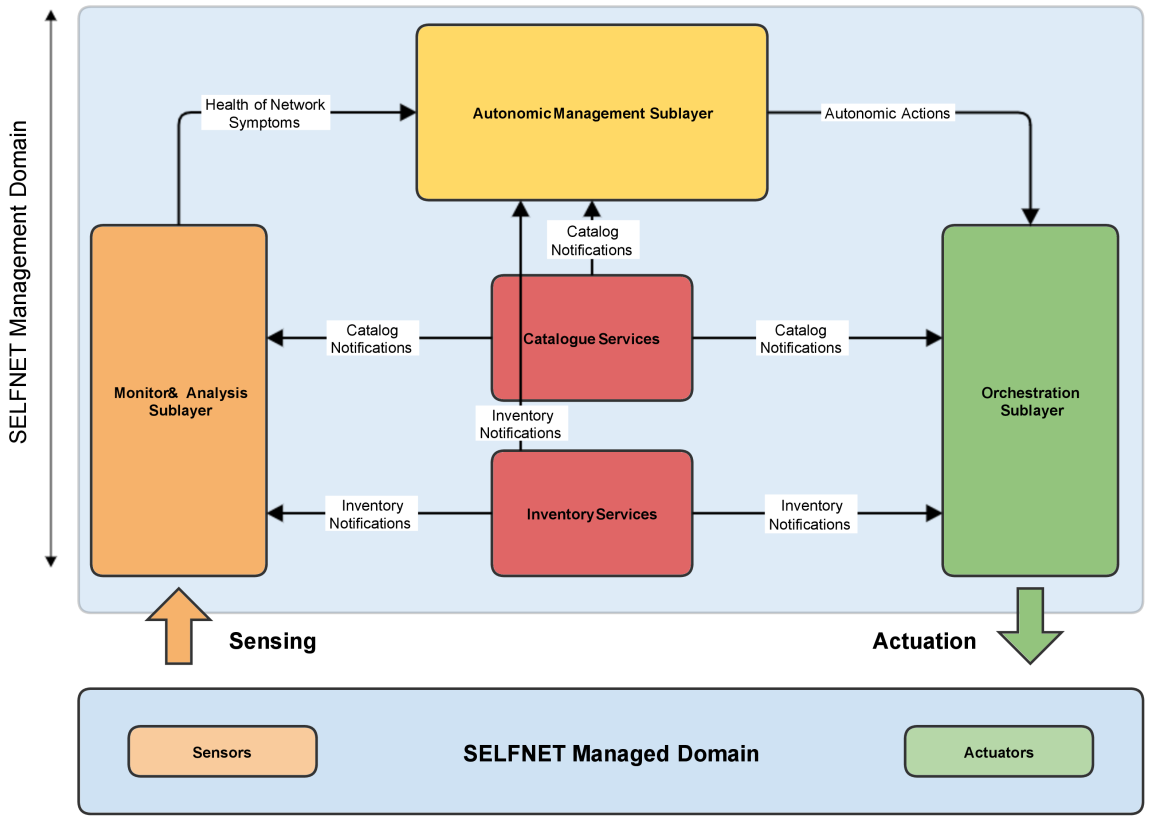


Figure 4.1: SELFNET Logical Architecture Level 0

The Monitor & Analysis Sublayer objective is to gather, aggregate and analyse sensor provided data originated from a 5G network environment composed of NFVs, SDNs and legacy network elements. It is its responsibility to provide HoN symptoms, by using prediction algorithms, which pro-actively indicate potential problems.

Similarly, the Autonomic Sublayer, considered the core of SELFNET, takes the provided HoN symptoms, performs a diagnosis of the most probable causes, and decides on the most appropriate tactics and actions to pro-actively prevent end-to-end services from being affected.

Furthermore, the Orchestration Sublayer is designed to enforce all actions defined in the autonomic process, which includes orchestrating the available heterogeneous network functions (e.g. SDN-Apps, VNFs) and actuators, up to their deployment and configuration processes.

Lastly, the Catalogue and Inventory Services are the entities accountable for managing and making available all on-boarding (Catalogue) and instantiation (Inventory) information for every other components, eliminating the replication of this data across sublayers. The Catalogue Framework and the Inventory Framework are complimentary. The Catalogue is a repository that contains all data regarding internal procedures present in other sublayers, for instance, information about which type of sensors exist, how their data is collected, which aggregation rules take the sensors counters to produce metrics, as well as how these sensors are to be deployed and configured. On the other hand, the Inventory Framework is the repository

that holds the records regarding all instantiation information (i.e. sensor, SDN, NFV and so on, instantiation details).

Figure 4.2 goes down one level on the architecture, zooming in into the Monitor & Analyser Sublayer and exposing the major high-level components: Monitoring layer, Aggregation & Correlation layer and the Analyser layer.

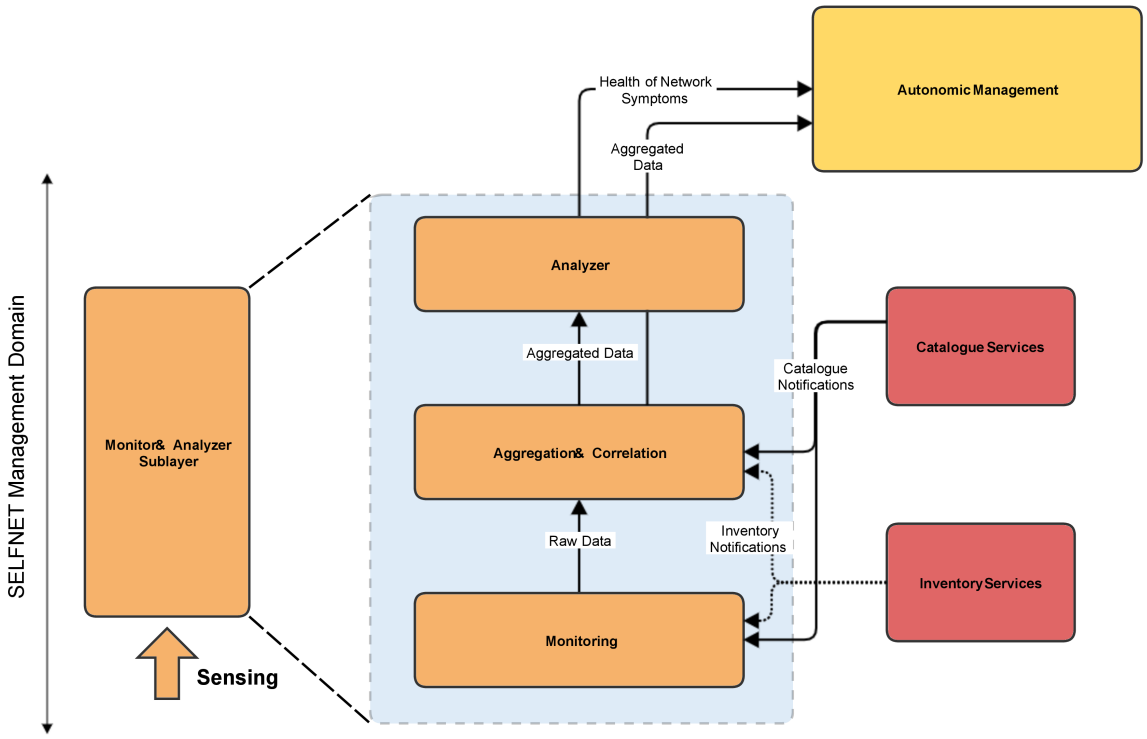


Figure 4.2: SELFNET Logical Architecture Level 1

Starting from the bottom, the Monitoring Framework collects data from sensors which is then persisted, for later use, and forwarded to the Aggregation Framework. Then, the Aggregation Framework aggregates and/or correlates data and provides the results to the Analyser Framework. It is to highlight that, in some specific scenarios, it is possible to forward the aggregated data directly to the Autonomic Framework when, for example, a threshold crossed alert that indicates a severe problem was detected. The Analyser role here is to provide HoN metrics, built by using the provided Aggregation Framework results and applying prediction algorithms. Consequently, the Autonomic Framework will have the necessary HoN symptoms and/or aggregated data to perform its autonomic functionalities. Moreover, Figure 4.2 also depicts interactions between the Catalogue and Inventory services with the Monitor & Analyser Sublayer components, these represent the on-boarding of configurations, e.g. life-cycle management of aggregation rules. With this in mind, it is now clear that the Aggregation Framework role is to be the entity that effectively processes all sensed data, creating meaningful data by correlating and aggregating it, as well as reducing the amount of stored data by transposing it into time-series data. This allows for upper layers to use this

information for analysis purposes and to take intelligent decisions. The framework not only aggregates data, but also offers threshold capabilities, together with notification mechanisms, further removing the other layers burden from constantly polling for new data.

4.2 Aggregation Detailed Architecture

In this section, the Aggregation Framework internal architecture is described in detail. The detailed perspective provided in this section is the level 2 perspective of the Aggregation Framework architecture, according to the terminology previously introduced in Section 4. In order to provide a comprehensive view of the level 2 architecture, it is described, in a step-by-step approach, how level 1 is decomposed into the level 2 perspective. Besides depicting the framework's architecture, a detailed view of the Monitoring Framework architecture is also provided, which interacts directly with the aggregation layer on the southbound (see Section 5.1.2).

Before delving into the detailed architecture inwards, establishing a clear definition of what raw data, aggregated data, counters, events and so on, are:

- **Raw Data:** it is data originated from the sensors itself and is characterized in two different types:
  - **Counters:** statistical data usually periodically reported;
  - **Events:** a specific occurrence that took place in sensed domain (e.g. a Virtual Machine (VM) created event). Raw events can be further extended into:
    - **Alarms:** are associated to a reactive indication that some occurrence has taken place (e.g. hardware failure).
- **Aggregated Data:** refers to already processed raw data and is split into two types:
  - **Metrics:** measurements usually built by aggregating raw counters (e.g. communication frequency metric built by aggregating the packet count between different network elements);
  - **Events:** an aggregated bundle of raw events, generally correlated together to provide meaningful context (e.g. a list of source IPs that communicated with the same destination IP). Aggregated events can also be extended into:
    - **Alarms:** an extension of the aggregated event that originates from a threshold crossed over already aggregated data (e.g. alarm triggered when the bandwidth of a certain network node exceeded a pre-defined value).

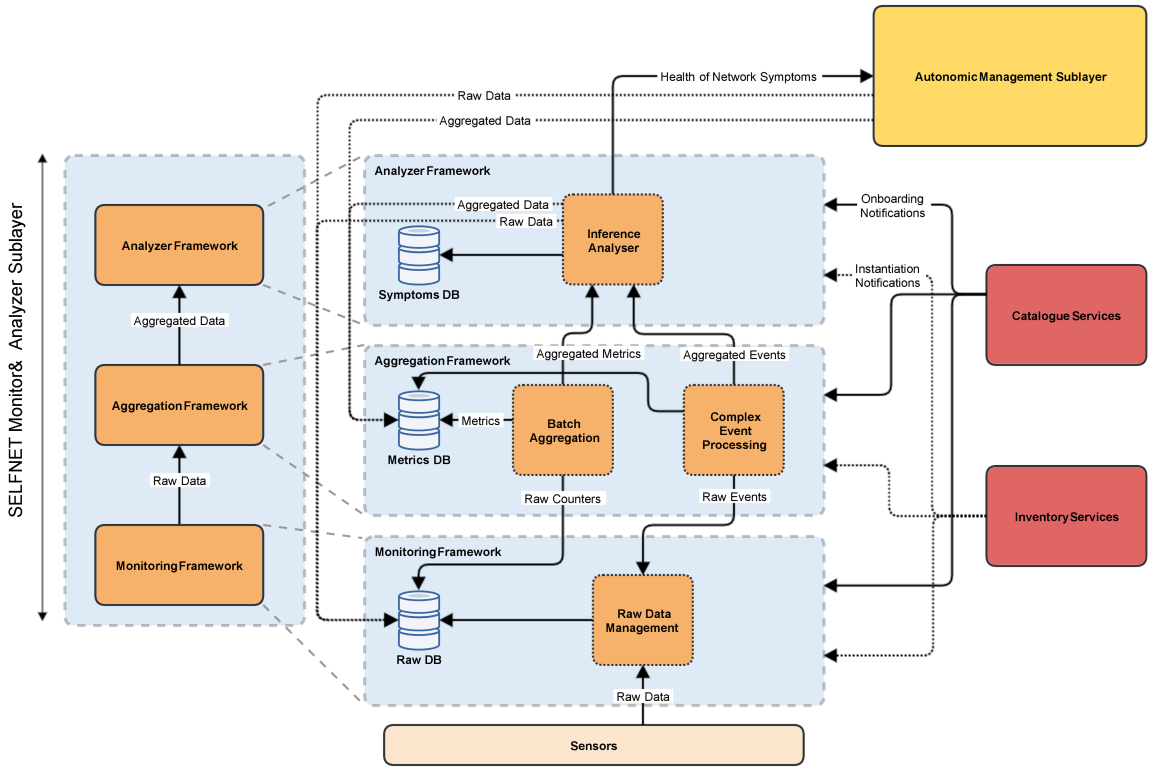


Figure 4.3: SELFNET Logical Architecture Level 2

Figure 4.3 makes the transition from the level 1 architecture (Figure 4.2) to the level 2 architecture depicted in Figure 4.4. The focus here will be the Monitoring and Aggregation Frameworks, the Analyser will be left out since it is not contained in this document's scope. Focusing our attention on the Monitoring Framework, and having in mind this framework's role covered in Section 4.1, we can see that collecting and persisting data is the Raw Data Management high-level component task, further detailed in Section 4.2.1. This data is persisted in the Raw Database (Raw DB), a database that holds all sensor gathered data, providing the most granular access to it.

Furthermore, we have the Aggregation Framework now divided into three major components: Metrics Database (Metrics DB), Batch Aggregation Framework (BAF) and the Complex Event Processing Framework (CEPF). Each one of them possess different roles, the Metrics DB has persisted, in a time-series fashion, all aggregated data.

This aggregated data is created inside BAF, by accessing the Monitoring Framework Raw DB to fetch raw data in batch and aggregate it, in a non-real-time pace (e.g every 30 seconds, 1 minute, 1 hour, etc) and based on pre-defined rules, thus producing metrics. BAF also contains threshold capabilities, outputting aggregation events, when they are crossed, to external systems that previously configured those threshold rules.

Lastly, CEPF provides a real-time, or near-real-time (i.e. provide a result within five seconds), approach to data aggregation/correlation. It processes data in a streaming way and produces aggregation events, similar to the ones provided by BAF, as its output. Those can range from a threshold crossed by a metrics calculated in real-time to a metric that

was aggregated/correlated in real-time. It is to note that this output is rule based, making the results vary from rule to rule. Although CEPF processes data in a streaming fashion, producing an output as soon as it can, the aggregation events it produces are always stored in the Metrics DB, thus providing historical data that may prove useful for future analysis. This component architecture is further detailed in Section 4.2.2.1 and its implementation can be seen in Section 6.2.1.

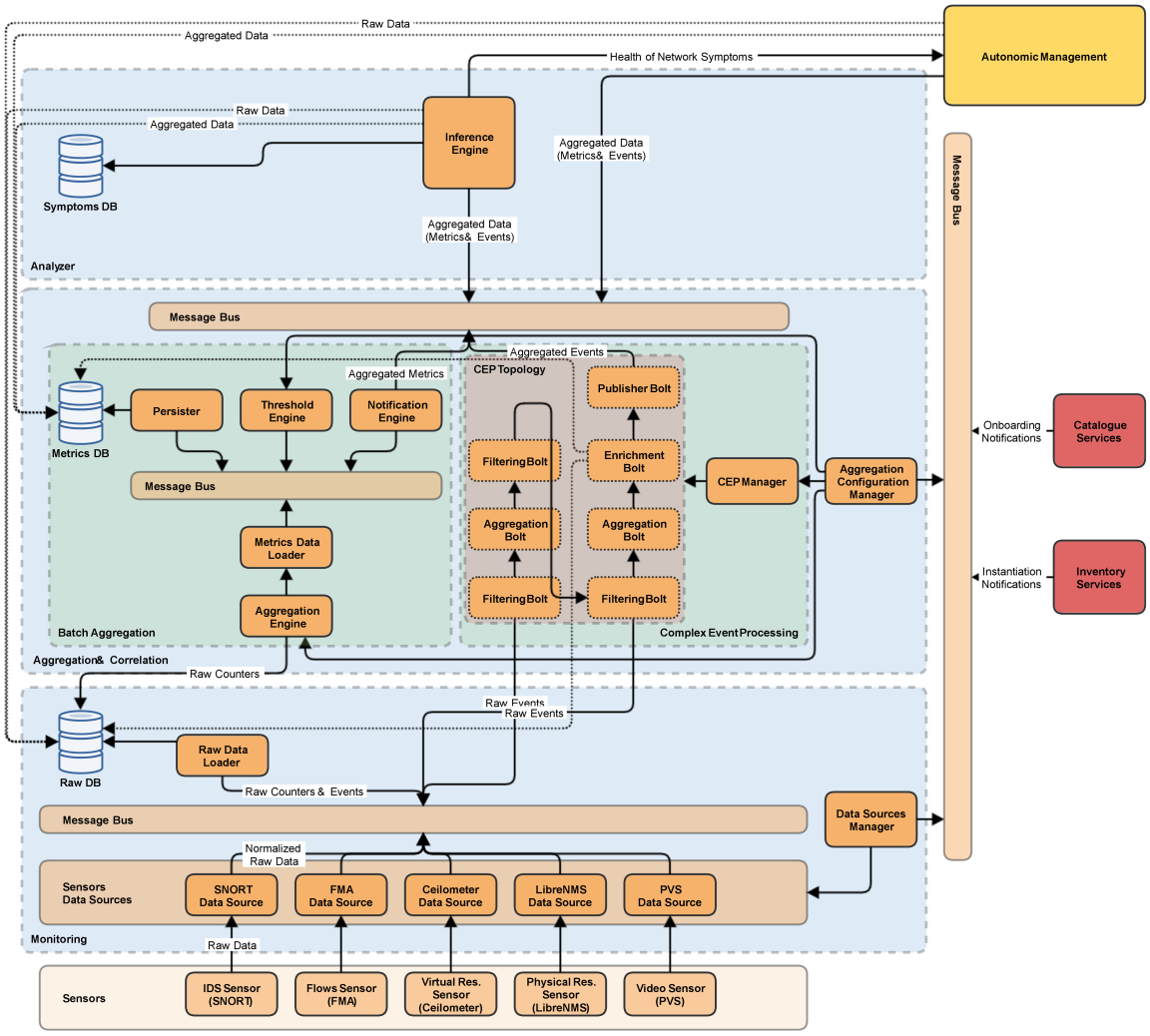


Figure 4.4: SELFNET Logical Architecture Level 2 Detailed

Figure 4.4 further explores the concepts here presented and will support the upcoming subsections in exposing the Raw Data Loader (RDL) and CEPF components, which are the main objectives of this thesis.

4.2.1 Monitoring Framework

Figure 4.4 contains a representation of the Monitoring Framework; however, it is depicted from the Aggregation Framework point of view, thus not addressing every component that the framework possesses. The reason being that it is not relevant to expose all components that do not belong to this document's scope. The exposure of this framework, in this document, is to provide context on how sensors report data and what path it follows until it is stored.

4.2.1.1 Raw Data Loader

As seen in Figure 4.4, Raw Data Loader (RDL) is the component, within the Monitoring Framework, that takes raw data (counters and events) from a message bus and persists them in the Raw Database.

This component task objective is to provide a single data entry point, that enforces a common data model in order to guarantee that data stored in the Raw Database is normalized, therefore minimizing the number of data models and simplifying data access to every other layer or external systems. This common data model is the Raw and Aggregation Data Model, which can be found in Section 5.3.1, and the Raw Database model is depicted in Section 5.3.2.

RDL internals and implementation are further described in Section 6.1.1.

4.2.2 Aggregation Framework

As seen in Section 4.1, the Aggregation Framework is composed of two main blocks, the Batch Aggregation Framework and the Complex Event Processing Framework. However, Figure 4.4 introduces a new component named Aggregation Configuration Manager. This new component is responsible for receiving information from the Catalogue Framework about new aggregation and threshold rules to be applied in BAF and CEPF.

4.2.2.1 Complex Event Processing Framework

The Complex Event Processing Framework (CEPF) main goal is to aggregate and correlate counters and/or events in real-time. Architecturally, the CEPF is split in five main areas:

1. Configuration area to enable dynamic configuration of real-time aggregation rules;
2. Filtering area to protect the upper processing areas from being flooded with large amounts of non-relevant raw counters and/or events;
3. Processing area which collects already filtered raw counters and/or events and produces aggregation events (triggered due to a threshold over a metric, aggregated alerts and aggregated alarms);
4. Persistence area for the produced metrics and/or aggregation events;
5. Publication area to deliver the real-time processing/aggregated data towards the external components of SELFNET.

Figure 4.5 illustrates the CEPF in detail. The ACM is the component responsible for interacting with the Catalogue Framework and propagating the configuration rules internally towards the CEP Manager. The latter is the element responsible for the life-cycle management of CEP topologies. It identifies and manages the required set of CEP internal entities (in this case Apache Storm Spouts and Bolts) that are required to implement the aggregation rules in real-time. The ACM, together with the CEP Manager, implements the configuration area.

Further details, as well as technological choices and implementation details about the CEP Manager are provided in section 6.2.1.1.

The CEP Engine is the entity responsible for performing the real-time data processing. Several types of real-time processing graphs, also known as "CEP Topologies" in the Apache Storm terminology, might be applied. For example, the CEP Engine might implement a single CEP topology per use-case/aggregation rule, or if the required real-time data processing across several use-cases/aggregation rules is similar, the same CEP topology can be applied for several aggregation rules. The decision on the type of CEP topologies that have to be applied to support the configured aggregation rules is made by the CEP Manager component. Internally, a CEP topology includes a set of internal nodes to implement the received aggregation rule, such as:

- **Filtering Node:** receives raw counters and/or raw events and filters the undesired ones;
- **Aggregation Node:** aggregates filtered raw counters and/or raw events based on the aggregation rules;
- **Routing Node:** routes data flowing through the topology based on which actions need to be performed next, for instance, an enrichment or a publish action;
- **Enrichment Node:** enriches data using external data sources e.g. metrics database, raw database, catalogue services;
- **Publisher Node:** as the name states, publishes the CEP output into a message bus or a database.

The CEP Engine embraces filtering, processing and publication areas described above. More details about the CEP Engine can be found in section 6.2.1.

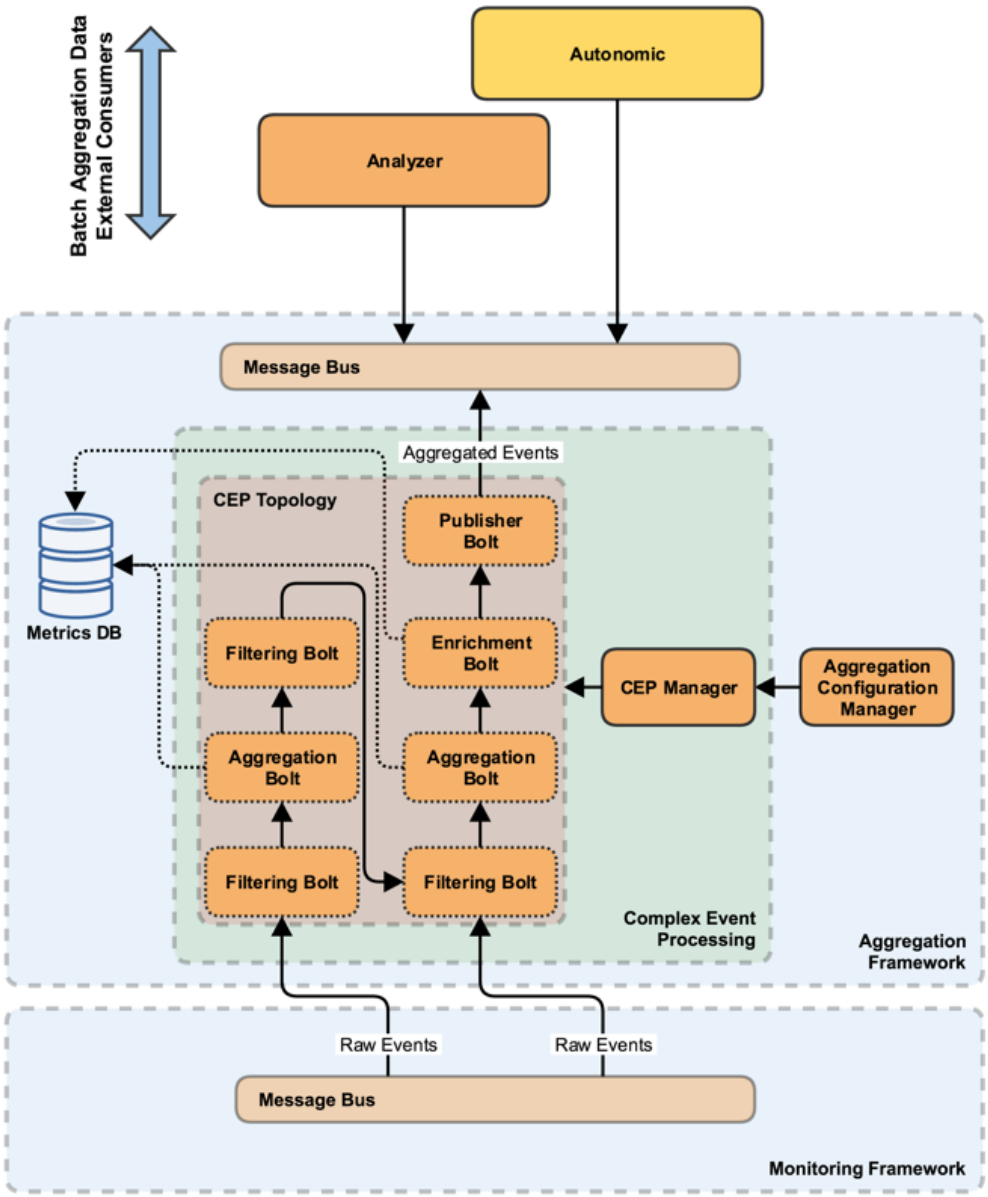


Figure 4.5: Complex Event Processing Engine Framework (CEPF)

4.3 SELFNET Use cases impact on Aggregation Architecture

Having introduced a high level overview of what SELFNET use cases are (see Section 1.3.1), it is important to also expose how these use cases will bring requirements and how they will exercise SELFNET Aggregation Architecture.

4.3.1 Self-Healing

The Self-Healing use case (SH-UC) aims to pro-actively detect potential network equipment failures, for example, a network failure caused by a power outage, and migrate the potentially affected existing network services to another Data Center Point-of-Presence (DC-PoP).

This UC will demand that a very large amount of energy related counters coming from physical energy sensors are persisted and aggregated in a periodical, non-real-time, approach. This will be critical to identify trends and energy fluctuations which might be an important indicator that a network equipment can be approaching a failure state. As a result, the Aggregation Framework, described in Section 4.2.2, must be capable to periodically (e.g. 20 second intervals) fetch energy related counters from the Monitoring Framework, briefly depicted in Section 4.1, and produce contextualized aggregated metrics that indicate energy fluctuations.

When threshold values for these metrics are crossed, the Analyser and/or Autonomic Frameworks must be notified in order to analyse and mitigate the potential network equipment failure risk. Additionally, besides energy-related counters, alerts and/or alarms might also be provided in real-time by the energy sensor, as well as by the sensor retrieving information from the physical and/or virtual infrastructure servers. These alerts and/or alarms might need to be correlated with each other in real-time to produce contextualized metrics about the potential network equipment failure. The Analyser and the Autonomic Frameworks will handle these notification events produced by the Aggregation Framework, to predict potential network link failures and decide on corrective measures, such as service migration for a DC-PoP not affected by the link failure.

4.3.2 Self-Optimization

The Self-Optimization use case (SO-UC) aims to optimize the Quality of Experience (QoE) of a 4K resolution video stream in congested network links.

Figure 4.6 illustrates the SO UC control loop. Sensing information about all the network flows traversing the network, as well as counters about the physical network equipment is sent towards the Monitor & Analyser Sublayer. All sensing information is: persisted (Monitoring Framework), aggregated (Aggregation Framework) to calculate the Congestion Index (CI) and analyzed (Analyser Framework) to evaluate the QoE impact in the end-user consuming the video-stream.

In order to enable the Analyser Framework to evaluate the end-user QoE, the Aggregation Framework must calculate and provide the CI for the video data path. The CI metric is calculated based on information retrieved from the flows sensor (FMA, see Section 5.2.1.1) and from the physical equipment sensors (e.g. SNMP, see Section 5.2.1.4).

Since it deals with congestion in video-streams, all the processing performed at the aggregation-level must be made in real-time, therefore stimulating the real-time components

of the Aggregation Framework. Finally, but of crucial importance, this UC also requires that large amounts of data (mainly network flows) are processed by the real-time aggregation components.

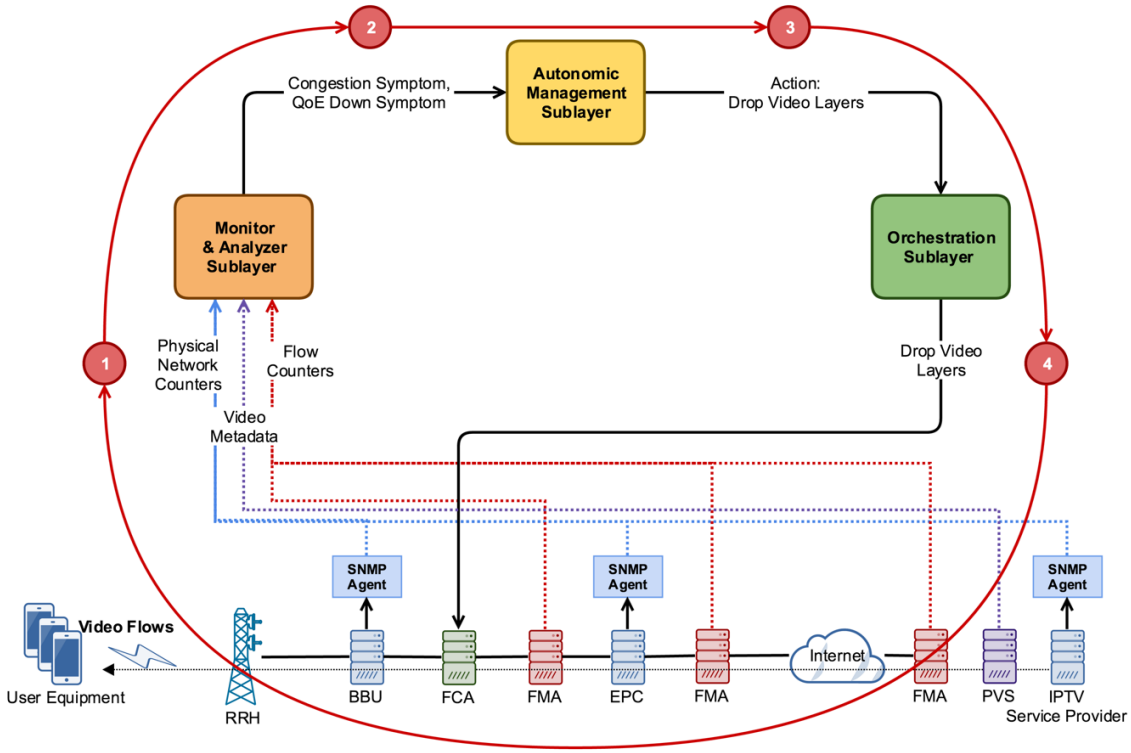


Figure 4.6: Self-Optimization Use-Case

4.3.3 Self-Protection

The Self-Protection use case (SP-UC) aims to identify cyber-attacks in advance and proactively take a set of countermeasures to isolate the attacker and the affected users (also known as zombies). The UC foresees two main loops. In the first loop of the SP UC, depicted in Figure 4.7, all the network flows in the data path are monitored through the Flow Monitoring Agent (FMA, see Section 5.2.1.1) sensor and raw counters are reported to the Monitor & Analyser Sublayer. Within the Monitor & Analyser Sublayer all flow counters must be: persisted (Monitoring Framework), aggregated (Aggregation Framework) and analysed (Analyser Framework) for the identification of a potentially suspicious network communication pattern.

In what concerns the Aggregation Framework, this UC requires that a large amount of network flows are aggregated in a non-real-time nature (e.g. 20 seconds) to identify suspicious network communication patterns. The latter is identified by aggregating all network flows that have the same source and destination nodes (e.g. using the IP address and communication ports) within pre-defined time periods. The aggregated flows must be persisted within the

Aggregation Framework, and when a pre-defined set of consecutive suspicious communication patterns between the same nodes is identified (e.g. frequency of and interval between sessions within a certain period), the Analyser Framework must be notified. The rationale to adopt a notification-based approach between the Aggregation and the Analyser Framework is to avoid the permanent queries from the Analyser towards the Aggregation without any context and, most probably, without any risky situation in place.

The persisted aggregated flows in the Aggregation Framework must be made available to external components of SELFNET (e.g. Analyser and/or Autonomic Management Frameworks) for further analysis. Moreover, and very importantly, the Aggregation Framework must enable external SELFNET components (e.g. Analyser and/or Autonomic Management Frameworks) to change the aggregation rules during runtime for taking advantage of the Aggregation Framework capabilities to produce useful views and projections of the network conditions/status.

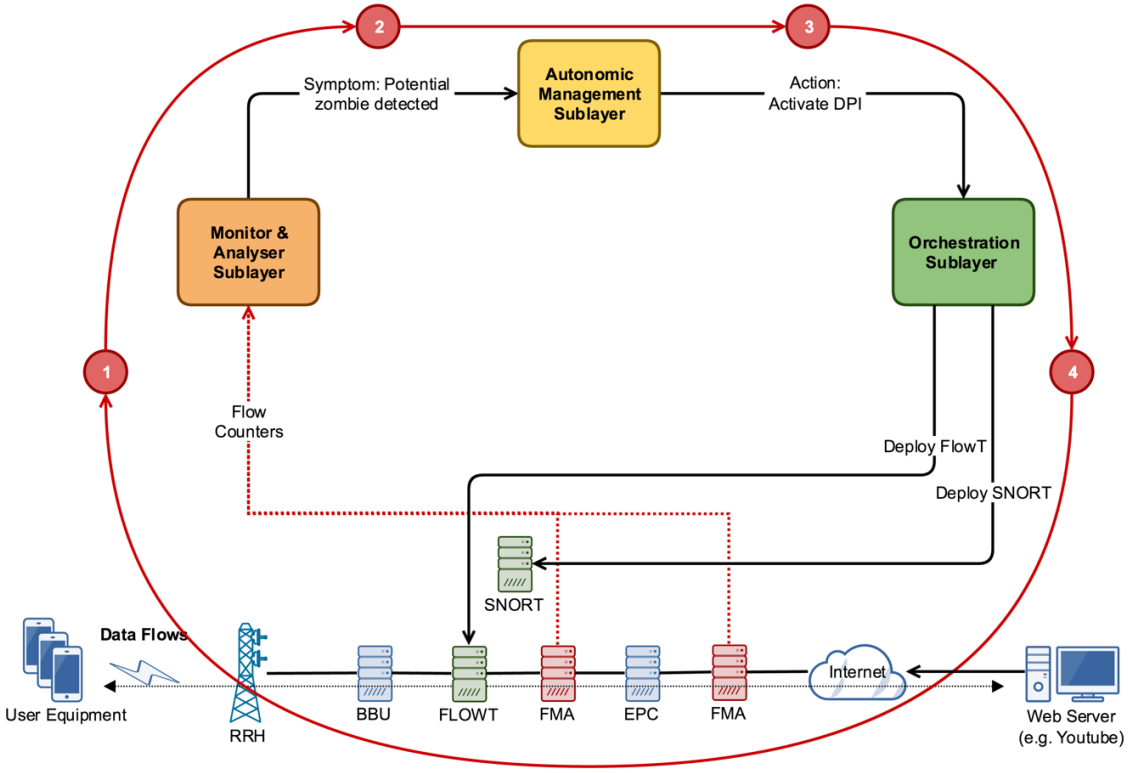


Figure 4.7: 1st Loop of the Self-Protection Use-Case

When a suspicious network communication pattern is identified, a report is generated towards the Autonomic Management Sublayer to be evaluated as symptom for further triggering a reaction in terms of a request that is sent to the Orchestration Sublayer to activate a Deep Packet Inspection (DPI) Virtual Network Function (VNF) (e.g. SNORT 5.2.1.2). The latter will confirm that the identified suspicious network communication is in fact a cyber-attack (also known as botnet). For the DPI function to be able to analyse the flows, the

Orchestration Sublayer also has to deploy a flow mirroring SDN application.

After SNORT and the flow mirroring are deployed and all the required configurations are performed by the Orchestration Sublayer, the second loop of the SP UC, represented in Figure 4.8, is initiated.

The DPI VNF sensor will trigger alert events towards the Monitor & Analyzer Sublayer when the suspicious communications are confirmed that they involve certain packet content. This information reaches the Aggregation Framework and must be correlated in real-time with all the SNORT alert events that are under the control of the same attacker (Command & Control Server). This will enable the identification of all the compromised user-equipment, also known as zombies, by the Command & Control Server. As in the first loop, also in this case it is under the responsibility of the Aggregation Framework to notify in real-time the analysis procedures with the aggregated alerts, also known as meta-alerts.

Finally, dynamic configuration procedures for the real-time procedures of the Aggregation Framework must also be provided to enable external SELFNET components to manage the aggregation rules according to its needs (for example, creation of a new aggregation rule to evaluate the performance of a deployed action coming from the machine-learning algorithms).

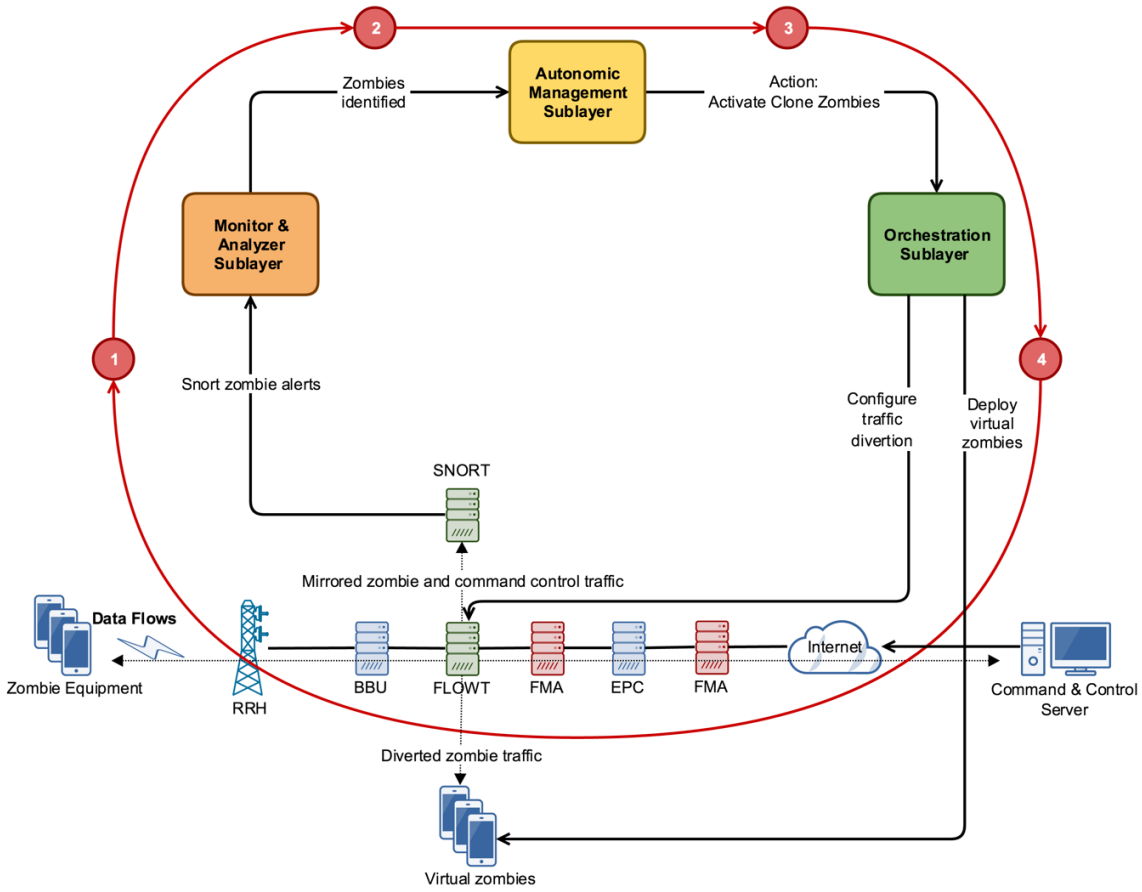


Figure 4.8: 2nd Loop of the Self-Protection Use-Case

4.4 Summary

This chapter started by introducing the three logical levels' approach used to expose the architecture with a progressive increase in detail. After having the approach in mind, the SELFNET architecture (Section 4.1) was presented with the first two logical levels, laying down the necessary information to understand the two upcoming sections.

In order to reach the next logic level, the Aggregation Detailed Architecture (Section 4.2) section further explored the Monitoring and Aggregation Frameworks components that belong to this thesis objectives. To finish off, the last section (Section 4.3) covered the impact which SELFNET use cases have on the presented architecture, thus making clear which functionalities need to be supported.

Chapter 5

Interfaces, Data Sources and Data Models

To better understand how data is produced, represented and how it is available to the Aggregation Framework components to be discussed in the Implementation section 6, this section dedicates its space to provide this information by firstly covering the entities producing data i.e. data sources (Section 5.2), secondly, which interfaces (Section 5.1) exist and how they are composed, and lastly, providing the data models (Section 5.3) used.

5.1 Aggregation Framework APIs

This section describes the interfaces exposed by Aggregation Framework towards external components, enabling a programmatic interaction at three levels: Southbound - used for data ingestion; Northbound - used for data to be outputted; Configuration - used to apply configurations over the Aggregation Framework components.

5.1.1 Northbound

The API exposed towards external systems, it is located at the North side of the Aggregation Layer. It is composed as a set of component specific APIs, due to the different aggregation strategies used, as an example: streaming or real-time processing implies that data is processed as it arrives, as a stream, making its output to also be provided in real-time, in a queue like mechanism; in opposition, batch aggregation performs its tasks offline, using a non-real-time cadence, aggregating data when it is available in the Raw Database to later persist it the Aggregated Metrics Database, thus making the usage of queries to retrieve data a natural process. These component specific APIs are described in the following subsections.

5.1.1.1 Aggregated Metrics Database API

This API is provided by the Monasca component present in the architecture revealed in Section 4.2.2. All external interactions with the Monasca service are done via its REST API [57]. The API supports storing and querying metrics measurements and statistics, CRUD operations on compound alarm definitions and notifications methods as well as read/delete operations on alarms.

The API authenticates all requests against the Keystone service, and all data is associated with a tenant to support multi-tenancy.

Metrics are described by a name and dimensions, which consists of a dictionary of (key, value) pairs, that allows a flexible, concise and self-describing representation.

When it comes to the Aggregation Framework, only a subset of the supported operations are to be used externally while the remaining ones are used internally or not used at all. The externally supported operations are presented in Table 5.1.

Metrics	List metrics [58]
	List dimensions values [59]
	List dimensions names [60]
Measurements	List measurements [61]
Metrics names	List names [62]
Statistics	List statistics [63]

Table 5.1: Monasca API - Read-only operations

The remaining operations, not listed here, are either unused or not particularly relevant regarding external interactions. In a general way we can state that all write operations (creating, updating, patching or deleting) are exclusively used within the Aggregation Framework and do not need to be exposed for external usage.

5.1.1.2 Raw Database API

The Raw Database API exposes the raw data persisted in the Raw DB to the external and internal components. 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). All data stored in this database follows the model defined in Section 5.3.2.

5.1.1.3 Alarm Notifications API

The Alarm Notifications API is represented in the Monasca Notification Engine, the Monasca component responsible for publishing alarm notifications whenever an alarm is triggered by the Monasca Threshold Engine. It supports several notification types such as Email, Pagerduty and Webhook. These notifications are plugins of the Notification Engine which, as the name suggests, is the core of the Monasca Notification component.

For the purpose of our solution, a new plugin was developed for SELFNET with the objective of publishing the alarm notifications to a Kafka topic exposing them to external consumers. The alarm notifications published on Kafka are in JSON format and follow the structure specified by the Raw and Aggregation Data Model (RADM), detailed in Section 5.3.1. Only the content of two specific fields of the model will be detailed here (resourceDescription and dataDefinition), leaving out the remaining ones as they are explained in the RADM section:

- **resourceDescription:**
  - It is essentially composed by fields of several IDs which are self-explanatory, all of them internal to all Monasca components (previously introduced in Figure 3.10);

- **dataDefinition:**
  - **severity:** the severity of the alarm which is an integer value between 0 and 3 with the following representation (mapped with the available Monasca severity levels):
    - 0 - LOW;
    - 1 - MEDIUM;
    - 2 - HIGH;
    - 3 - CRITICAL.
  - **newState:** the current state of the alarm (the three possible states are OK, ALARM or UNDETERMINED);
  - **oldState:** the previous state of the alarm (OK, ALARM or UNDETERMINED);
  - **alarmDescription:** the description of the alarm as set in the alarm definition;
  - **alarmTimestamp:** the unix epoch timestamp of the alarm in milliseconds;
  - **metadata:** an array of key value pairs of extra information;
  - **metrics:** the metrics that triggered the alarm:
    - **name:** the name of the metric associated with the alarm;
    - **dimensions:** the dimensions of the metric, composed of key value pairs;
    - **id:** the monasca internal ID of the alarm, if applicable.
  - **subAlarms:** an array with the subAlarms that together triggered the alarm:
    - subAlarmExpressions: a part of the overall expression of the threshold:
      - **function:** statistical function of the expression which can be a min, max, sum, count, avg or last;
      - **deterministic:** boolean - deterministic alarms only have two possible states, either OK or ALARM, while the nondeterministic ones can also have the UNDETERMINED state;
      - **period:** the period in seconds;
      - **periods:** the number of periods taken into account;
      - **threshold:** the threshold;
      - **operator:** the relational operator which can be a less than (lt, also <), greater than (gt, also >), less than equal (lte, also <=), greater than equal (gte, also >=);
      - **metricDefinition:** the metric used in the threshold expression:
        - **dimensions:** the dimensions of the metric, composed as key value pairs;
        - **id:** the monasca internal ID of the alarm, if applicable;
        - **name:** the name of the metric;
    - **currentValues:** an array of the current values of the metric (one per period if more than one period is specified);
    - **subAlarmState:** the state of the subAlarm (OK, ALARM or UNDETERMINED).

5.1.1.4 Aggregation Events API

Similar to the Alarm Notifications API, as seen in Section 5.1.1.3, the Aggregation Events API follows the same rationale about the usage of a messaging bus. It is a single Kafka messaging bus topic, used by the CEPF, described in Section 6.2.1, to output aggregation events. This outputted data is modelled using the RADM, detailed in Section 5.3.1, using its model to model events and alarms.

5.1.2 Southbound

Southbound interface is comprised of a single API. It is used to input sensor reported data into the Aggregation Layer. The API itself is a Kafka messaging bus, which follows a publish/subscribe communication model and provides a multiple topic usage used to separate different types of data, i.e. counters or events. In order to keep the reported input data normalized, it must be modelled according to the Raw and Aggregation Data Model (RADM) described in Section 5.3.1.

5.1.3 Configuration

The configuration interface of the Aggregation Framework will be provided by the Aggregation Configuration Manager (ACM), which unfortunately is still yet not defined, and so, it cannot be fully materialized here in this section, despite it has already been referenced in Section 4.2. However, ACM will provide an internal API for the Aggregation Framework components to propagate the externally received configurations. This API will be implemented over the Zookeeper service, mentioned in Section 3.1.1.

5.2 Data Sources

5.2.1 Sensors

This subsection will cover the relevant sensors and other components that generate data in the SELFNET environment.

5.2.1.1 Flow Monitoring Agent

Flow Monitoring Agent (FMA) is a piece of software categorized as a physical network function (PNF), designed to monitor network flows, with a great focus on Ultra High Definition (UHD) video flows. It reports flow counters and metadata (classified as an event) to a configurable message bus, thus being able to interact with the Southbound API (Section 5.1.2). This data is modelled after RADM (Section 5.3.1).

5.2.1.2 Snort

Snort is an open-source intrusion prevention system (IPS) that allows real-time traffic analysis and deep packet inspection (DPI)[50]. In the SELFNET context, the DPI feature is used to detect botnets and to identify users that were infected and turned into bots. Although this sensor does not report directly in the RADM format (Section 5.3.1), the Monitoring Framework supplies a SNORT data source, that translates the SNORT specific format into RADM. This data is sent to the Aggregation Framework via message bus as an event.

5.2.1.3 Ceilometer

The previously presented Ceilometer in Section 3.5.3 can also be seen as a data source, since it provides both counters and events originated from the virtualized environment (e.g. virtual machine CPU usage, Openstack virtual router current incoming packets, etc). In order for it to provide this data in the RADM model (Section 5.3.1), SELFNET provides a

modification to the publisher mechanism that Ceilometer possesses, thus being able to provide raw data in the proposed RADM format and to a message bus.

5.2.1.4 LibreNMS

LibreNMS is a multi-platform open source network monitoring system that supports a wide range of hardware and operating systems [51]. It is able to perform automatic devices discovery over several protocols (e.g. Link Layer Discovery Protocol (LLDP), Cisco Discovery Protocol (CDP), Border Gateway Protocol (BGP), amongst others) and monitor them by mainly using Simple Network Management Protocol (SNMP). Like the other sensors presented in this section, SELFNET also provides a data source that translates LibreNMS gathered data into the RADM model (Section 5.3.1) and publishes it to a message bus.

5.3 Data Models

5.3.1 Raw and Aggregation Data Model

The Raw and Aggregation Data Model (RADM) is a model that seeks to unify data flowing through the Aggregation Framework interfaces, representing statistical counters, events or alarms. It is comprised of a generic model that can be used to report different resource types using distinct reported data types. This is possible due to the model structure since it uses a set of generic and mandatory fields that allow to model any kind of data related to this scope (counters, events, alarms).

It envisions report data types as three different categories:

Category	**Description**
Counters	Any reported data that contains statistical fields, reported periodically or not. A good example would be the data produced by the FMA sensor (see Section 5.2.1.1), which can be reported periodically or only when anything on a flow changes. Regardless of its periodicity, from the aggregation point of view, it is statistical data comprised of a set of counters (e.g. outcomingOctets, lostPackets, etc).
Events	Event based reports regarding any change detected by sensors; it can range from a zombie detected event produced by the SNORT DPI sensor (see Section 5.2.1.2) to a simple VM instance created event originated from Ceilometer (see Section 5.2.1.3).
Alarms	A particular case of the event data type, where the data reported is relative to an alarm state transition.

Table 5.2: RADM Report Data Types

Below you can find the generic model together with a brief explanation about its fields. The following bulleted list depicts the model fields:

- **Data**:
  - **timestamp**: report time in epoch milliseconds;
  - **dataType**: type of data being reported, it can have one of three possible values: statistics, event, alarm;

- **reporterID**: a full distinguished name that uniquely identifies a reporter, it is built using fields and values from reporterDescription attribute;
- **resourceType**: an attribute that reflects the type of resource being reported, it can be a flow (see Section 5.2.1.1), virtual machine network counter (see Section 5.2.1.3), a snort dpi event (see Section 5.2.1.2), etc;
- **resourceID**: a full distinguished name that uniquely identifies the resource being reported, it is built using fields and values from resourceDescription attribute;
- **resourceDescription**: this attribute has a map as its value that represents all attributes describing the resource being reported, it is a string to string map;
- **dataDefinition**: a map that assumes two different types of values, it comes as a string to double map if the field dataType as the value statistics, or it comes as a string to string map if dataType is different than statistics:
  - **severity**: a field that must be included when dataType is alarm, it is expected to have a numeric value in order to keep severity values normalized;
- **reporterDescription**: this attribute has a map as its value that represents all attributes describing the reporter that reported this message, it is a string to string map.

Examples about this model can be found in Annex A.1.

5.3.2 Raw Database Data Model

Since there is a clear distinction between counters and events, as previously stated in Section 4.2, this section presents two tables that summarize how counters and events are stored in Raw DB: Table 5.3 refers to counters and Table 5.4 refers to events.

Column	Description
timestamp	Identifies the time (in milliseconds) when the raw counter (sample) was collected.
timepartition	Data partitioning column, derived from the timestamp of the data.
resourcetype	Identifies the type of resource from which the raw counters are being collected.
resourceid	Identifies the resource from which the raw counters are being collected.
reporterid	Identifies the sensor that is providing the data.
countertype	Identifies the type of counters present in each table entry.
resourcedescription	Describes the resource (and its hierarchy relationships, i.e., relation with other/underlying resources) from which data is being collected. Consists of a list of one or more resources hierarchy relationships. Each resource entry contains the following fields: • Name: - String that identifies the name of the resource; • Value: - Resource value.
datadefinition	Describes the raw counters. Consists of a list of one or more counters. Each counter entry contains the following fields: • Name: - String that identifies the name of the raw counter; • Value: - Raw counter numeric value.
reporterdescription	Describes the sensor reporting the data. Consists of a list of one or more attributes. Each entry contains the following fields: • Name: - String that identifies the name of the attribute; • Value: - Attribute value.

Table 5.3: Raw DB Counters structure

Column	Description
timestamp	Identifies the time (in milliseconds) when the raw event was collected.
timepartition	Data partitioning column, derived from the timestamp of the data.
resourcetype	Identifies the type of resource from which the raw events are being collected.
resourceid	Identifies the resource from which the raw events are being collected.
reporterid	Identifies the sensor that is providing the data.
dataType	Identifies the type of event present in the table entry (i.e. if it is an alarm or a normal event).
resourcedescription	Describes the resource (and its hierarchy relationships, i.e., relation with other/underlying resources) from which data is being collected. Consists of a list of one or more resources hierarchy relationships. Each resource entry contains the following fields: • Name: - String that identifies the name of the resource; • Value: - Resource value.
datadefinition	Describes the raw counters. Consists of a list of one or more events. Each counter entry contains the following fields: • Name: - String that identifies the name of the raw event; • Value: - Raw event string value.
reporterdescription	Describes the sensor reporting the data. Consists of a list of one or more attributes. Each entry contains the following fields: • Name: - String that identifies the name of the attribute; • Value: - Attribute value.

Table 5.4: Raw DB Events structure

Both CQL Data Definition Language files (DDL) that create these models as Cassandra tables are available at Annex B.1 (counters) and B.2 (events).

5.4 Summary

This chapter started by exposing the existing Aggregation Framework APIs: northbound, southbound and configuration; and how they provide access to the gathered and aggregated data. Moreover, the relevant SELFNET entities that produce sensing data were covered in order to provide an overview of what data is available to collect and process. Lastly, but not less important, the defined data models that provide a unified and standardized view over both sensed and aggregated data were presented.

Chapter 6

Implementation

This chapter covers all implementation details about the Raw Data Loader and Complex Event Processing Framework components, which respective architectures were already introduced in Section 4.2.

To provide a logical link between the architecture presented beforehand (Chapter 4) and this implementation chapter, the latter is organized in the following way: from the Monitoring Framework, the Raw Data Loader implementation will be covered, and from the Aggregation Framework, the Complex Event Processing Framework will be detailed.

6.1 Monitoring Framework

6.1.1 Raw Data Loader

Raw Data Loader (RDL) is the component within the Aggregation Layer responsible for loading raw data that is published via Southbound API (Section 5.1.2) into the Raw Database. This process implies that such data is parsed, validated and mapped in such a way that can be persisted. The next subsections will cover the RDL's internal architecture, what data is valid as input, how such data is mapped and how can RDL be configured.

6.1.1.1 Internal Architecture

RDL was designed using a modular approach so it can be easily extended to support new features or data models if necessary. As of the date, RDL works in a multi-threaded and asynchronous way, making use of the modern computational resources available. Figure 6.1 represents its architecture.

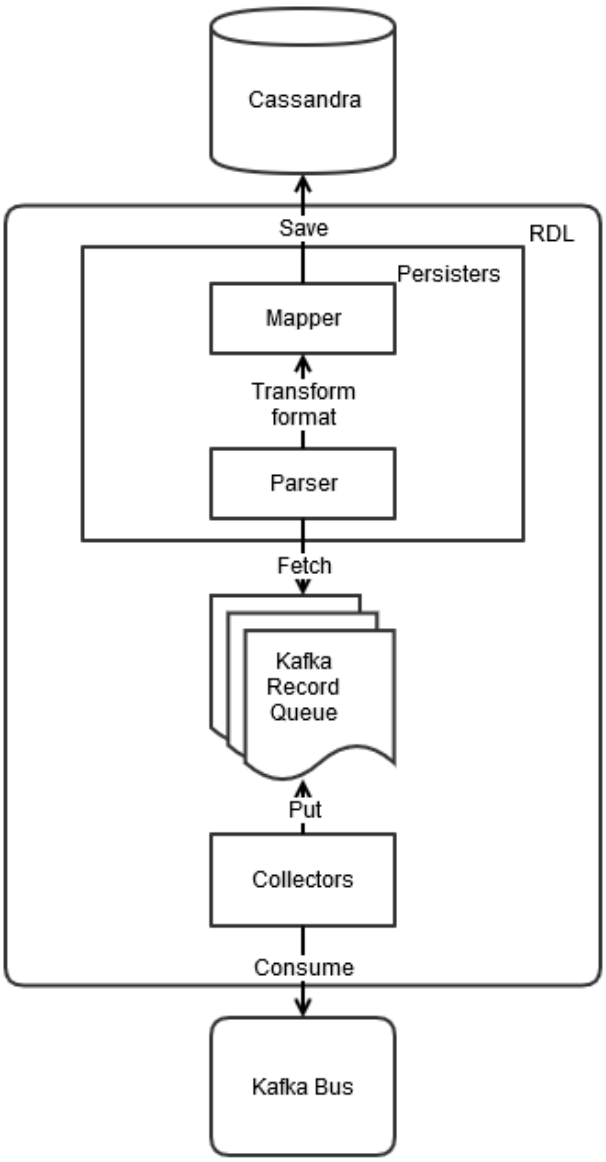


Figure 6.1: Raw Data Loader Architecture

From the bottom to the upper parts of RDL:

1. **Collectors:** a set of collector threads that consume data records from the configured Kafka topics and then append the consumed data to the Kafka Record Queue;
2. **Kafka Record Queue:** a concurrent data structure that is available to every collector and persister;
3. **Persisters:** like the Collectors, it is a set of persister threads that fetch Kafka records from the Kafka Record Queue and submits them to a transformation process so they can be stored in Cassandra:
   (a) firstly, they are parsed into JSON objects before being handed to the Parser that validates and transforms those JSON objects into classes that represent either

counters or events;

(b) those classes are given to Mapper so it can persist them into Cassandra.

Although not explicit in the Figure 6.1, RDL is able to load its configuration via a YAML file (see example in Annex C.1.1), specified at deployment time, allowing control over how many Persister and Collector threads to be used, which Kafka message bus and topics to use, where is Cassandra database located, amongst others.

6.2 Aggregation Framework

6.2.1 Complex Event Processing Framework

The Complex Event Processing Framework (CEPF) was designed as an alternative and complement to the Batch Aggregation Framework (BAF), presented in Section 4.1. While BAF is meant to process all available data, providing simple and complex metrics over multiple counters aggregated by different time periods, it inevitably is a batch processing system. Its priority is not fast extrapolation of data but reliable and extensive metric collection.

The concepts behind CEPF are the same of a real-time business intelligence (RTBI), but in a networking context. These are the ability to deliver real-time (or near to zero latency) access to information whenever it is required, and the ability to deliver intelligence or information about operations as they occur. Thus, CEPF was developed to provide real-time metrics over smaller segments of the network and data sources, according to demand and with low latency. Apache Storm is a distributed stream processing computation framework that targets exactly this scenario, meaning that data produced and collected continuously is naturally processed in the same way, following the streaming concept. It relies on its easy, by design, horizontal scalability to be a tool of many sizes, being able to deal with the different traffic load of network events. On a Storm cluster different topologies can be deployed to target different purposes and make use of optimized parallel computing capabilities. With this in consideration, CEPF has been defined as a set of topologies that run over a Storm cluster.

The main concept behind the developed topologies is a set of bolts that map simple but effective functions for event processing, while keeping these bolts stateless. This way is possible to generalize complex event aggregations into multiple sets of simple functions. The main functionalities defined are:

- **Kafka spouts:** These are the sources of events in the topology. Individually they are Kafka consumers threads that consume and track the topic partitions assigned to them. They ensure that every event is successfully consumed and processed on the topology, taking care of reproducing the event if necessary (e.g. in a failure situation). They also ensure mechanisms for throttling and maintaining a healthy flow of events in the topology. Finally, they also translate events from the Raw and Aggregation Data Model JSON format to Storm Tuples;
- **Filter bolts:** these are bolts that actively allow or block/discard events on the topology. From a filtering function, e.g. interface.id = xyz or avgBandwidth >500kbps, the events are allowed or not to progress in the topology;
- **Correlation/Aggregation bolts:** are used to join information from multiple events into a single one. They can be spatial and/or time aggregators in the sense that they can gather events from different sources and locations in the network with distinct time periods. These can be called aggregators bolts when they apply mathematical

formulae over the information contained in the events, as in calculating the average bandwidth, or correlation bolts when they simply join events while maintaining their relevant information, as in listing all IP addresses of a botnet from individual events. These are the most relevant bolts since they are the ones that require memory to store events over a short time period, and the ones that perform calculations over the events data;

- **Enrichment bolts:** are responsible for retrieving information from external sources and adding them to an event, e.g. collecting additional data from a database or enriching an event with rule defined metadata;
- **Thresholding bolts:** are the bolts that apply rule based thresholds over the events flowing through them. These can be effectively considered filtering bolts, however, due to their specialized nature they are given a distinct category. They only output threshold crossed events when the rule defined conditions are met;
- **Routing bolts:** as the name indicates, they are bolts that perform routing functionalities within a topology. Normally, bolts are chained together in a sequential way, e.g. a Kafka spout connected to a filter bolt and then connected to a publisher bolt, thus always applying the bolts functionalities in the same order. These bolts allow a topology to change this concept and guide the events following another logic;
- **Publisher bolts:** are responsible for outputting the events to the outside of the topology. This can be done to a database or the Kafka bus.

In CEPF, a generic and dynamic topology was defined and implemented with seven elements containing a kafka spout, a filtering bolt, an aggregation bolt, a routing bolt, a thresholding bolt, an enriching bolt and a publisher bolt. This is observed in Figure 6.2 and further detailed in Section 6.2.1.1.

This approach allows for more complex topologies, targeting the SELFNET use cases in section 4.3, to be defined by combining additional bolts or a topology built using other topologies.

6.2.1.1 Engine

CEP Engine is composed of the Apache Storm framework, used to run the developed topologies. In a sense, the actual engine are the topologies themselves which provide the computation capabilities needed to address the desired use cases. For this matter, the focus here will be on the developed topology named Generic Topology.

6.2.1.1.1 Generic Topology

The Generic Topology as firstly been designed to address the Self-Protection use case (Section 4.3.3) requirements. It was a topology that had a static behaviour, it would only process data incoming from various SNORT data sources (Section 5.2.1.2) that were reporting zombie detection events, supplying the topology with several reports about different source IPs communicating with one destination IP (recognized as the command and control server that issued attack vectors to zombie devices). These reports were grouped by their destination IP, and an aggregation function would compile all source IPs under a common list, with a five second periodicity. This same functionality is now addressed by making use of the Generic Topology and its runtime dynamic configuration feature.

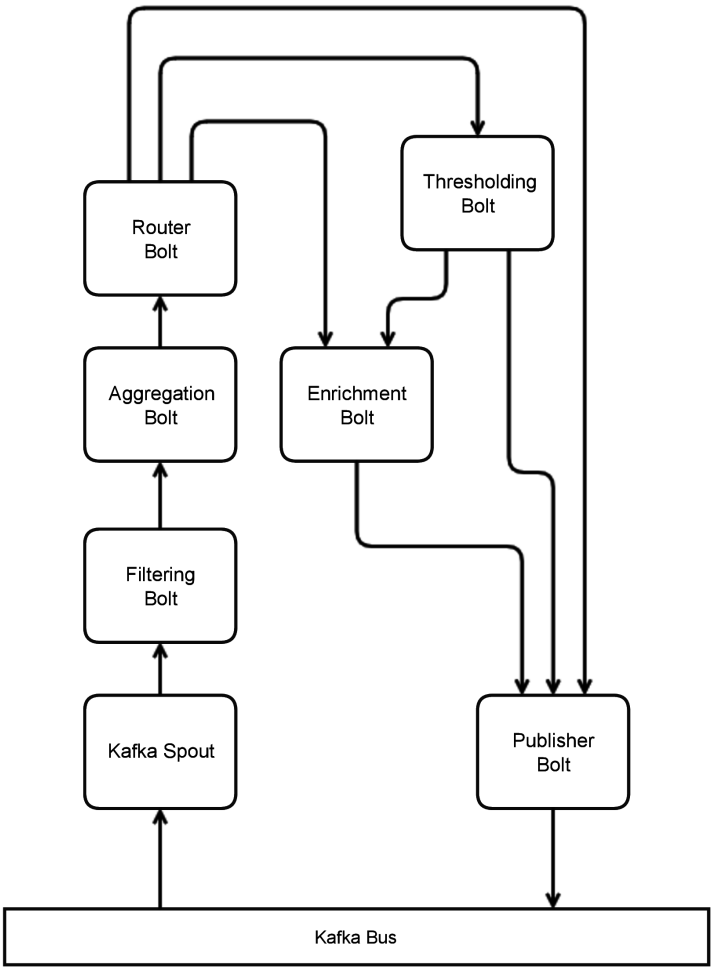


Figure 6.2: Generic Topology

Figure 6.2 presents the Generic Topology and its components, along with the Kafka message bus interaction. These components are described as follows:

- **Kafka Spout:** consumes Kafka records, validates their content and parses them into Storm Tuples, then it forwards the parsed data into the filtering bolt;
- **Filtering Bolt:** filters tuples based on a dynamic white-list, allowing only the tuples that meet the criteria set by the filter. This white-list is rule based and, since every rule has a unique identifier associated to it, data that successfully passed through the filter is tagged with this ID, allowing other bolts that receive this data to correctly identify the associated rule. At this stage, before data is emitted into the next bolt (the Aggregation bolt), it is also tagged with a composite group by key, built from the extracted value identified by the group by the condition specified in the rule and rule ID. This composite group by key allows to isolate the tagged data from being processed alongside with data associated to other rules that share the same group by key;
- **Aggregation Bolt:** is directly connected to the filtering bolt in a different way from how other bolts are connected to each other; these are connected in a none-grouping

way, the Aggregation Bolt, on the other hand, is connected to the Filtering Bolt in a fields-grouping way using the composite group by key, ensuring that multiple nodes of this bolt always receive data with the same key, useful when a topology is deployed into a cluster environment and configured to have multiple nodes (bolts and/or spouts) of the same type.
At the moment, this bolt only allows a static aggregation periodicity that must be defined at deployment time, forcing all data to be processed at the same rate, despite belonging to different rules. With the current topology implementation it would require a massive overhaul of this component in order to encompass a dynamic configuration of the aggregation period; however, this has been identified as future work and will surely be implemented when the opportunity arises.
By contrast, the aggregation functions to be applied to the arriving data are dynamic and based on the supplied rules. This can happen since data was previously tagged with the corresponding rule ID, therefore making the main functionality of this bolt actually dynamic. Currently it supports six aggregation functions:

1. List (LIST);
2. Count (COUNT);
3. Sum (SUM);
4. Average (AVG);
5. Maximum (MAX);
6. Minimum (MIN).

Figure 6.3 exposes how this bolt works internally.

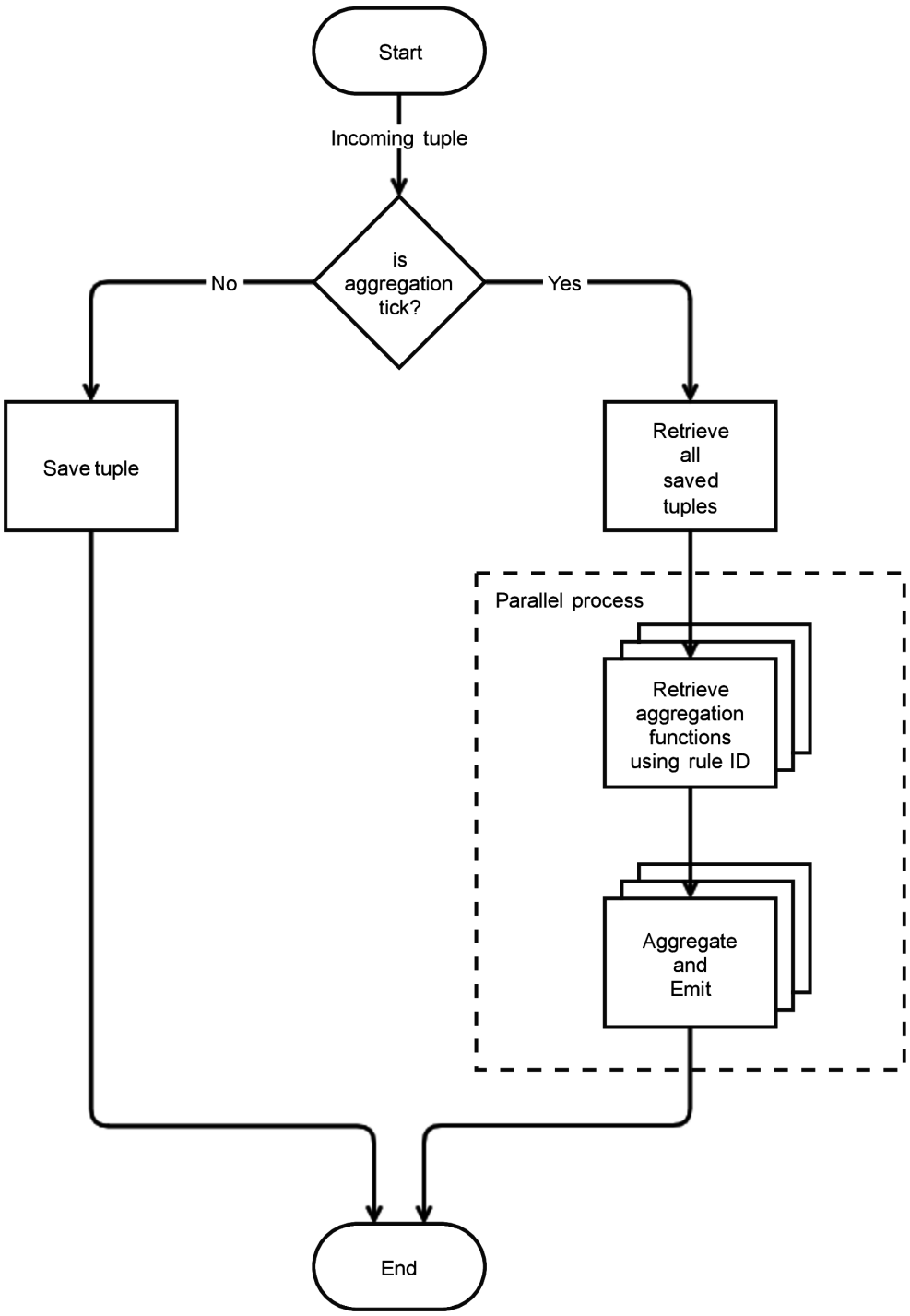


Figure 6.3: Aggregation process flowchart

It makes use of an underlying topology mechanism that sends a different kind of tuple called tick. This tick tuple does not contain any actual data, it just indicates that the aggregation period has passed. And so, given that bolts are stateless, this feature

sets the aggregation pace. An important note about tuples storage and their parallel processing: tuples are temporarily stored before their are aggregated, and are isolated by using the already mentioned composite group by key. That said, retrieving all saved tuples in order to process them is also an isolated procedure, by using the same group by approach. Therefore, several aggregation processes occur at the same time for each distinct group. Finally, by the end of each process, the aggregated data is emitted towards this topology's router;

- **Router Bolt:** provides, as its name already states, a routing functionality to this topology, allowing the data that has already been filtered and aggregated to be subjected to the remaining optional processes before being published. There are four different paths to follow that are directed by this router bolt, and that are decided upon the inferred actions to take (publish, enrich or threshold) from the rule associated to the incoming data tuples. These paths are:
  1. Router → Publisher;
  2. Router → Enricher → Publisher;
  3. Router → Threshold → Publisher;
  4. Router → Threshold → Enricher → Publisher.

  To accomplish this, data leaving the router bolt is tagged with two new fields:
  - Actions: the actions sequence to be performed after data leaves the router;
  - Next Action: the next action to be apply to data. It is consumed from the Actions sequence, allowing data to be correctly forwarded after being processed.

  Up to now, the already presented bolts were using a default stream transport to carry data from one bolt to another, and this practice was totally fine since each bolt was consuming and emitting tuples in a one to one ratio. This scenario is slightly different, as the router bolt receives data from the default stream but has to emit tuples to different bolts, as seem in Figure 6.2, in to follow the actions to take. For this matter, three streams were created to completely isolate tuples that need to be: enriched (Enrich stream), applied a threshold (Threshold stream), published (Publish stream); making this bolt able to emit tuples to the right streams;
- **Thresholding Bolt:** at the moment, the implementation for this bolt is just placeholder due to the fact that there is not any combination of sensors, data sources, use cases and catalogue rules that stimulate this functionality. Therefore, this bolt only forwards tuples without applying any sort of functions over them. However, the topology already has this module completely integrated and ready to be further developed in the future, when concrete requirements arrive;
- **Enrichment Bolt:** complements the arriving data with more information. As of the date, there is only one type of enrichment, metadata enrichment. By making use of a metadata field present in the rule that defined the aggregation, it is possible to enhance the previously processed data with static metadata specified in the rule. Tuples coming out of this bolt are directly sent to the Publisher;
- **Publisher Bolt:** is the exit towards the external world around the topology. In this case, it is represented by a Kafka messaging bus, where all results that can be metrics, events and alarms, are published. This bolt is responsible for receiving all processed data and parse it to the output model RADM (Section 5.3.1). Despite this implementation only covering the Kafka producer scenario, the Publisher bolt was built with a plugin like system, where various types of publishers can be used to better interface with other systems.

Having in mind that an Apache Storm topology like this one can be deployed in a cluster environment, precautions regarding rules synchronization over every bolt need to be taken. For this matter, Figure 6.4 depicts the interaction between every bolt in the topology with the CEP Manager component, responsible for deploying rules over the topology.

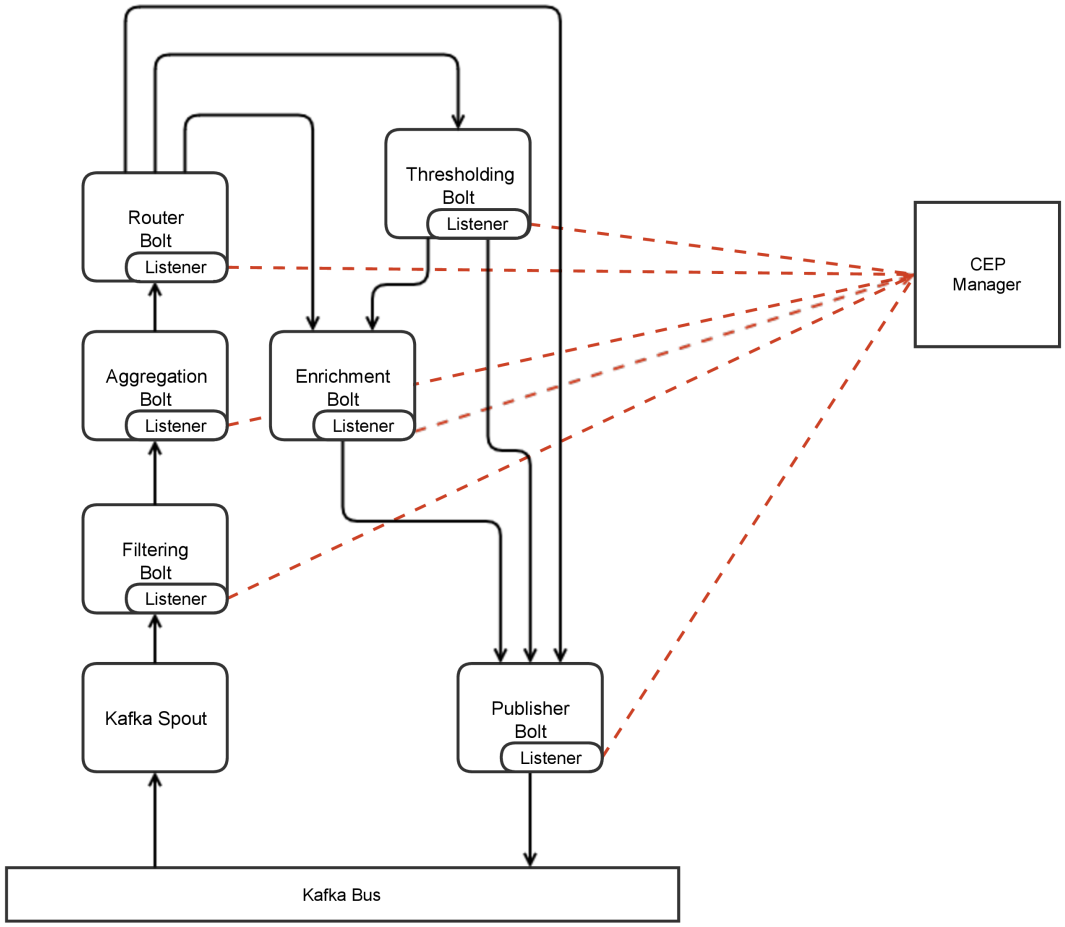


Figure 6.4: Generic Topology and CEP Manager Interaction

Despite Section 6.2.1.2 covered how aggregation rules are managed, it is of significant importance to expose how a bolt interfaces with the CEP Manager before delving into further implementation details. Figure 6.5 explores just that. Each bolt contains a local configuration manager that registers a listener callback in a Zookeeper Service path, provided at deployment time, in order to be notified for content changes on that certain path. Therefore, when the content for that node changes, the bolt is notified and provided it the new data, that hopefully, represents a set of rules. Here, bolts are to never write into Zookeeper, their role is to only read configurations provided by CEP Manager.

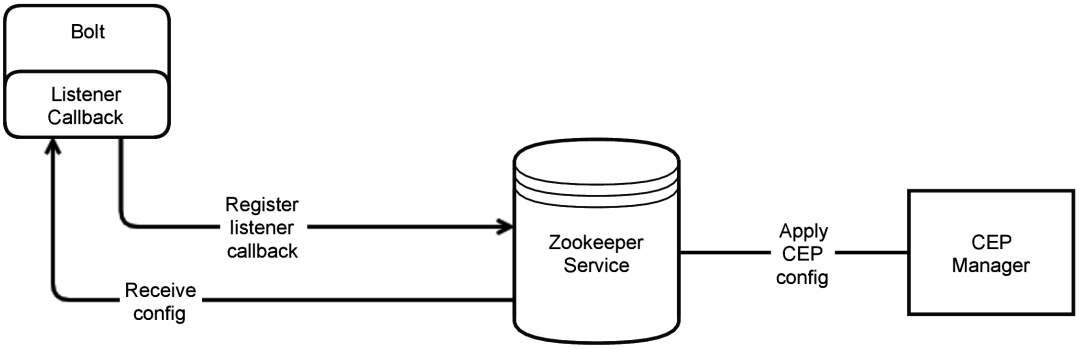


Figure 6.5: A bolt and CEP Manager Interaction

6.2.1.2 Manager

CEP Manager is the component in the Complex Event Processing Framework responsible for managing the topologies within the Framework, as well as their configuration that are dynamically introduced by the Aggregation Configuration Manager (ACM).

Despite the fact of this component being oblivious of the aggregation rules life-cycle management, it exposes the functionalities that enable this management to happen. It makes sure that the aggregation rules provided through the Aggregation Configuration Manager are instantiated on the right topologies, if they are applicable. Although not yet implemented, it is ready to be extended to support the control over the topologies that are running within the Engine (start, stop, pause) and even the deployment of new ones.

The gap between this component and the Aggregation Configuration Manager is closed using the coordination service Zookeeper. This ensures that any new data coming from ACM is properly available in a distributed environment as the CEP Engine.

Since ACM has not been implemented yet, there is no actual usage of the interface provided by CEP Manager, and so, in order to provide a way to test the aggregation rule deployment, this component was outfitted with a command line interface that allows a user to provide an JSON file containing a rule, so that it can be applied to the already running topology. An example for this rule is provided in Annex D.1.1, it refers to the aggregation rule used for the Self-Protection use case.

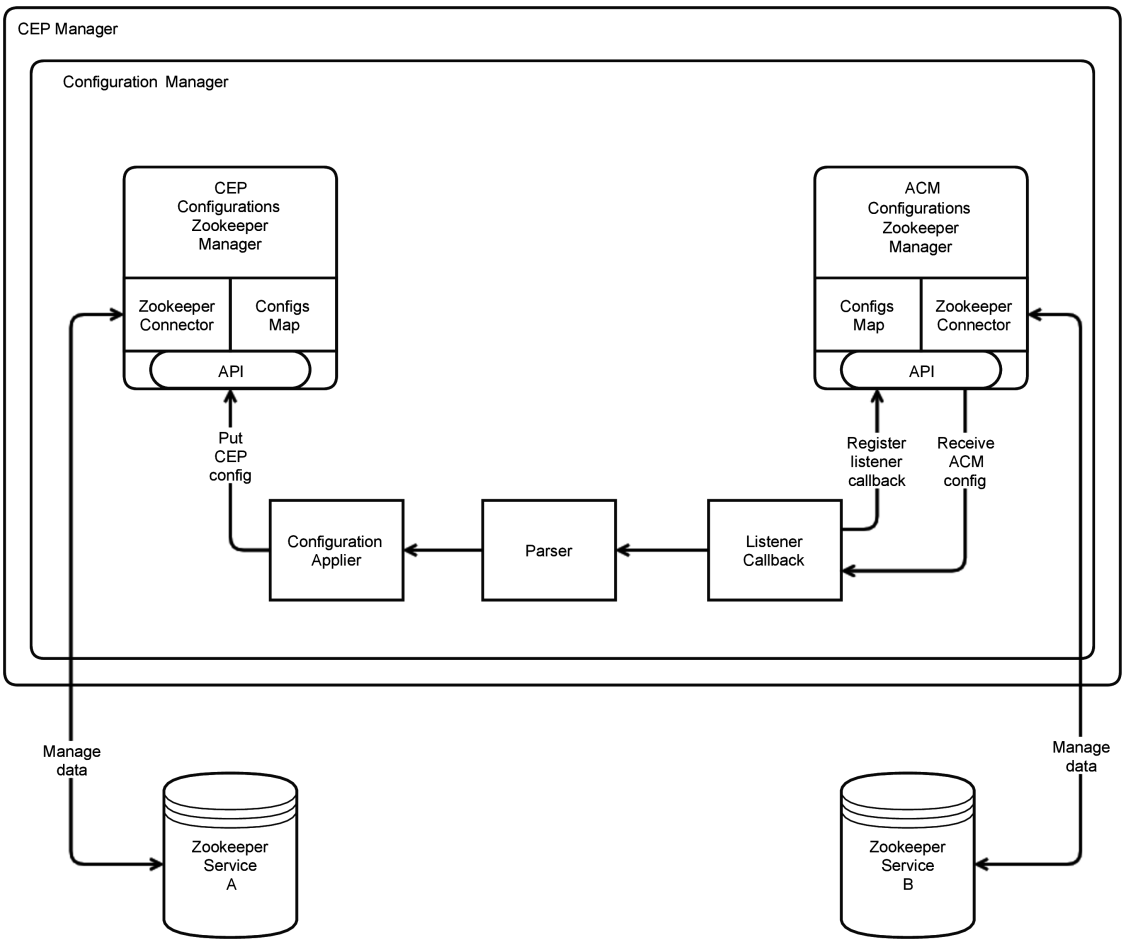


Figure 6.6: CEP Manager internal components

Figure 6.6 portraits CEP Manager internal components and how they communicate in order to deploy new ACM incoming configurations/aggregation rules while providing a mechanism to synchronize rules across different topologies. Firstly, CEP Manager is composed of a Configuration Manager that holds all logic in between the ACM interface up to the communication with a topology. By subdividing CEP Manager in this way simplifies the task of implementing a Topology Manager in the future, since the software is not tangled up in a monolithic way. The Configuration Manager is composed of:

- **ACM Configurations Zookeeper Manager:** connects to the a Zookeeper service in order to receive JSON strings over Zookeeper and to pass them into the internal Configuration Manager logic, while keeping a local cache of those strings;
  - **Zookeeper Connector:** effectively connects this manager to a Zookeeper service (Zookeeper Service A) to a node that ACM should connect as well, in order to exchange data. In this case, the */cep/acm_interface/config* path is used.
  - **Configs Map:** is the local cache of the data available at the Zookeeper path, which is kept in sync by using an automated mechanism of listener callback functions;

  - **API:** exposes a way to make use of read, write and listen to data from Zookeeper path nodes.
- **CEP Configurations Zookeeper Manager:** it is the same software module as the ACM Configurations Zookeeper Manager, but explores its functionalities in a distinct manner by connecting to different Zookeeper node paths and handling different data than JSON strings. Its logic in centred in receiving write requests to push Scala objects, representing configurations/rules into topologies;
  - **Zookeeper Connector:** connects this manager to a Zookeeper service, in this case represented by Zookeeper Service B, that in reality can be the same as Zookeeper Service A, but connected to a different path. As an example, this connector is used to connect to each topology root */cep/topology1*, so that latter it can be used to pass data into it;
  - **Configs Map:** represents a local cache of data that was written to the topologies Zookeeper path, for example */cep/topology1/config*;
  - **API:** the same as the last manager.
- **Listener Callback:** Configuration Manager makes use of this callback to receive data and notifications when the data in Zookeeper Service B (representing ACM interface, a Zookeeper node path, for instance */cep/acm_interface*), by registering it on the ACM Configurations Zookeeper Manager API;
- **Parser:** after the JSON strings, received via callback, are sent to this component, they are validated and parsed into an internal object structure that must be recognized by the topologies. The data is then forwarded into the applier;
- **Configuration Applier:** taking the objects given by the Parser, it is now a matter of serializing them and deciding to which topologies they must be given to. At this moment, since there is no Topology Manager component, these configurations are applied to the topologies indicated by their names, that are supplied during the CEP Manager deployment, via command line. This is realized by using the CEP Configurations Zookeeper Manager API.

All things considered, CEP Manager is implemented to run in a daemon-like mode, only processing incoming ACM configurations when notified to do so, parsing and applying them to the existing topologies, if applicable.

6.3 Summary

The implementation chapter exposed the developed software by this thesis work. It started in exploring Raw Data Loader and its mechanisms that allow sensed data to be categorized as counters and events, and then store it for historical data persistence purposes. At the same level, the Complex Event Processing Framework was introduced in detail, providing insight upon its Engine and Manager inner-working processes, showing a dynamic and rule-based real-time aggregation system.

Chapter 7

Application and Results

Having presented and explored all previous chapters that covered details from the motivation and objectives up to the implementation of the proof-of-concept, this chapter can now focus down on what applications have emerged from this research and their results in terms of performance and applicability to Altice Labs, SELFNET and their 5G network test-bed.

7.1 Applications

Here, applications refer to all software, models and tools that were developed throughout the research and implementation of this thesis. Given that, this section will present the achievements accomplished by this work and their impact on other entities.

7.1.1 Raw Data Loader

Raw Data Loader (RDL) was designed to be the data loader for SELFNET, that took care of parsing data and enforcing a unified model on the project. As of the date, July 2017, RDL is deployed in the test-bed for several months now, and has also been deployed in a SELFNET partner test-bed, University of West Scotland.

7.1.2 Raw and Aggregation Data Model

The Raw and Aggregation Data Model (RADM) was specifically designed and built for SELFNET's needs. It was discussed and suffered many changes since when it was first introduced in April 2017, and it has already reached a mature stage where SELFNET sensors and data-sources comply with the model.

7.1.3 Complex Event Processing Framework

The Complex Event Processing Framework (CEPF) has been in place, in the test-bed and in its last stable version, since June 2017, having been verified its functionalities in a SELFNET integration meeting in Aveiro, in July 2017, where it was integrated with the existing components available at the test-bed (i.e. sensors and software that processed CEPF output), and it has been also deployed over the University of West Scotland test-bed successfully. The results presented here are also useful to Altice Labs ALTAIA team to understand if this framework is a plausible strategy for dealing with events in a real-time fashion.

7.1.4 ASF Configurations over Zookeeper extension

This tool kit is a bi-product that emerged when developing CEP Manager, which is referred to as Configuration Manager in Section 6.2.1.2. It was born by extending and fusing two separate components supplied by Altice Labs: the Asynchronous Framework static map cache and their internal version of a Zookeeper watchdog. As briefly mentioned before, this Configuration Manager synchronizes a local cache map through Zookeeper while giving a way to register watchers and callbacks to further extract its functionality, and thus making this software a tool kit. Given this, the main objective of this tool, apart from serving its purpose in the software developed here, is to be integrated into Altice Labs tool set, after being properly tested and evaluated.

7.2 Scenario

Before presenting the tests and results obtained (Section 7.3), it is paramount to provide context about the testing environment and what use case was put in place to exercise the functional capabilities of the software. In order to achieve this goal, this scenario section will first introduce the underlying SELFNET test-bed (Section 7.2.1) and then the chosen use case (Section 7.2.2).

7.2.1 Test-bed

The testing scenario is based on the SELFNET test-bed, which itself is provided by Altice Labs and Instituto de Telecomunicações of Aveiro infrastructures. These are connected by an optical fiber link, allowing the Altice Labs data center to communicate with the flexible network environment, comprised of virtual and physical network elements such as: sensors, actuators, traffic interceptors; as well as, a Long Term Evolution (LTE) radio network connected to a virtualized Evolved Packeted Network (EPC) that has Internet connectivity, as seen in Figure 7.1.

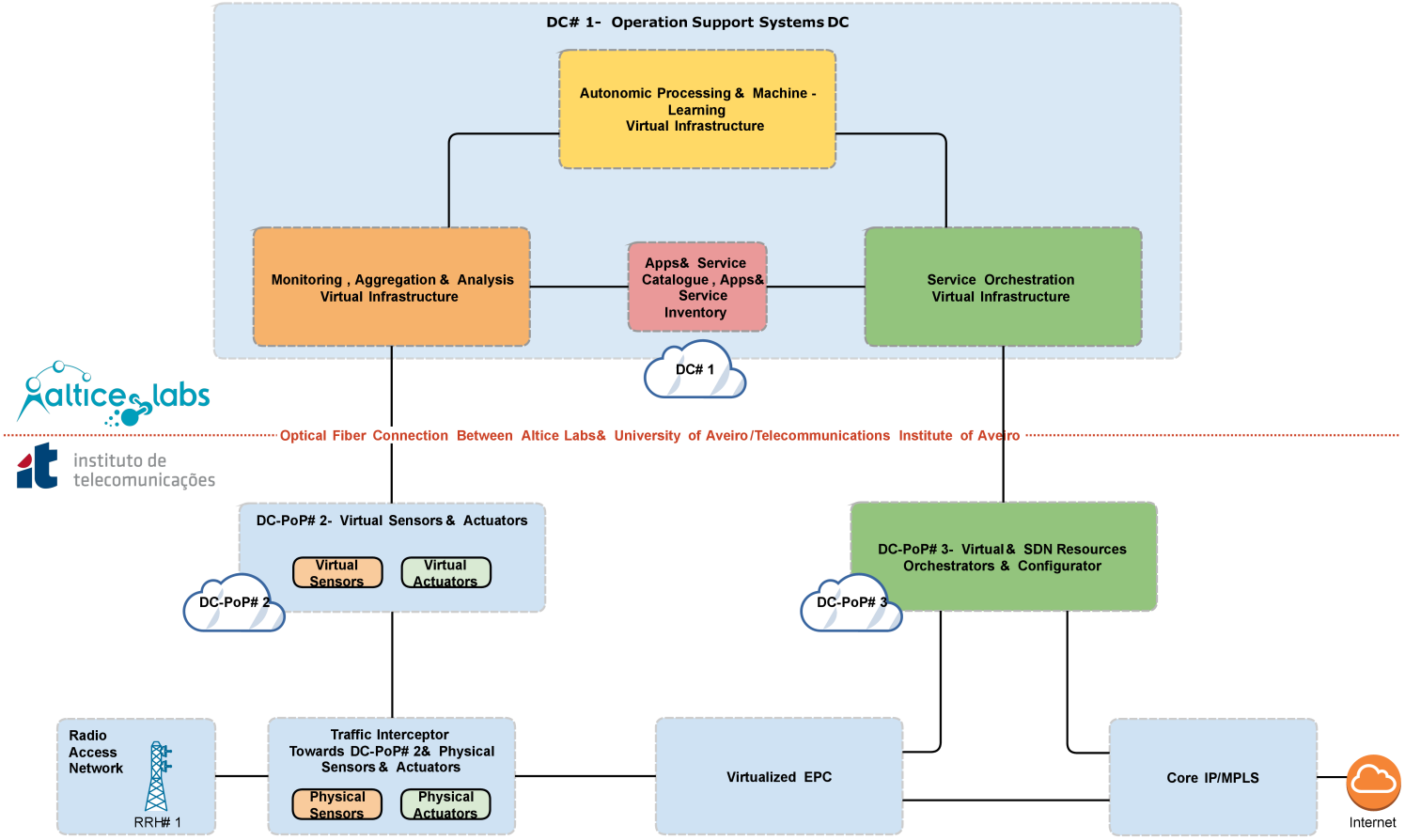


Figure 7.1: High Level Infrastructure Diagram

7.2.2 Use case

The chosen use case for this scenario's testing purposes was the SELFNET Self-Protection use case. The reason behind this choice lies in the fact that, out of the SELFNET use cases (Self-Protection, Self-Optimization and Self-Healing), it is the one that at the moment is more refined and mature with the current test-bed.

The main objectives of the use case, that was covered in Section 4.3.3, are to detect, identify and isolate a bot-net that has managed to infiltrate the managed network domain. For this matter there are two major steps in doing so:

1. Identify a bot-net communication pattern in the network by analysing network flows and then:
   (a) deploy a traffic mirroring virtual function that mirrors the suspicious traffic to a deep packet inspection function;
   (b) deploy a deep packet inspection virtual function to identify bots (also known as zombies) in the bot-net.
2. Collect data reported by the DPI function so that it can be aggregated in order to provide evidence of the entities that comprise the bot-net, to then be able to actuate in the network by:
   (a) deploying a honey-net, used to isolate all zombies and the server controlling them;
   (b) configuring the existing traffic mirroring function to divert the same traffic over to the deployed honey-net.

Illustrations of both step one and step two can be found in Section 4.3, referenced by Figure 4.7 and Figure 4.8, respectively.

7.3 Results

The Results section presents the performed load tests for RDL and CEP Engine Generic Topology, which are inserted in the second loop of the use case referred in Section 7.2.2. However, before presenting said results, a brief overview over the metrics and information extracted from the use case first loop.

This first loop aims to identify possible zombies that are communicating with a command & control server. To achieve this identification, the Batch Aggregation Framework periodically takes all network flows reported by FMA (see Section 5.2.1.1) and calculates two metrics for each unique combination of source IP, destination IP and destination port: average packet count, as seen in Figure 7.2, and communication frequency, depicted in Figure 7.3. Afterwards, the Analyser Framework would have previously configured a threshold rule in the Batch Aggregation Threshold Engine in order to trigger alarms that reported the potential zombies identification. Figure 7.4 depicts an example of the metrics values, together with the IPs and ports that identify zombies, that would trigger the alarms.

In this particular case, the data used to populate the presented charts originate from real sensing performed in the test-bed and with the presence of a real bot-net. Due to the vast amount of data fitted in those charts, Figure 7.2 and Figure 7.3 do not provide a comprehensive legend that lists all unique destination IP, destination port and source IP. On the other hand, Figure 7.4 can provide said list since it is relatively short. A brief note must be made in order to clarify the represented curves in Figures 7.2, 7.3 and 7.4. Each curve is a metric value over time, referent to a unique combination of source IP, destination IP and destination port; for

instance, Figure 7.2 shows the average packet count metric where each curve is a value tied to those three dimensions (source IP, destination IP and destination port).

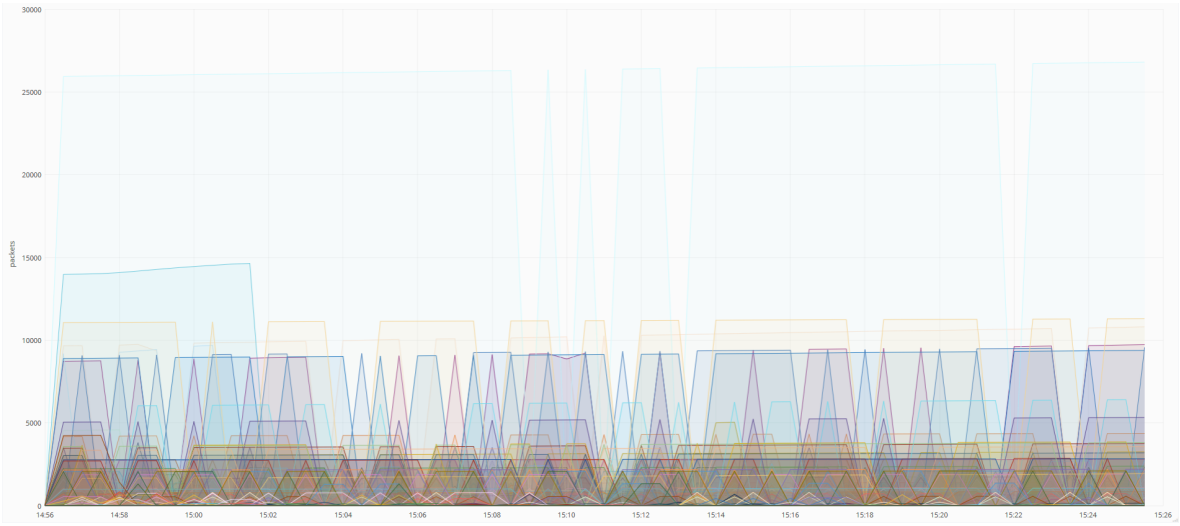


Figure 7.2: Batch aggregation - Average packet count metric for each unique combination of source IP, destination IP and destination port present in the network, with a sampling rate of 120 seconds on a 30 minute time window

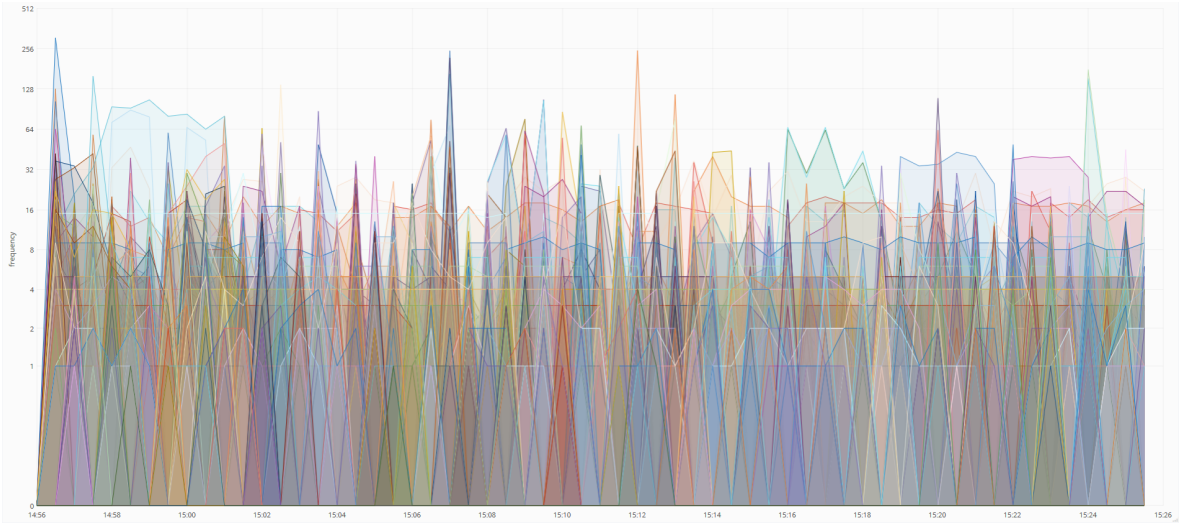


Figure 7.3: Batch aggregation - Communication frequency metric for each unique combination of source IP, destination IP and destination port present in the network, with a sampling rate of 120 seconds on a 30 minute time window

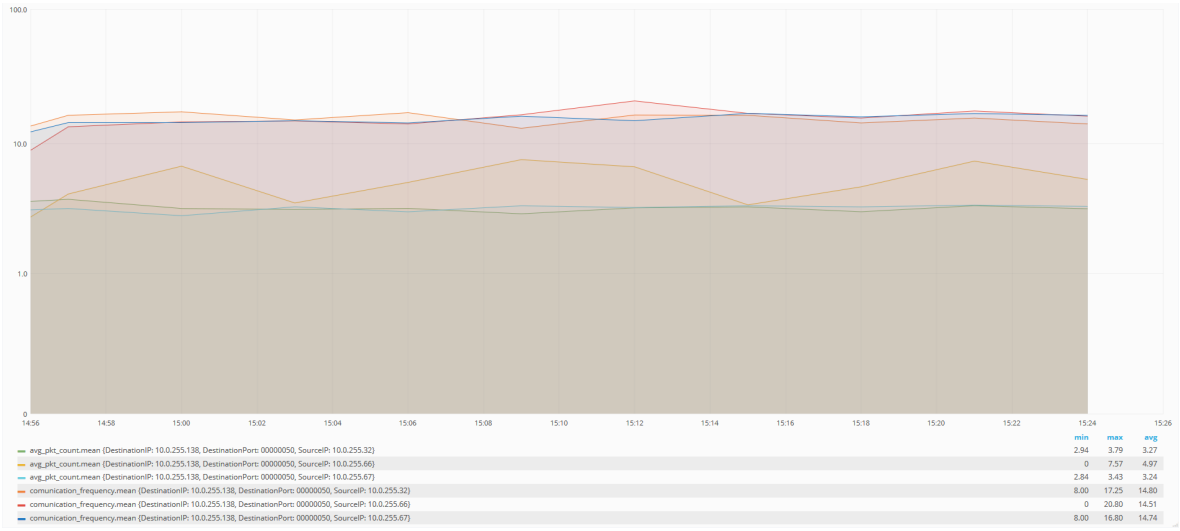


Figure 7.4: Average packet count and communication frequency metrics pairs for each unique combination of source IP, destination IP and destination port that have been identified as suspicious zombies

With this first loop zombie identification, SNORT can be deployed to inspect all packets exchanged between the zombies and their command & control server, producing new data to be sensed and processed of the Aggregation Layer. The next sections 7.3.1 and 7.3.2 will cover the second loop of the use case, for RDL and CEPF respectively.

7.3.1 Raw Data Loader

RDL is deployed in VM with 8GB RAM and 2 VCPUs, that is also used to deploy a Kafka broker, which serves all SELFNET test-bed components. RDL itself is configured to have 5 collector threads with 20 seconds of polling time and a batch consume size of 10 million records maximum, and 5 persister threads with 0 seconds of polling time (setting polling time to 0 implies no time internal between each polling action, ensuring the highest possible CPU usage), as seen in the Annex C.1.2.

To shed some light on RDL performance, two load test were performed, using a SNORT emulator that aimed to report:

- **Load 1:** 4826 events per second with 254 zombies per 19 botnets, during 280 seconds;
- **Load 2:** 64262 events per second with 254 zombies per 253 botnets, during 180 seconds.

Programmatically, these events are published to a Kafka message bus in best effort mode, thus making the actual data generation process to have different event publish rates. Another import remark to make is that the two tests have different duration times, the reason being the amount of data that they generate. Since RDL keeps records in-memory before storing them in to the database, there is a maximum amount of records that RDL can hold before crashing, therefore limiting the time duration of the data generation process. Having said that, for both tests, the data generator was ran until it first reached, or crossed, the 500000 events in-memory mark. Then, RDL was left running until it managed to process all generated events.

The event generator was located in a different VM than the RDL one for two main reasons: RDL VM had already several other software running there, limiting resources usage; the generator is CPU intensive and running it in the same VM as RDL would negatively impact the measurements.

Measurements were made in two parallel parts, manually and by using both RDL and the generator debug modes that output their statistics each second. That is to say, since measurements were taken from two different VMs, in a manual fashion and both software were Java Virtual Machines running on top of a VM, these measurements are not completely accurate yet they provide a good notion about RDL performance.

7.3.1.1 Test Load 1

Figure 7.5 and Table 7.1 represent the first test load measured values from RDL and the events generator. The test lasted 280 seconds, until the RAM safe limit was reached, having recorded 528352 events in-memory, where 708660 events were generated in 90 seconds. Visually, and with the aid of Table 7.1, it is possible to see that RDL can almost consume records at the same rate that they are published (87.457% ratio, 12,543% slower than the generator rate); however most of its performance is lost when persisting them into the database, presenting a storage ratio of 34.364% relative to the events consumed (65.636% slower than the consume rate).

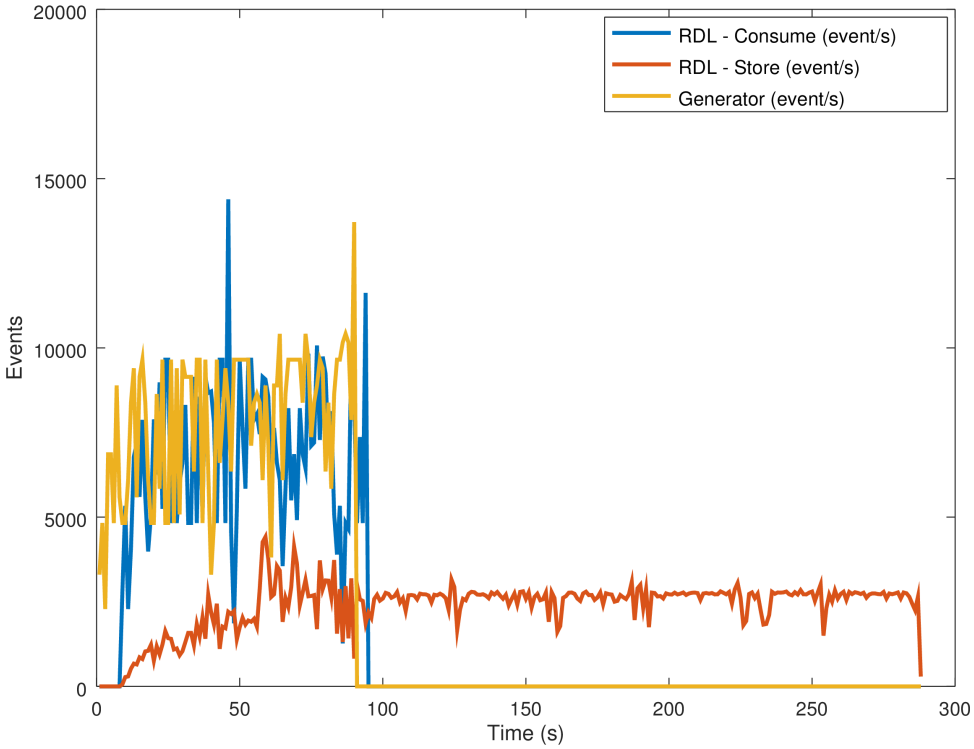


Figure 7.5: RDL Performance 1

Total Generated (event)	**Average Generated (event/s)**	**Total Consumed (event)**	**Average Consumed (event/s)**	**Consume Ratio**	**Total Stored (event)**	**Average Stored (event/s)**	**Storage Ratio**	**Maximum In-memory (event)**
708660	7874	592227	6886.4	87.457%	681536	2366.4	34.364%	528352

Table 7.1: RDL Performance Load 1, sample time 280 seconds

7.3.1.2 Test Load 2

Taking a look into Figure 7.6 and into Table 7.2 values, a big difference can be registered between this test and the first one: the sample time is shorter, 180 seconds, and the generator produced a similar amount of events, compared to the first test, but within a smaller time-frame, 40 seconds, until the in-memory limit was reached. The results to be taken from this experiment are analogous to the first one: RDL is able to consume records from the message bus with a 94.666% speed ratio (only 5.334 slower than the generator rate), and there is a huge performance drop when persisting data which is 75.297% slower than the consume rate.

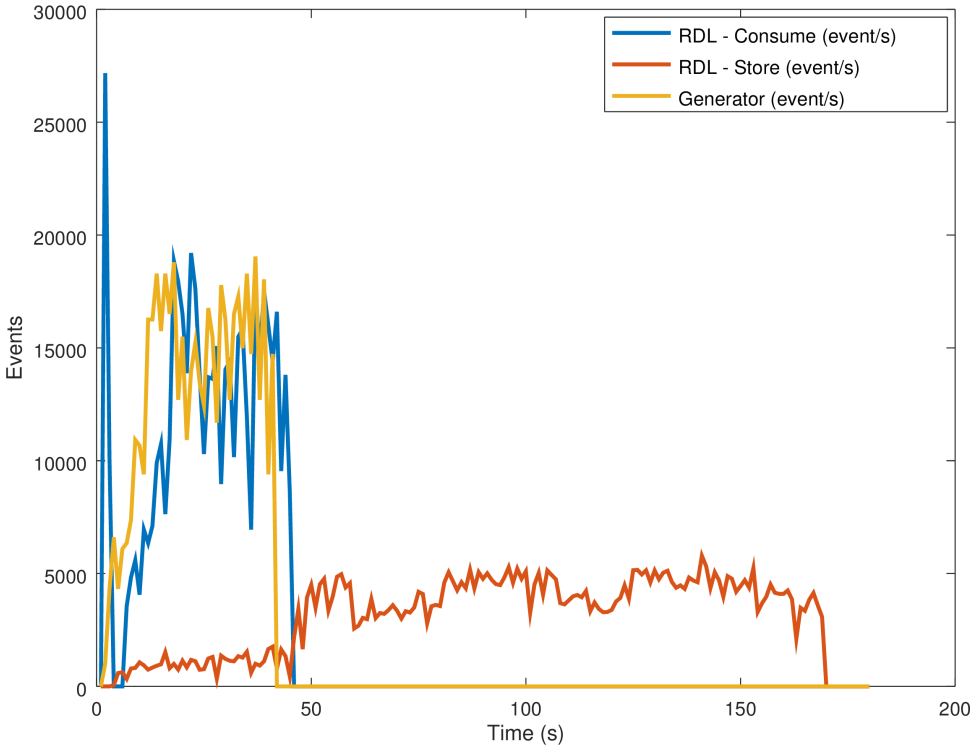


Figure 7.6: RDL Performance 2

Total Generated (event)	Average Generated (event/s)	Total Consumed (event)	Average Consumed (event/s)	Consume Ratio	Total Stored (event)	Average Stored (event/s)	Storage Ratio	Maximum In-memory (event)
527812	13195	512149	12491	94.666%	555442	3085.8	24.703%	531678

Table 7.2: RDL Performance Load 2, sample time 180 seconds

7.3.1.3 Conclusion

Taking these results into account, it is clear to see that RDL can be improved regarding its storing speed. From an implementation point of view, this slowed storage process can be caused by: parsing, database persistence or a combination of both. Finding the actual cause brings a new development required for this software, more accurate and complete measuring functionalities that allow to register how much time parsing and persisting takes, something that does not exist at the moment, enabling a way to analyse their performance and understand where the bottleneck is, which can only be in the parser, the persister or both. If the bottleneck is the parser, a possible solution would be reimplement it in a way that allows a parallel and asynchronous parsing process, instead of the actual static utility. On the other hand, if the bottleneck is the persister, the most likely solution to be applied is to discard the current persisting process, which involves a database mapper that takes objects and stores them in the database, to accommodate a new one where prepared statements are used to directly send stored queries to the database. Both proposed solutions will require investigation on their impact on the RDL performance and code maintainability. Moreover, and not invalidating the conclusion already drawn, RDL performance fits, for now, the dimension of the test-bed which does not have sensors configured to output data in rates that expose the revealed bottleneck.

7.3.2 Complex Event Processing Framework

Hereby it is presented the sole load test performed over the Complex Event Processing Framework. The focus here is the CEP Engine's Generic Topology, configured with an aggregation rule that lists all source IPs belonging to the same destination IP reported from SNORT (see Section 5.2.1.2). CEP Engine is deployed in a VM with very limited resources: 2GB RAM and 2 VCPUS; since Apache Storm (the framework used by CEP Engine) components take 1GB of the available RAM, the topology is left with little more than 1GB of RAM to use, which eliminates the possibility of having a thorough stress test on the topology.

Table 7.3 depicts the RAM occupied by the topology, alongside with the numbers of workers, executors, tasks and replication count of processing nodes. It is important to note that the amount workers is relative of configured RAM value for each one of them. In this case, the configuration used assigns 1024MB for each worker; however, the system does not possess these resources making it liberate only 832MB, and therefore limiting the workers number. Also in the table there are the executors and tasks which are configured in a simple way, one executor and tasks per processing node in the topology (any spout or bolt).

Workers	Executors	Tasks	Replication count	Assigned Mem (MB)
1	8	8	1	832

Table 7.3: Topology summary

An indispensable remark to make is that the data shown and presented here, in this section, has been retrieved using the Apache Storm UI, a component of Apache Storm that gathers statistical data about its system (e.g. Storm Supervisor, etc) and deployed topologies. The graph depicted in Figure 7.7 is the graphical view of the Generic Topology, where it is depicted the processing nodes, referred in Section 6.2.1.1.1 and presented in Figure 6.2, which are connected as follows: EventSpout-1 consumes records from a Kafka message bus, parses them to the Storm internal model (tuples) to then emit them towards the Event-FilteringBolt-1; there, tuples are filtered using a white-list and, if they pass the filter, they are forwarded to the EventAggregationBolt-1, where data is aggregated and then sent to the EventRoutingBolt-1 that decides the next operations to be performed over the tuples (enriched on the EventEnrichmentBolt-1, applied to a threshold on the EventTresholdingBolt-1, or published by the EventPublisherBolt-1); and some runtime statistics obtained via Storm UI, data stream names and their load (depicted at each arrow), and the average processing latency at each node. These statistical values are also available in Tables 7.4 and 7.5.

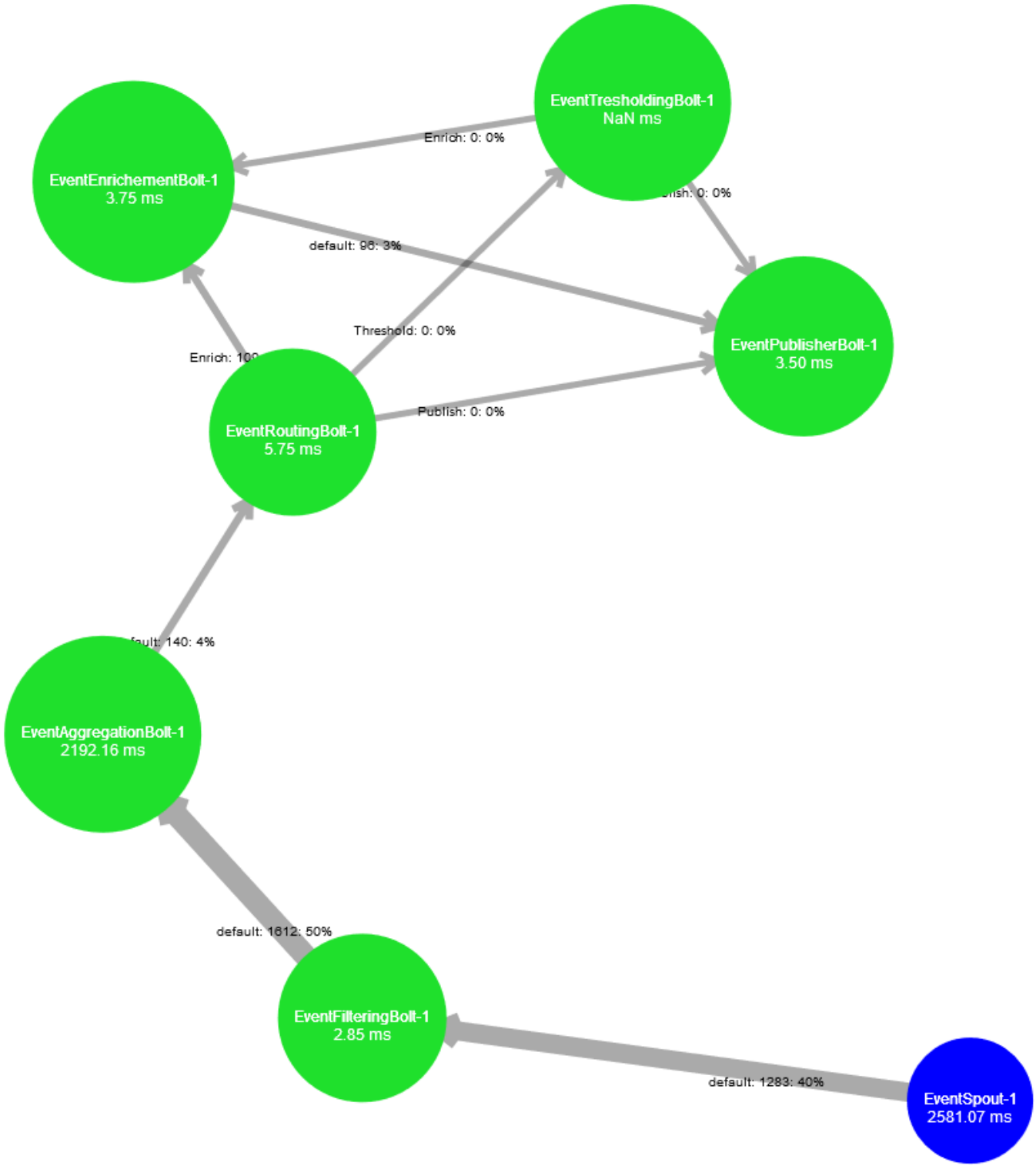


Figure 7.7: Generic Topology StormUI Load 1

7.3.2.1 Test Load 1

This test, as mentioned before, is a simple one. Its objective is to demonstrate a working example driven by the Self-Protection use case and what performance impact it has. For this matter, a SNORT emulator has been deployed and configured to report three zombie detection events (i.e. three different zombies) each second and equally spaced in time within the second, for 10 minutes. The only rule in place is the one referenced in Annex D.1.1, that impacts the topology in the following way:

- **FilteringBolt:** its white-list is populated with the filter *resourceType == snort-dpi-alert*, allowing only events reported by this type of resource to progress into the topology;
- **AggregationBolt:** it is supplied with the aggregation function *LIST(srcIP, classification)* to produce a list of sets that contain the source IP and classification of a zombie;
- **RoutingBolt:** it is configured to forward the events tagged with this rule in a way that they are enriched first and then published. This is achieved by looking into the rule and find if there is any metadata to enrich events with, which is the case;
- **Enrichment:** it is provided with the metadata to enrich events passing through this bolt;
- **Publisher:** it does the final arrangements to events before they are published: translate dimensions names (e.g. *dstIP* to *ccServerIP*), name the aggregation values (e.g *listSrcIP*), work out if the event is an alarm or just a normal event and provide the severity that is needed.

Having now the spout and bolts loaded with this rule, they processed these events for 10 minutes starting from a clean state. Table 7.4 shows data regarding the spout that consumes records from Kafka. In total it consumed and emitted 3157 events into the topology, and shows that the complete latency, the average time from when an event is consumed until it is acknowledged and outputted from the topology, is 2257,354 milliseconds, which is roughly half of the aggregation period, 5 seconds, making it the expected value since the generated data volume is constant throughout time. Furthermore, the emitted tuples match up with the acknowledged ones, confirming that all incoming data fits the rule in place.

ID	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked
EventSpout-1	1	1	1590	1590	2257,354	1586

Table 7.4: Topology Spouts 10 minutes stats

Most of the processing latency arises from the Aggregation Bolt, as can be seen in Table 7.5, which has an average of 2,193 milliseconds. This is caused by, the already mentioned, constant way that the generated events are produced and consumed. Moreover, with a glance over the process latency values, it is possible to state that aggregation operations are the ones that impact the most on the latency, something that is expected since the rule defines a five second aggregation period, and that other bolts have a significantly low impact on this metric.

Not only this metric is relevant from a testing point-of-view, but also, the amount of tuples emitted at each bolt. Table 7.5 reveals that all generated events are allowed to flow into the topology, by the filter, since the tuples emmited by *EventFilteringBolt-1* match the acknowledged ones by *EventAggregationBolt-1*; then the same bolt only emitted 120 tuples,

confirming that data is aggregated to then be routed, enriched with metadata and published.

A note should be made about the accuracy of the data presented here. Storm UI provides statistic values about what is happening in the topology; however, while all topology components can be clustered, which is not the case, the same components (i.e. spouts and bolts) are running in separate threads. This detail affects the measurements made by Storm UI, as it is a hard task to ensure that all measurements are made at the same exact time and that results will also exactly match between different threads. This matter is clearly reflected on Tables 7.4 and 7.5: one example would be the number of emitted tuples by EventAggregationBolt-1 and the number of acknowledged tuples by the EventRoutingBolt-1, they do not match but are fairly close.

ID	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked
EventAggregationBolt-1	1	1	120	120	0,006	2,193	1669	2159,346	1684
EventEnrichementBolt-1	1	1	129	129	0,001	4,800	100	3,167	120
EventFilteringBolt-1	1	1	1677	1677	0,006	2,259	1570	2,136	1543
EventPublisherBolt-1	1	1	0	0	0,001	2,571	128	2,571	140
EventRoutingBolt-1	1	1	151	151	0,001	6,667	120	6,429	140
EventTresholdingBolt-1	1	1	0	0	0,000	0,000	0	0,000	0

Table 7.5: Topology Bolts 10 minutes stats

7.3.2.2 Conclusion

Above all, the results here presented showed that the Generic Topology correctly works in this scenario, despite StormUI presenting slightly inaccurate values, and that in order to be able to perform a more serious series of experiments on this platform, two key aspects are required: better hardware, or VM resources, and a use case that uses sensors which are capable of generating the load needed for such test.

Despite this, a valid point can be made when the matter of introducing a new use case to the existing topology, which is: how can statistic data be collected from a topology that serves different use cases or rules concurrently? Well, StormUI will not be able to provide this, since it does not have notion about the rule isolation system running inside of the topology. A possible solution might be the implementation of a data structure or an agent that provides this from inside of the topology. While this would solve this problem, it would also further make the topology more complex and harder to maintain in the long run.

A simple solution, that could be used as a first approach, is to deploy test topologies which are configured to solely take data required from data sources defined by a certain use case, much alike the case presented here, and still use StormUI to take measures. As a starting point this definitively works, but the testing and deployment phases would slow down due to this personalised way of testing.

A conclusion that one can take from these series of thoughts is that there is no optimal way of testing CEP Framework, since the methodology is tied to the tests objective, to the deployment type and the underlying infrastructure. That said, it is also clear that CEPF is open to further improvements as the need for more complex scenarios arrive, like the Self-Protection use case.

7.4 Summary

This chapter started by establishing the four contributions made to Altice Labs and to the SELFNET project. Afterwards, and before displaying results, the scenario section was explored, where the SELFNET test-bed was introduced, followed by a clarification on which use case was to be exercised in the tests. After having provided context on the testing environment, the results section covered the tests, to which RDL and CEPF were subjected, while providing a critic vision about the extracted results. In a few words, RDL showed promising results regarding data consumption rates while revealing a storage throughput flaw, that has been identified and will be worked on. On the other hand, CEPF showed that infrastructure limitations are currently affecting load tests results. Despite this, CEPF showed that the proposed functionalities are in place and it still is open to new requirements.

Chapter 8

Conclusion and Future Work

All content presented here had the objective of not only presenting the work developed by this thesis, but also its context and impact outside of the academia. This is not to diminish the role that academia has in society and in the scientific world, but to enforce that opportunities which bring common work platforms that are able of connecting the academia to the public and private companies, must be seized in order to provide society and the market with more complete and realistic solutions. Although it might be pretentious to claim that this thesis work has managed to take a tiny step towards this vision, I think that there is evidence in this sense and that more can be done in this aspect.

This evidence refers to the accomplishments depicted in Section 7.1 which include three software pieces (Raw Data Loader, Complex Event Processing Framework and the ASF Configurations over Zookeeper extension, also known as Configuration Manager) and one data model (Raw and Aggregation Data Model). These were contributed to Altice Labs, SELFNET and SELFNET partners.

Results wise, both RDL and CEPF showed that their functionalities are in place and already being applied to SELFNET test-bed. However, the performance tests indicate that improvements are needed in both components. RDL tests assessed that consuming data from a message bus is done at acceptable rates, but parsing and persisting that data are not. These results reveal RDL's weaknesses and aspects to improve. Regarding CEPF and the tests performed to it expose that performance could not be properly assessed, due to inappropriate hardware requirements and inaccurate measuring tools.

Solution wise, all software exposed here can and will be further improved to meet new requirements and to fix known issues. In a future work point-of-view, Raw Data Loader will be improved to upgrade its performance and will be further matured to provide a solid platform that can take on new SELFNET requirements when they arrive, for instance, there might be the need for it to perform some enrichment operations before storing data. Regarding CEP Framework, this software is the one that has space for improvement, for several reasons: at the moment, and although it provides a dynamically configurable topology, it was only designed to support the Self-Protection use case. When the time comes to support the remaining use cases, its performance will become a relevant topic to test and discuss; another reason would be the pending integration with the Aggregation Configuration Manager, which will be a new requirement to integrate with the CEP Framework and its Manager. And at last, but not of lower importance, is the future work to be done over the Configuration Manager, the software extension built on top Altice Labs Asynchronous Framework map cache and Zookeeper, that

has to be done in order for it to be approved into Altice Labs tool set.

More concretely, the planned future work will address:

- Raw Data Loader performance issues by:
  - extending the Parser and Persister modules to provide performance indicators;
  - analyse these performance indicators and identify the bottleneck of the system, which can be the Parser, the Persister or both;
  - reimplement the identified bottlenecks.
- Complex Event Processing Framework in two fields:
  - Test coverage:
    - current test coverage is not optimal and does not provide a complete insight about the framework's performance. To address this, better hardware will be requested so that more complete tests can be written and used to assess the system.
  - New requirements:
    - as the SELFNET project continues and with the Self-Protection scenario already implemented at the Aggregation Framework, the other two use cases will be addressed. Due to the complex nature of the Self-Optimization use case (i.e. calculate complex metrics in real-time, correlating data from network flows and physical infrastructure network counters), a new topology will be developed, extending the Generic Topology (presented in Section 6.2.1.1.1) in order to address the use case. Regarding the Self-Healing use case, discussions about its course are still in place; however, regardless of their outcome, the minimum expected changes encompass new aggregation rules to be supported;
    - a integration requirement will emerge when the Aggregation Configuration Manager (ACM) is developed and introduced into the test-bed. For this matter, the CEP Manager component will need to be adjusted in order to correctly interface with ACM. The interfacing means changes to the current CEP Manager rule parser.
  - Improvement of the Generic Topology existing features:
    - aggregation functions are still relatively simple and will be extended to support complex formulae;
    - aggregation period is still static and a dynamic one will be implemented;
    - thresholding capabilities have not yet been developed and will now be addressed for the Self-Optimization use case;
    - enrichment functionalities will be further extended to not only support metadata enrichment, but to use Cassandra and InfluxDB as data sources as well, since they hold raw and aggregated data, respectively.
- Configuration Manager acceptance as a valid tool to integrate the existing Altice Labs tool set by subjecting it to their test procedures.

While the list above presented the future work relative to my participation in SELFNET, a transition will be made to the European project SLICENET [64], where the machine learning area will be researched and introduced to an environment similar to SELFNET's. The main difference between SELFNET and SLICENET participation is that, in SELFNET the data aggregation tasks were prioritized in order to be able to contribute to the Altice Labs ALTAIA project and to SELFNET itself; however, in SLICENET a different focus will be taken, the main objective is to take on the challenge of developing a proof-of-concept framework for inferring, diagnosing and solving potential anomalies in a 5G environment that supports

SDN and NFV, by using machine-learning concepts.

In a few words, and to summarize this last chapter, there is room for improvement and, although this thesis comes to an end, its work only provides a starting point for future work. Future work that will further contribute to Altice Labs, SELFNET, SLICENET and, ultimately and hopefully, to the future of 5G networks.

Bibliography

[1] International Telecommunications Union *"ITU towards "IMT for 2020 and beyond" - IMT-2020 standards for 5G"*, http://www.itu.int/en/ITU-R/study-groups/rsg5/rwp5d/imt-2020/Pages/default.aspx, last visited July 2017

[2] Jo Best *"The race to 5G: Inside the fight for the future of mobile as we know it"*, https://www.techrepublic.com/article/does-the-world-really-need-5g, last visited July 2017

[3] NGMN 5G White Paper, https://www.ngmn.org/uploads/media/NGMN_5G_White_Paper_V1_0.pdf, last visited July 2017

[4] The 5G-PPP/H2020 SELFNET project, "Framework for Self-Organized Network Management in Virtualized and Software Defined Networks", http://selfnet-5g.eu, last visited July 2017

[5] T-NOVA website, http://www.t-nova.eu, last visited July 2017

[6] UNIFY website, http://www.fp7-unify.eu, last visited July 2017

[7] CROWD website, http://www.ict-crowd.eu, last visited July 2017

[8] 5G-NORMA website, https://5gnorma.5g-ppp.eu, last visited July 2017

[9] MYCOM OSI PrOptima™web page, http://www.mycom-osi.com/products/proptima/service-and-network-performance-management, last visited July 2017

[10] ZTE Network Performance Management Service, http://www.zte.com.cn/global/services/categories/unicare/products/413041, last visited July 2017

[11] Altice Labs Operation Support Systems, "Altaia End-To-End Assurance Solution", http://www.alticelabs.com/content/products/BR_Altaia_ALB_EN.pdf, last visited July 2017

[12] P. Demestichas, A.Georgakopoulos, D. Karvounas, K. Tsagkaris, V. Stavroulaki, J. Lu, C. Xiong, J. Yao *5G on the Horizon: Key Challenges for the Radio-Access Network*, IEEE Vehicular Technology Magazine, Volume: 8, Issue: 3, September 2013

[13] O. Awobuluyi, J. Nightingale, Q. Wang, J. Calero *Video Quality in 5G Networks: Context-Aware QoE Management in the SDN Control Plane*, 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, October 2015

[14] J. Andrews, S. Buzzi, W. Choi, S. Hanly, A. Lozano, A. Soong, J. Zhang *What Will 5G Be?*, IEEE Journal on Selected Areas in Communications, Volume: 32, Issue: 6, June 2014

[15] A. Farshad, P. Georgopoulosy, M. Broadbent, M. Mu, N. Race *Leveraging SDN to Provide an In-network QoE Measurement Framework*, 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), May 2015

[16] L. Li. Z Mao, J. Rexford *Toward Software-Defined Cellular Networks*, 2012 European Workshop on Software Defined Networking, October 2012

[17] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula *DevoFlow: Scaling Flow Management for High-Performance Networks*, Proceeding SIGCOMM '11 Proceedings of the ACM SIGCOMM 2011 conference, August 2011

[18] M. Liyanage, J. Okwuibe, I. Ahmed, M. Ylianttila, O. Pérez, M. Itzazelaia, E. Oca *Software Defined Monitoring (SDM) for 5G Mobile Backhaul Networks*, 2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), June 2017

[19] C. Ramirez-Perez, V. Ramos *SDN Meets SDR in Self-Organizing Networks: Fitting the Pieces of Network Management*, IEEE Communications Magazine, Volume: 54, Issue: 1, January 2016

[20] P. Patil, A. Hakiri, Y. Barve, A. Gokhale *Enabling Software-Defined Networking for Wireless Mesh Networks in Smart Environments*, 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), November 2016

[21] G. Poulios, K. Tsagkaris, P. Demestichas, A. Tall, Z. Altman, C. Destr *Autonomics and SDN for Self-Organizing Networks*, 2014 11th International Symposium on Wireless Communications Systems (ISWCS), August 2014

[22] Zookeeper Web site, `http://zookeeper.apache.org/`, last accessed July 2017

[23] Zookeeper available at `https://zookeeper.apache.org/doc/trunk/zookeeperOver.html`, last visited July 2017

[24] Flávio Junqueira and Benjamin Reed O'Reilly *ZooKeeper: Distributed Process Coordination*, November 2013.

[25] Zookeeper Data Model, `https://www.dezyre.com/article/zookeeper-and-oozie-hadoop-workflow-and-cluster-managers/216`, last visited July 2017

[26] HashCorp Consul introduction, `https://www.consul.io/intro/index.html`, last visited July 2017

[27] Apache Kafka, "A Distributed Streaming Platform", `http://kafka.apache.org`, last visited July 2017

[28] Linkedin, Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines), https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines, last visited July 2017

[29] Apache Kafka introduction, http://kafka.apache.org/intro.html, last visited July 2017

[30] Sean T. Allen, Matthew Jankowski, Peter Pathirana Manning Publications Company, *Storm applied: strategies for real-time event processing*, 2015.

[31] Ankit Jain, Anand Nalya Packt Publishing Ltd *Learning Storm*, 2014.

[32] Apache Storm — Concepts, available in http://storm.apache.org/releases/1.1.0/Concepts.html, last visited July 2017

[33] The Apache Software Foundation Blog, "The Apache Software Foundation Announces Apache Cassandra Release 0.6", https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces3, last visited July 2017

[34] Benchmarking Top NoSQL Databases, https://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf, last visited July 2017

[35] InfluxData InfluxDB website, https://www.influxdata.com/time-series-platform/influxdb/, last visited July 2017

[36] InfluxDB Tops Cassandra in Time-Series Data & Metrics Benchmark, https://www.influxdata.com/influxdb-vs-cassandra-time-series, last visited June 2017

[37] Openstack Gnocchi website, https://wiki.openstack.org/wiki/Gnocchi, last visited July 2017

[38] Ceilometer, available in http://docs.openstack.org/developer/ceilometer/index.html, last visited July 2017

[39] Ceilometer Metrics, available in http://docs.openstack.org/admin-guide/telemetry-measurements.html, last visited July 2017

[40] Monasca Architecture, https://wiki.openstack.org/wiki/Monasca#Architecture, last visited July 2017

[41] About Monasca, http://monasca.io/about.html, last visited July 2017

[42] MongoDB Aggregation Framework, https://docs.mongodb.com/manual/aggregation, last visited July 2017

[43] MongoDB Aggregation Framework Pipeline Operations, https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline, last visited July 2017

[44] MongoDB Aggregation Framework Pipeline results limitation, https://docs.mongodb.com/manual/core/aggregation-pipeline-limits/#result-size-restrictions, last visited July 2017

[45] MongoDB Aggregation Framework Pipeline RAM limitation, https://docs.mongodb.com/manual/core/aggregation-pipeline-limits/#memory-restrictions, last visited July 2017

[46] Trifacta Wrangler homepage, https://www.trifacta.com/products/wrangler, last visited July 2017

[47] Trifacta Wrangler Editions, https://www.trifacta.com/products/editions, last visited July 2017

[48] Trifacta Wrangler Editions, https://www.trifacta.com/start-wrangling, last visited July 2017

[49] QoSient Argus Website, https://qosient.com/argus, last visited July 2017

[50] Snort Website, https://www.snort.org/, last visited July 2017

[51] LibreNMS Website, http://www.librenms.org/, last visited July 2017

[52] Apache Chukwa Website, http://chukwa.apache.org, last visited July 2017

[53] Pivotal Software, Inc., RabbitMQ: The Most Widely Deployed Open Source Message Broker, http://rabbitmq.com, last visited July 2017

[54] Apache Kafka vs RabbitMQ, http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/, last visited July 2017

[55] Openstack Keystone Website https://docs.openstack.org/keystone/latest, last visited July 2017

[56] MySQL Website https://www.mysql.com, last visited July 2017

[57] Monasca API Documentation, https://github.com/openstack/monascaapi/blob/master/docs/monasca-api-spec.md, last visited July 2017

[58] Monasca API Spec List Metrics, https://github.com/openstack/monascaapi/blob/master/docs/monasca-api-spec.md#list-metrics, last visited July 2017

[59] Monasca API Spec List Dimension Values, https://github.com/openstack/monasca-api/blob/master/docs/monasca-apispec.md#list-dimension-values, last visited July 2017

[60] Monasca API Spec List Dimension Names, https://github.com/openstack/monasca-api/blob/master/docs/monasca-apispec.md#list-dimension-names, last visited July 2017

[61] Monasca API Spec List Measurements, https://github.com/openstack/monasca-api/blob/master/docs/monasca-apispec.md#list-measurements, last visited July 2017

[62] Monasca API Spec List Names, https://github.com/openstack/monascaapi/blob/master/docs/monasca-api-spec.md#list-names, last visited July 2017

[63] Monasca API Spec List Statistics, https://github.com/openstack/monascaapi/blob/master/docs/monasca-api-spec.md#list-statistics, last visited July 2017

[64] SLICENET Website, https://5g-ppp.eu/slicenet, last visited July 2017

Appendix A

Data Models

A.1 Raw and Aggregation Data Model (RADM) examples for Self-Protection UC: Loop 1

A.1.1 Input - Aggregation (batch) counters


```
{
    "Data":[
        {
            "timestamp":1496149720665,
            "dataType":"event",
            "reporterID":"reporterHostName=fma2/reporterIP=10.10.0.11/
    reporterAppType=FMA",
            "resourceType":"FLOW",
            "resourceID":"flowLayer=0/flowHash=3BC05E59/l3Proto=2048/srcIP
    =192.168.1.17/destIP=192.168.1.255/l4Proto=17/srcPort=58063/destPort=1534/
    encapsulationLayer=0",
            "resourceDescription":{
                "flowLayer":0,
                "flowHash":"3BC05E59",
                "encapsulationLayer":0,
                "l3Proto":"2048",
                "srcIP":"192.168.1.17",
                "destIP":"192.168.1.255",
                "l4Proto":"17",
                "srcPort":"58063",
                "destPort":"1534",
                "packetStructure":"/mac:14/ip4:20/udp:8",
                "completePacketStructure":"",
                "firstPacketSeen":1496149720665
            },
            "dataDefinition":{
                "totalOctets":175148,
                "currentPktPerPeriod":707,
                "currentOutterOctetsPerPeriod":175148,
                "totalpktCount":707,
                "currentOctetsPerPeriod":175148,
                "timeLastPacketReceived":71964762173578,
                "sumInterPacketLagNS":0,
                "totalOctetsOutter":175148,
                "lastInterPacketRate":0,
```

```
                "timeLastPacketReceivedMs":1496149720665,
                "currentSumInterPacketLagNSPerPeriod":0
            },
            "reporterDescription":{
                "reporterHostName":"fma2",
                "reporterIP":"10.10.0.11",
                "reporterAppType":"FMA",
                "reporterEpochTime":1496149720665
            }
        }
    ]
}
```


A.1.2 Output - Aggregation (batch) - monasca threshold alarm


```
{
  "Data":
  [
     {
      "reporterDescription":{
       "reporterName":"aggregation-alarm-notifier"
      },
      "reporterId":"reporterName=aggregation-alarm-notifier",
      "dataType":"alarm",
      "timestamp":1493740126554,
      "dataDefinition":{
       "severity": 3,
       "newState":"ALARM",
       "oldState":"OK",
       "alarmDescription":"Potential Zombie Detected",
       "metadata": {
        "threshold_id": "id7",
        "metrics_ids": {
          "avg_pkt_count_id": "id2",
          "communication_frequency_id": "id3"
        }
       },
       "metrics":[
        {
           "dimensions":{
            "SourceIP":"192.168.12.203",
            "DestinationPort":"10850",
            "DestinationIP":"192.168.12.223"
           },
           "id":null,
           "name":"avg_pkt_count"
        },
        {
           "dimensions":{
            "SourceIP":"192.168.12.203",
            "DestinationPort":"10850",
            "DestinationIP":"192.168.12.223"
           },
           "id":null,
           "name":"communication_frequency"
        }
       ],
       "alarm_timestamp":1493740126394,
```

```
      "subAlarms":[
       {
          "subAlarmExpression":{
           "function":"AVG",
           "deterministic":true,
           "period":180,
           "threshold":6.0,
           "periods":1,
           "operator":"GTE",
           "metricDefinition":{
            "dimensions":{

            },
            "id":null,
            "name":"communication_frequency"
           }
          },
          "currentValues":[
           6.0
          ],
          "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":[
           2.0
          ],
          "subAlarmState":"ALARM"
       }
      ]
     },
     "resourceType":"aggregation-alarm-notification",
     "resourceDescription":{
      "notificationId":"ba3e5225-7a72-45a1-95d2-8be2784dec18",
      "alarmDefinitionName":"ZombieAlert",
      "alarmId":"5a26641f-98ae-4448-8fef-fc5a2d04f04d",
      "alarmDefinitionId":"391566de-7f71-497d-a015-4dc5722ec432",
      "tenantId":"3d483b83620d49a6afa0a601183b2b2a"
     },
     "resourceId":"tenantId=3d483b83620d49a6afa0a601183b2b2a/alarmDefinitionId
    =391566de-7f71-497d-a015-4dc5722ec432/alarmId=5a26641f-98ae-4448-8fef-
    fc5a2d04f04d"
    }
```

```
  ]
}
```

A.2 Raw and Aggregation Data Model (RADM) examples for Self-Protection UC: Loop 2

A.2.1 Input - aggregation (realtime) - two snort zombie detected event

```
{
  "Data":
  [
    {
      "timestamp":  1491487846000,
      "dataType":   "event",
      "reporterID": "reporterName=SnortAgent/reporterIP=193.136.93.101/
    reporterAppType=DPI",
      "resourceType": "snort_dpi_event",
      "resourceID": "sensorId=0/eventIdTimestamp=1490615204444/signatureId
    =10000001/generatorId=1",
      "resourceDescription":
      {
        "sensorId": "0",
        "signatureId": "10000001",
        "signatureRevision": "1",
        "generatorId": "1",
        "classificationId": "31",
        "eventType": "zombie_alert"
      },
      "dataDefinition":
      {
        "srcIP":       "155.54.205.1",
        "dstIP":       "155.54.205.4",
        "dstServiceName": "zeus_cc_server",
        "classification": "zombie_detected"
      },
      "reporterDescription":
      {
        "reporterName": "SnortAgent",
        "reporterIP": "193.136.93.101",
        "reporterEpochTime": "1491487846000",
        "reporterAppType": "DPI",
        "reporterMAC": "00:ff:12:34:56:34"
      }
    },
    {
      "timestamp":  1491487846000,
      "dataType":   "event",
      "reporterID": "reporterName=SnortAgent/reporterIP=193.136.93.101/
    reporterAppType=DPI",
      "resourceType": "snort_dpi_event",
      "resourceID": "sensorId=0/eventIdTimestamp=1490615204444/signatureId
    =10000001/generatorId=1",
      "resourceDescription":
      {
        "sensorId": "0",
        "signatureId": "10000001",
```

```
        "signatureRevision": "1",
        "generatorId": "1",
        "classificationId": "31",
        "eventType": "zombie_alert"
      },
      "dataDefinition":
      {
        "srcIP":       "155.54.205.2",
        "dstIP":       "155.54.205.4",
        "dstServiceName": "zeus_cc_server",
        "classification": "zombie_detected"
      },
      "reporterDescription":
      {
        "reporterName": "SnortAgent",
        "reporterIP": "193.136.93.101",
        "reporterEpochTime": "1491487846000",
        "reporterAppType": "DPI",
        "reporterMAC": "00:ff:12:34:56:34"
      }
    }
  ]
}
```


A.2.2 Output - aggregation (realtime) - botnet detected alarm


```
{
  "Data":
  [
    {
      "timestamp":  1491492704000,
      "dataType":    "alarm",
      "reporterID": "reporterName=aggregation-cep-engine/reporterEpochTime
    =1491492704000",
      "resourceType": "threshold-alarm",
      "resourceID":    "alarmID=ed34cb9e-1ade-11e7-93ae-92361f002671/type=
    realtime-threshold",
      "resourceDescription":
      {
        "alarmID": "ed34cb9e-1ade-11e7-93ae-92361f002671",
        "type": "realtime-threshold"
      },
      "dataDefinition":
      {
        "severity": 2,
        "metadata":{
          "rule_id": "id6",
          "classification": "unknown",
          "correlationName":  "IPsweep in short time"
        },
        "listSrcIP": [
          "155.54.205.1",
          "155.54.205.2"
        ],
        "ccServerIP":    "155.54.205.4",
        "dstServiceName":    "zeus_cc_server"
      },
      "reporterDescription":
```

```
        {
          "reporterName": "aggregation-cep-engine",
          "reporterEpochTime": "1491492704000"
        }
      }
    ]
}
```

Appendix B

APIs

B.1 Raw Counters DB API

```
--as dba:
grant ALL on keyspace selfnet to selfnet;
grant SELECT on keyspace system to selfnet;
grant SELECT on keyspace system_traces to selfnet;

--as user selfnet, on keyspace selfnet:
--counters table
create table selfnet_counters
(
  timestamp   timestamp,
  timepartition   timestamp,
  resourceType    text,
  counterType     text,
  resourceId    text,
  reporterId    text,
  resourceDescription map<text,text>,
  dataDefinition    map<text,double>,
  reporterDescription   map<text,text>,
  PRIMARY KEY ((resourceType,timepartition), timestamp, counterType, resourceId
    , reporterId)
);
```

B.2 Raw Events DB API

```
--as dba:
grant ALL on keyspace selfnet to selfnet;
grant SELECT on keyspace system to selfnet;
grant SELECT on keyspace system_traces to selfnet;

--as user selfnet, on keyspace selfnet:
--events/alarms table
create table selfnet_events
(
  timestamp     timestamp,
  timepartition     timestamp,
  resourceType    text,
  dataType    text,
  resourceId    text,
```

```
15   reporterId      text,
16   resourceDescription     map<text,text>,
17   dataDefinition      map<text,text>,
18   reporterDescription     map<text,text>,
19   PRIMARY KEY ((dataType,resourceType,timepartition), timestamp, resourceId,
       reporterId)
20 );
```

Appendix C

Configuration Files

C.1 Raw Data Loader Configuration

C.1.1 Yaml file example

```
version: 0.2

#kafka consumer configuration
kafka:
  #broker host:ip (default=localhost:9092)
  bootstrapserver: "localhost:9092"
  #topics list (default=test)
  topics:
    - "test"
    - "ceil"
  #autocommit (default=true)
  autocommit: true
  #records (default=10000000)
  records: 10000000
  #groupID (default=group_1)
  groupID: "group_1"
  #commitInterval (default=100)
  commitInterval: 100
  #poolTimeout (default=1000)
  poolTimeout: 1000

#cassandra mapper configuration
cassandra:
  #tableCounters (default=selfnet_counters)
  tableCounters:    "selfnet_counters"
  #tableEvents (default=selfnet_events)
  tableEvents:    "selfnet_events"
  #keyspace (default=selfnet)
  keyspace: "selfnet"
  #user (default=selfnet)
  user:     "selfnet"
  #password (default=selfnet)
  password: "selfnet"
  #host (default=localhost)
  host:     "localhost"
  #port (default=9042)
```

```
  port:      "9042"

#raw data loader configuration
rdl:
  #threads join timeout (default=1000)
  jointimeout:  1000
  #async boolean flag indicates if cassandra db insertions are async or not (
    default=true)
  async:        true
  #activates debugging mode (default=true)
  debug:        true
  #rdl-collector configuration
  collector:
    #threads polling period (default=5)
    pollingtime: 5
    #num of threads (default=1)
    threads:     1
  #rdl-collector configuration
  persister:
    #threads polling period (default=100)
    pollingtime: 100
    #num of threads (default=1)
    threads:     1
```

C.1.2 Yaml file deployed

```
version: 0.2

#kafka consumer configuration
kafka:
  #broker host:ip (default=localhost:9092)
  bootstrapserver: "192.168.89.226:9092"
  #topics list (default=test)
  topics:
        #Nix Ceilometer
    - "ceil"
        #SNORT emulator
    - "snort"
        #FMA metrics
    - "metrics"
        #FMA events
    - "events"
  #autocommit (default=true)
  autocommit: true
  #records (default=10000000)
  records: 10000000
  #groupID (default=group_1)
  groupID: "rdl2_consumer_group"
  #commitInterval (default=100)
  commitInterval: 20
  #poolTimeout (default=1000)
  poolTimeout: 100

#cassandra mapper configuration
cassandra:
  #tableCounters (default=selfnet_counters)
  tableCounters:    "selfnet_counters"
  #tableEvents (default=selfnet_evemts)
```

```
  tableEvents:    "selfnet_events"
  #keyspace (default=selfnet)
  keyspace: "selfnet"
  #user (default=selfnet)
  user:     "selfnet"
  #password (default=selfnet)
  password: "selfnet"
  #host (default=localhost)
  host:     "192.168.89.225"
  #port (default=9042)
  port:     "9042"

#raw data loader configuration
rdl:
  #threads join timeout (default=1000)
  jointimeout:  1000
  #async boolean flag indicates if cassandra db insertions are async or not (
    default=true)
  async:        true
  #activates debugging mode (default=true)
  debug:        true
  #rdl-collector configuration
  collector:
    #threads polling period (default=5)
    pollingtime: 20
    #num of threads (default=1)
    threads:     5
  #rdl-collector configuration
  persister:
    #threads polling period (default=100)
    pollingtime: 0
    #num of threads (default=1)
    threads:     5
```

Appendix D

Rules examples

D.1 Self-Protection use case

D.1.1 Botnet listing using SNORT

```
{
  "id":"uuid3",
  "name":"SelfProtection-CEP",
  "sensor_ref_list":[
    "SNORT"
  ],
  "dimensions":[
    {
      "name":"ccServerIP",
      "source_ref":"SNORT.snort_dpi_event.dstIP"
    },
    {
      "name":"dstServiceName",
      "source_ref":"SNORT.snort_dpi_event.dstServiceName"
    }
  ],
  "rules":[
    {
      "id":"id6",
      "name":"listSrcIP",
      "group_by":[
        "SNORT.snort_dpi_event.dstIP"
      ],
      "filters":[
        {
          "condition":"SNORT.snort_dpi_event.resourceType == snort-dpi-alert"
        }
      ],
      "formula":"LIST(Sensor.SNORT.snort_dpi_event.srcIP)",
      "period":5,
      "unit":"",
      "metadata":{
        "correlationName":"IPsweep in short time",
        "classification":"unknown",
        "rule_id":"100"
      }
```

```
        }
    ]
}
```