JOSÉ CASTILLO LEMA

A GENERIC NETWORK FUNCTION VIRTUALIZATION MANAGER AND ORCHESTRATOR FOR CONTENT-CENTRIC NETWORKS

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Doutor em Ciências.

SÃO PAULO

2019

JOSÉ CASTILLO LEMA

A GENERIC NETWORK FUNCTION VIRTUALIZATION MANAGER AND ORCHESTRATOR FOR CONTENT-CENTRIC NETWORKS

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Doutor em Ciências.

Área de concentração: Sistemas Eletrônicos

Orientador: Prof. Dr. Sergio Takeo Kofuji

SÃO PAULO

2019

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Castillo Lema, José
A generic Network Function Virtualization Manager and Orchestrator for Content-Centric Networks / J. Castillo Lema -- versão corr. -- São Paulo, 2019.
119 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1.Network Function Virtualization 2.Information-Centric Networking 3.Future Internet I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II.t.

AGRADECIMENTOS

A minha base, minha família, meus pais e minha irmã. As quinhentas vezes que meu pai me cobrou o estado da Tese.

A Denise, por ter sido a minha parceira e me aguantado durante grande parte do doutorado.

Ao meu orientador prof. Takeo, pela acolhida na USP e constante cobrança pela superação dos resultados.

Aos professores Augusto e Flávio, que sempre me apoiam.

Aos meus colegas do laboratório PAD, especialmente Stelvio e Cabrini, pelo companheirismo.

Ao meu colega Gerd, por todo o aprendizado nos últimos dois anos.

A Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

FOREWORD

The work described in this thesis was conducted at the High Performance and Pervasive Computing Systems Group (PAD) from the Laboratório de Sistemas Integráveis (LSI) at the Polytechnic School of the University of São Paulo (EPUSP) and at 5GinFIRE experimental testbed in Brazil in the Federal University of Uberlândia (UFU), within the context of the following projects:

- 5GinFIRE – A three years Research and Innovation action / project under the EU programme Horizon 2020 (Grant Agreement no. 732497) started on January 2017. The main 5GINFIRE goal is to build and operate an Open, and Extensible 5G NFV-based Reference (Open5G-NFV) ecosystem of Experimental Facilities that not only integrates existing FIRE facilities with new vertical-specific ones but also lays down the foundations for instantiating fully softwarised architectures of vertical industries and experimenting with them.
- Necos – Novel Enablers for Cloud Slicing – is a new EU-BR collaborative project, coordinated on the Brazilian side by INTRIG/UNICAMP, started in November 2017. The NECOS project addresses the limitations of current cloud computing infrastructures to respond to the demand of new services, as presented in two use-cases, that will drive the whole execution of the project. The first use-case is Telco service provider focused and is oriented towards the adoption of cloud computing in their large networks. The second use-case is targeting the use of edge clouds to support devices with low computation and storage capacity. The envisaged solution is based on a new concept “Lightweight Slice Defined Cloud (LSDC)” as an approach that extends the virtualization to all the resources in the involved networks and data centers and provides a uniform management with a high-level of orchestration.

The work done during this thesis resulted in the following publications:

- J. Castillo-Lema, A. Neto, F. Oliveira and S. Kofuji. “Mininet-NFV: Evolving Mininet with OASIS TOSCA NVF profiles Towards Reproducible NFV Prototyping", in 2019 2nd International Workshop on Advances in Slicing for Softwarized Infrastructures, IEEE Conference on Network Softwarization (NetSoft), Paris, France, June 2019.
- J. Castillo-Lema, A. Neto, F. Oliveira and S. Kofuji. “Network Function Virtualization in Content-Centric Networks", in 2019 Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), Workshop de Pesquisa Experimental da Internet do Futuro (WPEIF), Gramado, Brazil, May 2019.

The following software was made publicly available:

- Mini-NFV Framework (CASTILLO, 2018a)
- ETSI2TOSCA converter (CASTILLO, 2018b, p. 2)
- Locust CCN Client (CASTILLO, 2019)

RESUMO

GERENCIAMENTO E ORQUESTRAÇÃO DE VIRTUALIZAÇÃO DE FUNÇÕES DE REDE EM REDES ORIENTADAS Á CONTEÚDO

Objetivo: A virtualização de funções de rede (NFV) oferece uma maneira alternativa para projetar, implantar e gerenciar funções e serviços de rede, aproveitando as tecnologias de virtualização para consolidar funções de rede em plataformas de hardware de uso geral. Nos últimos anos, um grande esforço foi feito para desenvolver e amadurecer tecnologias NFV em redes IP. No entanto, pouca ou nenhuma tentativa foi feita para integrar NFV em redes orientadas à conteúdo (ICN). Este trabalho explora o uso e implementação de funções de redes virtualizadas (VNFs) em redes orientadas à conteúdo (ICN), e propõe a utilização do paradigma de redes de funções nomeadas (NFN) como meio para implementar funções e serviços de rede neste tipo de redes, distribuindo as funções e os serviços de rede através dos nós da rede e fornecendo flexibilidade para colocar dinamicamente funções na rede conforme necessário e sem a necessidade de um controlador central. Foi proposta a Arquitetura Dinâmica de Funções Nomeadas (DNFA), seguindo um modelo escalável e flexível que permite a gestão da implantação, monitoramento, otimização e gerenciamento do ciclo de vida automatizado de funções nomeadas sobre infraestruturas de rede, seguindo as orientações arquitetônicas para NFV do Instituto Europeu de Padrões de Telecomunicações (ETSI) e usando um modelo de perfil customizado da definição de metadados de VNF do modelo OASIS TOSCA NFV.

Descritores: Internet do Futuro, Redes orientadas à conteúdo, Virtualização das funções de rede

ABSTRACT

A GENERIC NETWORK FUNCTION VIRTUALIZATION MANAGER AND ORCHESTRATOR FOR CONTENT-CENTRIC NETWORKS

Network Functions Virtualization (NFV) offers an alternative way to design, deploy, and manage networking functions and services by leveraging virtualization technologies to consolidate network functions into general-purpose hardware platforms. On the past years extensive effort has been made to evolve and mature NFV technologies over IP networks. However, little or no attempts at all have been made to incorporate NFV into Information-Centric Networks (ICN). This work explores the use and implementation of Virtual Network Functions (VNFS) in Content-Centric Networks (CCN) and proposes the use of the Named Function Networking (NFN) paradigm as means to implement network functions and services in this kind of networks, distributing the network functions and services through the networks nodes and providing flexibility to dynamically place functions in the network as required and without the need of a central controller. The Dynamic Named Function Architecture (DNFA) was proposed, a scalable and flexible framework that allows the automated deployment, management, monitoring, optimization and lifecycle management of named functions over CCN infrastructures, following the European Telecommunications Standards Institute (ETSI) NFV architectural guidelines and using an extended customized OASIS TOSCA NFV profile template for VNF meta-data definition. Evaluation of the framework was carried out on a private OpenStack testbed scenario through a DPI VNF implementation.

Key words: Network Function Virtualization, Information-Centric Networking, Future Internet.

INDEX

ACRONYMS XII
1 INTRODUCTION 16
1.1 OBJECTIVES 19
1.2 CONTRIBUTIONS 19
1.3 ORGANIZATION 20
2 BACKGROUND 23
2.1 INFORMATION-CENTRIC NETWORKING 24
2.1.1 Content-Centric Networking 25
2.2 SERVICE-CENTRIC NETWORKING 29
2.3 NAMED FUNCTION NETWORKING 31
2.3.1 Orchestrating Function Names in NFN 33
2.3.2 Architecture and operations 34
2.3.3 Combining expression resolution with forwarding 35
2.4 CONCLUSION 38
3 RELATED WORK 40
3.1 NFV AIDING ICN IMPLEMENTATION 40
3.2 ICN AIDING NFV ORCHESTRATION 40
3.3 NFV IN ICN 41
3.3.1 NFV IN NFN 41
3.3.1.1 SERVERLESS PUBLIC CLOUD EVALUATION COMPARISON 43
3.4 NFV MANO FRAMEWORKS 43
3.4.1 ONAP 43
3.4.2 OSM 44
3.4.3 OPENSTACK TACKER 45
3.4.4 COMPARISON 46
3.5 CONCLUSION 47
4 DYNAMIC NAMED FUNCTION ARCHITECTURE 49
4.1 FUNCTIONALITIES 49
4.2 DIFFERENCES WITH IP-BASED NETWORK FUNCTION VIRTUALIZATION 50
4.3 ARCHITECTURE 51
4.3.1 DNFA Infrastructure 54

4.3.2 Named Function Manager 55
4.3.3 Orchestrator 57
4.3.3.1 DNFA Service Function Chaining 59
4.3.3.2 Orchestrator API 61
4.3.4 Service, NFN and Infrastructure Description 64
4.4 OPERATION 65
4.5 DESIGNING THE NAME SPACE 67
4.6 ADITIONAL CONSIDERATIONS 69
4.6.1 VIRTUAL NETWORK FUNCTIONS IN CONTENT-CENTRIC NETWORKS 69
4.6.2 PLACING AND DISTRIBUTING NAMED FUNCTIONS 71
4.6.3 Performance concerns 72
4.6.4 Security concerns 73
4.7 CONCLUSION 75
5 MINI-NFV 77
5.1 FUNCTIONAL ARCHITECTURE 77
5.2 ETSI NFV TEMPLATE SUPPORT 82
5.3 VNF MANAGEMENT 83
5.4 NFV ORCHESTRATION MANAGEMENT 84
5.5 TOPOLOGY CUSTOMIZATION 85
5.6 PARAMETRIZED TEMPLATES 85
5.7 EXPERIMENTS 86
5.8 CONCLUSION 89
6 IMPLEMENTATION AND EVALUATION 91
6.1 IMPLEMENTATION 91
6.1.1 DNFA Framework 91
6.1.2 Use Cases 91
6.2 EVALUATION 92
6.2.1 TOPOLOGY 93
6.2.2 CONSECUTIVE VNF INVOCATIONS 95
6.2.3 SEQUENTIAL VS PARALLEL IMPLEMENTATION 96
6.2.4 MASSIVE CONCURRENT VNF INVOCATIONS 97
6.3 CONCLUSION 103
7 CONCLUDING REMARKS 105
7.1 CONTRIBUTIONS 105

7.2 LIMITATIONS 106
7.3 FUTURE WORK 107
REFERENCES 108
APPENDIX A – EXTENDED NFV TOSCA PROFILE 114
APPENDIX B – IP/OPENSTACK TACKER VDPI TOSCA NFV PROFILE 116
APPENDIX C – DNFA VDPI TOSCA NFV PROFILE 118

LIST OF FIGURES

FIGURE 1 - AN EXAMPLE SHOWING CCN FUNCTIONALITIES 22
FIGURE 2 - THREE SCENARIOS THAT NFN MUST HANDLE..... 27
FIGURE 3 - NFN COMBINED RESOLUTION AND FORWARDING STRATEGY FOR CCN 30
FIGURE 4 ABRIDGED DNFA REFERENCE ARCHITECTURAL FRAMEWORK .. 52
FIGURE 5 COMPLETE DNFA REFERENCE ARCHITECTURAL FRAMEWORK 53
FIGURE 6 EXAMPLE OF AN HYBRID DNFA INFRASTRUCTURE 55
FIGURE 7 NETWORK FUNCTION JSON EXAMPLE REPRESENTATION 63
FIGURE 8 NAMED FUNCTION JSON EXAMPLE REPRESENTATION..... 63
FIGURE 9 NETWORK FUNCTION/NAMED FUNCTION DATA MODEL 64
FIGURE 10 EXAMPLE USE CASE 65
FIGURE 11 EXAMPLE OF NAME TREE STRUCTURE 68
FIGURE 12 OVERALL ARCHITECTURE WITHIN THE ETSI MANO ARCHITECTURAL FRAMEWORK..... 78
FIGURE 13 WORKFLOW FOR VNF DEPLOYMENT IN MINI-NFV 80
FIGURE 14 MEMORY CONSUMPTION 87
FIGURE 15 CREATION/TERMINATION TIMES 88
FIGURE 16 DELAY VS NUMBER OF VNFS..... 89
FIGURE 17 OPENSTACK VIRTUAL INFRASTRUCTURE TOPOLOGY 94
FIGURE 18 OVERLAY HYBRID CCN/NFN INFRASTRUCTURE TOPOLOGY 95
FIGURE 19 DELAY OF CONSECUTIVE DPI INVOCATIONS 96
FIGURE 20 DELAY OF MONOLITHIC vs PARALLEL DPI IMPLEMENTATION…..97
FIGURE 21 RESQUESTS PER SECOND..... 99
FIGURE 22 GLOBAL RESPONSE TIME 100
FIGURE 23 RESPONSE TIME PER CONTENT POPULARITY..... 101
FIGURE 24 INTER-NODE TRAFFIC IN REQUESTING RELAYS R 103

LIST OF TABLES

TABLE 1 - NFN COMPARISON 42
TABLE 2 - MANO COMPARISON 46
TABLE 3 - NETWORK FUNCTIONS REST API ATTRIBUTES 62
TABLE 4 - NAMED FUNCTIONS REST API ATTRIBUTES 62
TABLE 5 - MINI-NFV TOPOLOGY BENCHMARKS 88
TABLE 6 - DNFA BENCHMARKS BY CONTENT POPULARITY 102
TABLE 7 - IP BENCHMARKS BY CONTENT POPULARITY 102

ACRONYMS

API – Application Programming Interface
AWS – Amazon Web Services
BSS – Business Support Systems
CCN – Content-Centric Networking
CLI – Command Line Interface
CNT – Content Name Translation
CP – Connection Point
CPE – Customer Premises Equipment
CS – Content Store
DHCP – Dynamic Host Configuration Protocol
DNFA – Dynamic Named Function Architecture
DNS – Domain Name System
DONA – Data-Oriented Network Architecture
DPI – Deep Packet Inspection
DoS – Denial of Service
ECOMP – Enhanced Control, Orchestration, Management and Policy
ETSI – European Telecommunications Standards Institute
FaaS – Function as a Service
FIA – Future Internet Assembly
FIB – Forwarding Information Base
FNT – Function Name Translation
FP – Forwarding Path
GUI – Graphical User Interface
HTTP – Hypertext Transfer Protocol
HOT – Heat Orchestration Template
ICN – Information Centric Networking
ICT – Information and Communications Technology
IEEE – Institute of Electrical and Electronics Engineers
JSON – JavaScript Object Notation
IDS – Intrusion Detection System
IP – Internet Protocol

IPS – Intrusion Prevention System

ISG – Industry Specification Group

ISP – Internet Service Provider

IoT – Internet of Things

LCE – Leave Copy Everywhere

LTE – Long Term Evolution

LTE-A – Advanced

MANO – Management and Orchestration

NAT – Network Address Translation

NE – Network Element

NF – Network Function

NFN – Named Function Networking

NFV – Network Function Virtualization

NFVI – Network Function Virtualization Infrastructure

NFVO – Network Function Virtualization Orchestrator

NSD – Network Service Descriptor

OASIS – Organization for the Advancement of Structured Information Standards

ONAP – Open Network Automation Platform

OPEN-O – OPEN-Orchestrator Project

OPNFV – Open Platform for NFV

OSG – Open Source Group

OSI – Open System Interconnection

OSM – Open Source Mano

OSS – Operation Support Systems

PIT – Pending Interest Table

PURSUIT – Publish-Subscribe Internet Technology

PoP – Point of Presence

QoS – Quality of Service

REST – Representational State Transfer

SAIL – Scalable and Adaptative Internet Solutions

SCN – Service-Centric Networking

SDN – Software-Defined Networking

SFC – Service Function Chain

SoA – Service Oriented Architecture

TOSCA – Topology and Orchestration Specification for Cloud Applications

vCPE – Virtual Customer Premises Equipment

VDU – Virtual Deployment Unit

VIM – Virtualized Infrastructure Manager

VL – Virtual Link

VNF – Virtual Network Function

VNFC – Virtual Network Function Component

VNFD – Virtual Network Function Descriptor

VNFFG – Virtual Network Function Forwarding Graph

VNFFGD – Virtual Network Function Forwarding Graph Descriptor

VNFM – Virtual Network Function Manager

WAF – Web Application Firewall

WAN – Wide Area Network

WAVE – Wireless Access in Vehicular Environments

WiMAX – Worldwide Interoperability for Microwave Access

YANG – Yet Another Next Generation

1 INTRODUCTION

1 INTRODUCTION

Network functions, also known as ”middleboxes”, are playing an increasingly important role in modern networks, ranging from mobile networks, enterprise networks, to data-center networks. Surveys have been showing for some time (SHERRY; RATNASAMY, 2012) that the number of network functions is comparable to that of the forwarding devices, indicating their significance. Network functions improve the network performance (e.g., Wide Area Network – WAN Optimizer, web proxy and video transcoder, load balancer), enhance the security (e.g., Intrusion Detection System – IDS / Intrusion Prevention System – IPS) or monitor the traffic (e.g., lawful interception, passive network monitor).

The need to perform additional processing of packets of a data flow in the network before it is delivered to the destination has become an integral element of providing Internet services. These functions include the modification of the packet header (e.g., Network Address Translation – NAT, proxy), discard packets (e.g., firewall), collection of statistical information (e.g., Deep Packet Inspection – DPI) or even the modification of the payload (e.g., optimization and compression). They are provided in the form of policy control, security and performance optimization middleboxes. In an Internet Protocol (IP) based environment, these middleboxes have to be resident on the path of a flow, which implies that the traffic has to be deviated from its “natural” IP shortest path and forced through the middleboxes.

Conventionally, network functions were built in dedicated hardware for performance concerns, which incur high capital investment and operating expense. Furthermore, they were hard to manage. Their replacement and upgrade involve non-trial human labour. Considering this situation, Network Function Virtualization (NFV) (LI; CHEN, 2015) was proposed, aimed to address these issues by leveraging visualization technologies to consolidate network functions into general-purpose hardware platforms. NFV, along with Software-Defined Networking (SDN) paradigm, enables automated management of the whole life cycle of virtual network functions, leading to resource efficiency and expense reduction.

For the last 5 years Network Function Virtualization (NFV) has been one of the biggest trends in the telco space (“TELECOMS.COM INTELLIGENCE ANNUAL INDUSTRY SURVEY 2018”, 2018)0029. NFV introduces a new way to build service

provider networks, where instead of investing in expensive, carrier grade hardware appliances telecommunication providers deploy cheap Commercial Off-The-Shelf (COTS) servers and run network functions (like routers, session border controllers etc.) as pure software. The outcome is cost savings, less vendor lock-in and, most importantly, the ability to update software network functions and physical hardware on independent cycles.

NFV allows for middlebox functions to be present in greater number and positioned on-demand. Network service providers will increasingly adopt NFV to provide network resident functionality, not only for reducing CAPEX (MIJUMBI et al., 2016) but also for offering more flexibility to customers who would like customized processing of their packets. However, managing such a network of dynamically placed functions can be much more complex.

The initial perception of NFV was that virtualized capability should be implemented in data centers. This approach works in many (but not all) cases. NFV presumes and emphasizes the widest possible flexibility as to the physical location of the virtualized functions. Ideally, therefore, virtualized functions should be located where they are the most effective and least expensive. That means a service provider should be free to locate NFV in all possible locations, from the data center to the network node to the customer premises. This approach, known as distributed NFV, has been emphasized from the beginning as NFV was being developed and standardized, and is prominent in the recently released NFV European Telecommunications Standards Institute (ETSI) Industry Specification Group (ISG) documents (“ETSI GS NFV-SWA 001 V1.1.1 (2014-12)”,). For some cases there are clear advantages for a service provider to locate this virtualized functionality at the customer premises. These advantages range from economics to performance to the feasibility of the functions being virtualized.

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard put together by industry group Organization for the Advancement of Structured Information Standards (OASIS) that can be used to enable the portability of cloud applications and related IT services for telecom operators, as a data model that can be used by telecom carriers for creating templates or data descriptions of applications and infrastructure for cloud services. It is being used in conjunction with the ETSI management and orchestration (MANO) architecture to deliver NFV.

On the past years extensive effort has been made to evolve and mature NFV technologies over IP networks. However, Current routing protocols deployed in IP networks constrain how packets can be deviated from a well-defined path (e.g., shortest path) and thus cannot take full advantage of the great flexibility offered by NFV. Moreover, SDN imposes the use of a central controller, which may lead to performance penalties and jeopardizes scalability capabilities (BIANCO et al., 2010). Information-Centric Network (ICN) (AHLGREN et al., 2012) is a new networking paradigm that introduces Content Names to decouple the user interests from data location, improving in-network caching, management of mobility, multicast and peer-to-peer communications. Content-Centric Networking (CCN)(JACOBSON et al., 2009)is one of the most popular ICN architectures (JACOBSON et al., 2009).

Following the ICN line of thinking, Named Function Networking (NFN) (TSCHUDIN; SIFALAKIS, 2014) has proposed that names should not only refer to data but also to functions and computation tasks. In NFN the network's role becomes to resolve names to computations. By naming functions, the network starts acting like a computing machine, capable of not only caching content but also computation results. Note that in an NFN infrastructure the scope of programmability is beyond configuring links or data paths and processing individual packets, by contrast to the modern incarnation of programmable networks as in SDN (MCKEOWN et al., 2008) and thereby related programming environments (FOSTER et al., 2011). While NFN was originally planned for tempering, processing and delivering data, this work defends that named functions use can also be extended to deal with network management related functions.

The key contribution of this work is the Dynamic Named Functions Architecture (DNFA), a scalable and flexible framework that allows placing and orchestrating functions in a Content-Centric Network by leveraging the NFN layer, following the ETSI NFV architectural guidelines and an extended customized OASIS TOSCA NFV profile templates for VNF meta-data definition in a CCN context. Furthermore, DNFA aims to distribute network services/functions through the network infrastructure using the distributed nature of named functions and take advantage of its preferential execution opportunism to gain gratuitous parallelism and asynchronous computations. DNFA can bring many benefits, from reducing computational resources, power usage, response times and inter-node traffic to

better requisition rates and cost-efficient realization of network functions in software deployed over commodity hardware, without the need of a central controller for performing service chaining.

The DNFA evaluation is being carried out on a real testbed scenario, a private OpenStack cloud, and by network emulation experimentation through Mininet.

1.1 OBJECTIVES

The main objective of this work is to design and implement an integrated management architecture, including an orchestrator platform, for the automated deployment, lifecycle management (e.g. instantiation, configuration, update, scale up/down, termination, etc.) and orchestration of VNFs over CCN infrastructures.

Other specific objectives are:

- explore the use of named functions and the NFN paradigm as means to implement network functions and services in content-centric networks;
- distribute network services/functions through the network infrastructure using the distributed nature of named functions, and take advantage of its preferential execution opportunism to gain gratuitous parallelism and asynchronous computations;
- provide a native and purely content-based non central controller-dependant network function execution environment, allowing the creation of more agile content-centric networks;
- propose an extended TOSCA NFV template profile for CCN scenarios;
- model, implement, experiment, validate and evaluate the proposal and various use cases in a real testbed.

1.2 CONTRIBUTIONS

The DNFA proposal owns the following contributions:

- an integrated management architecture, including an orchestrator platform, for the automated deployment, lifecycle management and orchestration of named functions over CCN infrastructures;

- the use of named functions and the NFN paradigm to implement network functions and services in content-centric networks;
- the distribution of network services/functions through the network infrastructure taking advantage of the distributed nature of named functions, using its preferential execution opportunism to gain gratuitous parallelism and asynchronous computations;
- a native and purely content-based non central controller-dependant network function execution environment, allowing the creation of more agile content-centric networks;
- an extended TOSCA NFV template profile for CCN scenarios;
- a CCN name tree definition for NFV scenarios;
- by reducing computational resources usage, response times and inter-node traffic, gains in operational and capital benefits are expected: from improving operational efficiency and reducing power usage to better requisition rates and cost-efficient realization of network functions in software deployed over commodity hardware, without the need of a central controller.

1.3 ORGANIZATION

The remainder of the document is divided in four main chapters.

The second chapter presents the background for this work, highlighting not only the supporting technologies, but also other related approaches.

The third chapter introduces related proposals concerning NFV implementation in ICN, and highlights DNFA contributions.

The fourth chapter presents the DNFA reference framework, defining its main requirements and guidelines and the interaction between its elements. It explains the architecture implementation, and a detailed description of the mechanisms and main functions is presented, as well as the principal structures and agents constructed.

The fifth chapter introduces Mini-nfv, an orchestration/management NFV framework on top of Mininet implemented to test and aid the implementation of the DNFA framework.

The sixth chapter presents the DNFA implementation and the results of evaluations on a real testbed scenario, a private OpenStack cloud.

Finally, the seventh and last chapter presents the concluding remarks.

2 BACKGROUND

2 BACKGROUND

The way digital resources are used is going through a radical change. Nowadays, in fact, users are interested to share contents rather than to interconnect themselves to remote devices (AHLGREN et al., 2012). Without loss of generality, the imminent possibility that this trend will affect also how enhanced services will be offered and exploited in the so-called Future Internet cannot be neglected. Accordingly, Information and Communications Technology (ICT) infrastructure should be deployed taking into account this important aspect. From a technological point of view, the present is very rich of possibilities to create, for instance, wireless communication systems, based, for example, on the Internet of Things (IoT) paradigm (Hersent et al., 2012), Long Term Evolution (LTE) and LTE Advanced (LTE-A) specifications (Dahlman et al., 2011), newer 5G specifications (“NGMN - 5G White Paper”,), Worldwide Interoperability for Microwave Access (WiMAX) standard (Walke et al., 2006), and the Wireless Access in Vehicular Environments (WAVE) protocol stack (IEEE, 2009). Unfortunately, despite the current Internet architecture already supports the communication among all of these technologies, it will not be able to offer a highly scalable and efficient distribution of contents. Moreover, it manifests a set of challenges related to:

- the decoupling of contents from the knowledge of their location, the **location-dependence** issue: it is no more necessary for many services to identify the location of a given data (i.e., no need for IP addresses);
- **security** aspects: in line with the previous requirement, it is necessary to trust contents independently from the location and the identity of who is providing them;
- **availability** issues: users need to fetch contents in a fast, reliable, and effective way;
- the **mobility** of users: it is necessary to guarantee the seamless support of mobile users which should not experience any service interruption when moving across different access networks;
- the services **scalability**: the problems related to limited storage, bandwidth, and computational capabilities that affect service providers

when handling a huge number of users should not influence the behaviour and the quality of services;

- the tolerance to system failures, the **fault tolerance** issue: for all the application fields there is the need to increase the resilience of ICT services to system failures.

Unfortunately, it is widely recognized that the current Internet architecture is not able to efficiently face all the aforementioned challenges (AHLGREN et al., 2012). In fact, the Future Internet Assembly (FIA) has identified its fundamental limitations, classifying them in four main areas (GROUP, 2011): (i) the impossibility to process and handle, in real time, data packets exchanged within the network according to the information stored within them; (ii) the lack of content/context aware caching and storage capabilities in network routers; (iii) the inefficient transmission of content-oriented data; and (iv) the lack of flexibility and adaptive control mechanisms that could react in an autonomous manner to external events.

To overcome such obstacles, novel network architectures are being conceived, based on radically different Internet primitives. A summary of the most important proposals can be found in (PAN; PAUL; JAIN, 2011). Among them, particular interest has been achieved by the emerging Information-Centric Networking (ICN) paradigm which quickly gained popularity, thanks to its inherent ability to afford all the issues argued before, proposing new networking primitives for the Future Internet (MATSUBARA et al., 2013).

Based on these considerations, a change of perspective is required; in fact, ICT platforms should be information-centric rather than host-centric as in today Internet and, at the same time, should be able to embrace the wide availability of cutting-edge wireless communication systems.

2.1 INFORMATION-CENTRIC NETWORKING

The ICN approach is currently investigated and developed in several projects, such as Data-Oriented Network Architecture (DONA) (KOPONEN et al., 2007), Publish Subscribe Internet Technology (PURSUIT) (DIMITROV; KOPTCHEV, 2010), Scalable and Adaptive Internet Solutions (SAIL) (DANNEWITZ et al., 2013),

CONVERGENCE (MELAZZI et al., 2012) and Content-Centric Networking (CCN) (JACOBSON et al., 2009). Despite all the proposed architectures differ in some aspects (e.g., naming scheme, data integrity, data granularity, routing strategy for both requests and data), all of them assume a receiver-driven data exchange model based on content names.

With respect to the classical TCP/IP architecture, the ICN approach introduces the following novel and useful features (XYLOMENOS et al., 2014):

- addressing of contents through names that do not more contain any reference to their location;
- **in-network caching** strategies to speed up the distribution of contents among users, as well as reduce the traffic load at the server side;
- delivering of user demands toward the closest device that may satisfy the request by adopting **routing-by-name** approaches;
- simplified management of **mobility:** e.g., users that modify the point of access does not perform again the initialization of the connection, but they can continue to ask consecutive portion of the requested content;
- native support to **security** features: e.g., in ICN it is possible to encrypt and authenticate directly names and contents, without relaying to sophisticated schemes conceived instead for protecting the communication among nodes;
- simplified support to **peer-to-peer communications**;
- possibility to implement lean **Quality of Service** (QoS) aware strategies for handling the routing of requests and the delivery of data packets; and
- native support to **multicast communications**.

2.1.1 Content-Centric Networking

The ICN architecture known as Content-Centric Networking (CCN) (JACOBSON et al., 2009), is based on a “data-centric” approach: all contents are identified by a unique name, allowing users to retrieve information without having any awareness about the physical location of servers (e.g., IP address). Furthermore, it targets receiver-driven communications, based on the exchange of content chunks, name-based routing, and self-certifying packets.

A CCN network can be deployed by adopting a “clean-state” approach or considering an “overlay” layer. In the first case, the entire system is redesigned from scratch, thus connecting the CCN architecture directly to the lower technological layer. In the other case, instead, it is considered a feasible solution that allows to quickly design, experiment, and deploy the proposed novel architecture on top of the current Internet structure. Anyway, in both cases, CCN introduces in the protocol suite two novel layers: the strategy layer, which is in charge of disseminating messages within the network, and the security layer, which handles all security aspects.

CCN communications are driven by the consumer of the data and only two kinds of messages are exchanged: *Interests* and *Data*. A user may ask for a content by issuing an *Interest*, routed toward the nodes in possess of the required information; these nodes are triggered to reply with *Data* packets.

Each content is uniquely identified by the *Content Name*. CCN adopts a hierarchical structure for names; therefore, a name tree is introduced. It is formed by several components, each one made by a number of arbitrary octets (optionally encrypted), so that every name prefix identifies a sub-tree in the name space. An *Interest* packet can specify the full name of the content or its prefix, thus accessing to the entire collection of elements under that prefix.

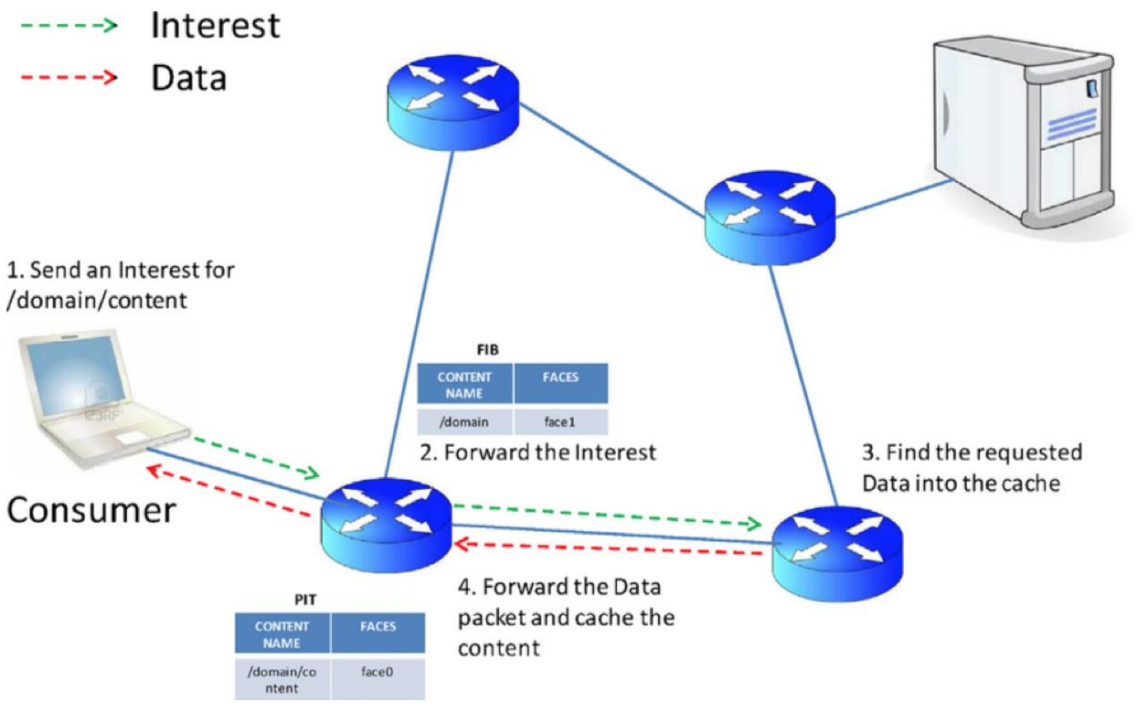
Other two novel aspects that characterize the behaviour of a CCN network are the routing and the distributed caching mechanism. Routing operations are performed by the strategy layer only for *Interest* packets. *Data* messages, instead, just follow the reverse path to the requesting user, allowing every intermediate node to cache the forwarded content. Moreover, each node receiving a *Data* packet decides whether to cache it or not, according to the adopted caching technique. In this way, the contents diffusion process is simplified, with a significant reduction both of server computational load and bandwidth consumption. In addition, since content exchange is based on content names themselves, multiple users interested to a particular content can share it using multicast suppression techniques over a broadcast medium.

To accomplish the aforementioned activities, a CCN node exploits three main structures:

- The **Content Store** (CS), which is a cache memory implementing different replacement policies, such as *least recently used*, *least frequently used* or *random*, where the received contents can be stored.
- The **Forwarding Information Base** (FIB), which is similar to a classical IP FIBs except for the possibility to have a list of *faces* (the name CCN uses for designing interfaces) for each content name entry; in this manner, *Interest* packets can be forwarded toward many potential sources of the required *Data*.
- The **Pending Interest Table** (PIT), which is a table used to keep track of the *Interest* packets previously forwarded upstream toward content sources; it saves information about the arrival *faces*. In this way, backward *Data* packets can be properly delivered to the right requesters.

To provide a further insight on the CCN paradigm, FIGURE 1 reports an example highlighting its main functionalities. In such an example, it is supposed that a consumer wants to download a content identified by the name */domain/content*. Hence, it expresses an *Interest* packet that will be forwarded by CCN routers according to information stored in FIB tables toward the device that has the requested content in its CS. In this operation, an intermediate CCN router stores all the forwarded requests in the PIT table, thus avoiding sending multiple *Interest* packets asking for the same content. Once the *Interest* packet reaches a node that can satisfy such a request, a corresponding *Data* packet will be generated and delivered to the consumer toward the reverse path. Any intermediate router, as shown in the figure, may cache the content; in this manner it is able to satisfy future requests for the same data.

FIGURE 1 AN EXAMPLE SHOWING CCN FUNCTIONALITIES



SOURCE: (PIRO et al., 2014)

Some optional fields stored into both *Interest* and *Data* packets may modify the execution of the CCN protocol in each node. For example, a device may prohibit that its generated contents are cached into the network, as well as a device sending the *Interest* packet could impose that the answer should be necessary generated by the publisher, thus bypassing any intermediate caches.

Since a *Data* packet is sent only as answer to a given *Interest* packet, CCN guarantees the flow balance. Moreover, the suppression of duplicated *Data* transmissions is achieved by using a *nonce* within the *Interest* packet. When the requested content is not received until a given timeout, the client could trigger the retransmission of the *Interest* packet; this provides a minimum level of reliability to the communication. Finally, CCN supports also some security features, such as the encryption and the authentication of both content name (or a part of it) and the content stored into the *Data* packet.

2.2 SERVICE-CENTRIC NETWORKING

While ICN strongly focuses on content retrieval, the Future Internet is expected to provide a more general support of services. Content delivery is merely one example of a service; other examples are content generation and manipulation as well as general processing services. This trend is also driven by cloud computing. (BRAUN et al., 2011, p.) proposed a Service-Centric Networking (SCN) scheme as an extension of CCN. Both content and services were considered as key design elements. SCN supports a variety of services including file storage and retrieval, audio/video streaming and recording, processing of stored images and video, on-line shopping, location-based services, cloud computing and telecommunication services.

Representational State Transfer (REST) is a style of software architecture pattern for web services based on Hypertext Transfer Protocol (HTTP). REST is completely stateless at the server and supports caching, and usually employs JavaScript Object Notation (JSON) as media type for the data. A REST-based service architecture could easily be implemented using CCN by encoding the content and the methods to be performed. However, the functionality of services is rather limited then and several complex services, in particular multimedia services, are rather difficult to be implemented inside the network. SCN aims to be more flexible and allow addressing service objects (functions), not only content objects (data) in object names.

SCN proposes to enhance content-centric networking to support general services. Data is not just retrieved, but can be processed before being presented to the user. The proposal to extend the use of names not only for content but also for services to be invoked. Services and content processed by services usually are two distinct entities that can reside at different locations in the network, and SCN provides explicit addressing for both entities. SCN leverages the concept of *Interest* and *Data* messages as defined by an underlying CCN infrastructure; a client sends *Interest* messages for services to be invoked and the results from service execution are returned in *Data* messages. The underlying CCN infrastructure should support name-based routing of *Interest* messages as well as routing of *Data* packets.

The advantages and benefits of SCN are the following:

- **No service lookup and service registry**: In traditional (web) service scenarios, services must be registered by the web service provider at a registry and must be looked up by the client before the service is invoked. In SCN, service registration is replaced by announcing service availability in the underlying CCN infrastructure, i.e., in the CCN routing tables. This results in a lower delay and avoids relying on an additional registry component. When invoking a service, no specific server needs to be addressed, but rather the service specified by its name. The CCN infrastructure automatically routes the service request encoded in an *Interest* message to the closest server supporting this service.
- **Caching of service data**: SCN leverages the features of the underlying CCN infrastructure. Thus, routers can cache data resulting from service calls and provide these data on subsequent requests. Although the benefits of caching are reduced for personalized services (e.g., commercial transactions), caching significantly reduces network traffic and response times for popular content and services. Caching is beneficial in case of mobility. Mobile users might request data or services when being mobile. This data might be stored in the network only, e.g., in case of cloud applications, and cached at network elements close to the user. After changing network access, users might request personal data again, with a high chance of being still cached close to the user.
- **Location-based services**: Traditional location-based services work as follows: 1) the client contacts a (central) server 2) the position of the client is determined, 3) the closest server is detected, and 4) the service request is redirected to the closest server. In SCN, location-based services can be easily built by deploying service entities at various locations and populating routing entries appropriately. Then, service requests (mapped to *Interest* messages) are routed to the closest (typically the local) server for processing the request independent of user locations.
- **Optimized service selection**: Service requests are sent by the client to the network using an object name that identifies the service. With the help

of the underlying CCN infrastructure, the request is routed to the most appropriate location. Optimizations can consider the distance between client and server or between server and data, etc. Therefore, a new instance of the service can be started on a resource close to the user that invokes the service or close to the data. Another option is to automatically deploy services on routers that previously received many service requests.

The authors in (SRINIVASAN et al., 2012) proposed an extension of CCN, which defines how a CCN node can implement a specific service after the reception of a given user request. This contribution represents follow up step toward the definition of a more complete platform enabling enhanced services in network environments.

2.3 NAMED FUNCTION NETWORKING

Overall, classic ICN looks like a distributed key-value store that answers name-lookup requests. Designing an information centric network as a passive content delivery infrastructure still leaves operations on data to happen “at the edge” and precludes an active role of the network beyond forwarding and caching. However, (TSCHUDIN; SIFALAKIS, 2014) cite three cases where the network can easily capitalize on computation tasks taken over from edge systems:

- **Cloud-style data transformation**: Content is generated in many forms (formats, volumes, quality, etc.), and clients need various transcoders for different devices and different uses. Instead of straining the content source to match an unbounded set of client requirements, the network can simplify things to let the network locate code, data, and find an execution place. Moreover, transformed content could be cached, avoiding duplicated efforts upon repeated requests.
- **Conditional information retrieval**: Filtering data is a known task which is more efficiently performed in the network, rather than letting clients download all possible data or having a source to service all possible filter requests. Clients can express filters in programs and let the network

decide the best course of action. At an extreme filtering case, a client may wish to retrieve information local to only one node despite the ICN trying to shield locality.

- **Data fusion**: Occasionally, an information centric network may also take over the role of a database and might have to aggregate information. The last example also highlights the trade-off between caching and generating content. Once some information has been computed, other clients do not trigger the same computation but get the cached result. In the case of volatile data sources, results would need to expire rather quickly, while a transcoded media format can remain cached as long as there is space. This trade-off between memory and re-compute cost is a basis for network-internal optimization which none of the edge nodes, client or source, would be capable of doing.

Using the data, rather than fetching it, is what users want. Moreover, there are cases where it is not feasible to transfer the data at all because of the sheer size, or security policy. In this case, “use” should happen in close locality of where the data is, implying that users must be able to express what function should be applied to the data of interest.

The authors in (TSCHUDIN; SIFALAKIS, 2014) suggested that an ICN should offer names for functions too, going one step further than SCN and enabling users to say what result they need by writing expressions that refer to data and function names; the network substrate would then be in charge of finding out how these results can be obtained, either by computing them or by looking them up in case it was already computed by others. Like in the case of removing locality-of-storage aspects from data names, in their approach they remove locality-of-execution: data caching was extended to also cover caching of results. Their work extended Named Data Networking to the realm of functional programming, by proposing that a name generally stands for a function. This function can be a constant mapping (variable lookup, as in ICN today), or a complex recipe involving many sub-operations that the network computes. Name resolution in ICN became a special case of expression resolution. A general λ-expression resolver was

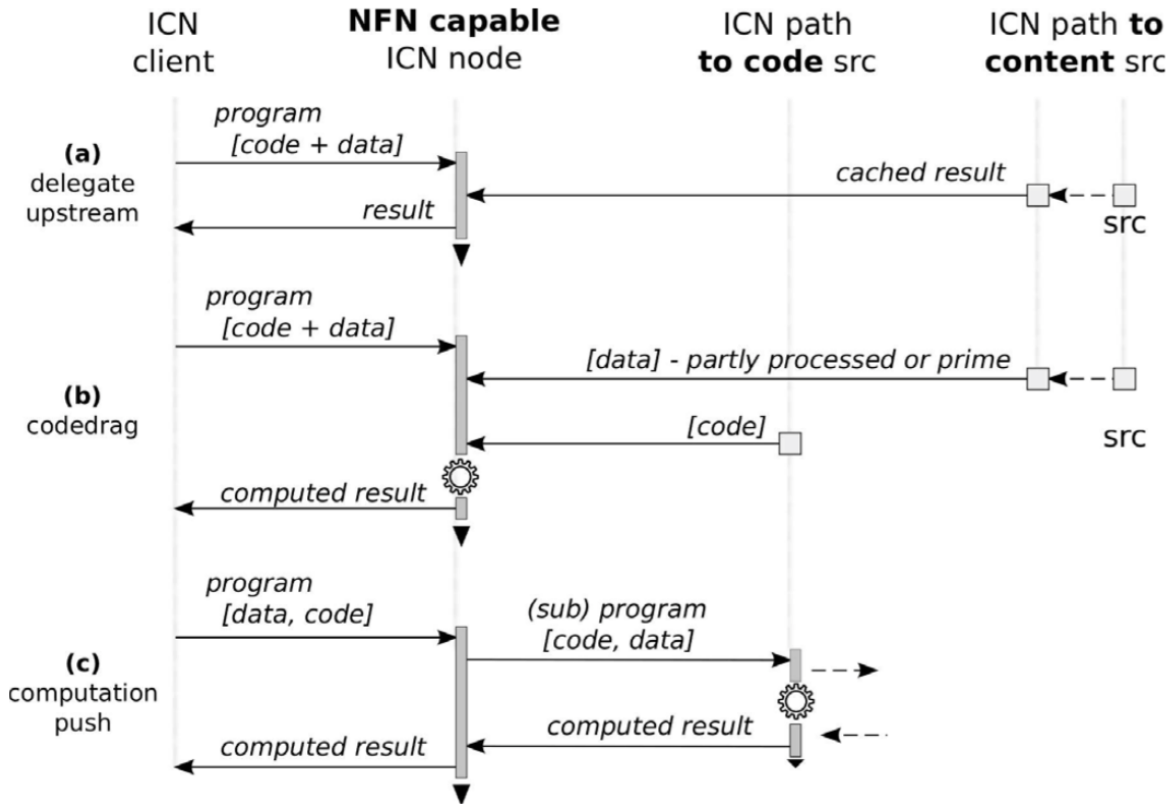
implemented, identified all memory accesses, and translated them directly to the pub/sub primitives of an ICN substrate.

This led to a generalization of “information access” so that it does not matter whether information is looked up or computed on the fly. The network is put into a position where it can make corresponding trade-offs: should it keep results in network memory or is it more economical to recompute it on demand, first fetching the code to do so. With mobile code technology well established, the network is in charge of moving data and code around and needs a way to orchestrate these moves.

2.3.1 Orchestrating Function Names in NFN

The authors in (TSCHUDIN; SIFALAKIS, 2014) reduced the general problem of controlling data and code movements to three prototypical scenarios that an ICN will be faced, represented in FIGURE 2.

FIGURE 2 THREE SCENARIOS THAT NFN MUST HANDLE



SOURCE: (TSCHUDIN; SIFALAKIS, 2014)

Given a complex content processing request, case a) of FIGURE 2 avoids recomputing or refetching information that exists elsewhere in the net. Case b) applies when information needs be produced, either because it never was computed before or is not timely available. Case c) covers a situation when some code or data is “pinned down” by either the owner or due to technical constraints. In this case the name resolution (execution) task is delegated (pushed) to the pinning site.

These three cases cover only simple tasks and by themselves could not deal with a more complex program. It remains to show how general name composition with λ-expressions can map to these three cases. Two major challenges are (i) to find a way to map arbitrary programs to ICN names and (ii) to develop some strategy for orchestrating the decomposition of complex names to computable and routable sub terms.

2.3.2 Architecture and operations

NFN was implemented by architecting the *NFN resolution engine* on top of an CCN system and by introducing network names for special constructs that give access to intermediate computation results, called nameable computation configurations.

The DNFA system uses the CCN substrate as a storage service for its internal data structures, as well as for intercepting client requests and to return (computed) results. In NFN’s untyped λ-calculus implementation, a computation configuration is a 4-tuple $< E, A, T, R >$ where:

E $name\ of\ an\ environment$

A $name\ of\ an\ argument$

T $term$

R $name\ of\ a\ result\ stack$

In that tuple, an environment E is a dictionary associating a term’s bound variables to closure names. A closure is a 2-tuple $E, T >$ associating an environment E with a term T, and serving as a context pointer during beta reductions. The argument stack is a sequence of closure names and is used to hold intermediate state during the computation of a term. All operations of the NFN resolution engine

are defined as op-codes over the argument stack. A term is a sequence of NFN-specific instructions, some of them parameterized with λ-expressions. Finally, the result stack is a normal operand-stack for interfacing with higher level languages and library functions (i.e. holding function side-effects during λ-expression evaluation). Instances of these data types are kept in the CCN's content store by name.

Although on first sight the untyped λ-calculus presented above seems nothing more than an elegant name-reshuffling machinery, it nevertheless allows to express program logic of arbitrary complexity, very compactly encoded in ICN names, and limited only by the maximum allowed length of a name (i.e. packet size). On the other hand, actual binary data processing operations cannot be efficiently handled at the name-manipulation level (although theoretically possible in reality it is impractical). For this reason, NFN assumes two levels of program execution: one regards the name-manipulation and the orchestration of computation distribution (handled by the functional untyped λ-calculus) and the other regards native code execution for actual data processing tasks at the identified execution site(s). (SIFALAKIS et al., 2014) developed a prototype in Java byte code for procedures written in the Scala language.

2.3.3 Combining expression resolution with forwarding

To spread the resolution of programs in a CCN network they were mapped to the CCN's specific naming scheme. (TSCHUDIN; SIFALAKIS, 2014) formulated a combined resolution and routing strategy, explained in this section. In the proposed scheme, the λ-expression:

$$f\big(g(data)\big)$$

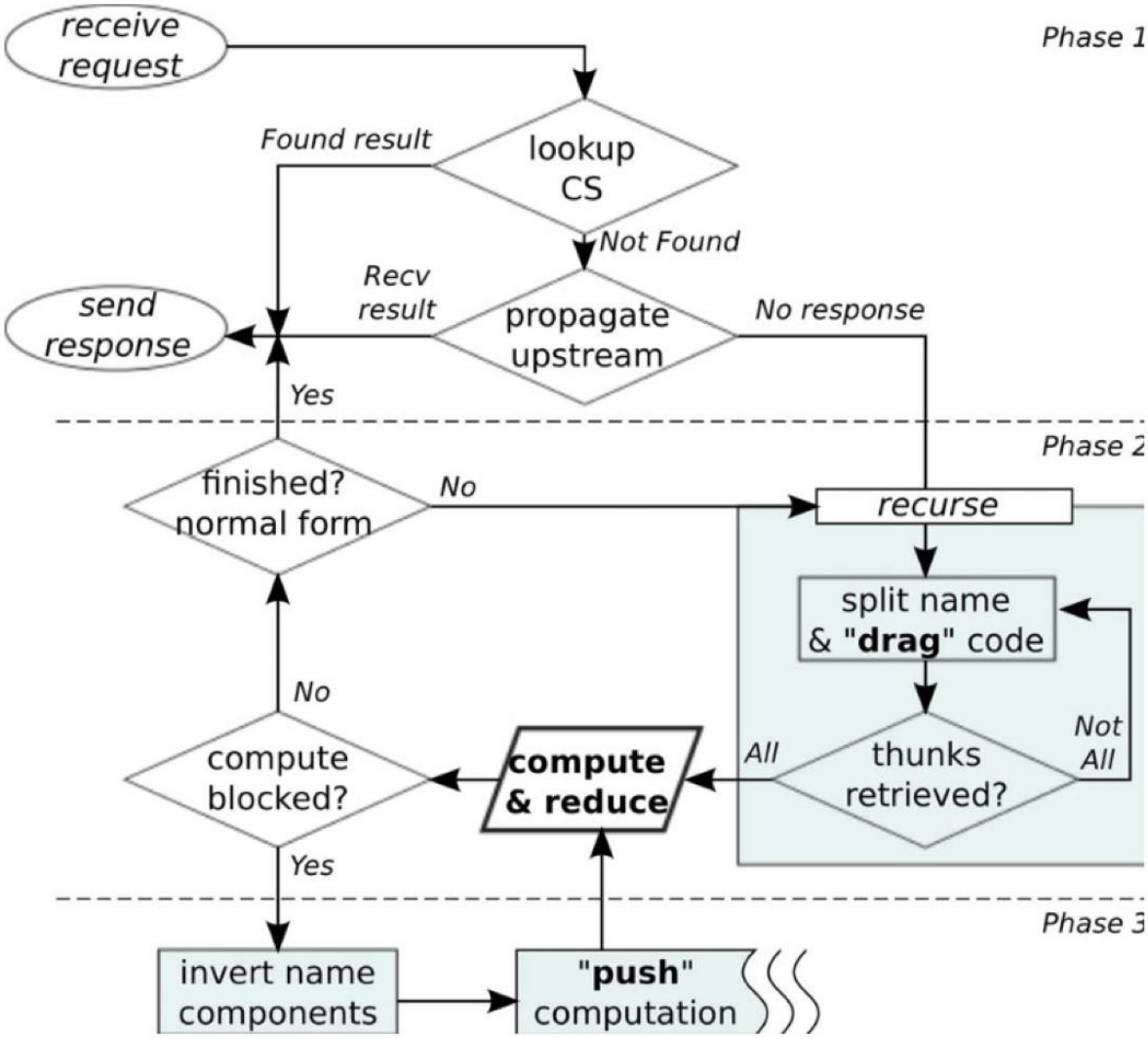
will be mapped in inverse order to a CCN name's components like in:

$$[ccn{:}\,nfn\,|data\,|\,g\,|\,f\,]$$

'[…]' brackets are used to enclose a CCN name and to separate its components. Moreover, this notation is used recursively, when a name component is itself a CCN name. Note that the components of this NFN name are themselves names.

The term inversion for CCN wire format has to do with CCN longest prefix-match forwarding philosophy. Assuming a chain of function applications $f\left(g\left(h(data)\right)\right)$, the inverted $[\ldots \mid data \mid h \mid g \mid f]$ encoding ensures that when the full expression is inquired but only intermediate results are available, partial results can be retrieved from the network along the path to the data, letting the network test first for $data \mid h \mid g \mid f$, then $data \mid h \mid g$, etc. FIGURE 3 gives an overview of the resolution and routing strategy for the combination of NFN and CCN, covering three phases.

FIGURE 3 NFN COMBINED RESOLUTION AND FORWARDING STRATEGY FOR CCN



SOURCE: (TSCHUDIN; SIFALAKIS, 2014)

Phase 1: Search by upstream delegation.

The first phase, which covers case a) in FIGURE 2, is the currently default forwarding strategy in CCN: when an *Interest* packet is received, a search in the local content store (CS) may resolve the request and satisfy immediately the pending *Interest*.

If the search fails, the *Interest* packet is propagated further up-stream according to the entries in the FIB. Since the leftmost component in the name that encodes a program refers to the innermost term of the expression, this is the target content for manipulation and due to prefix-based forwarding; the *Interest* packet travels towards the content source. This means that by default preference, the responsibility for the computation of the result is closer to the source (by contrast to the preference that the response is cached closer to the receiver). This is useful because it increases the probability of caching precalculated content as well as intermediate results in the middle of the network.

Phase 2: Search and reduce “by name”.

If the search for the complete name times out, the strategy enters the second phase and seeks in turn intermediate results that would enable it to complete the computation locally.

Recursively, starting from the outermost level of the expression, it will search in the ICN for intermediate results of the inner terms (sub-expressions), and in parallel search for the code of the outer term. This is done by tail-trimming the last (rightmost) component of the name in the received *Interest* and searching for each of the two parts separately. If the search fails for an inner term, the strategy recourses deeper in the inner sub-expression, until all functions and the content are requested independently (implying that no intermediate results were found).

Requests for intermediate results of inner terms or function code in fact retrieves only *thunks* (names of configurations). One can think of a *thunk* as a contract that the respective term can be made available, if needed. When all *thunks* become available the reduction of a pending λ-expression can progress. *Thunks* allow the evaluation of a named expression to progress even when results are not available yet (enabling asynchronous and parallel computations). Since in this process some terms might naturally disappear or cancelled, the respective *thunks* will never be executed to retrieve the code/data from the network. At this point the

strategy has implemented a call by name policy, which corresponds to case b) in FIGURE 2.

Phase 3: Inverse semantics and reduce “by value”

It can be that either the code or the required content cannot be obtained for some reason (no mobile code available or blocked content distribution), but that for example the code owner would be willing to execute the function for approved data. In this case phase 2 will not successfully complete. Phase 3 of the routing strategy will take care of this case which corresponds to scenario c) in FIGURE 2.

In λ-calculus the evaluation position of the terms affects the semantics in the order that reduction steps are performed. Given the 1:1 mapping of terms to name components in the *Interest* carrying a program, and also given the use of names for prefix-based routing/searching in CCN, this means that by changing the evaluation order of the λ-terms it is possible to influence the forwarding of the expression in the network.

If all else in phase 2 fails, it is taken advantage of this feature to swap the positions in the *Interest* name of the first components to affect the forwarding of the computation towards the code source. Computationally, the effect is that the resolution strategy has switched from call-by-name to call-by-value, which means that if at the remote end the component order is not recovered the reduction sequence will be different. But, due to the confluence theorem in λ-calculus, the final normal form of the reduced expression will be the same. Note that the strategy can resort to the 3rd phase either for the computation of the entire expression or any sub-term during the recursions of the 2nd phase.

2.4 CONCLUSION

This chapter explained some technologies and mechanisms that are used to support the DNFA architecture, through the work done in this Thesis.

An overview of the Future Internet paradigm, Information-Centric Networking, Service-Centric Networking and Named Function Networking was presented.

3 RELATED WORK

3 RELATED WORK

This chapter surveys relevant works in the context of NFV in Information Centric Networks. First, some works that leverage NFV to implement ICNs. Secondly, works that leverage ICN techniques to aid or improve some aspects of NFV Orchestration. Thirdly, works that have explicitly explored the use of VNFs in Information-Centric or Content-Centric Networks. Finally, some NFV Management and Orchestration frameworks.

3.1 NFV AIDING ICN IMPLEMENTATION

First efforts were focused on implementing ICN over SDN and OpenFlow (SYRIVELIS et al., 2012) (NGUYEN; SAUCEZ; TURLETTI, 2013) (SALSANO et al., 2013).

After wise, as NFV paradigm emerged and matured, further efforts were made to leverage NFV to implement ICNs.

In (RAVINDRAN et al., 2013) is proposed an ICN based edge-cloud service framework leveraging NFV technologies, implementing functional components of the ICN service platform controlled by an operator in an overlay deployment in an NFV realization.

In (ZHANG; ZHU, 2016) NFV is used to implement an information-centric wireless network virtualization technique, to transmit time-sensitive multimedia traffic data with the delay-bounded quality of service (QoS) guaranteed in SDN environments.

In (UEDA et al., 2016) is proposed an Network Function Virtualization Infrastructure (NFVI) assisted name-based forwarding scheme, offloading the partitioned exact match process to OpenvSwitch on NFVI and showing how ICN and NFVI could be linked and the benefits of this linkage.

3.2 ICN AIDING NFV ORCHESTRATION

The following works leverage ICN techniques to aid or improve some aspects of NFV Orchestration.

In (ARUMAITHURAI et al., 2014) is explored the potential of using information centric concepts within an SDN-based network management environment, especially focusing on service chaining.

In (ARUMAITHURAI et al., 2015) is demonstrated an Information Centric Networking based solution that complements SDN for service chaining and provides benefits such as scalability, flexibility and reliability.

Both these approaches have been followed in order to implement the Service Chaining service of the NFV Orchestrator in this work.

3.3 NFV IN ICN

From the early definitions of ICN, several works implemented various network functions into this kind of networks: access control (MARXER; SCHERB; TSCHUDIN, 2016), access privacy (MOHAISEN, 2017), lightweight authentication/secured routing (MICK; TOURANI; MISRA, 2018), publish-subscribe capabilities (MELAZZI et al., 2012), service discovery (QUEVEDO et al., 2017) and management operations (SUAREZ et al., 2016), among others.

However, few works have explicitly explored the use of virtual Network Functions in Information-Centric or Content-Centric Networks.

The work in (SUKSOMBOON et al., 2014) implemented through NFV cache state and content popularity coordination tasks from routers to a cache orchestrator where the offline optimal caching policy is computed while, leaving the simplest cache decision to the routers.

3.3.1 NFV in NFN

Several works have explored the use of Service-Centric Networks (SCN) and NFN to implement network services, as in the following.

The authors themselves of the NFN paradigm have already explored content access (MARXER; SCHERB; TSCHUDIN, 2016) and monitoring protocol (MANSOUR; TSCHUDIN, 2016) implementation trough NFN.

(TSCHUDIN; SIFALAKIS, 2014) propose to use NFN for on-the-fly data transformation (compression, encoding, transcoding, etc.) and data authentication.

(TSCHUDIN; SIFALAKIS, 2013) contemplates the following scenarios: media delivery orchestration, virtual content, content access control, active content (on-demand content draining, dynamic content updates), on-the-fly media subtitling, in-network stream-buffering and collective content-replacement across media caches.

(SIFALAKIS et al., 2014) mainly aims for data manipulation, through a series of proof-of-concept experiments designed to showcase the ability of NFN for dynamic distribution of computation tasks and interactions between static data and functions inside an ICN network; to test and identify occasions where NFN empowers network side decisions and optimisations; and finally to develop insights of how to improve the effectiveness of NFN.

TABLE 1 summarizes the main characteristics of the analysed proposals. Note that, in these works, named functions are manually deployed in absence of an integrated management framework, including an orchestrator platform, for the automated lifecycle management of VNFs.

TABLE 1 NFN COMPARISON

		Data Plane			**Control Plane**		
Authors	**Proposal**	**DT**	**DA**	**DM**	**CA**	**Mt**	**Generic VNF**
Braul *et al*., 2011	SCN	■		■			
Tschudin *et al*., 2013	NFN		■		■		
Tschudin *et al*., 2014	NFN	■					
Sifakalis et al., 2014	NFN			■			
Marxer *et al*., 2016	NFN				■		
Mansour *et al*., 2016	NFN					■	
Thesis	**DNFA**						■

Legend:

- DT: Data Transformation.
- DA: Data Authentication.
- DM: Data Manipulation.
- CA: Content-Access Control.
- Mt: Monitoring capabilities.

- Generic VNF: Support for an integrated management framework, including an orchestrator platform, for the automated lifecycle management of generic VNFs.

3.3.1.1 Serverless Public Cloud Evaluation Comparison

It is possible to make an analogy between the use for NFN that is being proposed in this work and the evolution of Serverless Function as a Service (FaaS) service in the biggest cloud provider, Amazon Web Services (AWS).

AWS Function as Service module (Amazon Lambda) was initially conceived for tempering, processing and delivering data, and recently (summer 2017) has released its Lambda@Edge service. This is still considered to be a part of AWS FaaS offering, but unlike the vanilla Lambda Functions the code is executed on the provider's edge location. It means that it is possible to trigger functions on the cloud edge and with lower latency. Some use cases include WAF scenarios and access content management.

This work defends that named functions can also be extended to deal with network management related functions.

3.4 NFV MANO FRAMEWORKS

NFV Management and Orchestration (NFV MANO) is a crucial part of the cloud infrastructure for unlocking the full potential of NFV. Two major frameworks ONAP and OSM will be discussed.

3.4.1 ONAP

The Open Network Automation Platform (ONAP) is an open-source software platform that enables the design, creation, and orchestration of services on an infrastructure layer on top of individual virtual network functions or SDN – or even a combination of the two. ONAP was formed as a combination of the Linux Foundation's OPEN-Orchestrator Project (OPEN-O), and the AT&T Enhanced

Control, Orchestration, Management and Policy (ECOMP) platform to form one unique code.

The main purpose of ONAP is to answer the rising needs of service providers to operate a common platform that will be able to provide efficient, end-to-end infrastructure management. The platform is also meant to speed up delivery of different on-demand services with the possibility of automating most of the processes.

ONAP architecture consists of two major architectural frameworks – design-time environment and execution-time environment – with numerous separate subsystems operating within them. A design-time environment is used as a development environment with all the functions and libraries needed for the development of new capabilities; the environment can be used to upgrade of existing capabilities through portal, role-based interfaces, and CLI. With a design-time environment framework, the Service Design and Creation can be used as visual tool for designing and modelling assets used in all ONAP components, while the POLICY subsystem is used to make policies and conditional rules.

The execution-time environment framework is used to execute all policies and rules prepared in design-time environment. These policies and rules are responsible for resource inventory, data collection, and analytics. They handle tasks such as performance monitoring, service orchestration for end-to-end service automation, and security based on the strong foundation inherited from ECOMP. In addition, using the Closed Loop Automation Management Platform, ONAP enables service providers to manage virtual network function life cycles and to automate deployment processes.

3.4.2 OSM

Open Source MANO (OSM) is an ETSI-hosted initiative for the development of open-source NFV Management and Orchestration software stacks. The initiative is fully aligned with the ETSI-developed NFV reference architecture. OSM is one of the first projects of the ETSI entity called Open Source Group (OSG), allowing open-source projects to be developed under ETSI.

Hosted by ETSI, the initiative is also based on components that have been around for some time like Telefonica's OpenMANO project, Canonical's Juju generic VNF Manager, and RIFT.io's orchestrator.

The idea behind OSM is to have all activities in development closely aligned with the evolution of the ETSI NFV standard. This enables vendors and operators to have an ecosystem that gives them the ability to deliver and deploy services in a cost-effective manner. In particular, OSM should allow network architects to take advantage of synergies between the ETSI NFV standard and the open-source approach. These synergies are important when developing Management and Orchestration infrastructure, an all-important part of the networks of future service providers. Like ONAP, the architecture of OSM consists of two major components: run-time scope and design-time scope.

The run-time scope component comprises different functions like automated service orchestration. The primary purpose of the component is to simplify service and lifecycle management. The component also performs provisioning for SDN controllers, including plugin models integrating multiple SDN controllers. The run-time scope also includes delivery of plugin models for integrating VIMs and multiple VIMs. Run-time scope's Generic VNF Manager is an important function to control VNFs with support for specific VNF Managers based on demand. Run-time scope supports integration of a new Physical Network Function into an automated service deployment, and a plugin model for the integration of multiple monitoring tools.

Design-time scope includes support for a model-driven environment fully aligned with the ETSI NFV standard. It also has functions for the creation, update and deletion of service definitions. Finally, it supplies a Graphical User Interface (GUI) to accelerate service design time, VNF onboarding, and deployment.

3.4.3 OpenStack Tacker

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface or API.

Tacker is a generic VNF Manager (VNFM) and NFV Orchestrator (NFVO) which helps in deployment and operation of VNFs within OpenStack. It is integrated with various OpenStack projects:

- Nova: the compute module interfacing with the hypervisor.
- Neutron: the network module, responsible (among other things) for implementing Service Function Chaining in OpenStack.
- Cinder: the block storage module.

OpenStack Tacker features include performance features like huge pages, CPU Pinning, NUMA topology and SR-IOV; service function chaining, networking slicing, scalability, high availability, resiliency and multisite enablement.

3.4.4 Comparison

OSM has been imagined as a framework that will follow the ETSI standard, with the scope to cover NFV orchestration (including network service onboarding, lifecycle, and performance management), VNF management (including VNF onboarding, lifecycle, fault, and performance management), VIM/SDN controllers, and security. On the other hand, ONAP covers all the above; and it also includes a unified design framework (supporting TOSCA and YANG) helping with end-to-end service orchestration and automation. Both implement multi-VIM support, unlike Tacker that is restricted to the OpenStack world.

TABLE 1 summarizes the main characteristics of the analysed proposals. None of the frameworks support any kind of ICN, neither as VNF Manager nor as NFV Orchestrator.

TABLE 2 MANO COMPARISON

MANO	VNFM	NFVO	TOSCA	Multi-VIM	ICN
OSM					
ONAP					
Tacker					
DNFA					

Legend:

- VFNM: Virtual Network Function Manager
- NFVO: Network Function Virtualization Orchestrator
- TOSCA: Support to TOSCA templating
- Multi-VIM: Multi-VIM support
- ICN: Information-Centric Networking support

3.5 CONCLUSION

This chapter summarizes relevant literature in the context of NFV in Information Centric Networks, and concludes that the analysis of the related work justifies the architecture proposed in this Thesis, revealing that none of the literature had proposed an integrated management architecture, including an orchestrator platform, for the automated deployment, management, monitoring, optimization and lifecycle management of VNFs over CCN infrastructures, neither envisioned the use of named functions as means of implementing network functions and services.

4 DYNAMIC NAMED FUNCTION ARCHITECTURE

4 DYNAMIC NAMED FUNCTION ARCHITECTURE

This chapter describes conceptually the functional architecture proposed by the **Dynamic Named Function Architecture (DNFA)**, a scalable and flexible framework that allows placing functions in the network by leveraging the Named Function Networking (NFN) layer, following the European Telecommunications Standards Institute (ETSI) Network Function Virtualization (NFV) architectural guidelines. Furthermore, DNFA aims to distribute network functions and services through the network infrastructure using the distributed nature of named functions and take advantage of its preferential execution opportunism to gain gratuitous parallelism and asynchronous computations. DNFA can bring many benefits, from reducing computational resources, power usage, response times and inter-node traffic to better requisition rates and cost-efficient realization of network functions in software deployed over commodity hardware, without the need of a central SDN controller to perform service chaining.

It is important to make clear the difference between network functions and named functions. Network functions are a generic form to refer the hardware or software ”middleboxes” that implement network services to improve the network performance, enhance the security or monitor the traffic, for example, and they can be hardware or software (typically implemented trough virtual network functions). On the other hand, named functions, through the NFN layer, extend classic Information-Centric Networks (ICN) such as in addition to the network resolving data access by name it also supports function definition and application to data in the same resolution-by-name process, empowering the network to select internally optimal places for fulfilling a potentially complex function computation.

4.1 FUNCTIONALITIES

The most relevant functionalities of the DNFA platform are:

- an integrated management architecture, including an orchestrator platform, for the automated deployment, lifecycle management (e.g. instantiation, configuration, update, scale up/down, termination, etc.) and orchestration of named functions over network infrastructures;

- the use of named functions and the NFN paradigm to implement network functions and services in content-centric networks;
- the distribution of network services/functions through the network infrastructure taking advantage of the distributed nature of named functions, using its preferential execution opportunism to gain gratuitous parallelism and asynchronous computations;
- exposure of functionalities such as: network service deployment and provisioning; service chaining configuration; and teardown;
- a native and purely content-based non central controller-dependant network function execution environment.

4.2 DIFFERENCES WITH IP-BASED NETWORK FUNCTION VIRTUALIZATION

Before diving more into the specifics of the DNFA architecture, it is important to make clear the differences between IP-based VNFs and DNFA-based VNFs.

- DNFA VNFs are inherently **distributed**, but considerations about where the code will be executed have to be taken. On the other hand, most IP-based VNFs reside at certain locations in the network, each responsible for a static portion of the flow path. With ever-growing network scale and traffic volume, these **centralized** virtual network functions become the performance bottleneck. Moreover, steering traffic to them incurs non-trivial overhead (path inflation).
- DNFA can provide **on-path** placement of network functions. On-path placement is desired for it introduces minimal inference to traffic engineering and obviates the need for complicated forwarding rules in routers. On the other hand, despite the variety, all existing IP-based NFV frameworks place virtual network functions at **off-path** servers.
- DNFA is a purely content-based non central controller-dependant network function execution environment, unlike IP-based NFV frameworks that rely in an SDN **central-controller** for service chaining operations.

4.3 ARCHITECTURE

The DNFA architecture is based on the European Telecommunications Standards Institute (ETSI) NFV architectural guidelines, and as far as possible the ETSI nomenclature was respected. ETSI NFV delivered a reference architecture guideline of a NFV framework (“ETSI GS NFV-SWA 001 V1.1.1 (2014-12)”,), focused on the functionalities necessary for the virtualization and the consequent operation of an operator's network, identifying the main functional blocks and the main reference points between those blocks.

DNFA is responsible for managing the virtualized infrastructure of an NFV-based solution. Its operations include:

- orchestrate the allocation, upgrade, release, and reclamation of NFVI resources and optimize their use, by keeping an inventory of the allocation of virtual resources to physical resources;
- support the management of VNF forwarding graphs by chaining named functions.

FIGURE 4 ABRIDGED DNFA REFERENCE ARCHITECTURAL FRAMEWORK

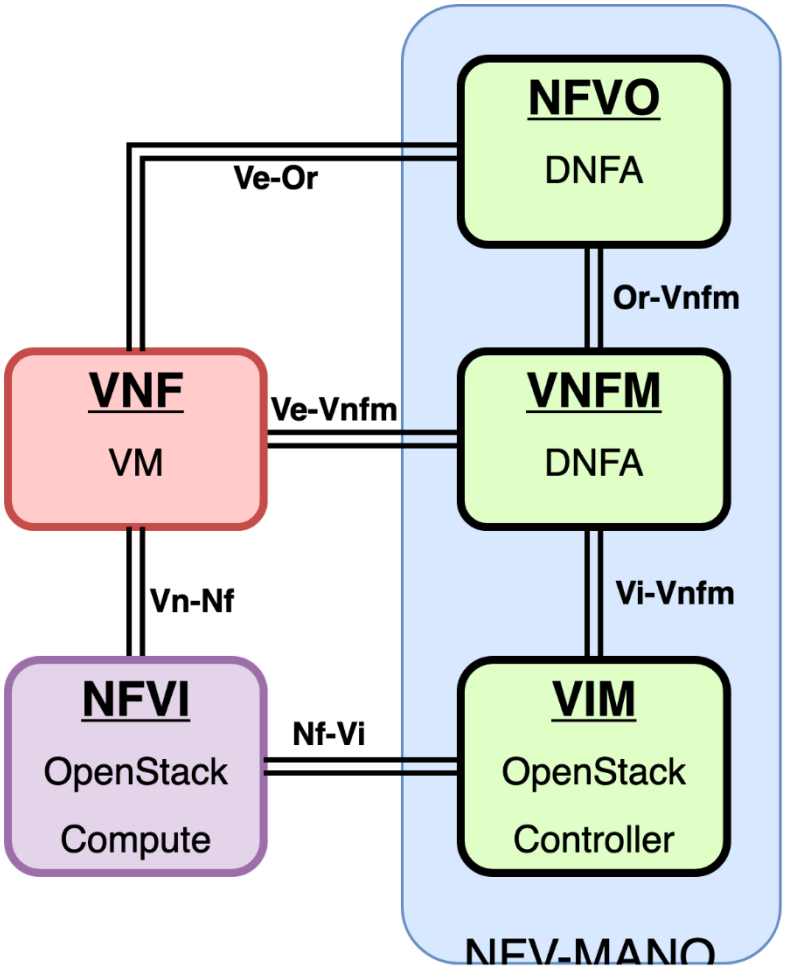


FIGURE 4 shows an abridged version of the ETSI MANO reference architecture diagram within DNFA scenario. Its architecture consists of the following building blocks:

- NFV Infrastructure (NFVI): OpenStack compute nodes in this work.
- Virtual Infrastructure Manager (VIM): OpenStack controller/API.
- VNF Manager (VNFM): the element responsible for lifecycle management (create, delete, …) of VNFs.
- NFV Orchestrator: the element responsible for service orchestration.

Different from its IP equivalent, note the absence of the link ***Or-Vi*** between the NFVO and the VIM, that will be further explained in Section 4.3.3.1. Basically, DNFA does not depend on the VIM SDN controller to implement service chaining operations and instead relies on NFN native named function chaining.

FIGURE 5 COMPLETE DNFA REFERENCE ARCHITECTURAL FRAMEWORK

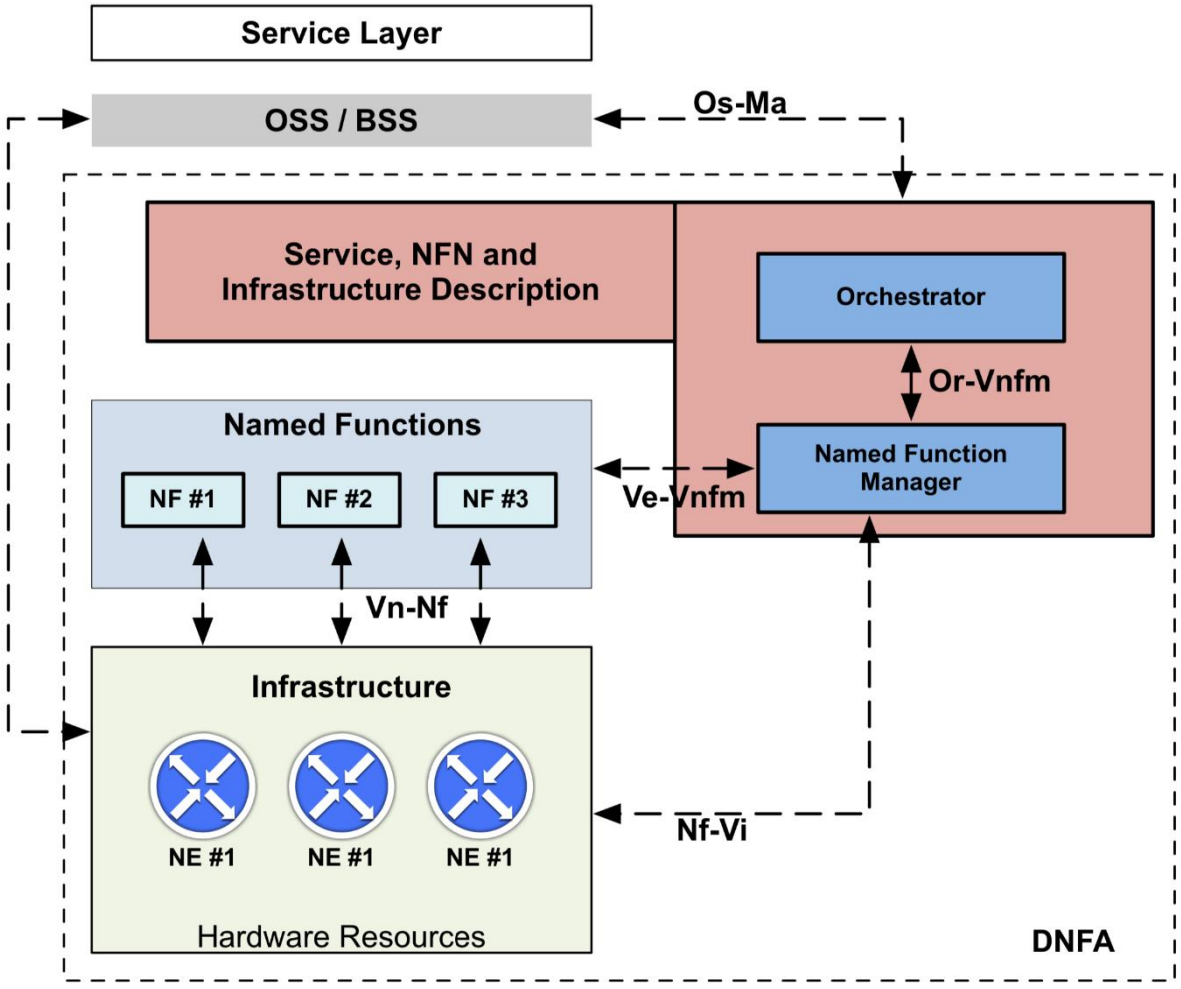


FIGURE 5 represents the complete reference architecture of a DNFA framework. The bottom left of the picture represents the *NFN Infrastructure*, all the Network Elements (NEs) that support NFN execution (ICN nodes, NFN capable or not). This infrastructure provides the necessary resources to DNFA physically span over several locations. Currently, only OpenStack is supported as Virtual Infrastructure Manager (VIM).

The middle left side of FIGURE 5 contains the *Named Functions*, which use the resources provided by the *NFN Infrastructure*. The right side contains the management and orchestration elements. The *Named Function Manager* is responsible for the lifecycle management of named functions (placement,

configuration, update, scale up/down, termination, etc.). Finally, the *Orchestrator* is responsible for the orchestration of network services.

In the top left corner are the *Operation Support Systems* (OSS) / *Business Support Systems* (BSS) of an operator as well as the *Service, NFN and Infrastructure Description*. The *Service, NFN and Infrastructure Description* provides information about the service (description, objective, limitations, etc.) and NFN information models.

Finally, the Service Layer represents where services provided to the end user are modelled.

4.3.1 DNFA Infrastructure

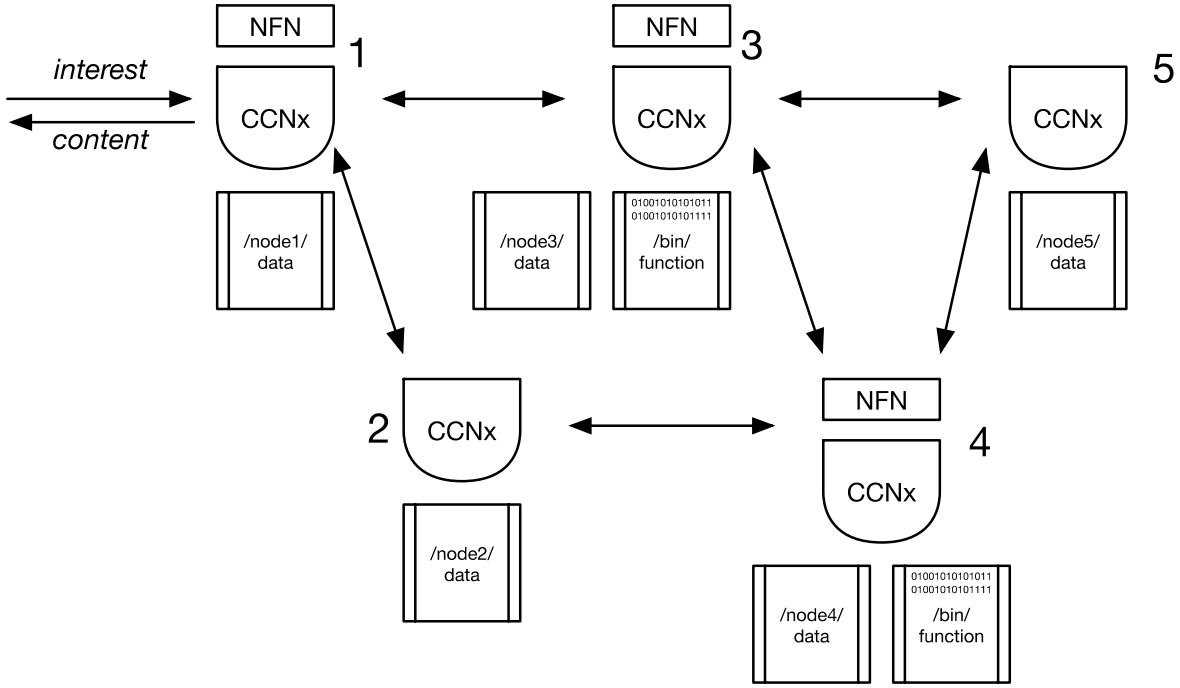
The *DNFA infrastructure* is formed by various Network Elements (NEs). These Network Elements are virtual machines containing CCN nodes (NFN capable or not). Note that the application processing/execution environment is optional in the sense that there is no requirement for all NEs to be capable of consuming data processing operations. For example, there may exist DNFA nodes that have a pure router/forwarder role, and only need the Abstract Machine for the distribution of computation tasks and caching of results (see the two level of program execution discussed in Section 2.3.2). Moreover, not all the NEs must be NFN capable (only VNF ones), as the DNFA framework will function on hybrid topologies, with both NFN capable (CCN-NFN nodes) and NFN non-capable (CCN-only) nodes.

On VNFs hosting the application processing/execution environment, a native code named function is registered in the NFN realm with a *publish* primitive by the DNFA orchestrator. This populates the CCN node's Forwarding Information Base (FIB) with the corresponding namespace entry.

FIGURE 6 represents an example of a hybrid DNFA Infrastructure, including CCN-NFN nodes as well as CCN-only nodes. It consists of five nodes, where two are pure CCN nodes and the other three are NFN nodes, two of them hosting the complete application processing/execution environment (*Node #3* and *#4*). Connections between nodes are bidirectional, and client requests always arrive at *Node #1* first (the NFN relay). Note that *Node #1* has a pure router/forwarder role, and only need the Abstract Machine for the distribution of computation tasks and

caching of results. As *Node #1* is the NFN relay, it processes all *Interest* packets that have the implicit postfix name component NFN.

FIGURE 6 EXAMPLE OF AN HYBRID DNFA INFRASTRUCTURE



DNFA is responsible for:

- orchestrate via OpenStack API the management of the lifecycle of the VNFs implemented as NFN nodes;
- initializing the FIBs of the VNF and requesting nodes and such that each node can reach every other node over the shortest path. When more than one paths are available, both are included.

4.3.2 Named Function Manager

The *Named Function Manager* is responsible for the entire lifecycle of VNFs implemented through named functions. It can request infrastructure resources and interact directly with them, e.g. to place or configure named functions. All VNFs instantiated by the DNFA Manager must receive the complete NFN application processing/execution environment.

This module can retrieve the required information about the named functions (e.g. named function configuration recipes) by accessing the *Service, NFN and Infrastructure* description module. Moreover, as part of the lifecycle management of a VNF this module is responsible for initializing and configuring the *faces* (the name CCN gives to interfaces, see Section 2.1.1) for all the network elements involved in the named functions execution.

An extended TOSCA Simple Profile for NFV is currently used in DNFA to define VNF Descriptors (VNFD) and VNF Forwarding Graph Descriptors (VNFFGD). These templates are then parsed and loaded into an active DNFA session, through the DNFA northbound API interface. A complete definition of the extended TOSCA DNFA NFV profile can be found in APPENDIX A.

VNFDs are templates that define the behavioral and deployment information of a particular VNF. Each VNF consists of one or more VNF components (VNFCs), represented in TOSCA as Virtual Deployment Units (VDUs). A VDU is the smallest part of a VNF and can be implemented as either a container or, as it is in this work, a VM. Apart from the usual set of parameters like CPU, RAM and disk, VNFD also describes all the virtual networks required for internal communication between VNFCs, called internal Virtual Links (VLs) and connected through Connection Points (CPs). VNFM can ask VIM to create those networks when the VNF is being instantiated.

Follows a basic VNFD template:

```
tosca_definitions_version:
  tosca_simple_profile_for_nfv_dnfa_1_0_0
description: Basic VBFD template
metadata:
  template name: sample_VNFd_userdata
topology_template:
node_templates:
    VDU1:
      type: tosca.nodes.nfv.VDU.DNFA
      capabilities:
        nfv compute:
          properties:
            num cpus: 1
            mem_size: 512 MB
            disk_size: 1 GB
    CP1 :
      type: tosca.nodes.nfv.CP.DNFA
      properties:
        management : true
        order: 0
```

```
    anti_spoofing_protection: false
  requirements:
    − virtualLink:
      node: VL1
    − virtualBinding:
      node: VDU1
VL1:
  type : tosca.nodes.nfv.VL.DNFA
  properties:
    network name : net mgmt
```

4.3.3 Orchestrator

The *Orchestrator* contains a northbound interface that exposes the following functionalities: network function advertisement; deployment and provisioning; monitoring and reconfiguration; and teardown. On the south side it has an interface to the different *Named Functions Managers*. Furthermore, it has access to the *Service, NFN and Infrastructure description* module that provides the *Orchestrator* high-level information about services and infrastructure. The actual service configuration and implementation are transparent to the orchestrator, establishing an abstraction layer towards the *Named Functions Managers*.

Regarding the NFV Orchestrator, the following capabilities were implemented:

- VNFs connected using a Service Function Chain (SFC), described in a VNF Forwarding Graph Descriptor (VNFFGD) and implemented through chained named functions;
- VNF placement policy, ensuring optimal placement of VNFs.

VNF Forwarding Graph or VNFFG feature in DNFA is used to orchestrate and manage traffic through VNFs. In short, abstract VNFFG TOSCA definitions are rendered into Service Function Chains (SFCs) and Classifiers. The SFC makes up an ordered list of VNFs for traffic to traverse, while the classifier decides which traffic should go through them, following determined Forwarding Paths (FPs). Similar from how VNFs are described by VNFDs, VNFFGs are described by VNF Forwarding Graph Descriptors (VNFFGDs).

The VNFFGD describe the chain as well as the classifier that will eventually be created to form a path through a set of VNFs. Its properties include policy (traffic match policy to flow through the path) and path (chain of VNFs and Connection Points).

In an IP-based scenario, the criteria and path sections TOSCA definitions of a forwarding path for VNF placement looks something like the following:

```
description: Sample IP VNFFG template
topology_template:
  description: Sample IP VNFFG template
  node_templates:
    Forwarding_path1:
      type: tosca.nodes.nfv.FP.Tacker
      description: creates path (CP12-> CP12)
      properties:
        id: 51
        policy:
          type: ACL
          criteria:
            - network_src_port_id: 05754e35
            - network_id: 3d582b05
            - ip_proto: 6
            - destination_port_range: 80 - 80
            - ip_dst_prefix: 192.168.120.2/24
            - ip_src_prefix: 192.168.120.1/24
        path:
          - forwarder: VDU1
            capability: CP1
```

In a CCN-based scenario within DNFA, using DFNA proposed extended TOSCA NFV template profile, criteria are defined as in the following:

```
description: Sample DNFA VNFFG template
topology_template:
  description: Sample DNFA VNFFG template
  node_templates:
    Forwarding_path1:
      type: tosca.nodes.nfv.FP.DNFA
      description: creates path (CP12-> CP12)
      properties:
        id: 51
        policy:
          type: ACL
          criteria:
            - content: sensitive_content
            - relay_src: relayA
            - placement_hint: delegate_upstream
        path:
          - forwarder: VDU1
            capability: CP1
```

No need to define IP-related information in the DNFA TOSCA extended NFV profile: network id, network ports, IP protocol, destination port range, source/destination prefix, etc. The only information needed is the content (or contents) desired to traverse the VNF (or VNF chain) for the DNFA classifier to enforce this configuration through face initialization/configuration of the network nodes and an optional placement hint (delegate upstream, codedrag or computation push), that DNFA enforces in a best effort basis following the placement strategies defined in Section 2.3.1.

Delegate upstream avoids recomputing or refetching information that exists elsewhere in the net. Codedrag applies when information needs be produced, either because it never was computed before or is not timely available. Computation push covers a situation when some code or data is “pinned down” by either the owner or due to technical constraints. In this case the name resolution (execution) task is delegated (pushed) to the pinning site.

4.3.3.1 DNFA Service Function Chaining

SFC is an SDN feature that has recently become available in vanilla OpenStack, through OpenStack Neutron SDN SFC. OpenStack Tacker creates and manages OpenStack SFC objects in order to implement service chaining. However, DNFA performs service chaining in a totally different way, without Neutron SDN controller assistance (note the absence of a dependency ***Vi-Or*** between the NFVO and the VIM in FIGURE 4, in contrast with other MANOs proposals).

NFN extends the ICN principle of naming content to naming functions. Furthermore, NFN natively supports named function chaining (see Section 2.3.3). Within DNFA, every instance that provides the same network function is referred by the same name, e.g., $/DPI$, $/analyser$, etc. When the network policy defined in a VNFFG requires a flow to go through a sequence of VNFs, the DNFA NFVO will enforce the sequence of functions to be executed through face initialization/configuration of the network nodes. E.g., a chain $/DPI/analyser/content$ implies that DPI and traffic analyser must be applied to the content before it arrives to its destination.

There might be some functions in the network that need to maintain state. In such a case, all the packets of a flow should go through the same instance, even though they may not care which actual instance they might use. This implies that the different instances (VNFs) for the same network function cannot be treated equivalently. Stateful firewalls could be an example of this kind. DNFA adopts the hierarchical name in ICN to meet this requirement. Instead of using the same name, the multiple instances share a common prefix (function name), but they have function-level unique ID. E.g., two stateful firewalls instances may be called $/firewall/\_A/content$ and $/firewall/\_B/content$ respectively.

In those scenarios, while advertising the prefix, the NFVO advertise the whole name instead of the function name itself. If a packet can go through any of the instances for a function, it just puts the function name in the header (e.g., $/firewall/analyser/content$). Otherwise, it will use the full name (e.g., $/firewall/\_A/analyser/content$). CCN/NFN nodes perform the longest-prefix match, and therefore the packet can be forwarded to the required function instance, if specified.

For the functions that require visibility of the bidirectional packets of a flow, NFVO module can also specify the function instance via its full name and create a VNF list in the reverse order. E.g., if the firewall function requires packets from both directions, the policy layer can create a chain $/firewall/content$ for one direction and chain $/firewall/response$ for the other.

This way, DNFA is enhanced with a service-name layer (built over NFN) to decouple the location of a particular network function instance from the identity of the function it provides. Such a decoupling facilitates the dynamic modification of the functions needed by a flow on the controller, and can significantly improve the flexibility for network management, alleviating the steering of flows through the diferent network functions needed, before it is delivered to the destination. DNFA helps to reduce the amount of state stored in the network and results in better scalability compared to the per-flow state solutions like Neutron SDN SFC.

DNFA intrinsically supports the presence of multiple instances for the same functionality and performs application-layer load-balancing among these nodes at any time (CCN are inherently load balanced). The faster response will be considered. This allows the dynamic adoption of new function instances and can also help fast recovery when a function instance/link fails. However, the downside is that if not

properly coordinated, two VNFs could end spending the double of computational resources than really needed.

Some advantages of DNFA Service Chaining model are:

- Flexibility: When there is a need to change the functions a flow traverses, IP-based solutions have to rely on the controller to build a new path that goes through a certain instance of each of these functions. This results in extra control overhead in both communication and latency for every flow whenever the set of functions are changed. This is not desirable since the controller in SDN design is supposed to generate the rules but not be involved in the real-time handling of packets
- Scalability: SDN solutions place rules for every flow on the switches. The number of rules stored in the network is proportional to the number of flows, the functions the flows require and the size of the network. It is very difficult to scale when the network has larger number of flows or the network itself grows larger.
- Reliability: When a VNF or a link fails, the switches in the existing SDN solutions have to rely on the central controller to build a new path for the flow. This increases the convergence time while dealing with such failures. Alternatively, the SDN controller may setup backup paths proportional to the number of hops for every flow to ensure quick convergence time. But this exacerbates the scalability problem.

The downside of DFNA approach for service chaining is the need to interact with non VNF nodes (in particular content-requesting nodes) to enforce configuration through face initialization/configuration (see link ***Ve-Or*** in Figure 4, absent in its IP equivalent).

4.3.3.2 Orchestrator API

The northbound interface was implemented as a RESTfull Application Programming Interface (API) Service. This API could also give feedback of service usage statics to service owners to scale service requirements.

TABLE 3 NETWORK FUNCTIONS REST API ATTRIBUTES

Attribute	Type	Default Value	Required	CRUD
Id	uuid	generated	yes	R
name	string	none	no	CRU
description	string	none	no	CRU
tenant_id	uuid	none	yes	RU
named_function_id	uuid list	none	yes	R

TABLE 4 NAMED FUNCTIONS REST API ATTRIBUTES

Attribute	Type	Default Value	Required	CRUD
id	uuid	generated	yes	R
name	string	none	yes	CRU
description	string	none	no	CRU
placement	uuid list	none	no	RU

TABLE 3 shows the main attributes (type, default values, if they are mandatory attributes and the operations allowed over each attribute) of the RESTful API associated to the network functions. The *id* attribute is a unique identifier generated by the orchestrator upon creation of the network functions. *Name* and *description* are optional attributes that help characterize the network function in a human readable way. The *tenant_id* identifies the client within the system. The *named_function_id* attribute specifies the named function(s) in charge of implementing the network function in question, and its attributes are shown in TABLE 4 . As with network functions, the *id* attribute is automatically generated by the orchestrator and the *placement* attribute may be used to specify the premises where the named function will be deployed.

FIGURE 7 NETWORK FUNCTION JSON EXAMPLE REPRESENTATION

```
{
  "_updated": "Tue, 10 May 2016 21:28:48 GMT",
  "description": "A test Network Function",
  "tenant_id": "41192201ec37573250ed01d5",
  "named_function_id": "[573250ed01d541192201ec37]",
  "_links": {
    "self": {
      "href": "network_function/5732529001d541199314c06d",
      "title": "Network_function"
    },
    "collection": {
      "href": "network_function",
      "title": "network_function"
    },
    "parent": {
      "href": "/",
      "title": "home"
    }
  },
  "_created": "Tue, 10 May 2016 21:28:48 GMT",
  "_id": "5732529001d541199314c06d",
  "_etag": "ec2a0ee625b43565a6c0590fd83c66d92c68aae8",
  "name": "Test"
}
```

FIGURE 8 NAMED FUNCTION JSON EXAMPLE REPRESENTATION

```
{
  "_updated": "Tue, 10 May 2016 21:21:49 GMT",
  "placement": "[3, 4]",
  "name": "/bin/function",
  "_links": {
    "self": {
      "href": "named_function/573250ed01d541192201ec37",
      "title": "Named_function"
    },
    "collection": {
      "href": "named_function",
      "title": "named_function"
    },
    "parent": {
      "href": "/",
      "title": "home"
    }
  },
  "_created": "Tue, 10 May 2016 21:21:49 GMT",
  "_id": "573250ed01d541192201ec37",
  "_etag": "9263ae59b29cae0f6786f19133351f3b05303145",
  "description": "A test Named Function"
}
```

FIGURE 7 and FIGURE 8 show the implemented JSON representation of the presented tables. Note that in this example, the network function named *Test* is implemented through the named function named $bin/function$, linked by the *named_function_id* attribute of the network function´s JSON. In this case, only one

named function was necessary to implement the network service (that may not always be de case).

4.3.4 Service, NFN and Infrastructure Description

The information about network topology, available network resources and information about network and named functions are stored in this component, as a database.

FIGURE 9 NETWORK FUNCTION/NAMED FUNCTION DATA MODEL

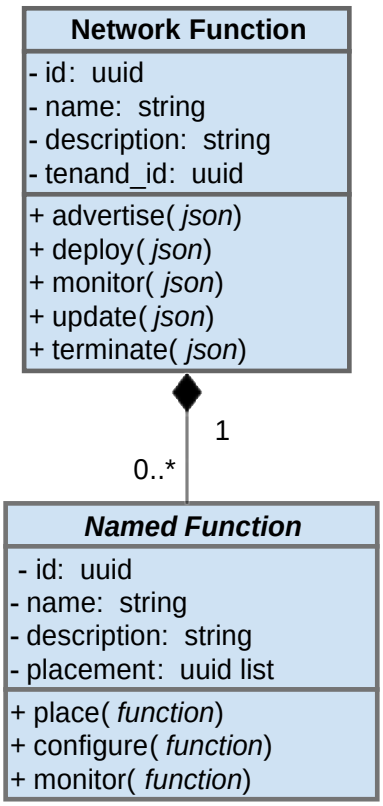


FIGURE 9 shows how network functions are modelled with respect to the relation with named functions. Note that various named functions may be needed to implement one functioning network function. This is achieved through a special syntax in the service chaining parameter $relay\_src$ of the DNFA extended TOSCA profile. For instance, $relay\_src: VNF1\&VNF2$ represents that the correspondent VNF

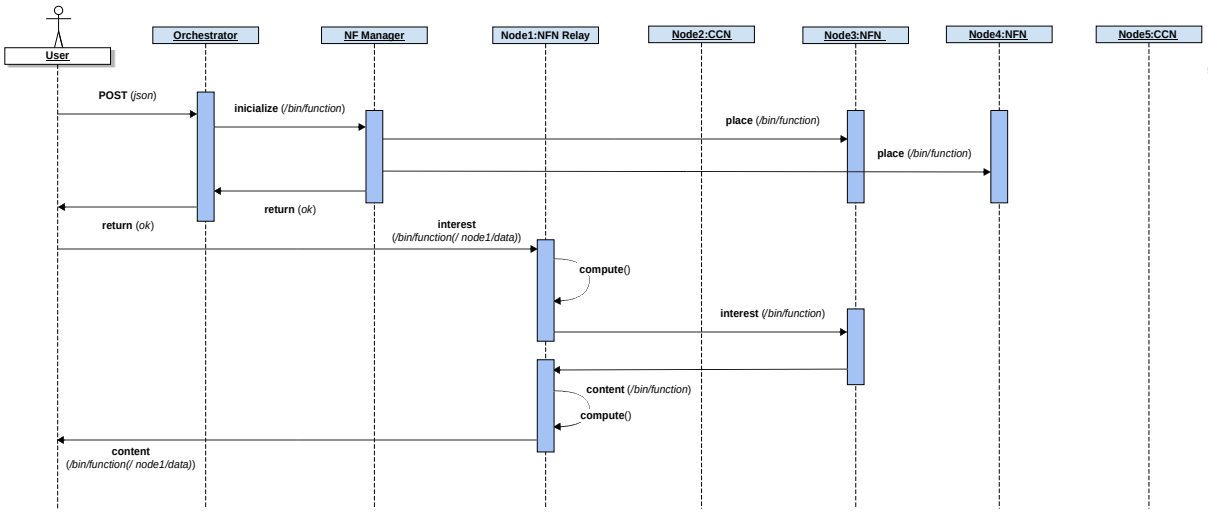
is implemented though two names functions VNF1 and VNF2 (that may be executed in parallel).

The shown network functions operations are implemented in the Orchestrator through a northbound REST API, and the named functions ones in the *Named Function Manager*, as stated before.

4.4 OPERATION

The whole process of a service deployment, named function registration and posterior execution can be seen in FIGURE 10, following the topology presented in FIGURE 6 (see Section 4.3.1).

FIGURE 10 EXAMPLE USE CASE



The process starts when a certain user (an ISP, for example) triggers the northbound REST API of the *Orchestrator* in order to deploy a new network service. A new network function is then deployed, following the JSON description of FIGURE 7, and then the *Orchestrator* triggers the *Named Function Manager* that creates two virtual machines to host the NFN execution environment (through VIM northbound API) and places the indicated named function following the JSON description of FIGURE 8 (see Section 4.3.3). Note that the *placement* attribute of the JSON named function description indicates the *Named Function Manager* where to place the indicated named function (in this case $bin/function$ has to be placed in *Nodes #3* and *#4*).

The execution of the named function corresponds to the *code drag* case for carrying out locally computations by first retrieving code and data (see Section 2.3.1). It starts by a user requesting the execution of $bin/function$ on $node\,1/data$. Following the default mapping of λ-expressions to CCN messages, the argument $node\,1/data$ becomes the first name component, which characterises *Node #1* as the recipient of the expression. When *Node #1* starts resolving the expression and the component $bin/function$ is encountered, it issues a second *Interest* to retrieve it, which is satisfied by *Node #3* (it could be satisfied by *Node #4* also). When *Node #1* receives the bytecode of the procedure it applies it to the locally available data and returns the function result in a content object to the client.

Note that the *User* that deploys the service and the *User* that executes the named function not necessarily has to be the same: e.g., an ISP could be in charge of the deployment of the service and a particular data center could be the final client to use the network service.

The following steps summarize the operation of the DNFA-MANO:

1. The DNFA-NFVO receives an instantiation request of an OSS.
2. The DNFA-NFVO sends a VNF instantiation request message to the DNFA-VNFM.
3. The DNFA-VNFM requests the VIM to allocate resources and establish a connection.
4. The VIM sends to the DNFA-VNFM a message confirming completion of the resource allocation.
5. The DNFA-VNFM performs faces configuration.
6. The DNFA-VNFM sends to the VNFM a message confirming completion of the resource allocation.
7. The DNFA-NFVO enforces the corresponding service chains into the NFN environment.
8. The DNFA-NFVO confirms, with the OSS, completion of the VNF instantiation.

4.5 DESIGNING THE NAME SPACE

The definition of the *name tree* is a key aspect of a CCN network because the *Content Name* field strongly influences the way *Interest* and *Data* packets are treated. Moreover, an optimized name space could give enormous advantages for routing operations and packet processing.

The definition of the name space is based on both contractable names and on-demand publishing concepts, firstly proposed in (JACOBSON et al., 2009). The former one supposes that each user is able to construct the name of a desired content through specific algorithms (e.g., it knows the structure of the *name tree* and the value that each field of the *Content Name* may assume). The latter one, instead, describes the ability to request a content that has not yet been published in the past, but it has to be created in answer to the received request.

Without loss of generality, it is possible to assume that a given service can be offered by a specific entity and that the same service can be managed by multiple providers, thus allowing the user to select one of them according to its preferences.

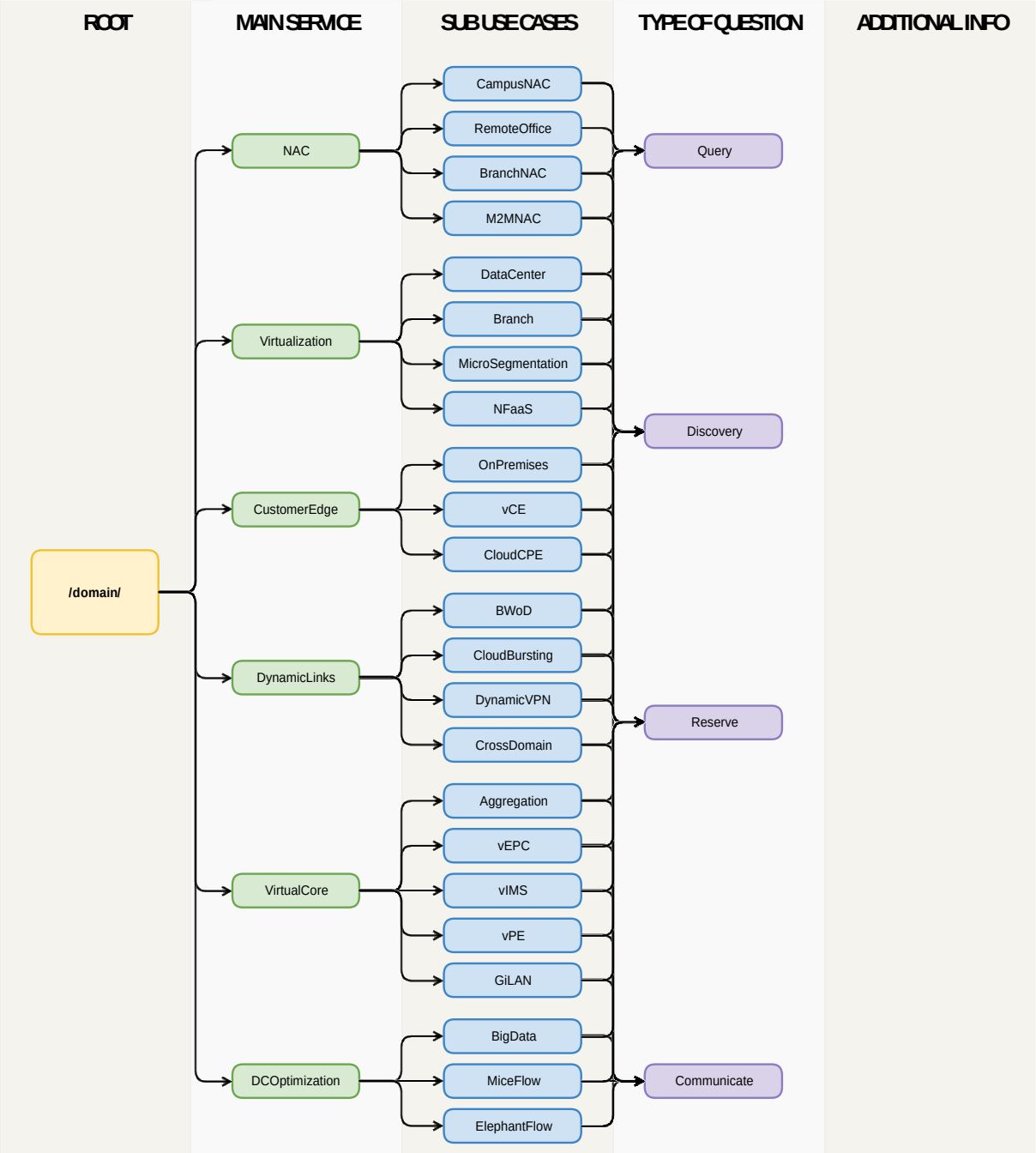
Therefore, the following *name tree* structure will be adopted hereafter:

$$/domain/main\_service/sub\_use\_case/type\_of\_question/additional\_info$$

Following the design introduced in (PIRO et al., 2014), starting from the root tree, which is identified with "/", it was introduced the *domain* field in order to explicitly indicate that a given service may be offered by a specific entity and within a specific coverage area. The *main-service* field specifies the macro category which the service belongs to. Then, the set of components listed in the *sub-use-case* field allows picking out a specific network service in a hierarchical structure of sub-categories. The *type-of-question* field explicit the user demand. Just to make few significant examples, it can be set equal to the value *discover*, *reserve*, *query* and *communicate*, in the case the user wants to discover a node offering a given network service, reserve a network service, interrogate a remote node, or communicate something to another device, respectively. Finally, components belonging to the *additional-info* field are used for appending to the *Content Name* specific optional values that can be exchanged among nodes during the negotiation and the execution of the network service itself.

Generally, the part of the *Content Name* before the *additional-info* field serves for routing purposes, i.e., for delivering the *Interest* packet to a node that is able to satisfy the user request. The *additional-info* field could provide further details useful for the publisher to elaborate and generate a more specific answer to be included into the *Data* packet.

FIGURE 11 EXAMPLE OF NAME TREE STRUCTURE



It has been pictured in FIGURE 11 an example of the *name tree* generated by adopting the proposed hierarchical structure, based in the taxonomy presented in ("Most Common SDN & NFV Use Cases Defined", 2012). Obviously, this *name tree* has to be considered as a possible starting point for conceiving more complete names space structures, which would encompass a very large number of network services that, probably, have not been yet considered, conceived nor envisaged. Thanks to its high flexibility, in fact, it is possible to extend the *name tree* by introducing new categories and sub-categories of network services, as well as different kind of questions and additional details to append at the end of the name.

4.6 ADITIONAL CONSIDERATIONS

4.6.1 Virtual Network Functions in Content-Centric Networks

Prior to implementing VNFs in CNNs, some considerations must be made regarding what kind of VNFs would make sense in a pure CCN scenario and the suitability of traditional IP-based VNFs in CCNs.

The macro use case that will be focused in this work is the *Customer Edge* one, because is the one that is generally being tackled first from a network function perspective. *Customer Edge* use case focuses on how to connect a remote office, or remote branch, to a central data center network, and extending the central data center's network services into that remote office. In order to deliver this connectivity, a Customer Premises Equipment (CPE) device is used. Generally, the types of device used would be a router, switch, gateway, firewall, etc. (these are all CPEs) providing anything from Layer 2 Quality of Service (QoS) services to Layer 7 intrusion detection. The Layers 2 and 7 references are from the Open Systems Interconnection (OSI) Model. vCPE denotes the virtual CPE. This particular use case has two forms, or variants: remote vCPE (or customer-premise deployed) and centralized vCPE (deployed within the data center).

For starters, some frequent IP-based VNFs do not make much sense in CCNs:

- Network Address Translation (NAT): No need for traditional IP NAT in CCNs. However, some form of Content Name Translation (CNT) - or

Function Name Translation (FNT) in the NFN case - could become handy in some occasions. For instance, lets picture a large and complex VNF. FNT could be used to divide the function into smaller functions and distribute them along the network.

- Domain Name System (DNS): DNS would be redundant, as all content in CCNs already has a label/name/identifier.
- Dynamic Host Configuration Protocol (DHCP): Addresses have no use in content networks, so address designation makes no sense. However, DHCP is used for more than address designation, as it delivers additional information to the hosts. This initial configuration propagation could be useful to initialize content nodes, or to propagate face or interest information.
- Load Balancer: CCN is inherently balanced. Each interest is transmitted through all the corresponding configured faces and the associated content can be delivered through any of those faces.
- Content Caching: CCN has native build-in support for content caching.

There are others IP-based VNFs that make all the sense in CCN networks, with some adjustments:

- Content Access: Content Access is of paramount importance in CCNs. As discussed in Section II, several works have already explored this subject (FOTIOU; MARIAS; POLYZOS, 2012) (MOHAISEN, 2017), including an implementation trough the NFN paradigm (MARXER; SCHERB; TSCHUDIN, 2016).
- Firewall: Classic L3 firewalls do not make much sense in CCNs. Interests are propagated in some specific and configured faces. If a face is not desired to be used for receiving content, just avoiding the propagation of interest through it would be enough to guarantee this. However, maybe it is needed all root content except one specific sub content. In this case would make sense to implement a named-based firewall, filtering according to content name. In the other hand, content-based firewalls

would be possible, somehow like IP Web Application Firewalls (WAFs), where the filtering is made according to the content of the packets.

- Deep Packet Inspection: For collection of statistical information purposes, avoiding out any non-compliance to protocol, spam, viruses, intrusions, and any other defined criteria to block the packet from passing through the inspection point.
- Traffic analyser: Alerts would need a push paradigm. Several works have already explored the implementation of publish-subscribe capabilities in CCNs (QUEVEDO et al., 2018).
- Content classifier and marker: Classify content flowing through according to a particular policy and either select them for special treatment or mark them, in particular for differentiated services and Quality of Service (QoS) applications.
- Anonymiser: Anonymiser functions can be implemented in various ways to hide information of the data sender.
- Route propagator: The FIBs of CCN nodes are usually manually initialized, in absence of a dynamic routing standard currently for CCN.

4.6.2 Placing and distributing Named Functions

Nowadays, most network functions are resided at certain locations in the network, each responsible for a static portion of the flow space. With the ever-growing network scale and traffic volume, these centralized network functions have become the performance bottleneck. Moreover, steering traffic to them incurs non-trivial overhead. There already exist in the literature some documented cases of distributed network functions successfully implemented: distributed monitoring (SEKAR et al., 2008), distributed redundant elimination (ANAND; SEKAR; AKELLA, 2009) and distributed intrusion detection / prevention systems (SEKAR et al., 2010).

The demand for such a distributed infrastructure relates to the requirements imposed by different network functions. For example, there are certain functions that need to be in a certain geographical location, while others do not. As an example, certain CPE functions like the set top boxes should be closer to the costumer site, in Points of Presence (PoPs), or even within the site (see previous Section). Web

proxies would better to be closed to the clients in order to optimize user experience and reduce bandwidth consumption. On the other hand, some network functions that encrypt the traffic may inflate the bandwidth utilization and are suggested to be placed near the destination. These placement considerations are especially important in the Internet, where location dependency impacts the performance significantly and Internet Service Providers (ISPs) have started deploying micro-datacenters.

Named functions are inherently distributed. But considerations about where the code will be executed have to be taken and should be implemented through DNFA placement hints (described in Section 4.3.3). How network functions are placed directly determines some important metrics, such as operating cost and end-to-end latency. To achieve the benefits from NFN ability to conveniently place named functions anywhere anytime, a number of network functions placement strategies were studied for different network functions orchestration frameworks. Apart from centralized network functions, some other forms of network functions exist towards different objectives, and their placement is according quite different (LI; QIAN, 2016).

DNFA has the ability to discover places where computations can take place. But in this process, it is also possible for clients to “give hints” to NFN for preferential placement of computations, through placement hints. This must be exploited to properly optimize network functions, such that preferences for the distribution of computations are expressed (but they cannot be enforced) for individual sub-expressions (in the program), thus achieving preferential opportunism in the NFN distribution of computations.

This preferential opportunism may be used to gain gratuitous parallelism and asynchronous computations with *thunks* (see Section 2.3.3). (SIFALAKIS et al., 2014) managed to reduce the execution time of some experiments by half by properly exploiting this effect.

4.6.3 Performance concerns

(SIFALAKIS et al., 2014) performed a first analysis of the performance overhead of NFN-over-CCN. Although the mentioned experiments are only indicative (given its non-quantitative character), nevertheless two standing out observations

were made. The first is that NFN processing is definitely affordable in the time scales of operation and decisions in CCN. Even with a very crude prototype implementation all NFN related operations across up to 3-4 nodes took only a fraction of the typical CCN *Interest* timeout; this confirms the lightweight type of operation it adds to the ICN forwarding plane.

The second observation regards the overhead that the current CCN architecture inflicts on NFN, witnessed in the effects of timeouts (as an implicit feedback mechanism) on the total completion time of NFN client requests. The agility of NFN in making decisions is seriously limited by the absence of fast feedback after probing for computations that are infeasible or for content/code that cannot be retrieved. The effect is more pronounced in NFN than in CCN only, since a typical NFN request entails several CCN level transactions.

Moreover, as stated early, named functions may be implemented so they can take advantage of preferential opportunism in NFN distribution of computations and gratuitous parallelism through *thunks*. Software network functions are expected to be implemented with the same level of performance as hardware-based realizations. Some experiments performed by (SIFALAKIS et al., 2014) took an execution time in the order of seconds. This is unacceptable for some data-path intensive functions, and so some performance optimizations must be taken into account (data path offload, etc.).

4.6.4 Security concerns

Since active networking research is well known the danger of potentially uncontrolled use of computing and forwarding resources. The security implications of having the possibility to spawn arbitrary computations from the middle of the network through malicious programs masqueraded as normal requests (even by accident) must be considered. In this case the obvious problem are divergent or looping programs.

Another security concern analogous to active network research is dependability: injection of malicious code, preventing insertion of or cleaning up caches from bogus results, trusting that hosts faithfully execute the provided code or do not leak private data, are but a few obvious security problems.

According to (TSCHUDIN; SIFALAKIS, 2014), these NFN security-related issues are heavily mitigated or even vanished e.g. in private clouds (pointing out to seek policy-related solutions as by routing policies).

Although NFN was architected with security as a prime concern, security-implications were taken into consideration. The principles of caching and re-using computation results, and the removal of *locality-of-execution*, are two top-listed features of NFN, which can minimize the effectiveness of Denial of Service (DoS) attacks towards specific targets in the network (more than what is actually possible in today's Internet).

Call-by-name expression resolution warrants that a request for evaluating a sub-expression will be dispatched in the network only if the result of that sub-expression will actually be used, making the plausibility of an attack not deterministically discernible.

Thunks also have the potential to protect against waste of computation resources: the orchestrating NFN node can ensure that *thunked* computations are spawned only if their enclosing expression is feasible and allowed.

All these features, although not securing the network, they nevertheless intend to limit or localise the effects of a possible attack and make it difficult to plan against specific targets.

Additionally, access control to functions can be protected by similar means as in today´s Service Oriented Architectures (SoAs). Data cannot be altered in NFN, only new data can be generated from other data and while older fade-away from caches. Every entity that generates new data must sign them according to data authorship rules in CCN, and when source data are transformed by some function, chained signing can be used to assert and verify the function owner, its inputs and its outputs (independently of its location).

In resume, it is important to note that: (a) distribution of computation tasks in the network does not entail forwarding state or cache state alterations (ephemeral content may appear in a cache as a by-product, which in absence of popularity will be eventually erased); and (b) computations may or may not take place, leaving the computation placement and resource allocation decision entirely to the network, thus avoiding single points of failure, compensating for routing failures, and partly protecting against DoS attacks targeting a specific host or service.

4.7 CONCLUSION

This chapter described conceptually the functional architecture proposed by DNFA, that allows placing network functions and services in the network by leveraging the NFN layer and without the need of a central SDN controller to perform Service Chaining.

The main operations and example use cases of the DNFA architecture, inspired by the ETSI NFV framework, were discussed.

Also, additional placing and distribution, performance and security concerns were considered.

4 MINI-NFV

5 MINI-NFV

As part of this work, Mininet was extended to support NFV Orchestration and VNF Manager capabilities, in order to aid DNFA prototyping and experimentation. This chapter will present mini-nfv, a framework for NFV orchestration on top of Mininet.

Mininet (LANTZ; HELLER; MCKEOWN, 2010) is a network emulator, which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow (MCKEOWN et al., 2008) for highly flexible custom routing and Software-Defined Networking experimentation.

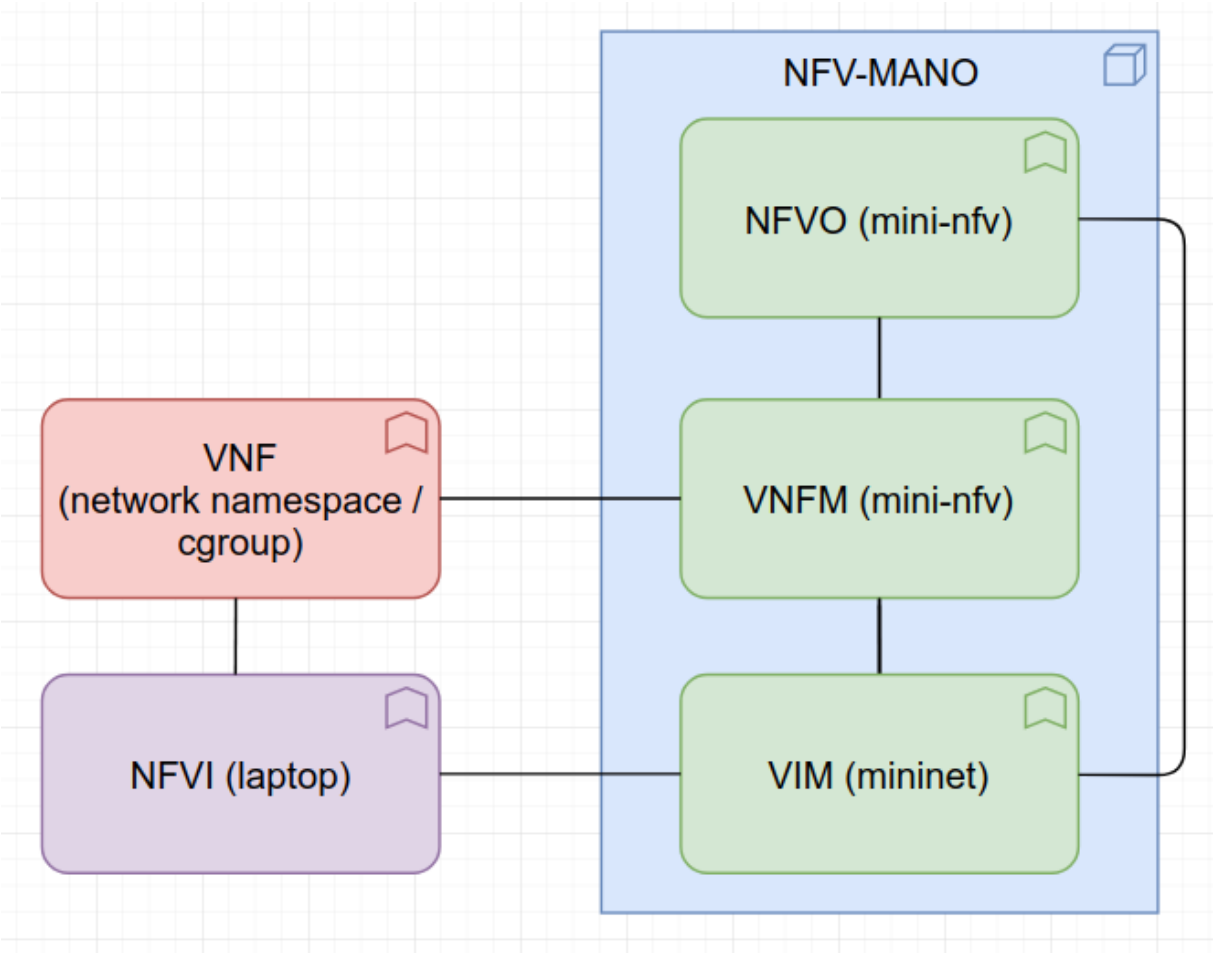
Mininet has shown itself as an excellent tool for agile network/SDN/NFV experimentation. Various NFV related works make use of Mininet for prototyping, testing, and implementation. However, to the best of our knowledge, neither opted for TOSCA or ETSI as a standard way to define a data model that can be used for creating templates or data descriptions of applications and infrastructure for cloud services.

For this Thesis, mini-nfv was developed, a framework for NFV Orchestration with a general purpose VNF Manager to deploy and operate virtual Network Functions and Network Services on Mininet. Its goal is to alleviate the developers' tedious task of setting up a whole service chaining environment or complex infrastructure environments (i.e., OpenStack) and let them focus on their work (e.g., developing a particular VNF, prototyping, implementing an orchestration algorithm or a customized traffic steering). By using mini-nfv, and its parametrized OASIS TOSCA or ETSI NVF profiles it was also intended to bring some level of standardization, reproducibility, and replicability to the NFV experimentation world.

5.1 FUNCTIONAL ARCHITECTURE

Mini-nfv is also based on the ETSI MANO Architectural Framework. FIGURE 12 shows a basic overview of the proposed architecture within the ETSI MANO Architectural Framework.

FIGURE 12 OVERALL ARCHITECTURE WITHIN THE ETSI MANO ARCHITECTURAL FRAMEWORK



Mini-nfv is responsible for managing the virtualized infrastructure of an NFV-based solution (through Mininet's API). Its operations include:

- Keeping an inventory of the allocation of virtual resources to physical resources. This allows mini-nfv to orchestrate the allocation, upgrade, release, and reclamation of NFVI resources and optimize their use.
- Supporting the management of VNF forwarding graphs by organizing virtual links, networks, subnets, and ports.

More specifically, the following capabilities are supported:

- Regarding the NFV catalog:
  - VNFDs
  - VNFFGD

  - NSD

- Regarding the VNF Manager:
  - Basic lifecycle of VNFs (create/update/delete/scale)
  - Initial configuration of VNFs

- Regarding the NFV Orchestrator:
  - Templatized end-to-end NSDs using decomposed VNFs.
  - VNF placement policy, ensuring optimal placement of VNFs.
  - VNFs connected using a Service Function Chain (SFC), described in a VNFFGD.
  - Symmetrical and asymmetrical traffic from and to the VNFs (currently Tacker/OpenStack Virtualized Infrastructure Manager (VIM) driver only support asymmetrical unidirectional traffic).
  - A new mirroring mode, designed for monitoring-alike VNFs, where the traffic is mirrored and not rerouted.

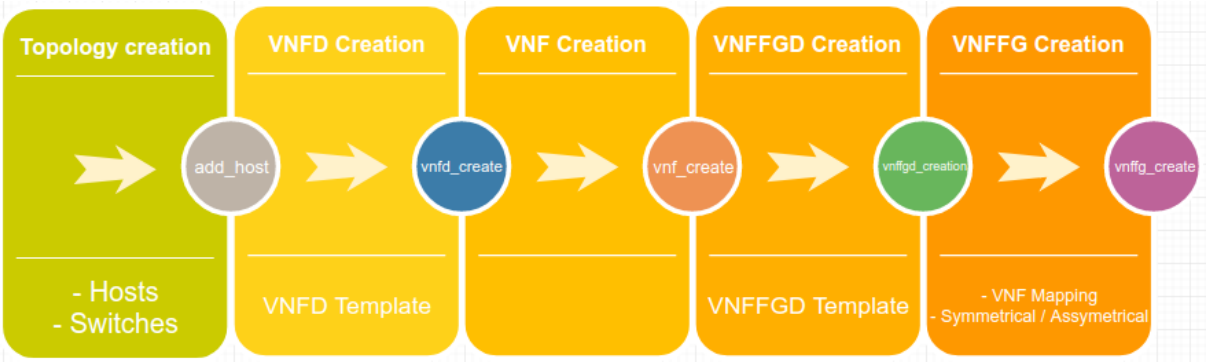
On top of that, mini-nfv supports:

- Parametrized templates, to automate the scale-out of VNF deployments and NFV orchestration graphs within TOSCA templates.
- Network definition via virtual Links description.
- IP/MAC definition via Connection Points description.
- Emulation of numbers of CPUs and flavor properties through Mininet's $CPULimitedHost$ class.
- Initial configuration on the VNFs through cloud-init scripts (“cloud-init - The standard for customising cloud instances”,).
- Network slicing experimentation through an extended TOSCA Virtual Link definition implemented by Mininet's $TCIntf$ class.

FIGURE 13 depicts a standard flow for VNF instantiation. Steps 2 (VNFD creation) and 4 (VNFFGD creation) are optional as it is possible to deploy VNFs and

VNFFGs passing directly the template as a parameter through direct VNF instantiation (further information in Section IV-D).

FIGURE 13 WORKFLOW FOR VNF DEPLOYMENT IN MINI-NFV



Mini-nfv can be used with or without Mininet's default controller. The first way can be useful to test the VNF Manager functionality with full connectivity between VNFs hosts and NFV Orchestration capabilities without the need of running POX (REPO, 2019), Ryu (“Ryu SDN Framework”,) or other SDN controllers. However, if needed, mini-nfv also supports the integration of a third party SDN controller into the experimentation environment.

Mini-nfv was purposely built on top of Mininet, and not as a fork. This way mini-nfv installation and upgrade are simpler, and automatically it evolves itself alongside with Mininet development.

Mini-nfv is free software, publicly available (CASTILLO, 2018a) and distributed under the terms of the GNU General Public License version 2.

TOSCA Simple Profiles for NFV (“TOSCA Simple Profile for Network Functions Virtualization (NFV)—Version 1.0”,) are currently used to define Network Service Descriptors (NSD), VNF Descriptors (VNFD) and VNF Forwarding Graph Descriptors (VNFFGD) using TOSCA templates. These templates are then parsed and loaded into an active Mininet session, through the mini-nfv command line interface (CLI).

Follows a TOSCA basic packet counter VNFD template:

```
tosca_definitions_version:
  tosca_simple_profile_for_nfv_1_0_0
description: Packet count with user_data
metadata:
  template name: sample_VNFd_userdata
```

```
topology_template:
node_templates:
  VDU1:
    type: tosca.nodes.nfv.VDU.MiniNFV
    capabilities:
      nfv compute:
        properties:
          num cpus: 1
          mem_size: 512 MB
          disk_size: 1 GB
    properties:
      user_data_format: RAW
      user_data: |
        #!/bin/sh
        FILE = $(ifconfig | head -n 1 | cut -f 1 -d '')
        mkdir exp
        watch -n 60 "date >> exp/$FILE;
          ifconfig | grep eth0 -C 5 | e g r e p 'RXjTX' > exp/$FILE" &
  CP1:
    type: tosca.nodes.nfv.CP.MiniNFV
    properties:
      management: true
      order: 0
      anti_spoofing_protection: false
    requirements:
      - virtualLink:
          node: VL1
      - virtualBinding:
          node: VDU1
  VL1:
    type: tosca.nodes.nfv.VL
    properties:
      network name: net mgmt
      bandwidth: 512K
```

- Virtual Deployment Unit (VDU): Is the Mininet *namespace* and *cgroup* that hosts the network function. Mini-nfv supports emulation of number of vCPUs and flavor properties trough Mininets *CPULimitedHost*, and user data, custom commands to be run on VDU once it is spawned (cloud-init scripts). This could be useful to perform an initial configuration on the VNF.
- Connection Point (CP): Mininfv supports IP and MAC definition via CP.

Mini-nfv also supports network slicing experimentation through an extended TOSCA NFV specification. There are two ways to accomplish this:

- At a VL level: an extended VL definition implemented by Mininet's *TCIntf* class, interfaces customized by traffic control (tc) utility that allows

specification of bandwidth limits as well as delay, loss and max queue length. This approach is the less invasive way concerning standard TOSCA NFV profile:

$tosca.nodes.nfv.VL$:
 $derived\_from$: $tosca.nodes.network.Network$
 $properties$:
  $bandwidth$:
   $type$: $string$
   $required$: $false$
   $description$: $link\ bandwidth\ (ex. 256K)$
  $delay$:
   $type$: $string$
   $required$: $false$
   $description$: $link\ delay\ (ex. 5ms)$
  $loss$:
   $type$: $string$
   $required$: $false$
   $description$: $loss\ percentage\ (ex. 5\%)$
 $capabilities$:
  $virtual\_linkable$:
   $type$: $tosca.capabilities.nfv.VirtualLinkable$

- A new high-level network slice constructor $tosca.nodes.nfv.NS$. This way, all root definitions (VDU, VL, CP, FP, ...) could optionally inherit or become part of a certain network slice previously defined, and adaptations or reconfigurations would be accomplished at a per slice level, rather than per individual virtual link.

For instance, VNFD defined in the shown VNFD template is accommodated in a 512K network slice specified through an extended VL definition.

5.2 ETSI NFV TEMPLATE SUPPORT

Besides OASIS TOSCA NFV profile template, mini-nfv supports ETSI NFV template format (nfv-sol-libs: ETSI GS NFV SOL001 v2.5.1, 2019) as well. This support is delivered through $etsi2tosca$, a conversor explicitly developed for this project and publicly available in (CASTILLO, 2018b, p. 2). As far as the author or the Thesis knows, there are no other converters between these data models. For instance, OPNFV Parser Project (“Parser - Parser - OPNFV Wiki”,) is limited to Netconf/YANG to TOSCA conversion.

5.3 VNF MANAGEMENT

VNFs, VNFDs, VNFFGs, and VNFFGDs are created, listed, shown, updated and deleted from the mini-nfv command-line interface (CLI). From the CLI it is also possible to insert hosts and switches, along with troubleshooting and debugging.

The syntax and workflow used for implementing VNFs and VNFFGs in mini-nfv respects OpenStack Tacker as much as possible.

Main commands regarding VNF management implemented include:

- Onboarding VNF: TOSCA VNFD templates can be onboarded onto mini-nfv active session catalog using the following command: $VNFD\_create\ -VNFD-file\ <yaml\ file\ path><VNFD-NAME>$
- Deploying VNF using mini-nfv catalog: In this method, a TOSCA VNFD template is first onboarded into mini-nfv VNFD catalog. This VNFD is then used to instantiate VNFs. This is the most common way of creating VNFs in mini-nfv.
    - Onboard a TOSCA VNFD template:
      $VNFD\_create\ -VNFD-file\ <yaml\ file\ path><VNFD-NAME>$
    - Create a VNF:
      $VNF\_create\ -VNFD-name\ <VNFD-FILE-NAME><VNF-NAME>$
- Direct VNF Instantiation: In this method, the VNF is directly created from the TOSCA template without onboarding the template into mini-nfv VNFD Catalog: $VNF\_create\ -VNFd-template\ <VNFD-FILE-NAME><VNF-NAME>$
- Finding VNFM status: Status of various VNFM resources can be checked using the following commands:
  $VNFD\_list$
  VNF_list
  $VNF\_show\ <VNF-NAME>$
  $VNFD\_show\ <VNFD-NAME>$
  $VNFD\_template\_show\ <VNFD-NAME>$
- Scaling VNF: Horizontally scale (scale in/out) the number of VNF instances: $VNF\_scale\ <VNF-NAME{:}replica-number>$

- Deleting VNFs and VNFDs: VNFs and VNFDs can be deleted using the following commands:

  $VNF\_delete\ < VNF - NAME >$

  $VNFD\_delete\ < VNFD - NAME >$

5.4 NFV ORCHESTRATION MANAGEMENT

VNFFG can be instantiated from VNFFGD or directly from VNFFGD template by separate mini-nfv commands. This action will build the chain and classifier necessary to realize the VNFFG.

- Creating the VNFFGD:

  $VNFFGD\_create\ - VNFFGD - file\ < yaml\ file\ path >< VNFFGD - NAME >$

- Creating the VNFFG: To create a VNFFG, VNF instances of the same VNFD types listed in the VNFFGD must be created first. The defined VNFD must include the same Connection Point definitions as the ones declared in the VNFFGD template. Failure to do so will result in an error when trying to create a VNFFG.
  - To create VNFFG from VNFFGD:

    $VNFFG_{create} - VNFFGD - name\ < VNFFGD - name$

    $> -VNF - mapping < VNF - mapping$

    $> -symmetrical\ < boolean >< VNFfg - name >$
  - To create VNFFG directly from VNFFGD template without iniciating VNFFGD:

    $VNFFG\_create\ - -VNFFGD - template\ < VNFFGD - template$

    $> - - VNF - mapping\ < VNF - mapping$

    $> - - symmetrical\ < boolean >< VNFFG - name >$
- Status of various NFVO resources can be checked using the following commands:

  $VNFFG\_list$

  $VNFFGD\_list$

  $VNFFG\_show\ < VNFFG - NAME >$

  $VNFFGD\_show\ < VNFFGD - NAME >$

  $VNFFGD\_template\_show\ < VNFFGD - NAME >$

- VNFFGs and VNFFGDs can be deleted using the following commands:
  $VNFFG\_delete\ <VNFFG-NAME>$
  $VNFFGD\_delete\ <VNFFGD-NAME>$

VNF mapping allows a list of logical VNFD to VNF instance mapping. It is used to declare which specific VNF instance to be used for each VNF in the Forwarding Path.

The symmetrical argument is used to indicate if reverse traffic should also flow through the path. This creates an additional classifier to ensure return traffic flows through the chain in a reverse path. Otherwise, this traffic would be routed normally and does not enter the VNFFG. Different from OpenStack Tacker, mini-nfv supports both symmetrical and asymmetrical types.

5.5 TOPOLOGY CUSTOMIZATION

Mini-nfv allows to create dynamic topologies by adding hosts and switches into an active mini-nfv session through the add_host command:

$add\_host\ <HOSTNAME>\ [<IP1/masc>\ldots]$

Complex topologies can be stored into plain text files and loaded into an active mini-nfv session through the Mininet's $source$ command, that reads commands from an input file:

$source\ <file>$

5.6 PARAMETRIZED TEMPLATES

To allow the scale-out of VNF deployments and NFV orchestration graphs within TOSCA templates, mini-nfv supports Jinja (“Jinja2 Documentation (2.10)”,), a full-featured and designer-friendly template engine for Python, with an integrated sandboxed execution environment.

This way, inside the TOSCA templates, variables or expressions can be defined, as in for example:

ip_dst_prefix: ‘{{ ip_dst }}’

It is also possible to define tags, which control the logic of the template:

$\{\%\ ip\_masq\ ==\ 24\ \%\}$

Variables will then be replaced with values when the templates are rendered. This can be done interactively through mini-nfv CLI, as in:

$px\ import\ yaml;\ net.values = yaml.load('---nnip\_dst: 10.0.40.10/24')$

This way, using only one parametrized template it is possible to deploy any number of VNFs/VNFDs/VNFFGs with varying parameters.

5.7 EXPERIMENTS

Mini-nfv was built on top of Mininet, and it makes use of its emulated links, hosts, switches and controllers. VNFs are implemented in hosts (isolated network namespaces), and VNFFGs are implemented through sets of virtual Ethernet pairs (virtual wires connecting two virtual interfaces). Mini-nfv uses Mininet's $CPULimitedHost$ class to achieve emulation of numbers of vCPUs and flavor properties and Mininet's $TCIntf$ class to implement network slicing experimentation capabilities.

In this way, mini-nfv scalability is coherent with Mininet's scalability itself (LAI; FU; MOORS, 2015). Lightweight virtualization is the key to scaling to hundreds or even thousands of nodes while preserving interactive performance. In this section, it is measured memory consumption, overall topology creation, and clean-up time stamps and delay, considering VNFD deployment templates defined in Section 5.3.

FIGURE 14 shows the memory consumption of a varying number of VNFs and VNFFGs topologies. It is observed a roughly 1 MB overhead for each VNF, almost two orders of magnitude less than a virtual machine would require.

FIGURE 14 MEMORY CONSUMPTION

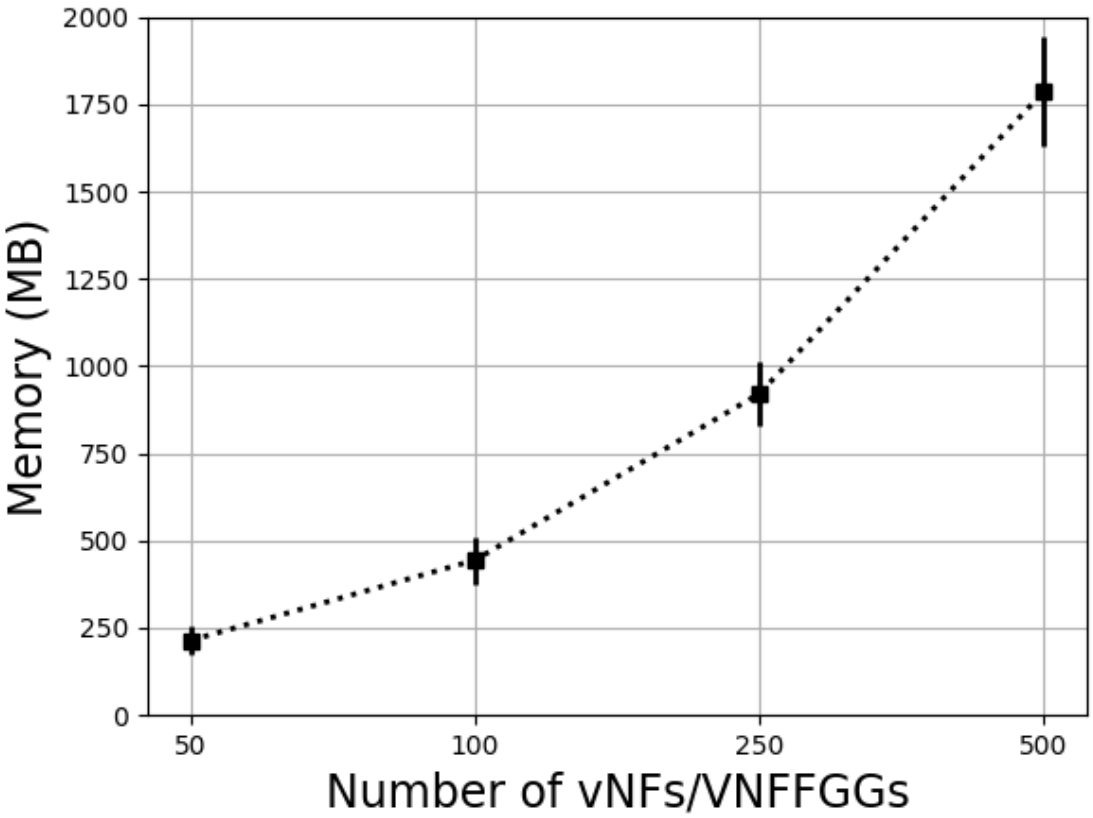


FIGURE 15 registers the time required to create and destroy a variety of topologies with mini-nfv. Creating such topologies with hundreds or thousands of nodes in OpenStack (or other cloud environment) will require several hours, and a lot of cloud resources. Even more, an individualized template for each VNF/VNFFG will be needed, as OpenStack Tacker does not support parametrized templates.

FIGURE 15 CREATION/TERMINATION TIMES

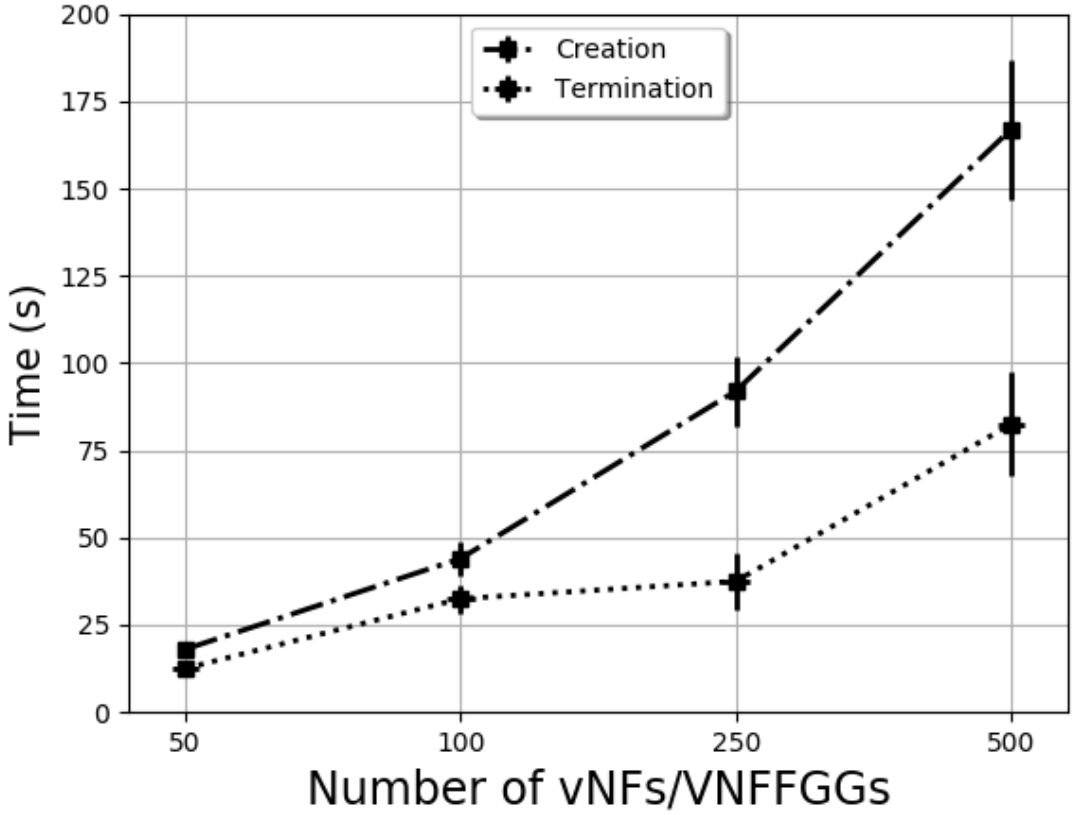


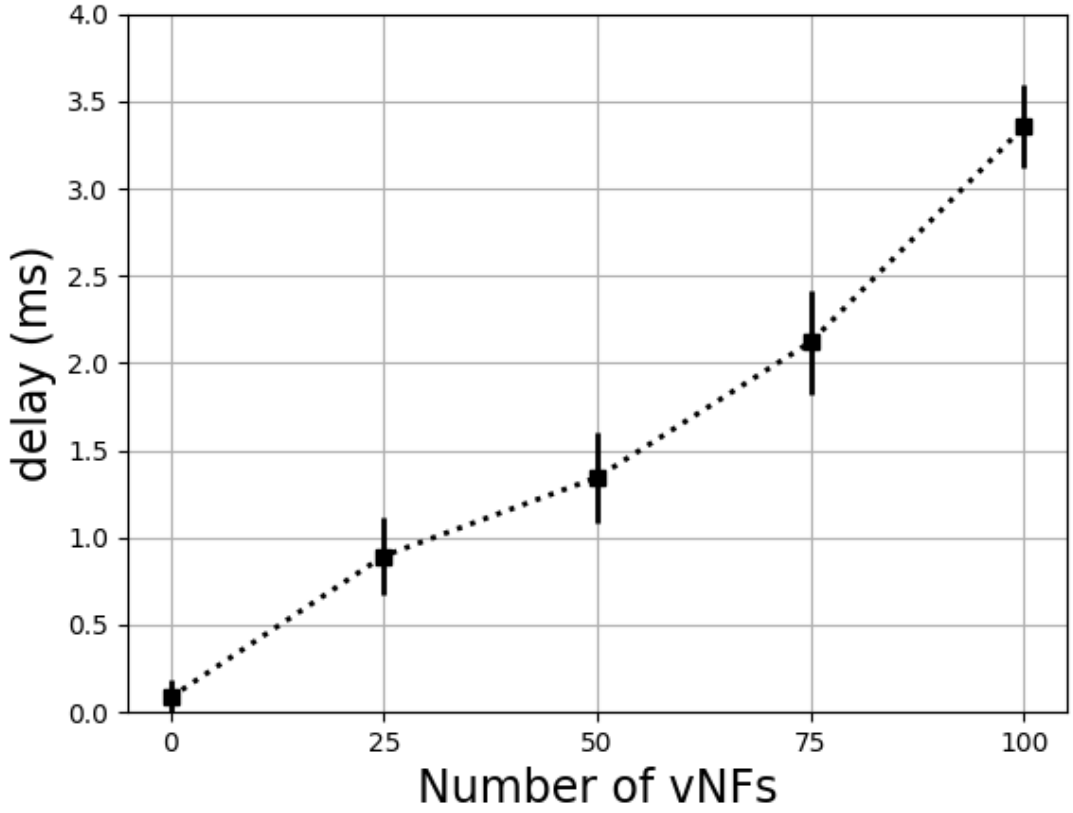
TABLE 5 consolidates memory consumption, creation/destruction times and number of network elements created by the experiment.

TABLE 5 MINI-NFV TOPOLOGY BENCHMARKS

VNFs	VNFFGs	Hosts	Links	Mem (MB)	Start (s)	Stop (s)
50	50	100	150	214	18	12
100	100	200	300	444	44	32
250	250	500	750	921	92	37
500	500	1000	1500	1787	167	82

FIGURE 16 illustrates the delay of traffic through a varying number of chained VNFs. As expected, it is perceived a linear increment of the delay as the number of VNFs grows.

FIGURE 16 DELAY VS NUMBER OF VNFS



Mini-nfv scales to the large topologies shown (over 1000 VNFs) because Mininet virtualizes less and shares more. The file system, user ID space, process ID space, kernel, device drivers, shared libraries and other common code are shared between processes and managed by the operating system. The roughly 1 MB overhead for a host is the memory cost of a shell process and small network namespace state; this total is almost two orders of magnitude less than the 70 MB required per host for the memory image and translation state of a lean virtual machine, thus allowing bigger topologies.

5.8 CONCLUSION

This chapter introduced Mini-nfv, an NFV orchestration/manager framework on top of Mininet.

6 IMPLEMENTATION AND EVALUATION

6 IMPLEMENTATION AND EVALUATION

This chapter will describe the methodology used for the implementation of the different aspects of the architecture (framework, infrastructure support and use cases) and the results of the performance evaluation of the DNFA framework, analysed at on a real testbed scenario, a private OpenStack cloud.

6.1 IMPLEMENTATION

6.1.1 DNFA Framework

In order to implement the reference architecture, CCN-lite (“CCN-lite Project”,) was chosen. CCN-lite is a reduced and lightweight (yet functionally interoperable) implementation of the CCN protocols. It supports named functions for letting clients express results instead of accessing only raw data. It admits Java, Scala or Python to host function execution and to interface to an NFN network. CCN-lite has been included in the RIOT operating system (“RIOT operating system for IoT”,) for the Internet of Things.

At the Service Layer, Eve and MongoDB were chosen to implement the REST API. Eve (“eve Pyhton REST API Framework”,) is an open source Python REST API framework that allows to build and deploy highly customizable, fully featured RESTful Web Services. MongoDB (“MongoDB”,) is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB avoids the traditional table-based relational database structure in favour of JSON-like documents with dynamic schemas.

6.1.2 Use Cases

CCN-lite admits Java, Scala or Python to host function execution and to interface to a NFN network. In this work Python was chosen.

As use case to stress the DNFA framework, Deep Packet Inspection (DPI) was chosen. The vDPI VNF calculates the hash function of the content of the packet, weeding out any non-compliance to protocol, spam, viruses, intrusions, and any

other defined criteria to block the packet from passing through the inspection point. The vDPI has a sequential behavior implemented through a sequential task queue.

For the implementation and deploy of the IP-based VNF, the following combination has been used:

- OpenStack Tacker, the project responsible for implementing a generic VNFM and NFVO in OpenStack clouds. At the input consumes TOSCA-based templates, converts them to Heat Orchestration Templates (HOTs) which are then used to spin up the virtual network functions on OpenStack.
- Scapy (“Scapy”,), a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more.

Each execution of the vDPI takes roughly 30 *ms* in the given flavor with 4 vCPUs (2 socket/2 NUMA node configuration) and 8 GB of memory in the IP implementation. DNFA implementation takes between 40-50 ms. This difference in execution times comes from the overhead of the CCN-lite python NFN execution environment, only intended for testing purposes and not for production use.

APPENDIX B lists the templates used for the IP vDPI Implementation trough OpenStack Tacker and APPENDIX C lists their equivalent for the DNFA implementation.

The implemented use case was compared in the next section to its IP VNF equivalent, in terms of network performance, scalability and flexibility.

6.2 EVALUATION

This section will present the results of the performance evaluation of the DNFA framework, analysed at on a real testbed scenario, a private OpenStack cloud through the vDPI VNF implementation described in the previous section.

6.2.1 Topology

FIGURE 17 depicts the virtual topology spanned up in OpenStack. Relays D contain the desired contents, while relays R act as requisitioners. Relays D and R are placed in different subnets.

A virtual machine containing the DNFA Orchestrator and Manager was created in the management network. It needs to access the Internet in order to communicate with the VIM (OpenStack public northbound API). It was responsible for spinning up two relays P, that have been implemented both as an IP-based vDPI VNF and a DNFA vDPI function. They work in an active/active high availability setup. Virtual machines hosting relays P were configured within a *flavor* with 4 vCPUs (2 socket / 2 NUMA node configuration) and 8 GB of memory (complete definition in APPENDIX B and APPENDIX C).

Note that relays P must be part of the management network, for the DNFA MANO to initialize the FIBs of the vDPIs. Relays R must be part of the management network as well, for the DNFA classifier to enforce service chaining configuration through face initialization/configuration of the network nodes.

FIGURE 17 OPENSTACK VIRTUAL INFRASTRUCTURE TOPOLOGY

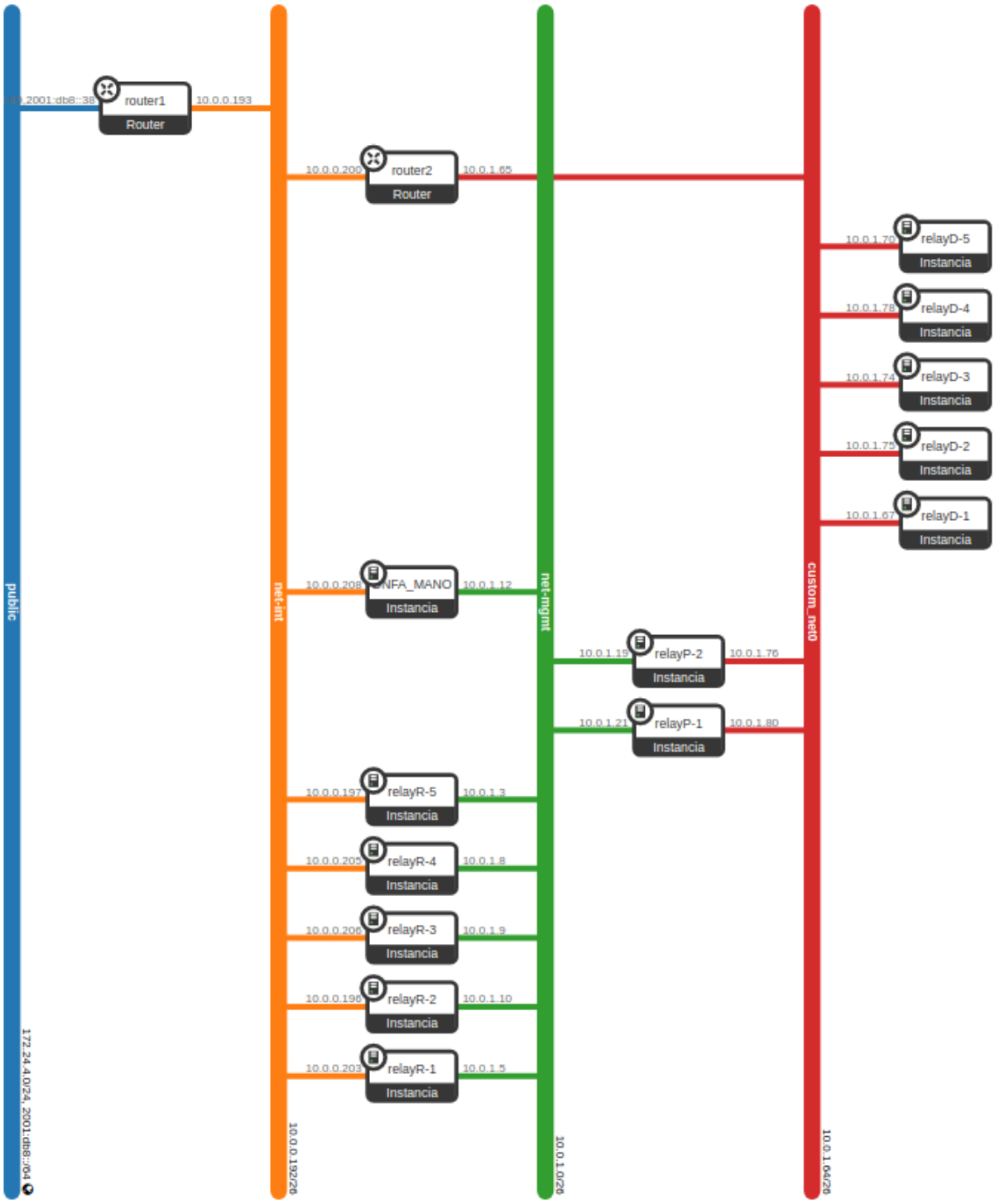
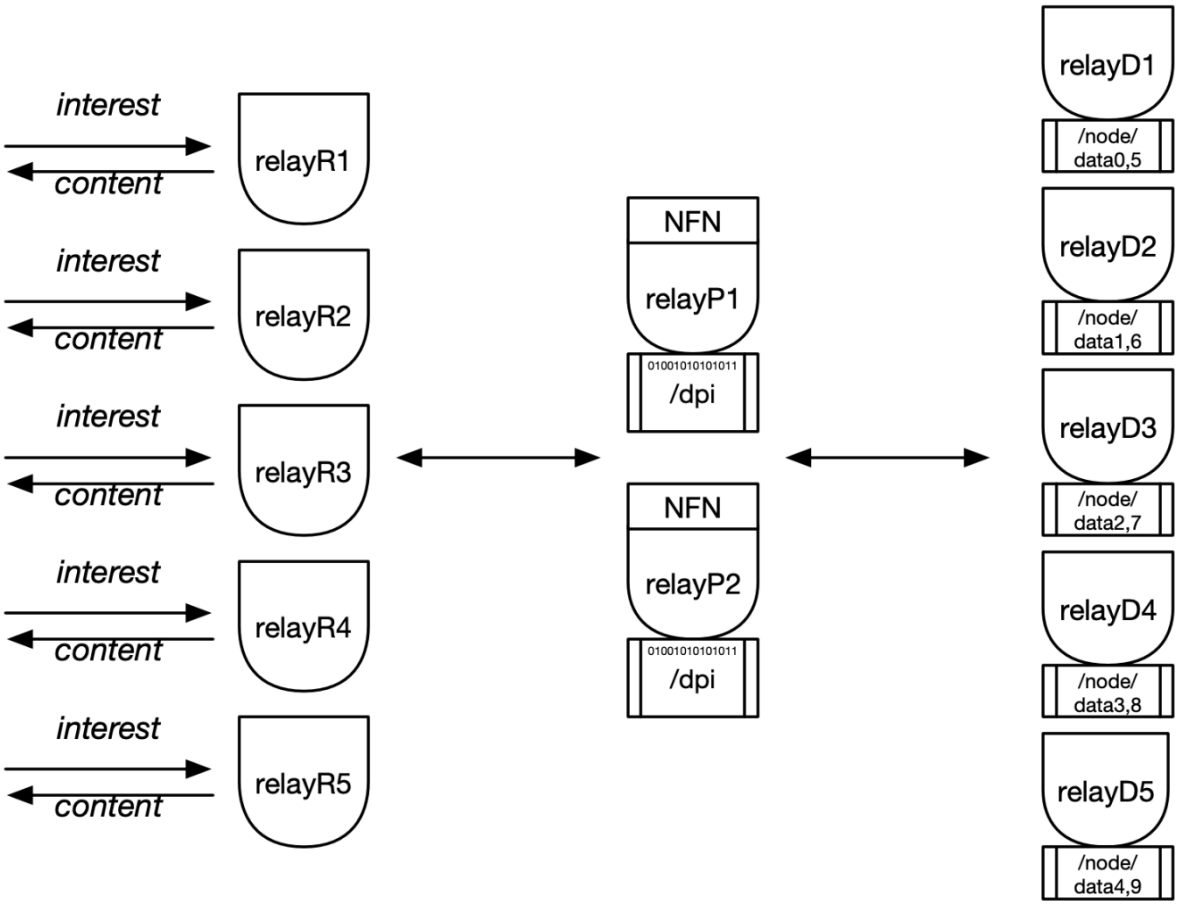


FIGURE 18 depicts the overlay hybrid CCN/NFN infrastructure build on top of the OpenStack virtual infrastructure. Relays P and R are NFN-enabled: relays P because all VNFs instantiated by the DNFA Manager must receive the complete NFN application processing/execution environment and relays R have a pure

router/forwarder role, and only need the NFN Abstract Machine for the distribution of computation tasks and caching of results. Relays D are the only pure CCN nodes of the overlay infrastructure.

FIGURE 18 OVERLAY HYBRID CCN/NFN INFRASTRUCTURE TOPOLOGY

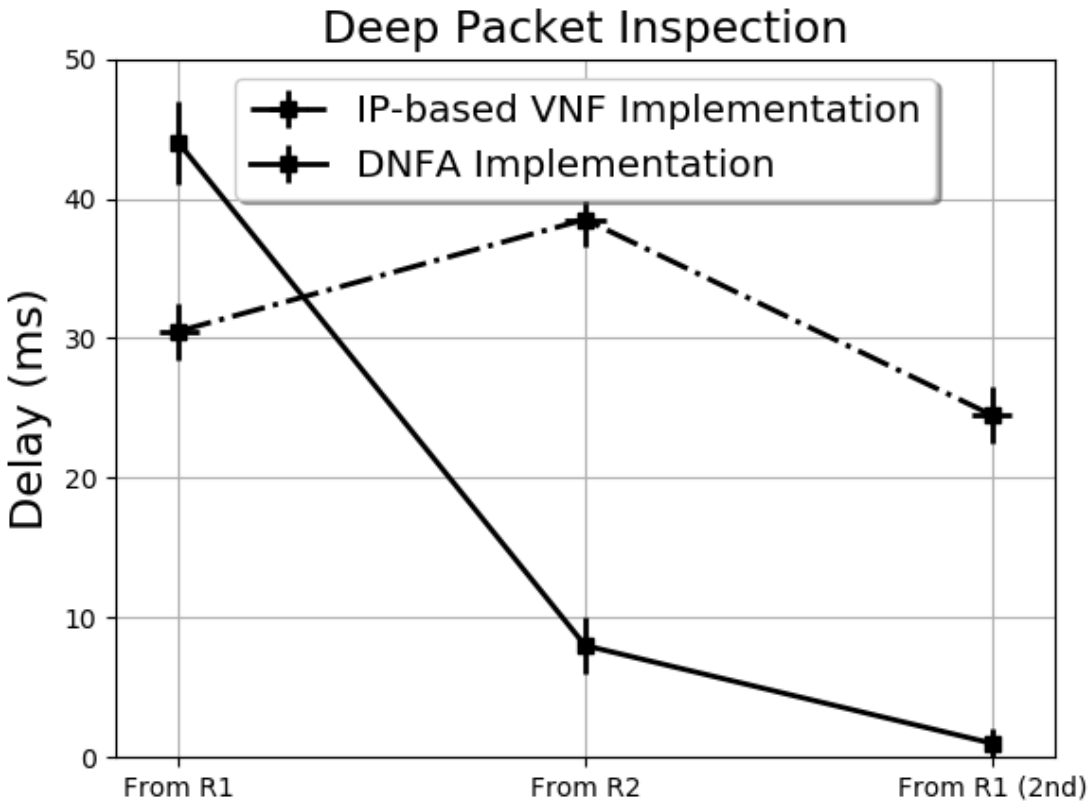


6.2.2 Consecutive VNF invocations

FIGURE 19 pictures response time from various sequential vDPI invocations as means to deconstruct where each delay time comes from. First invocation from R1 (Relay R1), followed by an identical invocation (requisition of the same content) from R2 and followed by a second and final requisition from R1. For the first requisition it is observed an almost equal delay response both from the classic IP VNF implementation and from its DNFA equivalent. However, in the second request (from R2) it is observed a significant difference. This is because, in the case of the DNFA implementation, the desired (computational) content is already cached within RP, so

its recovery is faster than its IP alternative. Finally, in the third solicitation (the second one from R1) has the computational result cached within R1 itself. Note the capability of the computational caching scheme to reduce the amount of redundant inter-node traffic in a half in this particular scenario.

FIGURE 19 DELAY OF CONSECUTIVE DPI INVOCATIONS



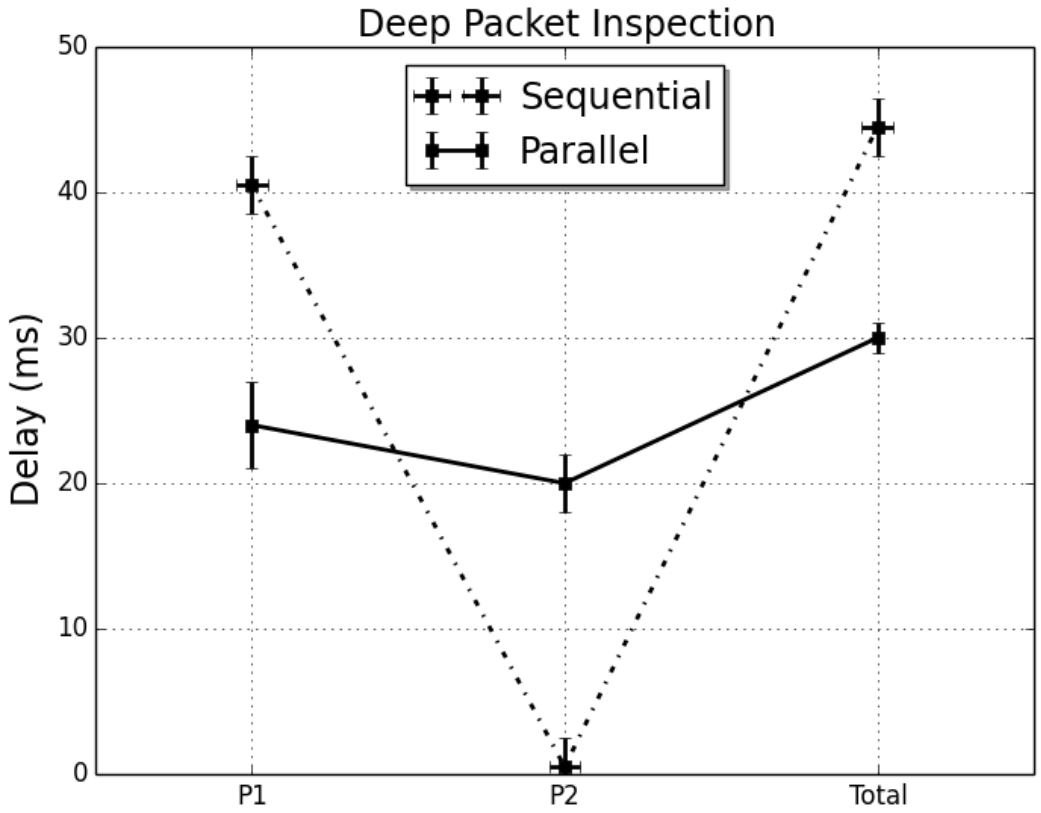
6.2.3 Sequential vs Parallel implementation

As described in Section 4.3.4, various named functions may be used to implement one functioning network function. This is achieved through a special syntax in the service chaining parameter $relay\_src$ of the DNFA extended TOSCA profile. For instance, $relay\_src: VDPI1\&VDPI2$ represents that the correspondent VDPI VNF is implemented though two non-interdependent named functions VDPI1 and VDPI2 (that may be executed in parallel for instance in relayP1 and relayP2).

FIGURE 20 pictures response from executing VDPI in a monolithic/sequential fashion (in relayP1) versus executing it through two non-

interdependent named functions (VDPI1 running in relayP1 and VDPI2 running in relayP2). The figure shows execution times in relayP1, relayP2 and total execution time. Note that in the parallel scenario total execution time is significantly better as execution times are divided between relayP1 and relayP2.

FIGURE 20 DELAY OF MONOLITHIC vs PARALLEL DPI IMPLEMENTATION



6.2.4 Massive concurrent VNF invocations

Locust (“Locust - A modern load testing framework”,) was used as load testing tool. It allows the definition of user behaviour with Python code and swarming any system with hundreds or thousands of simultaneous users. Locust was built with HTTP as its main target, but it can easily be extended to load test any request/response-based system. For the purpose of this Thesis it was wrote and made publicly available a custom client to support CCN requests (CASTILLO, 2019).

Network nodes have been assigned constant cache sizes, a cache expiration time of 10 seconds and a Leave Copy Everywhere (LCE) cache policy. 50 users

were spawned with a hatch rate of 2 user spawned per second, equally distributed between five relays R. Each user made consecutive and random requisitions from 200 to 400 *ms*. Content requests have been modelled as Poisson processes with $\lambda = 10$ to simulate content popularity. The popularity ordering is determined by the request frequencies for each object. The duration of the experiment was of 250 seconds.

It was analysed the performance of the DNFA implementation by focusing on the following metrics: requisitions per second, total number of requests delivered, global response time, response time per content popularity and inter-node traffic.

FIGURE 21 shows the requisitions per second (RPS) delivered by each implementation against the number of users. As stated before, the vDPI VNF has a sequential behavior implemented through a sequential task queue. The DNFA implementation caches both most popular content and the computation result of the virtual network function invocation managing to deliver more than 3 times requisitions per second than its IP-based alternative.

FIGURE 21 RESQUESTS PER SECOND

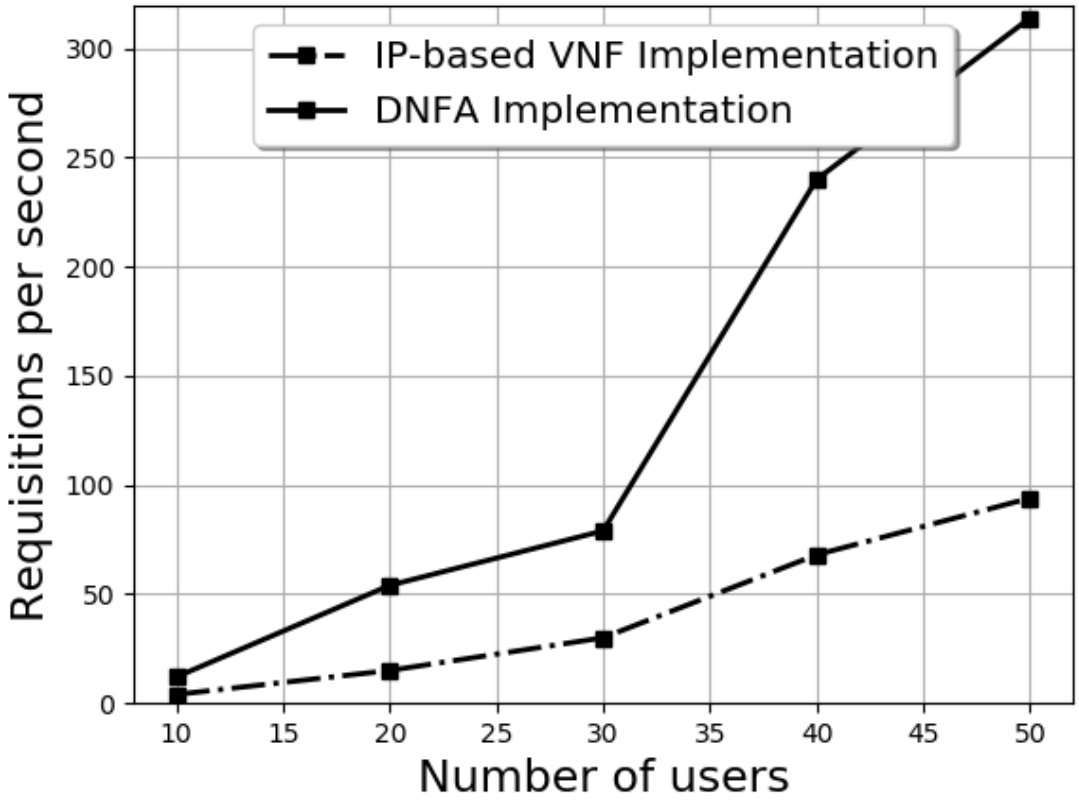


FIGURE 22 depicts the global response time delivered by each implementation against the number of users. The DNFA implementation shows a pike in the beginning of the experiment, where it has no cached content whatsoever, but rapidly manages to reduce global response time by one-third as the cache hit ratio increases and stabilizes.

FIGURE 22 GLOBAL RESPONSE TIME

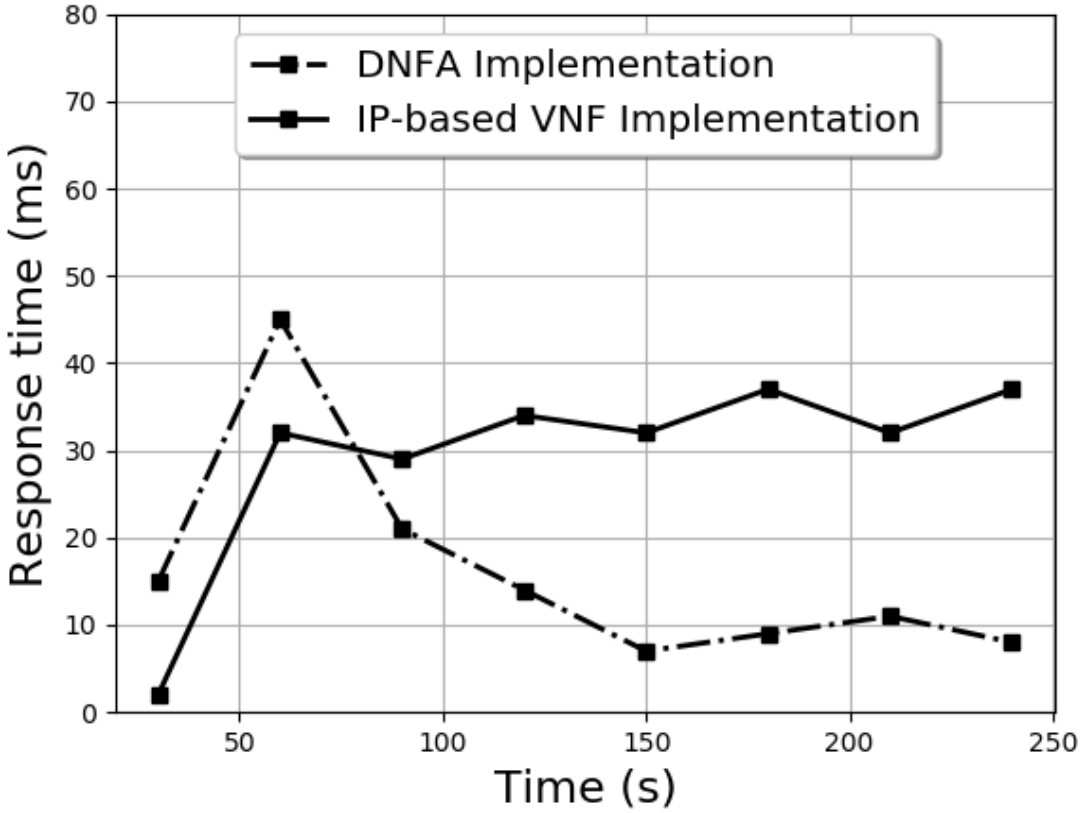


FIGURE 23 shows the response time delivered by each implementation per access frequency. As stated before, content requests have been modelled as Poisson processes with $\lambda = 10$, simulating content popularity. The popularity ordering is determined by the request frequencies for each object. Content represented by $k = 10$ is the most popular (and consequently the most requested) and contents represented by $k = 0$ and $k = 20$ are the least popular/requested. As expected, in the DNFA implementation more popular content displays a higher cache hit ratio and consequently a lower response time than rarely accessed content. The IP implementation shows a constant response time, regardless of the access frequency.

FIGURE 23 RESPONSE TIME PER CONTENT POPULARITY

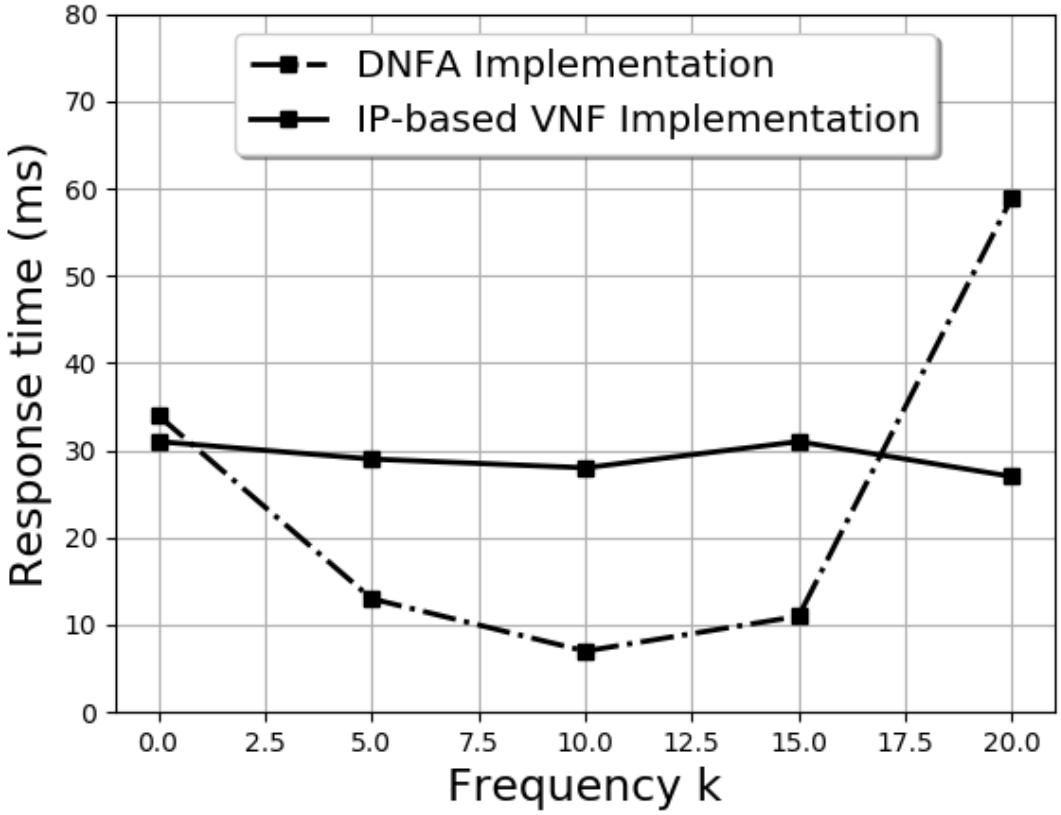


TABLE 6 consolidates number of requests, median/average/min/max times and number of requisitions per second by content popularity in the DNFA load testing experiment. TABLE 7 shows its equivalents for the IP-based experiment.

TABLE 6 DNFA BENCHMARKS BY CONTENT POPULARITY

k	Requests	Median (ms)	Average (ms)	Min (ms)	Max (ms)	RPS
0	9	29	34	2	117	0
2.5	22	7	15	3	77	2.9
5	82	9	13	4	102	3.75
7.5	942	4	6	3	75	81.5
10	2188	5	7	2	70	166.2
12.5	948	5	7	2	117	48.4
15	100	7	11	4	107	9.5
17.5	14	12	14	3	89	0.3
20	1	59	58	59	81	0
Total	4306	-	-	-	-	312.5

TABLE 7 IP BENCHMARKS BY CONTENT POPULARITY

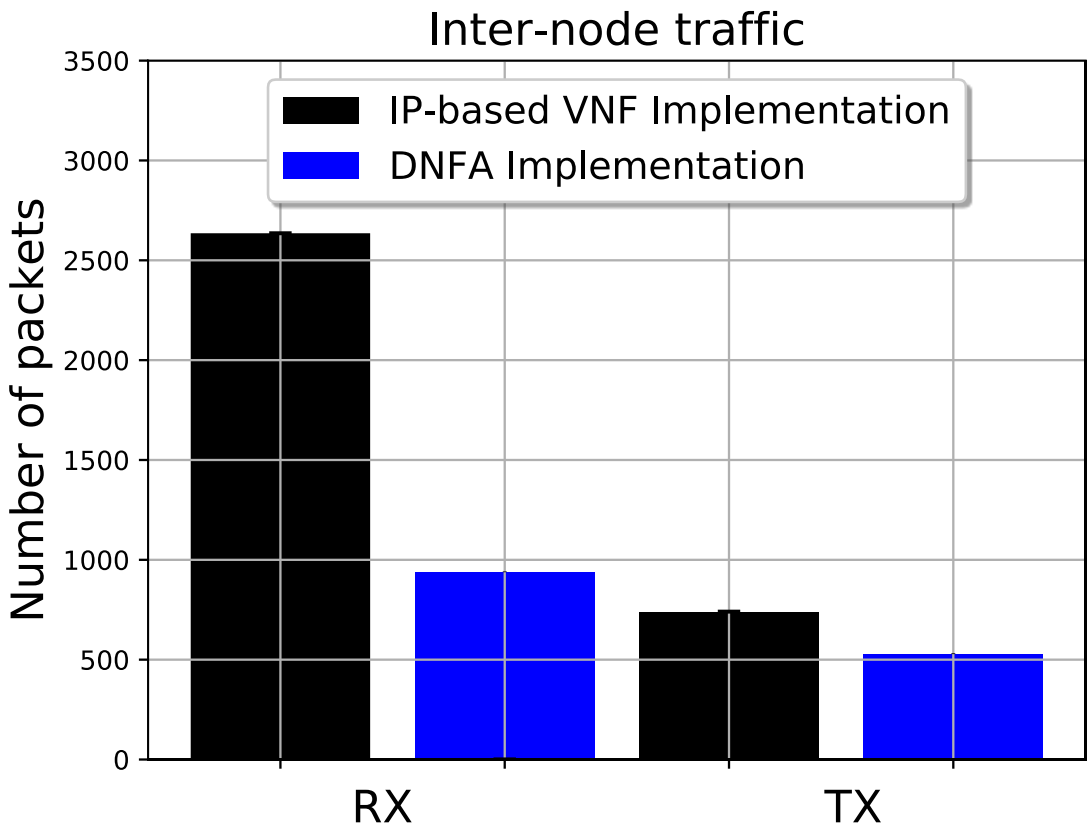
k	Requests	Median (ms)	Average (ms)	Min (ms)	Max (ms)	RPS
0	3	31	31	30	32	0
2.5	7	31	31	30	33	3.1
5	32	31	31	30	102	14.4
7.5	271	31	31	30	88	22.8
10	709	30	29	29	66	30.2
12.5	281	31	31	30	112	17.3
15	28	31	31	30	31	4.3
17.5	12	30	30	30	55	2.4
20	1	30	30	30	47	0.6
Total	1344	-	-	-	-	95.1

Note that even though execution time of the vDPI is slightly better in the IP scenario than DNFA's (maximum execution time significantly better as well), DNFA manages to deliver better results as the cache hit ratio grows. Almost all objects (all but content 9 corresponding to k=20) in the DNFA scenario where cached at some point, therein the minimum execution times of < 5 ms.

The cache hit ratio corresponds to the fraction of content requests served by caches deployed within the NFN network and reflects the capability of the caching scheme to reduce the amount of redundant inter-node traffic. FIGURE 24 depicts the inter-node traffic corresponding to the requesting relays R, measured in number of

packets both for received traffic (RX) and transmitted traffic (TX). The most significant difference is manifested in the received traffic, where DNFA managed to reduce the number of packets in over 2.5 times.

FIGURE 24 INTER-NODE TRAFFIC IN REQUESTING RELAYS R



6.3 CONCLUSION

This chapter presented the results of the performance evaluation of the DNFA framework, analysed at on a real testbed scenario, a private OpenStack cloud.

The data plane analysis confirms reduced computational resource usage, response times and inter-node traffic and better requisition rates.

6 CONCLUDING REMARKS

7 CONCLUDING REMARKS

This chapter will present the concluding remarks.

This work explored the use and implementation of VNFs in CCNs, and proposed the use of the NFN paradigm as means to implement network functions and services in this kind of networks, distributing the network functions and services through the networks nodes and providing flexibility to dynamically place functions in the network as required and without the need of a SDN central controller to perform Service Chaining. The Dynamic Named Function Architecture (DNFA) was proposed, a scalable and flexible framework that allows the automated deployment, lifecycle management and orchestration of named functions over content-centric network infrastructures, using an extended TOSCA NFV profile template.

DNFA can bring many benefits, from reducing computational resources, power usage, response times and inter-node traffic to better requisition rates and cost-efficient realization of network functions in software deployed over commodity hardware, without the need of a central controller.

7.1 CONTRIBUTIONS

The contributions of this work, in line with the specific objectives presented in Chapter 1, are:

- an integrated management architecture, including an orchestrator platform, for the automated deployment, management, monitoring, optimization and lifecycle management (e.g. instantiation, configuration, update, scale up/down, termination, etc.) of named functions over network infrastructures;
- the use of named functions and the NFN paradigm to implement network functions and services in content-centric networks;
- the distribution of network services/functions through the network infrastructure taking advantage of the distributed nature of named functions, using its preferential execution opportunism to gain gratuitous parallelism and asynchronous computations;

- a native and purely content-based non central controller-dependant network function execution environment, allowing the creation of more agile content-centric networks;
- an extended TOSCA NFV template profile for CCN scenarios;
- a CCN name tree definition for NFV scenarios;
- by reducing computational resources usage, response times and inter-node traffic, gains in operational and capital benefits are expected: from improving operational efficiency and reducing power usage to better requisition rates and cost-efficient realization of network functions in software deployed over commodity hardware, without the need of a central controller.

7.2 LIMITATIONS

Main limitations include:

- Many IP-based VNFs are not suited for CCN scenarios.
- VNF instantiation by VNFM, as performed by DNFA, should only be used for testing purposes, and since a VNF only makes sense as a part of a Network Service, the intended way is to use an NSD to instantiate all VNFs.

Additionally, DNFA was implemented to validate the ideas of this Thesis, and lacks several functionalities of more mature and commercial MANOs:

- Monitoring;
- Autoscaling;
- Performance management;
- Multisite enablement;
- Fault management;
- Hardware offloading and advanced network configurations: SR-IOV, Data Plane Development Kit (DPDK), Non-Uniform Memory Access (NUMA), CPU pinning, etc.

7.3 FUTURE WORK

As future work is planned to:

- integrate DNFA with an automated NFV performance benchmarking testing framework, like e.g. (ROSA; BERTOLDO; ROTHENBERG, 2017);
- support an NFV integration test suite, like e.g. Open Platform for NFV (“OPNFV”,);
- extend DNFA support to other VIMs: OpenVIM, AWS, VMWare, etc.;
- extend the ETSI NFV template format to support DNFA integration;
- enlarge DNFA tests with bigger topologies and multiple VNFs;
- perform trace-based analysis of the DNFA framework, defining a characteristic ICN workload for NFV evaluation;
- in IP-based scenarios, explore the advantages and changes needed (new triggers, placement orchestration, etc.) for running VNFs in on-premises Serverless frameworks (i.e.: OpenFaas, Fission, Kubeless, Apache OpenWhisk, etc.).

REFERENCES

AHLGREN, B. et al. A survey of information-centric networking. **IEEE Communications Magazine**, v. 50, n. 7, p. 26–36, jul. 2012.

ANAND, A.; SEKAR, V.; AKELLA, A. SmartRE: An Architecture for Coordinated Network-wide Redundancy Elimination. In: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2009. Disponível em: <http://doi.acm.org/10.1145/1592568.1592580>. Acesso em: 2 ago. 2019.

ARUMAITHURAI, M. et al. Exploiting ICN for Flexible Management of Software-defined Networks. In: Proceedings of the 1st ACM Conference on Information-Centric Networking, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2014. Disponível em: <http://doi.acm.org/10.1145/2660129.2660147>. Acesso em: 9 maio. 2019.

ARUMAITHURAI, M. et al. Prototype of an ICN based approach for flexible service chaining in SDN. In: 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), **Anais**... In: 2015 IEEE CONFERENCE ON COMPUTER COMMUNICATIONS WORKSHOPS (INFOCOM WKSHPS). abr. 2015.

BIANCO, A. et al. OpenFlow Switching: Data Plane Performance. In: 2010 IEEE International Conference on Communications (ICC), **Anais**... In: 2010 IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS (ICC). 2010.

BRAUN, T. et al. Service-Centric Networking. In: 2011 IEEE International Conference on Communications Workshops (ICC), **Anais**... In: 2011 IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS WORKSHOPS (ICC). jun. 2011.

CASTILLO, J. **Mini-nfv framework.** Disponível em: <https://github.com/josecastillolema/mini-nfv>. Acesso em: 14 out. 2018a.

CASTILLO, J. **etsi2tosca - Converts ETSI NFV templates into TOSCA profile.** Disponível em: <https://github.com/josecastillolema/etsi2tosca>. Acesso em: 13 fev. 2019b.

CASTILLO, J. **Locust CCN client**. Disponível em: <https://github.com/josecastillolema/locust-ccnclient>. Acesso em: 15 mar. 2019.

CCN-lite Project. Disponível em: <http://www.ccn-lite.net>. Acesso em: 2 maio. 2016.

cloud-init - The standard for customising cloud instances. Disponível em: <http://cloud-init.org/>. Acesso em: 22 abr. 2018.

DANNEWITZ, C. et al. Network of Information (NetInf) - An Information-centric Networking Architecture. **Comput. Commun.**, v. 36, n. 7, p. 721–735, abr. 2013.

DIMITROV, V.; KOPTCHEV, V. PSIRP Project – Publish-subscribe Internet Routing Paradigm: New Ideas for Future Internet. In: Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2010. Disponível em: <http://doi.acm.org/10.1145/1839379.1839409>. Acesso em: 12 dez. 2014.

ETSI GS NFV-SWA 001 V1.1.1 (2014-12). Disponível em: <http://docplayer.net/3007035-Etsi-gs-nfv-swa-001-v1-1-1-2014-12.html>. Acesso em: 2 ago. 2019.

eve Pyhton REST API Framework. Disponível em: <http://python-eve.org/>. Acesso em: 2 maio. 2016.

FOSTER, N. et al. Frenetic: A Network Programming Language. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2011. Disponível em: <http://doi.acm.org/10.1145/2034773.2034812>. Acesso em: 2 ago. 2019.

FOTIOU, N.; MARIAS, G. F.; POLYZOS, G. C. Access Control Enforcement Delegation for Information-centric Networking Architectures. In: Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2012. Disponível em: <http://doi.acm.org/10.1145/2342488.2342507>. Acesso em: 18 ago. 2019.

GROUP, E. Fia. **Fundamental Limitations of current Internet and the path to Future Internet**. [s.l: s.n.]

JACOBSON, V. et al. Networking Named Content. In: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2009. Disponível em: <http://doi.acm.org/10.1145/1658939.1658941>. Acesso em: 2 ago. 2019.

Jinja2 Documentation (2.10). Disponível em: <http://jinja.pocoo.org/docs/2.10/templates/>. Acesso em: 4 jun. 2018.

KOPONEN, T. et al. A Data-Oriented (and beyond) Network Architecture. **ACM SIGCOMM Computer Communication Review**, v. 37, n. 4, p. 181, 1 out. 2007.

LAI, J.; FU, Q.; MOORS, T. Rapid IP Rerouting with SDN and NFV. In: 2015 IEEE Global Communications Conference (GLOBECOM), **Anais**... In: 2015 IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM). dez. 2015.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2010. Disponível em: <http://doi.acm.org/10.1145/1868447.1868466>. Acesso em: 2 ago. 2019.

LI, X.; QIAN, C. A survey of network function placement. In: 2016 13th IEEE Annual Consumer Communications Networking Conference (CCNC), **Anais**... In: 2016 13TH IEEE ANNUAL CONSUMER COMMUNICATIONS NETWORKING CONFERENCE (CCNC). jan. 2016.

LI, Y.; CHEN, M. Software-Defined Network Function Virtualization: A Survey. **IEEE Access**, v. 3, p. 2542–2553, 2015.

Locust - A modern load testing framework. Disponível em: <https://locust.io/>. Acesso em: 15 mar. 2019.

MANSOUR, D.; TSCHUDIN, C. Towards a Monitoring Protocol Over Information-Centric Networks. In: Proceedings of the 3rd ACM Conference on Information-Centric Networking, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2016. Disponível em: <http://doi.acm.org/10.1145/2984356.2984378>. Acesso em: 31 dez. 2018.

MARXER, C.; SCHERB, C.; TSCHUDIN, C. Access-Controlled In-Network Processing of Named Data. In: Proceedings of the 3rd ACM Conference on Information-Centric Networking, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2016. Disponível em: <http://doi.acm.org/10.1145/2984356.2984366>. Acesso em: 31 dez. 2018.

MATSUBARA, D. et al. Toward future networks: A viewpoint from ITU-T. **IEEE Communications Magazine**, v. 51, n. 3, p. 112–118, mar. 2013.

MCKEOWN, N. et al. OpenFlow: Enabling Innovation in Campus Networks. **SIGCOMM Comput. Commun. Rev.**, v. 38, n. 2, p. 69–74, mar. 2008.

MELAZZI, N. B. et al. Publish/subscribe over information centric networks: A Standardized approach in CONVERGENCE. In: Future Network Mobile Summit (FutureNetw), 2012, **Anais**... In: FUTURE NETWORK MOBILE SUMMIT (FUTURENETW), 2012. jul. 2012.

MICK, T.; TOURANI, R.; MISRA, S. LASeR: Lightweight Authentication and Secured Routing for NDN IoT in Smart Cities. **IEEE Internet of Things Journal**, v. 5, n. 2, p. 755–764, abr. 2018.

MIJUMBI, R. et al. Network Function Virtualization: State-of-the-art and Research Challenges. **IEEE Communications Surveys & Tutorials**, v. 18, n. 1, p. 236–262, 2016.

MongoDB. Disponível em: <https://www.mongodb.org/>. Acesso em: 2 ago. 2019.

Most Common SDN & NFV Use Cases Defined. Disponível em: <https://www.sdxcentral.com/sdn-nfv-use-cases/>. Acesso em: 13 maio. 2016.

nfv-sol-libs: ETSI GS NFV SOL001 v2.5.1. [s.l.] Nextworks s.r.l., 2019.

NGMN - 5G White Paper. Disponível em: <https://www.ngmn.org/5g-white-paper/5g-white-paper.html>. Acesso em: 12 maio. 2019.

NGUYEN, X. N.; SAUCEZ, D.; TURLETTI, T. **Providing CCN functionalities over OpenFlow switches**. [s.l: s.n.]. Disponível em: <https://hal.inria.fr/hal-00920554/document>. Acesso em: 28 dez. 2018.

OPNFV. Disponível em: <https://www.opnfv.org/>. Acesso em: 12 maio. 2019.

PAN, J.; PAUL, S.; JAIN, R. A survey of the research on future internet architectures. **IEEE Communications Magazine**, v. 49, n. 7, p. 26–36, 2011.

Parser - Parser - OPNFV Wiki. Disponível em: <https://wiki.opnfv.org/display/parser/Parser>. Acesso em: 15 fev. 2019.

PIRO, G. et al. Information Centric Services in Smart Cities. **J. Syst. Softw.**, v. 88, p. 169–188, fev. 2014.

QUEVEDO, J. et al. ICN as Network Infrastructure for Multi-Sensory Devices: Local Domain Service Discovery for ICN-Based IoT Environments. **Wireless Personal Communications**, v. 95, n. 1, p. 7–26, 1 jul. 2017.

QUEVEDO, J. et al. Internet of Things Discovery in Interoperable Information Centric and IP Networks. **Internet Technology Letters**, v. 1, n. 1, p. e1, 2018.

RAVINDRAN, R. et al. Towards software defined ICN based edge-cloud services. In: 2013 IEEE 2nd International Conference on Cloud Networking (CloudNet), **Anais**... In: 2013 IEEE 2ND INTERNATIONAL CONFERENCE ON CLOUD NETWORKING (CLOUDNET). nov. 2013.

REPO, N. O. X. **The POX network software platform.** [s.l: s.n.]

RIOT operating system for IoT. Disponível em: <http://www.riot-os.org/>. Acesso em: 2 ago. 2019.

ROSA, R. V.; BERTOLDO, C.; ROTHENBERG, C. E. Take Your VNF to the Gym: A Testing Framework for Automated NFV Performance Benchmarking. **IEEE Communications Magazine**, v. 55, n. 9, p. 110–117, set. 2017.

Ryu SDN Framework. Disponível em: <https://osrg.github.io/ryu/>. Acesso em: 28 abr. 2019.

SALSANO, S. et al. Information centric networking over SDN and OpenFlow: Architectural aspects and experiments on the OFELIA testbed. **Computer Networks**, Information Centric Networking. v. 57, n. 16, p. 3207–3221, 13 nov. 2013.

Scapy. Disponível em: <https://scapy.net/>. Acesso em: 24 abr. 2019.

SEKAR, V. et al. CSAMP: A System for Network-wide Flow Monitoring. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, Berkeley, CA, USA. **Anais**... Berkeley, CA, USA: USENIX

Association, 2008. Disponível em: <http://dl.acm.org/citation.cfm?id=1387589.1387606>. Acesso em: 2 ago. 2019.

SEKAR, V. et al. Network-wide Deployment of Intrusion Detection and Prevention Systems. In: Proceedings of the 6th International COnference, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2010. Disponível em: <http://doi.acm.org/10.1145/1921168.1921192>. Acesso em: 2 ago. 2019.

SHERRY, J.; RATNASAMY, S. **A Survey of Enterprise Middlebox Deployments**. [s.l.] EECS Department, University of California, Berkeley, fev. 2012. . Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-24.html>.

SIFALAKIS, M. et al. An Information Centric Network for Computing the Distribution of Computations. In: Proceedings of the 1st International Conference on Information-centric Networking, New York, NY, USA. **Anais**... New York, NY, USA: ACM, 2014. Disponível em: <http://doi.acm.org/10.1145/2660129.2660150>. Acesso em: 2 ago. 2019.

SRINIVASAN, S. et al. CCNxServ: Dynamic service scalability in information-centric networks. In: 2012 IEEE International Conference on Communications (ICC), **Anais**... In: 2012 IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS (ICC). jun. 2012.

SUAREZ, J. et al. A secure IoT management architecture based on Information-Centric Networking. **Journal of Network and Computer Applications**, v. 63, p. 190–204, 1 mar. 2016.

SUKSOMBOON, K. et al. Pending-interest-driven cache orchestration through network function virtualization. In: 2014 IEEE Global Communications Conference, **Anais**... In: 2014 IEEE GLOBAL COMMUNICATIONS CONFERENCE. dez. 2014.

SYRIVELIS, D. et al. Pursuing a Software Defined Information-centric Network. In: 2012 European Workshop on Software Defined Networking, **Anais**... In: 2012 EUROPEAN WORKSHOP ON SOFTWARE DEFINED NETWORKING. out. 2012.

TELECOMS.COM INTELLIGENCE ANNUAL INDUSTRY SURVEY 2018. p. 37, 2018.

TOSCA Simple Profile for Network Functions Virtualization (NFV)—Version 1.0. Disponível em: <http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html>. Acesso em: 12 set. 2018.

TSCHUDIN, C.; SIFALAKIS, M. Named Functions for Media Delivery Orchestration. In: Packet Video Workshop (PV), 2013 20th International, **Anais**... In: PACKET VIDEO WORKSHOP (PV), 2013 20TH INTERNATIONAL. dez. 2013.

TSCHUDIN, C.; SIFALAKIS, M. Named functions and cached computations. In: Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th, **Anais**... In: CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE (CCNC), 2014 IEEE 11TH. jan. 2014.

UEDA, K. et al. Towards the NFVI-Assisted ICN: Integrating ICN Forwarding into the Virtualization Infrastructure. In: 2016 IEEE Global Communications Conference (GLOBECOM), **Anais**... In: 2016 IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM). dez. 2016.

XYLOMENOS, G. et al. A Survey of Information-Centric Networking Research. **IEEE Communications Surveys Tutorials**, v. 16, n. 2, p. 1024–1049, Second 2014.

ZHANG, X.; ZHU, Q. Information-centric network virtualization for QoS provisioning over software defined wireless networks. In: MILCOM 2016 - 2016 IEEE Military Communications Conference, **Anais**... In: MILCOM 2016 - 2016 IEEE MILITARY COMMUNICATIONS CONFERENCE. nov. 2016.

APPENDIX A – EXTENDED NFV TOSCA PROFILE

The TOSCA ***L3AddressData*** data type is defined as follows:

```
tosca.datatypes.nfv.L3AddressData:
 derived_from: tosca.datatypes.Root
 properties:
  ip_address_assignment:
   type: Boolean
   required: false
  floating_ip_activated:
   type: Boolean
   required: false
  ip_address_type:
   type: string
   required: false
   constraints:
    - valid_values: [ipv4, ipv6, ccnx]
  number_of_ip_address:
   type: integer
   required: false
```

The TOSCA ***AddressData*** data type is defined as follows:

```
tosca.datatypes.nfv.AddressData:
 derived_from: tosca.datatypes.Root
 properties:
  address_type:
   type: string
   required: true
   constraints:
    - valid_values: [mac_address, ip_address, ccnx_name]
  l2_address_data:
   type: tosca.datatypes.nfv.L2AddressData
   required: false
  l3_address_data:
   type: tosca.datatypes.nfv.L3AddressData
   required: false
```

The TOSCA ***ConnectivityType*** data type is defined as follows:

```
tosca.datatypes.nfv.ConnectivityType:
 derived_from: tosca.datatypes.Root
 properties:
  layer_protocol:
   type: string
   required: yes
   constraints:
    - valid_values: [ethernet, mpls, odu2, ipv4, ipv6, pseudo_wire, ccnx ]
  flow_pattern:
   type: string
   required: false
```

The TOSCA ***ForwadingPath*** data type is defined as follows:

```
tosca.nodes.nfv.FP:
 derived_from: tosca.nodes.Root
 properties:
  policy:
   type:
    type: string
    required: false
    description: name of the vendor who generate this VL
   criteria:
    type: list
    content:
     type: string
    relay_src:
     type: string
    placement_hint:
     type: string from ['delegate_upstream','codegrag','computation_push']
 requirements:
  - forwarder:
    capability: tosca.capabilities.nfv.Forwarder
```

APPENDIX B – IP/OPENSTACK TACKER vDPI TOSCA NFV PROFILE

The TOSCA ***VNFD*** template was defined as follows:

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Deep Packet Inspector
metadata:
  template name: DPI_VNFd_userdata

topology_template:
node_templates:
    vDPI:
      type: tosca.nodes.nfv.VDU.Tacker
      capabilities:
        nfv compute:
          properties:
            num cpus: 4
            mem_size: 8192 MB
            disk_size: 10 GB
      properties:
        image: linux_ubuntu_16_64b_base
        availability_zone: nova
    CP1 :
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        management : true
        order: 0
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL1
        - virtualBinding:
            node: vDPI
    CP2 :
      type: tosca.nodes.nfv.CP.Tacker
      properties:
       order: 1
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL2
        - virtualBinding:
            node: vDPI
    VL1:
      type : tosca.nodes.nfv.VL
      properties:
        network name : net_mgmt
    VL2:
      type : tosca.nodes.nfv.VL
      properties:
        network name: custom_net0
        vendor: Tacker
        ip_version: 4
        cidr: '10.0.1.64/26'
```

The TOSCA ***VNFFGD*** template was defined as follows:

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: DPI VNFFG template

topology_template:
  description: DPI VNFFG template
  node_templates:
    Forwarding_path1:
      type: tosca.nodes.nfv.FP.Tacker
      description: creates path (CP12-> CP12)
      properties:
        id: 51
        policy:
          type: ACL
          criteria:
            - network_id: 3d582b05_b996_4df2_9b3f_a1a26b2f1383
            - destination_port_range: 80 - 80
            - ip_dst_prefix: 10.0.0.192/26
            - ip_src_prefix: 10.0.1.64/26
        path:
          - forwarder: vDPI
            capability: CP2
  groups:
    VNFFG1:
      type: tosca.groups.nfv.VNFFG
      properties:
        vendor: tacker
        version: 1.0
        number_of_endpoints: 1
        dependent_virtual_link: [VL2]
        connection_point: [CP2]
        constituent_vnfs: [vDPI]
      members: [Forwarding_path1]
```

APPENDIX C – DNFA vDPI TOSCA NFV PROFILE

The TOSCA ***VNFD*** template was defined as follows:

```
tosca_definitions_version: tosca_simple_profile_for_nfv_dnfa_1_0_0

description: Deep Packet Inspector

metadata:
  template name: DPI_VNFd_userdata
topology_template:
node_templates:
    relayP:
      type: tosca.nodes.nfv.VDU.DNFA
      capabilities:
        nfv compute:
          properties:
            num cpus: 4
            mem_size: 8192 MB
            disk_size: 10 GB
      properties:
        image: linux_ubuntu_16_64b_nfn
        availability_zone: nova

    CP1 :
      type: tosca.nodes.nfv.CP.DNFA
      properties:
        management : true
        order: 0
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL1
        - virtualBinding:
            node: relayP
    CP2 :
      type: tosca.nodes.nfv.CP.Tacker
      properties:
       order: 1
        anti_spoofing_protection: false
      requirements:
        - virtualLink:
            node: VL2
        - virtualBinding:
            node: relayP

    VL1:
      type : tosca.nodes.nfv.VL.DNFA
      properties:
        network name : net_mgmt
    VL2:
      type : tosca.nodes.nfv.VL
      properties:
        network name: custom_net0
```

The TOSCA ***VNFFGD*** template was defined as follows:

$tosca\_definitions\_version$: $tosca\_simple\_profile\_for\_nfv\_dnfa\_1\_0\_0$

$description$: $DPI\ DNFA\ VNFFG\ template$

$topology\_template$:
  $description$: $DPI\ DNFA\ VNFFG\ template$
  $node\_templates$:
    $Forwarding\_path1$:
      $type$: $tosca.nodes.nfv.FP.DNFA$
      $description$: $creates\ path\ (CP12 -> CP12)$
      $properties$:
        id: 51
        $policy$:
          $type$: ACL
          $criteria$:
            $-\ content$: $sensitive\_content$
            $-\ relay\_src$: $relayP$
            $-\ placement\_hint$: $delegate\_upstream$
          $path$:
          $-\ forwarder$: $relayP$
            $capability$: $CP2$
  $groups$:
    $VNFFG1$:
      $type$: $tosca.groups.nfv.VNFFG$
      $properties$:
        $version$: 1.0
        $number\_of\_endpoints$: 1
        $dependent\_virtual\_link$: [$VL2$]
        $connection\_point$: [$CP2$]
        $constituent\_vnfs$: [$relayP$]
      $members$: [$Forwarding\_path1$]