

Universidade Federal do Espírito Santo

Cristina Klippel Dominicini

Programmable, Expressive, Scalable,
and Agile Service Function Chaining
for Edge Data Centers

Vitória-ES
2019



PROGRAMMABLE, EXPRESSIVE, AND AGILE SERVICE FUNCTION CHAINING FOR EDGE DATA CENTERS

Cristina Klippel Dominicini

Tese submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Aprovada em 23 de agosto de 2019:

Prof. Dr. Magnos Martinello
Orientador

Prof. Dr. Moisés Renato Nunes Ribeiro
Coorientador

Prof. Dr. Vinícius Fernandes Soares Mota
Membro Interno

Prof. Dr. Rafael Pasquini
Membro Externo

Prof. Dr. Luciano Paschoal Gaspar
Membro Externo

Prof. Dr. Christian Rodolfo Esteve Rothenberg
Membro Externo

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
Vitória-ES, 23 de agosto de 2019.

Ficha catalográfica disponibilizada pelo Sistema Integrado de
Bibliotecas - SIBI/UFES e elaborada pelo autor

D671p Dominicini, Cristina Klippel, 1985-
 Programmable, Expressive, Scalable, and Agile Service
 Function Chaining for Edge Data Centers / Cristina Klippel
 Dominicini. - 2019.
 190 f. : il.

 Orientador: Magnos Martinello.
 Coorientador: Moisés Renato Nunes Ribeiro.
 Tese (Doutorado em Informática) - Universidade Federal
 do Espírito Santo, Centro Tecnológico.

 1. Computação em nuvem. 2. Redes de computadores. 3.
 Rede de computador - Protocolos. I. Martinello, Magnos. II.
 Ribeiro, Moisés Renato Nunes. III. Universidade Federal do
 Espírito Santo. Centro Tecnológico. IV. Título.

CDU: 004

Dedico esta tese ao meu marido, Rodolfo, e aos meus pais, Eugélica e Waldir.

Agradecimentos

A Deus pela proteção, força e bênçãos recebidas durante esta caminhada.

Aos meus orientadores, Profs. Magnos Martinello e Moisés Ribeiro, pela orientação, pelo exemplo, pelo tempo dedicado, pelo conhecimento compartilhado, pela paciência e pelas oportunidades de crescimento. O trabalho deles é grande fonte de inspiração pela dedicação às pessoas e pela busca por excelência.

Ao Prof. Rodolfo Villaça, pelas discussões e idéias valiosas para evolução deste trabalho.

Ao Prof. Gilmar Vassoler, pelo apoio essencial para concepção deste trabalho.

Ao Prof. Diego Mafioletti pelo apoio com os experimentos com as SmartNICs.

À Profa. Ana C. Locateli pelo apoio com os estudos sobre campos finitos.

Ao Profs. Christian Rothenberg e César Marcondes pelas contribuições ao trabalho na banca de qualificação.

Aos meus pais, Eugélica e Waldir, por todo amor e dedicação. Eles são os meus heróis e meus maiores exemplos de vida.

Ao meu marido, melhor amigo e companheiro, Rodolfo, pelo apoio e amor ao longo dessa caminhada que fizemos juntos.

Aos meus amigos e à minha família que sempre me apoiaram. Em especial, ao meu irmão, Wagner, e à minha cunhada, Luciana.

A todos os amigos e colaboradores do Núcleo de Estudos em Redes Definidas por Software (NERDS), por todos os aprendizados e momentos de convivência que ficarão na memória. Em especial, Alextian, Rafael, Sabrina, Márcia, Eduardo Zambon, Renato, Maxwell, Leandro, Dione, Rodolfo Ribeiro, Diego Cardoso, Isabella, Leonardo, Matheus, Rodolfo Valentim, Pedro, João Henrique, Victor, Ricardo, Epaminondas, Fernando, Pablo, Vinicius, Rodrigo, Roberto, Luis Fernando e João Paulo.

Ao Instituto Federal do Espírito Santo pela oportunidade de afastamento para realização do doutorado.

Ao Programa de Pós-Graduação em Informática (PPGI) da Universidade Federal do Espírito Santo pela estrutura oferecida para desenvolvimento desta tese de doutorado.

A CAPES, CNPq, FAPES e RNP pelo apoio financeiro aos projetos de pesquisa do NERDS.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

O paradigma de computação de borda transfere a capacidade de processamento de grandes centros de dados remotos para centros de dados menores e distribuídos na borda da rede. Essa mudança exige soluções de virtualização de funções de rede (network functions virtualization, NFV) que possam gerenciar e combinar eficientemente um grande número de serviços dinâmicos em um centro de dados com poucos recursos, ao mesmo tempo que garantam que os requisitos de desempenho sejam atendidos.

No entanto, os mecanismos de roteamento das redes de centros de dados tradicionais não são adequados para a composição dinâmica desses serviços, pois são complexos, rígidos, sujeitos a grandes atrasos de propagação de informações de controle e com escalabilidade limitada pelo tamanho das tabelas de encaminhamento. Além disso, as soluções tradicionais de encadeamento de funções de serviço (service function chaining, SFC) são frequentemente desacopladas das decisões de roteamento de rede e restringem as opções de seleção de caminhos por parte da engenharia de tráfego. Dessa forma, o orquestrador NFV não consegue explorar toda a capacidade da rede.

Para resolver esses problemas, esta tese investiga uma proposta de SFC que seja programável, expressiva, escalável, e ágil para permitir a orquestração dinâmica e eficiente da infraestrutura de rede de centros de dados de borda. Essa proposta é composta por três soluções inter-relacionadas que exploram tecnologias de virtualização e programabilidade de redes de centros de dados com equipamentos de rede comoditizados. A primeira delas, chamada VirtPhy, é uma arquitetura programável que tira proveito das propriedades topológicas de centros de dados centrados em servidores para orquestração de NFV. A segunda solução, chamada KeySFC, é um esquema de SFC independente de topologia, que explora o conceito de redes *fabric* com uma separação clara entre: (i) comutadores de borda baseados no paradigma de redes definidas por software (software-defined networking, SDN) que classificam fluxos para SFC; e (ii) comutadores de núcleo que executam um mecanismo de roteamento de fonte baseado no sistema numérico de resíduos (residue number system, RNS), que elimina a necessidade de tabelas de encaminhamento. Por fim, a terceira solução, chamada PolKA, é um mecanismo que estende o roteamento de fonte com RNS usando Galois Fields de dois elementos, $GF(2)$, de forma que a representação binária usada no sistema de roteamento seja mais próxima às operações disponibilizadas em equipamentos de rede comoditizados.

Como prova de conceito, foram implementados protótipos de todas as soluções propostas com tecnologias de produção de redes de centros de dados, tais como OpenFlow, OpenStack, Open vSwitch e P4. Os resultados dos testes funcionais e de desempenho mostraram que as propostas conseguem habilitar o encadeamento de funções de rede em centros de dados para computação de borda de forma programável, expressiva, escalável, ágil e com baixo custo. Dessa forma, o esquema de SFC proposto consegue entregar ao orquestrador de NFV mecanismos que permitam que a engenharia de tráfego tome decisões otimizadas na seleção de caminhos de redes. Esta tese também abre caminho para a exploração em esquemas de SFC de várias propriedades do roteamento de fonte baseado em RNS, que podem agregar funcionalidades como segurança, reação rápida a falhas e encaminhamento sem reescrita do pacote.

Palavras-chave: NFV, SFC, computação de borda, SDN, redes de centros de dados, RNS, equipamentos comoditizados, roteamento de fonte.

Abstract

The edge computing paradigm transfers processing power from large remote data centers (DCs) to distributed DCs at the edge of the network. This shift requires the ability to provide network functions virtualization (NFV) solutions that can efficiently manage and combine a large number of dynamic services in a resource-constrained DC, while ensuring that performance requirements are met.

However, the routing mechanisms of traditional data center networks are not adequate for the dynamic composition of these services, because they are complex, rigid, subject to large delays in the propagation of control information, and limited by the size of switches' routing tables. In addition, traditional service function chaining (SFC) solutions in the service overlay are often decoupled from routing decisions in the network underlay, and restrict path selection options by traffic engineering. In this way, the NFV orchestrator cannot explore the full capacity of the network.

To tackle these issues, this thesis investigates a programmable, expressive, scalable, and agile SFC proposal that allows dynamic and efficient orchestration of the network infrastructure of edge DCs. This proposal is composed of three interrelated solutions that exploit virtualization and programmability technologies of DC networks with commodity network equipment. The first one, called VirtPhy, is a programmable architecture that takes advantage of the topological properties of server-centric DCs for NFV orchestration. The second solution, called KeySFC, is a topology-independent SFC scheme that exploits the concept of fabric networks with a clear separation between: (i) edge switches based on software-defined networking (SDN) that classify flows for SFC; and (ii) core switches that implement a source routing mechanism based on the residue number system (RNS), which eliminates the need for routing tables. Finally, the third solution, called PolKA, is a mechanism that extends RNS-based source routing using Galois Fields of two elements, $GF(2)$, so that the binary representation of the routing system is closer to the available operations in commodity network equipment.

As proof-of-concept, prototypes of all proposed solutions were implemented with production DC technologies, such as OpenFlow, OpenStack, Open vSwitch and P4. The results of functional and performance tests showed that the solutions can enable SFC in edge DCs in a programmable, expressive, scalable, agile and low cost manner. Thus, the proposed SFC scheme provides mechanisms to the NFV orchestrator that allow traffic engineering to make optimized decisions in the selection of network paths. This thesis also paves the way for exploring various RNS-based source routing properties in SFC schemes, which can provide features such as route authenticity, fast failure reaction, and forwarding without packet rewrite.

Keywords: network functions virtualization, software-defined networking, service function chaining, edge computing, data center networks, source routing, commodity equipment, residue number system.

List of Acronyms

BSS	business support systems
CAPEX	capital expenditures
COTS	commercial off-the-shelf
CRC	cyclic redundancy check
CRT	Chinese remainder theorem
DC	data center
DCN	data center network
ECMP	equal cost multi path
ETSI	European Telecommunication Standards Institute
FEC	forward error correction
GF	Galois field
HPC	high performance computing
IETF	Internet Engineering Task Force
ILP	integer linear programming
ISG	Industry Specification Group
LDP	label distribution protocol
LSPs	label-switched paths
MAC	media access control
MANO	management and orchestration
MEC	mobile edge computing
MME	mobility management entity
MNO	mobile network operator
MPLS	multi-protocol label switching
NERDS	núcleo de estudos em redes definidas por software

NFV network functions virtualization

NFVI NFV infrastructure

NFVO NFV orchestrator

NSH network service headers

OAM operation and management

ONF Open Networking Foundation

OPEX operational expenses

OSM Open Source MANO

OSS operational support systems

OvS Open vSwitch

PGW packet data network gateway

PISA protocol-independent switch architecture

PNF physical network function

PoC proof-of-concept

RAN radio access network

RNS residue number system

SDN software-defined networking

SF service function

SFC service function chaining

SFF service function forwarder

SFP service function path

SI service index

SP service provider

SPF shortest path first

SPI service path identifier

SR source routing

SR-TE segment routing traffic engineered

SSR strict source routing

STP spanning tree protocol

ToR top-of-rack

VIM virtualized infrastructure manager

VM virtual machine

VMAC virtual MAC

VNF virtualized network function

VNF-FG VNF forwarding graph

VNF-FGE VNF forwarding graph embedding

VNFM VNF manager

List of Figures

1.1	SFC embedding: service overlay mapped into underlay network.	24
1.2	SFC workflow. Adapted from [Zhang et al. 2018].	25
1.3	VNF-FGE example.	26
1.4	Traffic engineering workflow. Adapted from: [Tso et al. 2016].	26
1.5	Example scenarios for edge data centers.	32
1.6	Overview of works proposed in this Thesis.	35
2.1	Example of fat-tree topology. Source: [Al-Fares et al. 2008].	38
2.2	Example of server-centric topologies with 8 nodes and 12 links: (a) Twin, and (b) Hypercube. Source: [Vassoler 2015].	39
2.3	Tableless strict source routing example.	41
2.4	Example of <i>routeID</i> computation. Adapted from [Martinello et al. 2014].	43
2.5	Edge computing paradigm. Source [Shi and Dustdar 2016].	44
2.6	NFV implementation of network functions using virtualization techniques over standard hardware. Source: [Han et al. 2015].	46
2.7	ETSI NFV Architecture.	47
2.8	Reference architecture from RFC 7665. Source: [Castanho et al. 2018].	49
3.1	Comparison between data center network architectures.	52
3.1.a	Network-centric	52
3.1.b	Server-centric	52
3.2	VNF-FGE problem in network-centric topologies : (a) service overlay layer: VNF forwarding graph; (b) server layer: VNF embedding in physical servers and logical connections between servers; (c) network underlay layer: routing in physical network. Adapted from: [Dominicini et al. 2017].	53
3.3	VNF-FGE problem in server-centric topologies : (a) service overlay layer: VNF forwarding graph; (b) server layer: VNF embedding in physical servers and logical connections between servers; (c) network underlay layer: routing in physical network. Adapted from: [Dominicini et al. 2017].	53
3.4	SFC scenarios: (a) Legacy networks: PNFs, and SDN hardware switches for SFC. (b) Network-centric cloud: VNFs, and SDN software and hardware switches for SFC. (c) Server-centric cloud: VNFs, and SDN software switches for SFC. Source: [Dominicini et al. 2017].	54
3.5	SFC problem.	55
3.5.a	SFC layers.	55
3.5.b	Current SFC approaches.	55
3.5.c	SFC with algorithmic SSR.	55

3.6	Comparison between flow entries (represented by stars) in traditional SDN and algorithmic SSR approaches for dynamic migration from SFC1 ($VM_S \rightarrow IDS \rightarrow VM_D$) to SFC2 ($VM_S \rightarrow FW \rightarrow VM_D$).	57
3.6.a	SFC1: traditional SDN approach.	57
3.6.b	SFC2: traditional SDN approach.	57
3.6.c	SFC1: algorithmic SSR approach.	57
3.6.d	SFC2: algorithmic SSR approach.	57
3.7	Layers of TRIIAD architecture. Source: [Vassoler and Ribeiro 2017].	63
4.1	Network-centric approach vs. VirtPhy. Source: [Dominicini et al. 2017].	68
4.2	Challenges, enablers, and design principles. Source: [Dominicini et al. 2017].	69
4.3	VirtPhy and ETSI NFV standard. Source: [Dominicini et al. 2017].	71
4.4	NFV orchestration in VirtPhy. Source: [Dominicini et al. 2017].	72
4.5	Forwarding mechanism using MAC to locate and identify a VNF. Source: [Dominicini et al. 2017].	76
4.6	Basic SFC strategy using SDN. Source: [Dominicini et al. 2017].	78
4.7	SFC example in a hypercube with degree 3. Source: [Dominicini et al. 2017].	80
4.8	SFC test: (a) Service request. (b) SFC scenario: $VM_{source} \rightarrow SFF1 \rightarrow SFF2 \rightarrow VM_{dest}$. Source: [Dominicini et al. 2017].	82
4.8.a		82
4.8.b		82
4.9	SFC Test 1: iperf traffic for SFC $VM_{source} \rightarrow SFF1 \rightarrow SFF2 \rightarrow VM_{dest}$: (a) Traffic at source. (b) Traffic at SFF1. (c) Traffic at SFF2. (d) Traffic at destination. Source: [Dominicini et al. 2017].	85
4.9.a		85
4.9.b		85
4.9.c		85
4.9.d		85
4.10	SFC tests for traffic passing through SFC and normal traffic going directly from source to destination. (a) Test 2: Jitter over time. (b) Test 3: Latency over time. Source: [Dominicini et al. 2017].	86
4.10.a		86
4.10.b		86
4.11	SFC Test 4: Source throughput vs. Destination throughput for SFC traffic and normal traffic going directly from source to destination. Source: [Dominicini et al. 2017].	87
5.1	KeySFC architecture, and example for chain $VM_S \rightarrow SF1 \rightarrow VM_D$.	90
5.2	VMAC address format.	92
5.3	KeySFC example ($VM_S \rightarrow SF1 \rightarrow VM_D$).	93
5.3.a	Detailed scenario.	93
5.3.b	Overlay and underlay.	93
5.3.c	Forwarding operations.	93
5.4	KeySFC prototype testbed.	95
5.5	Reference scenarios exploring different placement strategies: (a) in SC1 , 1 SF and all VMs in different servers; (b) in SC2 , 1 SF allocated with VM_D ; (c) in SC3 , 2 SFs and all VMs in different servers; and (d) in SC4 , 2 SFs allocated in the same server.	97

5.6	Throughput results for functional tests: (a) in SC1 , SF1 is off from 20s to 30s; and (b) in SC3 , SF1 is off from 20s to 30s and SF2 is off from 40s to 50s.	97
5.7	Jitter and latency results comparing reference scenarios with baseline (BL).	97
5.5.a	SC1.	97
5.5.b	SC2.	97
5.5.c	SC3.	97
5.5.d	SC4.	97
5.6.a	SC1 ($VM_S \rightarrow SF1 \rightarrow VM_D$).	97
5.6.b	SC3 ($VM_S \rightarrow SF1 \rightarrow SF2 \rightarrow VM_D$).	97
5.7.a	Baseline (BL).	97
5.7.b	Latency over time for SFC scenarios.	97
5.7.c	Box plot of jitter for SFC scenarios.	97
5.8	Scenarios for evaluating the impact of SFC length.	99
5.9	Latency results when chain length varies from 1 to 10.	99
5.8.a	All SFs in the same server.	99
5.8.b	Consecutive SFs in alternating servers.	99
5.9.a	Scenario of Fig. 5.8.a.	99
5.9.b	Scenario of Fig. 5.8.b.	99
5.10	Traffic engineering for migration from Path 1 to Path 2: (a) Path 1 with concurrent 400Mbps UDP traffic, (b) Path 2 with no concurrent traffic, and (c) Throughput results at VM_D during migration test.	101
5.10.a	Path 1	101
5.10.b	Path 2	101
5.10.c	Throughput at VM_D	101
5.11	Traffic engineering scenarios for different paths using SF redundancy: (a) Path 1 with 600Mbps UDP traffic, (b) Path 2 with 400Mbps UDP traffic, and (c) Path 3 with no concurrent traffic.	102
5.11.a	Path 1	102
5.11.b	Path 2	102
5.11.c	Path 3	102
5.12	Throughput results when SFC migrates from Path 1 to Path 2, and from Path 2 to Path 3 (see Figure 5.11).	102
5.13	Example of Multi-domain SFC.	103
6.1	Example of source routing using PolKA.	108
6.1.a	Unicast : polynomial o_i directly represents label of the transmitting port.	108
6.1.b	Multicast : polynomial o_i represents transmitting states of ports.	108
6.2	Comparison between SSR headers for Sourcey and PolKA.	115
6.2.a	Sourcey header.	115
6.2.b	PolKA header with fixed length.	115
6.2.c	PolKA header with variable length.	115
6.3	Comparison between Sourcey and PolKA complete pipelines.	116
6.3.a	Sourcey.	116
6.3.b	PolKA.	116
6.4	Example fabric network in DCs.	121
6.5	Comparison between Sourcey and PolKA core pipelines.	122

6.5.a Sourcey.	122
6.5.b PolKA.	122
6.6 Example of CRC format.	123
6.7 Example of CRC calculation. Adapted from [Kurose and Ross 2013].	124
6.8 Linear topology.	132
6.9 Linear fabric topology.	132
6.10 Linear scenario: comparison between PolKA and PolKA-Var solutions.	133
6.11 Linear and fabric scenarios: comparison between Sourcey and PolKA solutions.	134
6.11.aRTT small packet.	134
6.11.bRTT big packet.	134
6.11.cRTT background traffic.	134
6.11.dJitter.	134
6.11.eFCT.	134
6.12 Two-tier scenario: agile path migration of TCP flow for allocation of maximum bandwidth in Sourcey and PolKA.	135
6.12.aMigration of TCP flow A ($H_{11_1} \rightarrow H_{21_1}$) from Path 1 to Path 3.	135
6.12.bSourcey: throughput at destination H_{21_1} .	135
6.12.cPolKA: throughput at destination H_{21_1} .	135
6.13 Migration of UDP flow from Path 1 to Path 3 with no concurrent traffic in PolKA.	137
6.13.aPolKA: throughput at source (H_{11_1}).	137
6.13.bPolKA: throughput at S_1 .	137
6.13.cPolKA: throughput at S_2 .	137
6.13.dPolKA: throughput at destination (H_{21_1}).	137
6.14 SmartNIC setup.	139
6.15 Netronome Agilio CX 2x10GbE SmartNIC. Source: https://www.netronome.com/products/agilio-cx/	139
6.16 Timestamps header.	140
6.16.aSourcey.	140
6.16.bPolKA.	140
6.17 Comparison of test cases for Sourcey and Sourcey Baseline scenarios.	142
6.17.aLow throughput and small packets.	142
6.17.bLow throughput and big packets.	142
6.17.cHigh throughput and small packets.	142
6.17.dHigh throughput and big packets.	142
6.18 Comparison of PolKA and PolKA Baseline test cases.	143
6.18.aLow throughput and small packets.	143
6.18.bLow throughput and big packets.	143
6.18.cHigh throughput and small packets.	143
6.18.dHigh throughput and big packets.	143
6.19 Comparison of Sourcey and PolKA test cases.	144
6.19.aLow throughput and small packets.	144
6.19.bLow throughput and big packets.	144
6.19.cHigh throughput and small packets.	144
6.19.dHigh throughput and big packets.	144
7.1 SFC edge pipeline.	147

7.2	KeySFC with PolKA: example SFC.	149
7.3	SFC tests for increasing chain length: comparison between Sourcey and PolKA solutions.	151
7.3.a	SFC description.	151
7.3.b	RTT.	151
7.3.c	Jitter.	151
7.3.d	FCT.	151
7.4	SFC tests for increasing number of hops per SFC segment: comparison between Sourcey and PolKA solutions.	152
7.4.a	SFC description.	152
7.4.b	RTT.	152
7.4.c	Jitter.	152
7.4.d	FCT.	152
7.5	KeySFC with PolKA: SFC migration example.	153
7.6	Migration of UDP flow from SFC 1 to SFC 2 in KeySFC with PolKA.	154
7.6.a	Throughput at source (VM_S).	154
7.6.b	Throughput at S_3	154
7.6.c	Throughput at S_{10}	154
7.6.d	Throughput at destination (VM_D).	154
7.7	Fast failure reaction for SFC $VM_S \rightarrow VNF \rightarrow VM_D$: (a) topology, (b) unprotected path for segment $VNF \rightarrow VM_D$, and (c) protected path for segment $VNF \rightarrow VM_D$	155
7.7.a	155
7.7.b	155
7.7.c	155
7.8	UDP test for unprotected path: results for failure of link S_4 - S_6	156
7.8.a	Throughput at source (VM_S).	156
7.8.b	Throughput at destination.	156
7.9	UDP test for protected path: results for failure of link S_4 - S_6 with the deflections of the fast failure reaction mechanism.	157
7.9.a	Throughput at source (VM_S).	157
7.9.b	Throughput at destination (VM_D).	157
7.9.c	Throughput at S_3 (Port 4).	157
7.9.d	Throughput at S_5 (Port 2).	157
7.9.e	Throughput at S_7 (Port 3).	157
7.10	TCP test for protected path: results for failure of link S_4 - S_6 with the deflections of the fast failure reaction mechanism.	158
7.10.a	Throughput at source (VM_S).	158
7.10.b	Throughput at destination (VM_D).	158
7.10.c	Throughput at S_3 (Port 4).	158
7.10.d	Throughput at S_5 (Port 2).	158
7.10.e	Throughput at S_7 (Port 3).	158
A.1	Traditional networking architecture versus OpenFlow architecture. Source: [Sherwood et al. 2009].	180
A.2	OpenFlow architecture. Source: [Mckeown et al. 2008].	181
A.3	PISA architecture. Source: [P4.org 2018b].	182
A.4	Example of P4 workflow. Source: [P4.org 2017].	183

A.5	Lookup table in P4. Source: [P4.org 2017].	183
A.6	OpenStack Diagram. Source: https://www.openstack.org/software/	185
A.7	Networking of compute node in OpenStack. Source [Dominicini et al. 2017].	186
B.1	VNF-FGE model example.	187
B.2	(a) Comparison between the Fat-Tree and the Hypercube: No. of links vs. no. of servers, and CAPEX vs. no. of servers. (b) Results from ILP model: no. of provisioned VNF instances, network utilization, and processing capacity utilization vs. traffic factor.	190
B.2.a	190
B.2.b	190

List of Tables

2.1	Routing methods classification. Adapted from [Abts and Kim 2011].	40
3.1	Comparison between SFC related works.	64
3.2	Related works and requirements.	67
4.1	Flow table at switch S1.	78
4.2	Flow rules for SFC example.	81
4.3	Flow Rules for SFC Test.	84
5.1	Simplified flow tables at <i>Eswitches</i>	93
5.2	Flow entries at S1 for paths 1 and 2.	101
6.1	Maximum $len(R)$ for example topologies.	111
6.2	Maximum $len(R)$ for different unicast SSR mechanisms in DC topologies.	112
6.3	Sourcey: example lookup table.	119
6.4	PolKA: example lookup table.	120
6.5	Emulated setup: software versions.	129
6.6	P4 programs: Source lines of code (SLOC).	131
6.7	Sourcey: Table entries at edge switch E_{11} for destination H_{21_1}	136
6.8	PolKA: Table entries at edge switch E_{11} for destination H_{21_1}	136
6.9	Physical setup: software versions.	140
6.10	SSR headers based on destination IP address	141
6.11	SmartNIC test scenarios: deployed P4 programs.	141
7.1	Table entries in edge switches for example SFC.	148
7.2	Table entries in edge switches before SFC migration.	153
7.3	Table entries in edge switches after SFC migration.	153

List of codes

6.1	Sourcey: example P4 code for defining headers.	117
6.2	PolKA: example P4 code for defining headers.	118
6.3	Sourcey: example P4 code for declaration of lookup table.	119
6.4	Sourcey: example P4 code for declaration of an action.	119
6.5	PolKA: example P4 code for declaration of a lookup table.	120
6.6	PolKA: example P4 code for declaration of an action.	120
6.7	PSA: hash extern and supported hash algorithms. Source: [P4.org 2019b] .	125
6.8	v1model: hash extern and supported hash algorithms. Source:[P4.org 2019a]	126
6.9	Example P4 code for calculating the output port using CRC operation. . .	126
6.10	Commands of switch CLI used by control plane application.	129
6.11	Usage example of <code>simple_switch_CLI</code> command for modifying table entry.	136
7.1	P4 code for table declaration.	147
7.2	P4 code for action declaration.	148

Contents

1	Introduction	22
1.1	Research problem	23
1.1.1	SFC workflow	24
1.1.2	VNF-FG embedding and deployment gaps	25
1.1.3	Problems of current SFC solutions	28
1.1.4	Research question	29
1.1.5	Hypotheses	29
1.2	Scope	31
1.3	Objectives	32
1.4	Proposal and contributions	33
1.5	Text Structure	35
2	Background	37
2.1	Data center networking	37
2.1.1	Network-centric DCNs	38
2.1.2	Server-centric and Hybrid DCNs	38
2.2	Routing	40
2.2.1	RNS-based SSR	42
2.3	Edge Computing	44
2.4	NFV	45
2.4.1	Concepts and objectives	45
2.4.2	ETSI NFV reference architecture	46
2.5	SFC	48
2.6	Concluding remarks	50
3	Comparison with the state of the art	51
3.1	Discussion on the SFC underlay layer	51
3.1.1	Limitations of current network infrastructures	51
3.1.2	Topology: Network-centric vs. server-centric DCNs	53
3.1.3	Routing: per-hop table-based vs. algorithmic SSR	55
3.2	Related works	58
3.2.1	RNS-based SSR	58
3.2.2	NFV Orchestration	60
3.2.3	SFC	63
3.3	Concluding remarks	67

4	VirtPhy: NFV Orchestration Architecture	68
4.1	Proposal	68
4.1.1	Challenges, enablers, and design principles	69
4.1.2	Architecture design	70
4.1.3	NFV orchestration architecture	72
4.1.4	NFV in a server-centric infrastructure	73
4.2	Proof-of-concept	75
4.2.1	The hypercube topology and routing mechanism	75
4.2.2	The testbed	76
4.2.3	SFC using SDN	77
4.2.4	SFC in OpenStack and Hypercube	79
4.3	Evaluation	82
4.4	Concluding remarks	87
5	KeySFC: SFC Scheme	89
5.1	Proposal	89
5.1.1	Architecture	89
5.1.2	KeySFC underlay routing design	90
5.1.3	How does KeySFC work?	91
5.1.4	KeySFC control plane	94
5.2	Proof-of-concept and evaluation	95
5.2.1	Prototype	95
5.2.2	Reference SFC scenarios	96
5.2.3	Functional test	97
5.2.4	Performance tests	98
5.2.5	Traffic engineering enabled by KeySFC	100
5.3	Multi-domain SFC	103
5.4	Concluding remarks	104
6	PolKA: SSR Mechanism	105
6.1	Proposal	105
6.1.1	Mathematical background	105
6.1.2	Unicast source routing	107
6.1.3	Multicast source routing	109
6.1.4	Scalability of the bit length of the <i>routeID</i>	110
6.1.5	Control plane	113
6.2	Design based on P4 architecture	114
6.2.1	Sourcey pipeline	114
6.2.2	PolKA pipeline	115
6.2.3	Headers and lookup tables	117
6.2.4	Discussion on fabric networks and pipeline differences	121
6.2.5	Reuse of CRC hardware for implementing <i>modulo</i>	123
6.3	Proof-of-concept and evaluation	127
6.3.1	Emulated setup with Mininet and bmv2	127
6.3.2	Physical setup with SmartNICs	138
6.4	Concluding remarks	144

7	Integration of PolKA and KeySFC	146
7.1	Design	146
7.2	Proof-of-concept and evaluation	150
7.2.1	Linear topology	150
7.2.2	Programmable, expressive, scalable, and agile migration	152
7.2.3	Exploitation of RNS properties: fast failure reaction	154
7.3	Concluding remarks	159
8	Conclusion	160
8.1	Conclusions	160
8.2	Future works	162
8.3	Publications	163
	Bibliography	166
A	Enabling technologies	179
A.1	SDN	179
A.1.1	OpenFlow	180
A.1.2	P4 language	182
A.1.3	Open vSwitch (OvS)	184
A.1.4	Mininet	184
A.2	OpenStack	184
A.2.1	Networking	185
B	Comparison between the Fat-Tree and the Hypercube topologies	187
B.1	The optimization model	187
B.2	Tradeoff analysis based on port cost	189
B.3	Tradeoff evaluation for growing traffic demands	190
B.4	Analysis	191

Chapter 1

Introduction

Network functions virtualization (NFV) has attracted a lot of attention in recent years as a potential solution for reducing costs and providing scalable network services [Mijumbi et al. 2016]. In parallel, emerging trends in telecommunication networks, such as edge computing, internet of things, smart cities, and industry 4.0, portend new types of services and transfer processing power from large remote data centers (DCs) to distributed DCs at the edge of the network [Martini et al. 2015, Mao et al. 2017].

Current DC networks are composed by a large set of network functions that can be connected in order to provide a network service. Most of the times, this composition of network functions is done in a static way, but NFV enables dynamic methods to manage and combine a set of service functions (SFs). Thus, the adoption of NFV in edge data centers brings new challenges, because this shift requires the ability to provide NFV solutions that can efficiently orchestrate a large number of dynamic services in a resource-constrained DC.

In this scenario, one of the fundamental challenges in NFV is how to enable the composition of customized services by steering a large number of flows across a set of SFs to provide service function chaining (SFC) [Halpern and Pignataro 2015]. It is worth noting that SFC changes the traditional end-to-end routing paradigm in DC networks (DCNs), because it requires the dynamic insertion of virtualized nodes between source and destination: packets from source are steered to a host and delivered to a virtualized SF to be processed; then, packets return to the network and are routed to the next SF, repeating these steps until they reach the destination [Dominicini et al. 2017]. This is a far more complex challenge than routing between two end nodes, since it involves capturing, classifying, and steering the traffic for each virtualized SF.

To provide composite services, a service overlay topology is built "on top" of the existing network underlay topology [Quinn and Nadeau 2015]. To ensure such services achieve maximum performance, the network controller or distributed agents in charge of traffic engineering should be able to select among all possible paths in the underlay network according to dynamic demands [Jyothi et al. 2015], when deciding the routing

paths between SFs.

However, there are fundamental problems in traditional SFC approaches that compromise efficient network orchestration. Firstly, most of the current works underestimates the importance of the network infrastructure that interconnects the physical servers hosting the SFs. Indeed, they normally perceive the network as a mere way to provide connectivity for the overlay, but the topology has great impact on network orchestration. In this way, the SFC solutions are frequently decoupled from the routing decisions on the underlay [Quinn et al. 2018], since they leave to the routing mechanisms the responsibility of deciding how to deliver packets between SFC segments. Secondly, routing mechanisms of traditional DCNs are not designed for these dynamic and composite service requests, because they are usually complex, rigid, and subject to large propagation delays of control information [Martinello et al. 2014, Soliman et al. 2012, Bhamare et al. 2016]. Thirdly, the number of states is limited by the size of forwarding tables in switches [Jin et al. 2016], and the traffic engineering is usually restricted to a set of shortest paths between SFC endpoints. This may prevent the orchestrator to select non-shortest paths for avoiding congestion or faulty paths [Tso et al. 2016].

Therefore, the traffic engineering of SFC requests is restricted to sub-optimal solutions, because current SFC mechanisms do not allow: to explore all the network capacities of the underlay topology when mapping the service overlay; and to explicitly select amongst all the existing paths per chain segment and agilely modify these paths for variable demands.

On the other hand, emerging networking architectures and equipment are allowing for softwarization of DCNs and flexible dataplane programmability at line rate, both at the switch and the network interfaces, opening up unprecedented opportunities for the development of innovative networking solutions [McCauley et al. 2019, P4.org 2017].

In this context, this thesis investigates a SFC solution that can provide greater synergy between the NFV orchestration and the underlying DC infrastructure. This section describes the limitations of current SFC mechanisms of the network underlay, defines the research problem, and presents our proposal to build a SFC solution that solves this problem.

1.1 Research problem

This section describes the SFC workflow, the problem of embedding SFC requests in the network infrastructure, and the existing gaps in the SFC mechanisms of the underlay network to deploy state-of-the-art optimization solutions for NFV orchestration. Finally, it defines the research question and hypotheses investigated by this thesis.

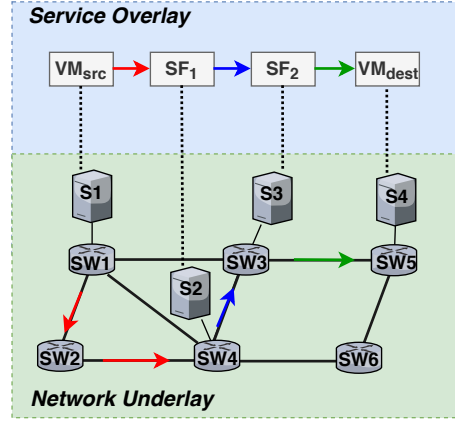


Figure 1.1: SFC embedding: service overlay mapped into underlay network.

1.1.1 SFC workflow

Fig. 1.1 shows the embedding of a SFC request in the NFV Infrastructure using the concept of service overlay layer (or simply overlay) and network underlay layer (or simply underlay).

The overlay is represented by a graph that defines the logical connections between VMs in SFC. This graph is known as the *virtual network function forwarding graph* (VNF-FG) that is the representation of a connection chain of SFs in which the order is important. The underlay is represented by a topology graph with links between physical nodes. We consider that each hop in the overlay is an SFC segment, which can be mapped to zero or more hops in the underlay. Thus, an end-to-end network service can be described as a VNF-FG interconnected by a network infrastructure underlay [ETSI NFV ISG 2014].

For example, Fig. 1.1 shows a VNF-FG ($VM_{src} \rightarrow SF_1 \rightarrow SF_2 \rightarrow VM_{dest}$) that has three SFC segments: $VM_{src} \rightarrow SF_1$, $SF_1 \rightarrow SF_2$, and $SF_2 \rightarrow VM_{dest}$. The virtual machines (VMs) of this chain are placed into servers S1, S2, S3 and S4, respectively. The first segment is mapped to two hops in the underlay network ($SW1 \rightarrow SW2$ and $SW2 \rightarrow SW4$), the second segment is mapped to a single hop in the underlay network ($SW4 \rightarrow SW3$), and the third segment is mapped to a single hop in the underlay network ($SW3 \rightarrow SW5$).

Fig. 1.2 shows how the main components of the NFV architecture interact in the SFC workflow [ETSI NFV ISG 2014, Zhang et al. 2018] (more details about the ETSI NFV reference architecture on Section 2.4.2). Initially, the Orchestrator receives service requests from customers and the service modeling generates SFC description models for the VNF-FGs. After that, the NFV Orchestrator gathers information about the physical infrastructure, including server resource usage and network status, to define the available capacity from processing nodes and network. Based on this information, the NFV Orchestrator takes two decisions: (i) placement: deciding the optimal location of SFs in physical servers, considering service requirements and resource constraints [Mijumbi et al. 2016]; and (ii) SFC: deciding on how to steer the traffic flows across an ordered set of SFs that

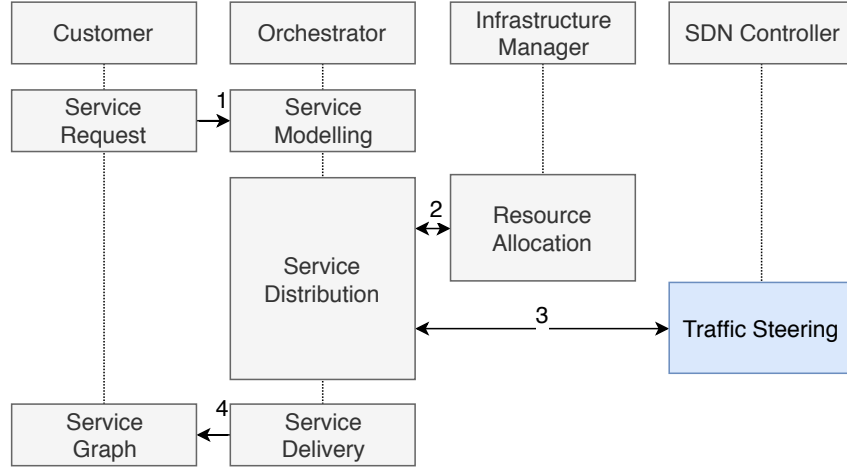


Figure 1.2: SFC workflow. Adapted from [Zhang et al. 2018].

compose the service [Halpern and Pignataro 2015].

Then, these decisions are distributed to the Infrastructure Manager for resource allocation, which includes the placement of SFs and determination of paths between them. In the next step, SFC descriptions and the selected paths are distributed to the software-defined networking (SDN) controller, which translates the policy requirements into rules for traffic steering in the data plane. The traffic steering mechanisms refer to the operations involved in directing the traffic to reach the intermediate SFs of a specific chain, and consider two routing levels [Hantouti et al. 2018]: (i) an overlay routing between SFC elements, and (ii) an underlay routing to ensure network reachability (e.g., IP, and MPLS). Finally, the service is fully provisioned for the customers.

1.1.2 VNF-FG embedding and deployment gaps

The embedding of SFC requests in the network infrastructure, as described in the last section, is known in the literature as the VNF-FG embedding (VNF-FGE) problem, which is \mathcal{NP} -hard [Herrera and Botero 2016, Luizelli et al. 2017].

Consider the following notation for the VNF-FGE problem: \mathbf{n} is a physical server node; \mathbf{t} is a SF type; \mathbf{v} is a SF instance of type \mathbf{t} in a physical node \mathbf{n} ; \mathbf{r} is a service request; and $\mathbf{p} \in \mathbf{P}(\mathbf{r})$ is a path in the set of possible paths to provision a request \mathbf{r} . A service request $\mathbf{r} = \langle \mathbf{s}(\mathbf{r}), \mathbf{d}(\mathbf{r}), \mathbf{T}(\mathbf{r}), \mathbf{b}(\mathbf{r}) \rangle$ consists of the source node $\mathbf{s}(\mathbf{r})$, the destination node $\mathbf{d}(\mathbf{r})$, the bandwidth demand $\mathbf{b}(\mathbf{r})$, and a service chaining containing the sequence of SF types through which the traffic must pass, $\mathbf{T}(\mathbf{r})$ (e.g., $\mathbf{T}(\mathbf{r}) = t_1 \rightarrow t_2 \rightarrow t_3$).

The VNF-FGE problem consists in allocating the instances \mathbf{v} in the physical hosts (placement decision) and defining the paths \mathbf{p} (SFC decision) for a set of service requests in order to find an optimal solution according to a specific objective function (e.g., minimize the number of SF instances [Luizelli et al. 2015], and minimize OPEX [Bari et al. 2015]).

A example of the VNF-FGE problem is shown in Fig. 1.3 for a topology composed by

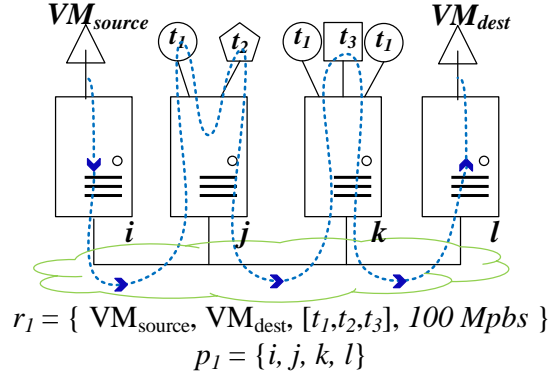


Figure 1.3: VNF-FGE example.

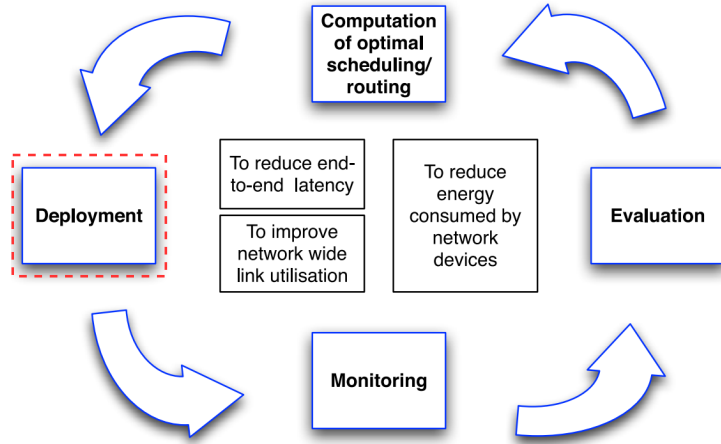


Figure 1.4: Traffic engineering workflow. Adapted from: [Tso et al. 2016].

four interconnected physical server nodes: i , j , k , and l . It abstracts the interconnection topology between the servers, represented as a green cloud in the figure. The service request r_1 defines a 100 Mbps traffic originated in virtual node VM_{source} (hosted at server i) and addressed to virtual node VM_{dest} (hosted at server l). This traffic must pass through a SF instance of type t_1 , then, through a SF instance of type t_2 , and finally, through a SF instance of type t_3 , before it reaches the final destination. In this example, an instance of t_1 and a instance of t_2 (both allocated in server j), and a instance of t_3 (allocated in server k) will serve this request. Thus, in this example, the NFV Orchestrator chose the path $p_1 = \langle i, j, k, l \rangle$, considering the servers.

The Orchestrator runs a VNF-FGE algorithm that makes embedding decisions, according to the defined optimization objectives [Herrera and Botero 2016]. One of its responsibilities is to perform traffic engineering, which selects the paths (or routes) for each SFC segment for efficient resource usage. Fig. 1.4 illustrates a typical traffic engineering workflow, consisting of a control loop with the following steps [Tso et al. 2016]: (i) monitoring and evaluation of metrics of interest; (ii) based on the monitoring information, the selected VNF-FGE algorithms perform computation of an optimal resource usage solution; (iii) the solution is deployed on the network infrastructure.

Many important works have investigated optimization models and heuristic algorithms to solve the VNF-FGE problem using simulations [Bhamare et al. 2016, Herrera and Botero 2016], considering different optimization objectives (e.g., minimize OPEX [Bari et al. 2015], the number of SF instances [Luizelli et al. 2015], or deployment costs). In [Bari et al. 2016] and [Luizelli et al. 2015], the authors formalized the placement and chaining problem and proposed an Integer Linear Programming (ILP) model and a heuristics to solve the problem. Other works developed models and heuristics to tackle the scalability of this problem for larger scenarios [Mechtri et al. 2016, Luizelli et al. 2017, Beck and Botero 2017]. Another relevant approach is the study of the VNF-FGE problem in multi-domain or multi-cloud scenarios [Bhamare et al. 2017, Sun et al. 2019].

Moreover, DC traffic presents significant workload variation in short timescales due to the start and end of user requests in a shared infrastructure [Tso et al. 2016]. To adapt to such dynamic workloads, some works also developed solutions to enable the reallocation of previously determined placement and chaining decisions according to NFV demands [Leivadeas et al. 2017, Draxler et al. 2018, Miotto et al. 2019].

These optimization solutions require deployment options that allow the selection of any available resource and enable the exploitation of the rich path redundancy of the underlying DC network. For instance, the traffic engineering may need to select a set of paths and load-balance between them, choose non-shortest paths for avoiding congestion or faulty paths, or quickly change paths to adapt to highly variable demands [Jyothi et al. 2015, Tso et al. 2016]. Also, to ensure that the actual resource usage is compatible with the resource allocation models, the traffic engineering must be able to specify each networking element in the path for each SFC segment.

In summary, the SFC mechanisms in the underlay should meet the following requirements in order to enable the deployment of the optimization models for traffic engineering:

- **Programmable:** Flow control is decoupled from network hardware and managed by software using application programming interfaces (APIs).
- **Expressive:** It is possible to select any available path between two endpoints, and specify each forwarding element in a path between two endpoints.
- **Scalable:** It supports the encoding of a diverse set of paths between each source-destination pair for the SFC segments and minimizes the control plane overhead.
- **Agile:** It allows quick changes to the paths, considering the convergence time to apply these changes in all the affected nodes.

However, while many optimization solutions have been proposed in the literature, much less attention has been dedicated to the development of the underlay SFC networking mechanisms that enable the deployment of such resource allocation solutions

in the network infrastructures of DCs (highlighted in red in Fig. 1.4). According to [Bhamare et al. 2016], the optimization studies for SFC lack practical values and need modifications to suit to the SFC architecture, while the problem of deployment of dynamic function chains is an open challenge that needs to be addressed. In [Hantouti et al. 2018], authors present a comprehensive survey of traffic steering techniques for SFC and conclude that current solutions are not efficient enough to be deployed in real-life networks, mainly due to scalability and flexibility limitations. The next section details the problems of current solutions for SFC deployment.

1.1.3 Problems of current SFC solutions

As discussed in the previous section, a SFC solution has to allow the network orchestration to react in a timely and flexible manner to workload variation and should not restrict the traffic engineering on the selection of paths for the SFC segments. In this way, the network orchestration can make optimal use of the underlay resources according to dynamic demands. **Nevertheless, current SFC mechanisms do not offer the levels of programmability, expressiveness, scalability, and agility that are necessary to deploy the current state-of-the-art resource allocation solutions in the network underlay.**

The first problem is that many existing SFC mechanisms consider the network underlay that interconnects the servers in Fig. 1.3 as a mere way to provide connectivity to the service overlay layer. They normally do not offer adequate mechanisms for traffic engineering to define the paths between SFC segments, and sometimes leave the responsibility of selecting paths to routing methods that are separated from the SFC decisions in the overlay. For instance, in the network service headers (NSH) protocol [Quinn et al. 2018], the decisions for routing (in the underlay) and SFC (in the overlay) are completely decoupled and executed by different mechanisms.

The second problem is that, even when the SFC solutions allow the traffic engineering to specify all networking nodes in a path, their underlay routing mechanisms have limited scalability and require the insertion of additional constraints in the resource allocation models. For example, traditional flow-based methods for SFC install a large number of flow rules in the switches to steer traffic and the number of states is limited by the size of forwarding tables [Jyothi et al. 2015]. Therefore, the path options are commonly restricted to a single shortest path or a small set of shortest paths between each pair of segment endpoints. Another example is the Segment Routing protocol [Clad et al. 2018], which can enable SFC over a MPLS infrastructure, but restricts the maximum number of nodes that can be specified in a path depending on the MPLS equipment (about 3 or 5 [Abdullah et al. 2019]) and delegates the routing between these nodes to the underlying network mechanisms. In this way, the traffic engineering algorithms have to

take into account specific constraints of path encoding as additional objective functions [Moreno et al. 2017], which may lead to inefficient traffic distribution and network congestion [Abdullah et al. 2019].

The third problem is that most SFC solutions, such as traditional flow-based methods for SFC and the NSH protocol, rely on rigid table-based routing mechanisms in the underlay. Thus, a path migration may involve changing table entries in all the forwarding elements of that path, which may lead to control plane overhead and long convergence time to configure changes. Therefore, it is difficult to provide agile path selection in response to highly dynamic traffic requests.

The fourth problem is that SFC mechanisms normally only support network infrastructures that employ dedicated network equipment (e.g., switches and routers) to forward traffic, which are known as network-centric topologies. Thus, they miss an opportunity to exploit SFC in server-centric or hybrid topologies, where servers are directly interconnected and perform both forwarding and processing tasks [Dominicini et al. 2017].

The described problems have direct impact on NFV orchestration for large data centers, and are also present in smaller infrastructures used for edge DCs, where resource constraints are more pronounced and efficient resource usage is essential. Moreover, edge DCs require low cost solutions that can be deployed with commercial off-the-shelf (COTS) equipment (low capital expenses) and reduced operating costs related to configuration of forwarding states (low operational expenses).

1.1.4 Research question

To tackle the aforementioned problems, this thesis aims to address the following question:

- How to design a **programmable, expressive, scalable, and agile SFC solution** that enables dynamic and efficient orchestration of the network underlay?

More specifically, this thesis investigates this research question considering the scope of **edge data centers with COTS equipment**, as described in Section 1.2.

1.1.5 Hypotheses

To investigate the defined research question, we formulated the following hypothesis:

- **Algorithmic forwarding can replace forwarding tables in the design of scalable SFC solutions.** Differently from table-based methods, algorithmic forwarding defines the output port based on information about the current and destination nodes using a fixed logic [Abts and Kim 2011]. Thus, this approach can reduce the number of forwarding states for SFC. The routing algorithm can be specific of some topologies [Chen et al. 2011], such as the XOR operation in the hypercube

[Dominicini et al. 2017], or a topology-independent algorithm, such as the residue number system (RNS) [Martinello et al. 2014].

- **Using strict source routing (SSR), it is possible to specify any topological path between SFC segments and design agile and expressive SFC solutions.** Differently from per-hop methods, in SSR, the responsibility of defining the route belongs to the source of packets (or edge nodes), which can specify all the elements of the path to destination [Sunshine 1977, Filsfils et al. 2018]. The route information can be inserted in the packet header, and used by each node to define the output port. Thus, it reduces the control plane overhead to create and modify paths, and can achieve optimal throughput performance when compared to per-hop approaches [Guo et al. 2010, Soliman et al. 2012, Martinello et al. 2014, Filsfils et al. 2015, Jyothi et al. 2015, Jin et al. 2016].
- **The residue number system (RNS) can be used in the design of SFC solutions to provide SSR with algorithmic forwarding.** The RNS is a number system based on the Chinese remainder theorem (CRT) [Chang et al. 2015]. The route information is represented as a number according to RNS and the nodes receive identifiers as pairwise co-prime numbers. Using this mechanism, the output port in each node can be directly calculated by the *modulo* (remainder of division) of the route identifier of the packet by the node identifier, without using tables [Martinello et al. 2014]. In addition, the properties of RNS can provide some special features, such as fast failure reaction and packet forwarding without header modification (see Section 3.2.1).
- **SDN and fabric paradigms offer a programmable solution for the automation and control of SFC.** SDN solutions promote the separation between the control plane and the data plane, allowing a logically centralized controller to dynamically manage and program network elements [Farhady et al. 2015]. In this approach, a controller deploys, in software, several networking functionalities, such as traffic engineering, network monitoring, and routing algorithms [Kotronis et al. 2012]. Another important concept that has been applied to SDN is the idea of fabric networks [Casado et al. 2012, Martinello et al. 2014], which separates edge and core network elements. The edge provides flexible and complex network services, while the core is only responsible for basic and efficient packet transport. This approach can be extended to intra-DC server-based networking, more specifically to the SFC problem with the use of SDN-enabled software switches in the edge and algorithmic forwarding in the core.
- **SFC solutions can be implemented in any DCN and take benefit from topology characteristics to execute traffic steering using COTS equip-**

ment. Most NFV solutions only focus on network-centric DCNs, neglecting the potential contribution of hybrid and server-centric DCNs. This work, on the other hand, argues that SFC solutions should support any DCN. In this way, the application can select the topology that better suits its requirements. Moreover, server-centric and hybrid topologies have been traditionally used in high performance computing (HPC) systems. Many of these topologies have their own efficient routing mechanisms, offer high path diversity, and can leverage high performance SFs while reducing capital expenditures (CAPEX), as they can be built using COTS servers [Li and Yang 2016, Li et al. 2016].

1.2 Scope

Although the solutions presented here could be applied to a wide variety of SFC use cases, such as core networks or multi-cloud scenarios, the focus of this thesis is to enable SFC in private or public edge data centers, as illustrated in Fig. 1.5. The selection of edge data centers delimits the following scope for the application of our SFC proposals:

- The solutions are designed for edge DCs based on COTS servers and switches.
- The scale depends on the processing requirements of the client edge applications, but we intend to cover DCs ranging from some dozen of servers (e.g., private enterprise edge DCs for specific purpose, such as Industry 4.0 or healthcare) up to hundreds of servers (e.g., distributed public DCs that represent edge clouds for Internet of Things applications, such as Smart Cities or Smart Homes).
- The topologies of these edge DCs may be network-centric, server-centric, or hybrid, as explained in Section 2.1.
- The main focus is a single DC scenario with several users (multi-tenancy) in a single administrative domain. The multi-cloud case where a service chain is distributed over servers hosted at different DCs and administrative domains is discussed in Section 5.3.
- SFs may include traditional network layer functions as well as application functions in upper OSI layers.
- Chains may contain cycles, as long as the physical paths of each SFC segment are loop free. For instance, we support chains where the flow passes more than one time through the same SF.
- We don't explore the specific case of chains with branches. Though, this can be studied in future works with the exploration of multicast in PolKA, as discussed in Section 8.2.

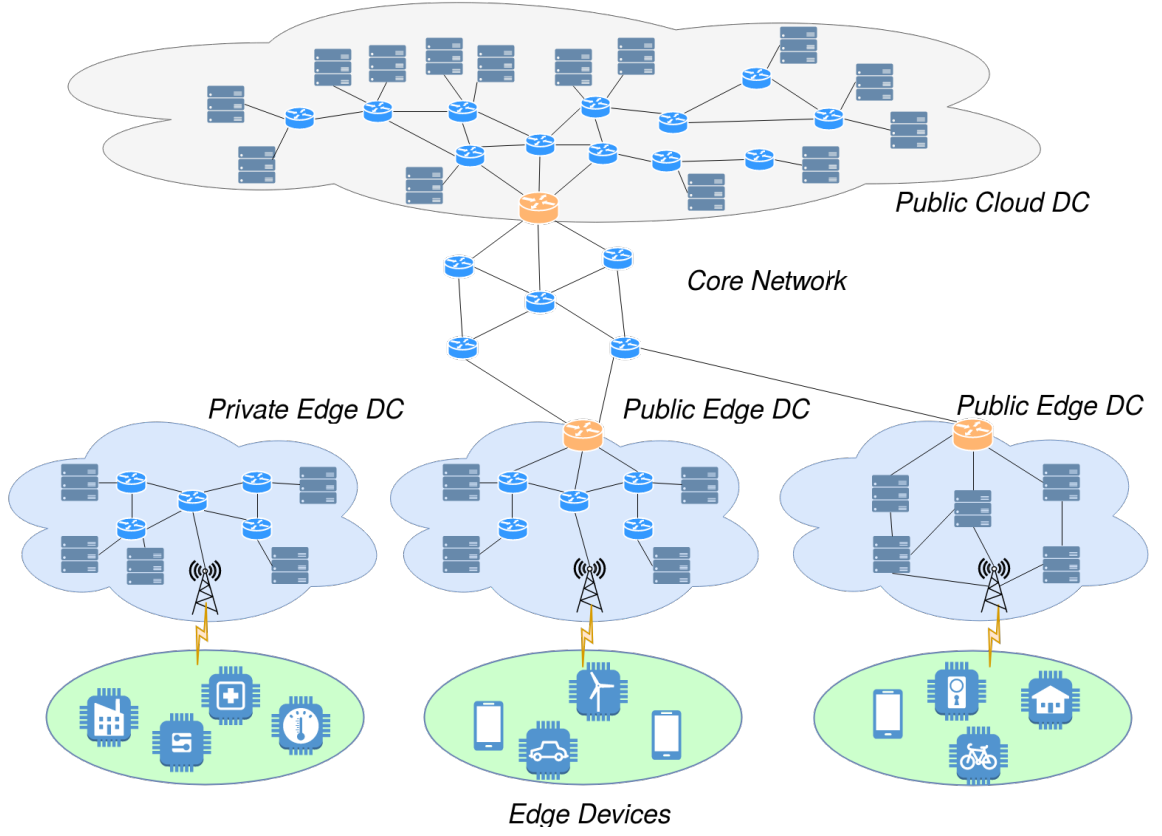


Figure 1.5: Example scenarios for edge data centers.

1.3 Objectives

In order to answer the defined research question and test the hypothesis, this work has the following general objectives:

- To develop a **NFV orchestration architecture** that supports server-centric topologies, and is interoperable with industry standards for NFV to perform SFC tasks.
- To develop an **SFC scheme** that supports any DCN (network-centric, server-centric, or hybrid), and is based on SSR, algorithmic forwarding, RNS, SDN, and fabric networks.
- To develop a efficient **RNS-based source routing mechanism** that can be implemented using COTS network hardware.

As a specific objective, this work aims to implement prototypes of all the proposed solutions to be used as proof-of-concept for functional and performance tests in a close-to-production environment.

1.4 Proposal and contributions

The first observation of this research was that current NFV solutions are tailored to network-centric DCNs. Thus, many options of COTS topologies based on server-centric or hybrid DCNs that could be adopted for low cost edge DCs were not yet exploited. The second observation was that traditional approaches to SFC base their routing mechanisms in tables, which restricts the way the traffic engineering can select and change paths for SFC in the underlay network.

Based on these observations, we proposed **VirtPhy**, a programmable NFV orchestration architecture based on server-centric networks. To this end, VirtPhy applied the concept of server-based network fabric to SFC: core software switches execute algorithmic forwarding, and SDN-enabled edge software switches use flow tables only for classifying SFC flows. The main contributions of VirtPhy are:

- We proposed how server-centric DCNs can fit in the NFV ETSI reference architecture and provide efficient SFC with algorithmic routing, while eliminating the need for tables and hardware switches.
- We explored routing mechanisms for SFC that consider characteristics of the underlay network topology (e.g., XOR routing in the hypercube topology), and can take advantage of the node position to place SFs in servers that are already part of the path between source and destination.
- We implemented and evaluated a hypercube prototype of VirtPhy using OpenStack and OpenFlow technologies, proving it is viable to deploy these ideas in production DCs. The software switches in the servers were implemented using Open vSwitch (OvS), and the datapath in core nodes was modified to perform algorithmic XOR routing.

VirtPhy offers a way to explore the specificities of server-centric DCNs, and their inherited routing algorithms that can offer efficient forwarding mechanisms. On the other hand, the solution cannot be implemented in topologies that do not provide their own routing algorithms. Furthermore, the choice of using per-hop algorithmic forwarding does not allow the full path specification by the traffic engineering mechanisms. So, the study of VirtPhy pointed out that a more expressive and flexible SFC scheme could be designed.

To explore the advantages brought by VirtPhy and tackle its limitations, we proposed a novel SFC scheme, called **KeySFC**, that explores SSR in SFC. To this end, it uses a topology-independent algorithmic forwarding mechanism based on RNS [Martinello et al. 2014]. For each SFC segment, SDN-enabled edge software switches classify flows for SFC and embed a RNS identifier in the packet header, which encodes the information about all the forwarding elements in the path. When packets reach the

(hardware or software) core switches, the forwarding engine executes a *modulo* operation over the path identifiers and the node identifiers to discover the output ports. In this way, it eliminates table lookups in core nodes, reducing the number of forwarding states and control plane overhead.

The contributions of KeySFC can be summarized as follows:

- We proposed an SFC scheme that supports the use of different topologies (server-centric, network centric, or hybrid).
- We proposed an SFC scheme that offers the following features to traffic engineering for each chain segment: (i) selection of any available path, (ii) specification of all the forwarding elements in the path, and (iii) agile path migration.
- We implemented and validated KeySFC in a proof-of-concept testbed orchestrated by OpenStack, demonstrating its feasibility in production DCs. Results for latency, jitter, and throughput indicate that the selected RNS-based SSR mechanism can provide efficient packet transport.

For KeySFC, we extended the VirtPhy prototype to add support for RNS-based SSR, and the necessary SDN control plane functionalities. The data plane implementation was accomplished by modifying the OvS datapath to perform the *modulo* operation.

However, the use of software switches in the core has performance limitations for scenarios that require very low latency and jitter. Besides, the *modulo* operation does not map to the instruction set of commodity network hardware. The work in [Martinello et al. 2017] demonstrated it is possible to achieve high performance by implementing KeyFlow in NetFPGAs, but it requires specialized hardware.

To meet both cost and performance requirements, we propose **PolKA**, a novel SSR approach that can be implemented in programmable network switches. To this end, we reinterpreted the original RNS-based SSR mechanism, based on integer arithmetic [Wessing et al. 2002, Martinello et al. 2014], to a binary polynomial arithmetic using Galois field (GF) of order 2 [Shoup 2009, Bajard 2007].

The main contributions of PolKA are:

- We brought RNS-based SSR expressiveness closer to elementary binary operations. The immediate benefit of this approach is to enable the reuse in RNS-based SSR of commodity embedded network functions that are based on polynomial arithmetic. Also, to the best of our knowledge, this is the first time a work applies the CRT theorem with Galois field to solve routing problems.
- We developed a technique that allows the execution of the polynomial *modulo* operation by reusing common CRC (cyclic redundancy check) operations, which are also based on GF(2) polynomials and supported by P4 standard architectures.

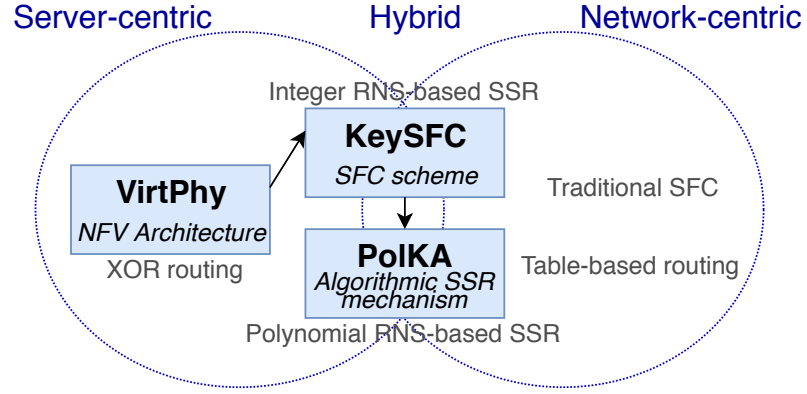


Figure 1.6: Overview of works proposed in this Thesis.

- We implemented the proposal in P4 emulated and hardware prototypes. The tests demonstrated that it is possible to take advantage of RNS properties using PolKA without losing performance when compared to traditional SSR methods.
- We integrated PolKA as the SSR mechanism of KeySFC scheme, demonstrating our SFC solution can be implemented in commercial P4-enabled network equipment.

In summary, this thesis proposes three interrelated solutions: (i) **VirtPhy**, a novel NFV orchestration architecture based on server-centric topologies (presented in Chapter 4); (ii) **KeySFC**, a topology-independent SFC scheme that uses RNS-based SSR and network fabric (presented in Chapter 5); and (iii) **PolKA**, an algorithmic SSR mechanism based on RNS and Galois Field polynomials that can be implemented over commodity network hardware (presented in Chapter 6).

The combination of these three solutions builds up a powerful framework to enable SFC in edge DCs. Fig. 1.6 shows an overview of works proposed in this thesis, and their correlation to traditional approaches.

1.5 Text Structure

The remainder of this work is structured as follows. In Chapter 2, we give some important background to understand this work. Then, Chapter 3 describes in more details the state of the art, and the comparison with related works. Chapter 4 presents our proposal and prototype for a NFV orchestration architecture, called VirtPhy. Chapter 5 proposes, implements and evaluates our SFC scheme based on SSR, named KeySFC. Then, Chapter 6 describes a proposal, called PolKA, for a novel SSR mechanism. Chapter 7 describes the integration between PolKA and KeySFC. Finally, we outline the conclusions and future works in Chapter 8.

Chapter 2

Background

This chapter describes some of the fundamental background and key concepts that are relevant to understand this work, including DCNs, routing, edge computing, NFV, and SFC. In addition, [Appendix A](#) provides a brief outline of the enabling technologies in SDN and cloud computing for the solutions proposed in this thesis.

2.1 Data center networking

DCs are the infrastructure that enables the operation of cloud computing platforms. They can host hundreds of thousands of physical servers and networking equipment that are located in the same environment due to common environmental, safety, and maintenance requirements [[Barroso et al. 2013](#)]. However, the rapid adoption of the cloud computing paradigm has brought several challenges for researchers and providers as current DC architectures still cannot fully meet the requirements of low cost, scalability, security, flexibility, resilience, quality of service, performance, and energy efficiency required by cloud computing [[Bilal et al. 2014](#)].

In this context, several of these challenges are directly related to the increasing demand for connection and bandwidth between the servers that compose the DCs as the processing capacity and the number of these servers increases. On one hand, it is possible to add processing and storage capacity to the DC by simply increasing the number of processing and storage elements. On the other hand, there is no direct solution to horizontally increase the capacity of networks [[Barroso et al. 2013](#), [Abts and Kim 2011](#)]. In addition, management and migration of virtual machines (VMs) bring new addressing and routing challenges, requiring network state to be changed dynamically.

Several architectures have been proposed to create efficient and scalable DC networks, such as Fat-Tree [[Al-Fares et al. 2008](#)], VL2 [[Greenberg et al. 2009](#)], Hypercube [[Saad and Schultz 1989](#)], DCell [[Guo et al. 2008](#)] and BCube [[Guo et al. 2009](#)]. These proposals can be classified as network-centric or server-centric architectures, according to the need for dedicated network equipment to forward traffic.

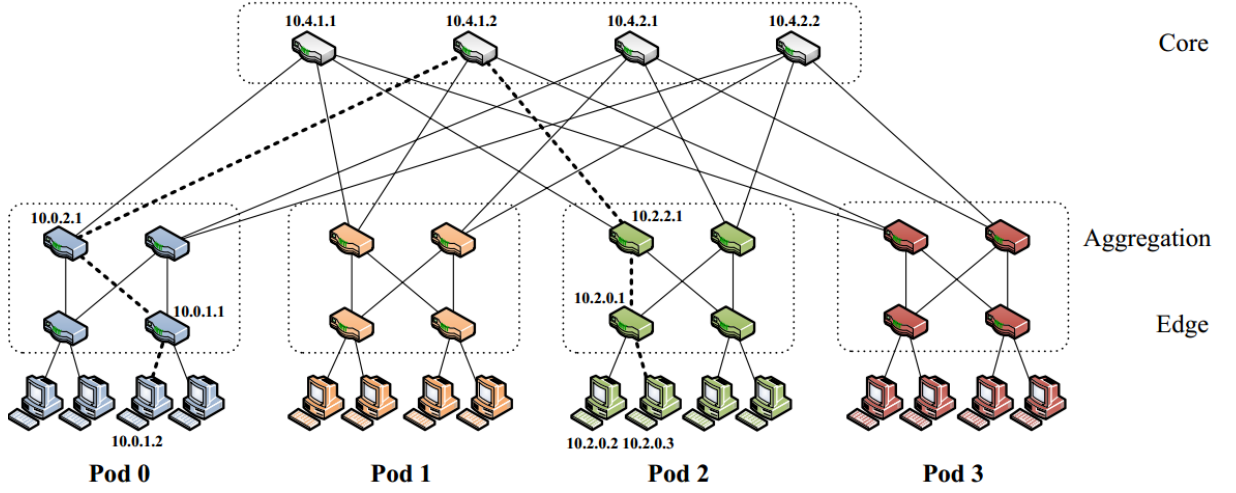


Figure 2.1: Example of fat-tree topology. Source: [Al-Fares et al. 2008].

The network topology defines how nodes are connected, and plays a central role in both the performance and cost of the network [Abts and Kim 2011]. In this work, we focus on DCNs that can leverage good performance for edge DCs at less cost by exploring commodity switches and servers.

2.1.1 Network-centric DCNs

Network-centric architectures use dedicated network equipment to forward network traffic. Traditional DCNs consists of a tree of routers or switches with progressively more specialized equipment in the top of the network hierarchy, such as three-tiered or two-tiered networks [Al-Fares et al. 2008]. The problem with these approaches is that they become very expensive, but do not deliver full aggregate bandwidth. To tackle these issues some solutions, such as Fat-tree[Al-Fares et al. 2008], Portland [Niranjan et al. 2009] and VL2 [Greenberg et al. 2009] have emerged with the objective to replace fewer larger and expensive switches by various small commodity switches.

Fig. 2.1 shows an example of a Fat-tree DCN with three levels: the edge switches that connect the servers to the aggregation switches, which, in turn, are connected to the core switches. This is typically how network-centric topologies, consisting of physical servers grouped in racks that are interconnected through top-of-rack (ToR) switches, can scale in number of servers and guarantee interconnectivity.

2.1.2 Server-centric and Hybrid DCNs

Server-centric architectures do not demand dedicated network equipment, and use servers for both forwarding and data processing tasks. Hypercube [Saad and Schultz 1989], Torus [Abts and Kim 2011], and Twin [Vassoler et al. 2014] are examples of server-centric topologies. Fig. 2.2 shows an example of a Twin and a Hypercube topology, both with 8 nodes

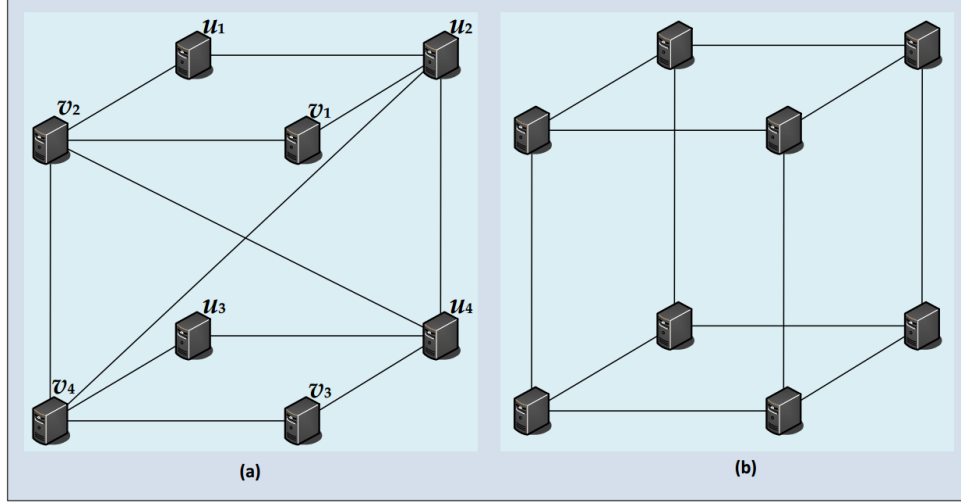


Figure 2.2: Example of server-centric topologies with 8 nodes and 12 links: (a) Twin, and (b) Hypercube. Source: [Vassoler 2015].

and 12 links.

The choice of network topology defines important aspects such as average and maximum hop count, fault tolerance, number of links per number of servers, and routing properties. For example, when compared to other traditional topologies, Twin topologies have a low link cost, and present special properties of resilience under both normal and faulty operating conditions [Vassoler et al. 2014]. Other topologies present inherent routing mechanisms that can be used to achieve better performance, such as the XOR mechanism in the Hypercube.

Other topology models are a hybrid of the basic network-centric and server-centric interconnection models, where forwarding tasks are performed by commodity servers and switches. Example of hybrid topologies are DCell [Guo et al. 2008], and BCube [Guo et al. 2009], which are able to reduce number of links and network diameter, respectively [Vassoler et al. 2014].

In the last years, server-centric networks have been relegated to a secondary role when DCs started to demand thousands to tens of thousands of computers. In this work, we show that server-centric topologies can be a suitable option for edge DCNs, which need high performance with low costs, but require a smaller scale of servers.

Another trend in this direction is server-based networking, which offloads the networking processing tasks to low cost networking co-processor devices (or SmartNICs¹) in the servers. With this approach, it is possible to save CPU cycles to process application tasks, while moving virtual switching tasks to these specialized devices.

¹<https://www.netronome.com/>

Table 2.1: Routing methods classification. Adapted from [Abts and Kim 2011].

Classification	Method	Description
Routing decision	Source routing	The routing path is determined at the source and the path computation only needs to be done once for each packet.
	Per-hop routing	At each hop enroute to the destination, the packet goes through path computation to determine the next hop.
Forwarding	Algorithmic	Based on the current node and destination information, a fixed logic can be used to determine the output port.
	Table-based	A lookup table can be implemented whose inputs are either the source and destination, and the table returns the appropriate output port.
Hop count	Minimal routing	Minimal number of hop count between source and destination is traversed. Depending on the topology, there might be multiple minimal paths.
	Nonminimal routing	The number of hop count exceeds the minimal hop count. The objective may be to increase path diversity and/or improve network throughput.

2.2 Routing

The routing algorithm defines the path a packet takes from source to destination, and explores the network capacities that can be potentially delivered by the topologies [Chen et al. 2011]. Table 2.1 shows a classification of routing methods according to different metrics [Abts and Kim 2011].

OpenFlow networks traditionally adopt per-hop routing with table-based forwarding, since a logically centralized SDN Controller distributes the forwarding states in the flow tables of the network switches. These flow tables contain entries that match flows to forwarding actions, and new flows trigger new state distribution to all the devices along the path [Soliman et al. 2012]. Regarding the hop count, the path is calculated by the SDN Controller, which can decide to choose the minimal routing or not, depending on traffic engineering aspects.

The distribution and management of forwarding states in the network bring some limitations to this approach, because the control plane signaling generates overhead traffic, and the switches in the path have to wait for a response from the Controller when a new flow arrives. Besides, table lookup is a time-consuming operation subject to high variability depending on the number of entries [Martinello et al. 2014]. In addition, the number of paths that can be represented is restricted to the size of forwarding tables.

On the other hand, **source routing (SR)** eliminates complex routing tasks from core nodes by placing the responsibility of route selection at source or edge nodes [Sunshine 1977], which add a route label in the packet header to specify the nodes in the routing path. In this way, SR can decrease forwarding table sizes, facilitate mainte-

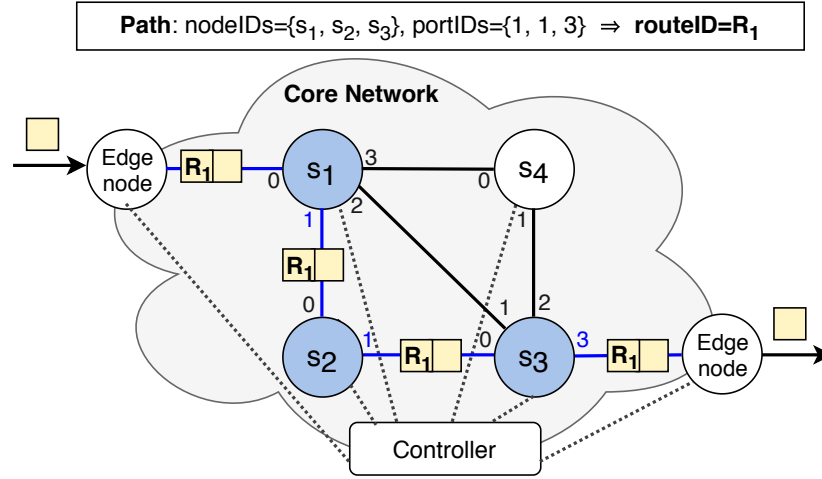


Figure 2.3: Tableless strict source routing example.

nance, reduce control plane overhead, and support optimal throughput performance when compared to per-hop table-based approaches [Martinello et al. 2014, Filsfils et al. 2015, Jin et al. 2016, Jyothi et al. 2015, Ren et al. 2018, Guo et al. 2010, Soliman et al. 2012, Stephens et al. 2011].

In some SR methods, such as multi-protocol label switching (MPLS), per-node forwarding is based on lookup tables whose input is the route label of the packet header (table-based forwarding). However, concatenating label-switched paths (LSPs) in MPLS demands interactions via label distribution protocol (LDP) [Rosen et al. 2001].

As an alternative, there are tableless SR methods where the forwarding is based on operations over the route label (**algorithmic forwarding**), rather than relying on table lookups, centralized controllers or communication with distributed peer nodes [Jin et al. 2016, Martinello et al. 2014]. For SDN systems, this also reduces the control signaling and latency related to path setup convergence [Soliman et al. 2012].

Also, SR methods can be classified as [Jin et al. 2016]: (i) **strict SR (SSR)**, if they specify the entire routing path, as in SecondNet [Guo et al. 2010]; or (ii) **loose SR**, if they determine only some hops that the packet must go through, as in Segment Routing [Filsfils et al. 2015]. In this work, we reference as **algorithmic SSR** the methods that use SSR with algorithmic forwarding.

Fig. 2.3 illustrates a generic algorithmic SSR approach in a fabric network with centralized control, composed by two edge nodes and four core nodes (s_1, s_2, s_3, s_4). When a packet arrives, the ingress edge node communicates with the controller, which gathers information about the network and calculates a route label (*routeID*) for that packet. In this example, the *routeID* R_1 maps a path specified by a set of nodes ($\text{nodeIDs} = \{s_1, s_2, s_3\}$) and their respective output ports ($\text{portIDs} = \{1, 1, 3\}$). The ingress node adds the *routeID* to the packet, and core nodes decide output ports based on an operation over this label. When the packet reaches an egress edge node, the route label is removed.

Port Switching is a conventional SSR method that implements algorithmic forwarding by representing the *routeID* as a stack of ports and the forwarding operation as a stack pop [Jin et al. 2016, Guo et al. 2010]. Other methods, as Segment Routing, represent the path to any destination as a list of segment addresses [Filsfils et al. 2015] and update the head pointer in each hop. In these methods, the list (or stack) is embedded by the ingress node in the packet header, and used by each node in the path to take its forwarding decision.

This thesis proposes to implement algorithmic forwarding by using a simple and efficient arithmetic operation between a path or destination identifier and a node identifier. This can be achieved by exploring specific routing algorithms provided by some topologies [Chen et al. 2011], such as the XOR operation in the hypercube [Dominicini et al. 2017] (adopted in the VirtPhy architecture in Chapter 4). Another alternative is the use of some topology-independent SSR algorithm, such as the *modulo* operation using the Residue Number System (RNS) [Martinello et al. 2014] (adopted in the KeySFC scheme in Chapter 5). The next section describes how to use RNS-based SSR.

2.2.1 RNS-based SSR

The residue number system (RNS) [Garner 1959] has been known as an alternative number system, based on the Chinese remainder theorem (CRT), that can satisfy certain critical constraints in performance, security, and power consumption for various applications, ranging from digital signal processors, cryptography, and networking [Chang et al. 2015]. It also appears as a promising way to enable SSR.

A RNS-based SSR mechanism represents the *routeID* as a number according to the RNS [Chang et al. 2015] and the *nodeIDs* as pairwise co-prime numbers. Then, the output port in each node is given by the *modulo* (remainder of division) of the *routeID* of the packet by its *nodeID*.

The logic for computing *routeIDs* exploits mathematical properties from RNS [Martinello et al. 2014]. Let $S = \{s_1, s_2, \dots, s_N\}$ be a multiset of the N *nodeIDs* of the core nodes on the desired path, in which all elements are pairwise co-prime numbers. Besides, let $P = \{p_1, p_2, \dots, p_N\}$ be a multiset of N outgoing ports, where p_i is the output port of the packet at the core node s_i . Also, let M be an upper bound value, defined as $M = \prod_{s_i \in S} s_i$.

Then, a natural number $0 \leq R < M$ can be represented by a *residue set* given a basis modulo set S :

$$R \xrightarrow{\text{RNS}} \{p_1, p_2, \dots, p_N\}_S, \quad \text{where } p_i = R \text{ modulo } s_i \quad (2.1)$$

To define a route, it is necessary to find out the value of R (the explicit *routeID*), given a modulo set S (the *nodeIDs*), and its RNS representation P (the core node output ports). The Chinese Remainder Theorem states that it is possible to reconstruct R through its

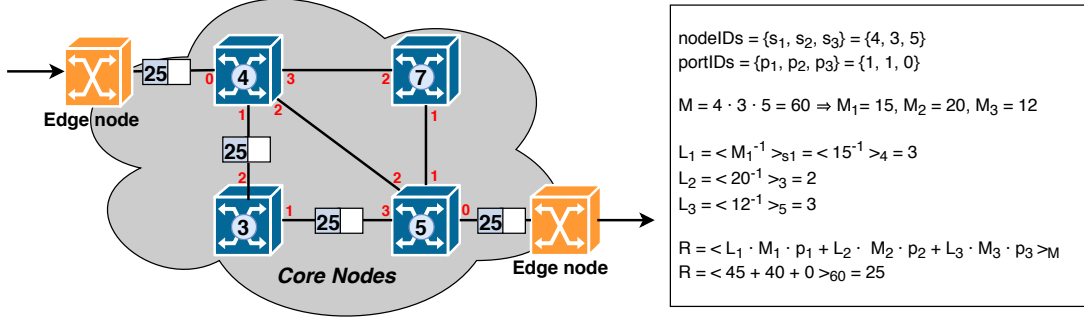


Figure 2.4: Example of *routeID* computation. Adapted from [Martinello et al. 2014].

residues in a RNS as follows, where $\langle a \rangle_b \equiv a \text{ modulo } b$ [Martinello et al. 2017]:

$$R = \langle \sum_{s_i \in S} p_i \cdot M_i \cdot L_i \rangle_M \quad (2.2)$$

$$M_i = M/s_i \quad (2.3)$$

$$L_i = \langle M_i^{-1} \rangle_{s_i} \quad (2.4)$$

Eq. (2.4) means that L_i is the modular multiplicative inverse of M_i . In other words, L_i is an integer number such that:

$$\langle L_i \cdot M_i \rangle_{s_i} = 1 \quad (2.5)$$

Fig. 2.4 shows an example of a *routeID* computation based on RNS. For this path, the *routeID* is 25, which binds the output ports $P = \{1, 1, 0\}$ to the nodes $S = \{4, 3, 5\}$, respectively. In fact, $\langle 25 \rangle_4 = 1$, $\langle 25 \rangle_3 = 1$, and $\langle 25 \rangle_5 = 0$.

The algorithm complexity for *routeID* computation is $\mathcal{O}(\text{len}(M)^2)$ [Shoup 2009], where $\text{len}(M)$ is the number of bits of the binary representation of M . It is important to note that the *routeID* is computed by the SDN Controller only when chains are created or updated, while core nodes only execute one *modulo* operation per packet. Also, in our scheme the *routeID* represents the path between two servers and does not grow with respect to the number of VMs and SFs, which is much larger and dynamic. Besides, the SDN Controller may proactively compute the *routeID* for the paths between a pair of servers that the NFV Orchestrator has decided to consider, and use this information when necessary to apply orchestration decisions.

As explained in [Martinello et al. 2014], the bit length of the *routeID* depends on: (i) the switch with the largest number of ports; (ii) the size of the core network; and (iii) the number of hops for the selected path. The works in [Martinello et al. 2017, Liberato et al. 2018, Ren et al. 2018, Ren et al. 2017] have already performed extensive scalability analysis of the RNS scheme and showed that is possible to use legacy small headers, such as MAC addresses, to encode the *routeID* for DCN topologies of reasonable size. In addition, [Ren et al. 2018, Ren et al. 2017] proposed methods to reduce the

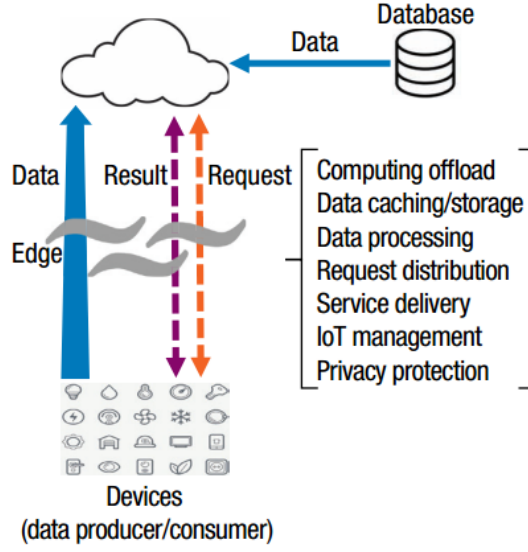


Figure 2.5: Edge computing paradigm. Source [Shi and Dustdar 2016].

length of the *routeID*.

2.3 Edge Computing

Edge computing is a paradigm in which substantial computing and storage resources, which also can be called cloudlets, micro data centers, or fog nodes, are placed close to mobile devices or sensors at the Internet’s edge to deliver highly responsive cloud services for mobile computing [Satyanarayanan 2017].

In edge computing, the cloud collects data from existing databases (as traditionally done), and also from end devices (e.g., sensors and mobile phones), which act as both data consumers and data producers, as shown in Fig. 2.5. The nodes at the edge DCs may perform many computing tasks, such as data processing, caching, service delivery, and privacy protection.

ETSI defines the term mobile edge computing (MEC) as a new platform that provides IT and cloud computing capabilities within the radio access network (RAN) in close proximity to mobile subscribers [ETSI MEC ISG 2014]. Moreover, Cisco proposed the concept of fog computing as a generalized form of MEC where the definition of edge devices is broader [Mao et al. 2017]. The terms mobile edge computing, edge computing, and fog computing are overlapping, and will be used interchangeable in the context of the present work.

In recent years, the interest in edge computing has grown dramatically as pointed out by the last Gartner’s report [Kasey Panetta 2017] on “Hype Cycle for Emerging Technologies”, which classified edge computing as “innovation trigger” with mainstream adoption expected by 2019-2022. As a result, we can foresee the emergence of this new kind of DC that presents huge differences from centralized cloud DCs, since edge DCs are geographi-

cally distributed, have much smaller scale in terms of resources, are located close to the end users, and can support latency-critical applications [Mao et al. 2017]. To operationalize innovative services in edge computing, the mobile network operators (MNOs) will need new technologies that enable the orchestration of services across many distributed DCs as opposed to their traditional centralized model.

2.4 NFV

This section presents the main concepts related to NFV that are essential for understanding this thesis.

2.4.1 Concepts and objectives

The presence of proprietary hardware-based network appliances, known as middleboxes, is a key part of the operation of today’s computer and telecommunications networks, supporting a diverse set of network functions such as firewalls, intrusion detection systems, load balancers, NAT, caches, and proxies [Martins et al. 2014]. To get an idea, in corporate networks, the number of these middleboxes is equivalent to the number of routers and switches deployed [Sherry et al. 2012].

However, the presence of these middleboxes brings several problems, such as [Martins et al. 2014, ETSI NFV ISG 2014, Han et al. 2015]: networks have become very complex with a wide variety of proprietary elements; the time to bring new services and functionalities to the market is high, as it depends on the production of new hardware; the operation of the networks is costly and depends on specialized knowledge in each proprietary platform; the costs of acquiring equipment to meet network demands are high, but they quickly reach obsolescence; lack of flexibility and scalability, as resources cannot be moved according to demands, and need to be scaled to the peak scenario; there are big barriers to innovation, since it requires a great investment to develop a device in hardware; and the technologies of different manufacturers are incompatible with each other and do not allow reuse of hardware and software.

To address these problems and implement a less costly and more agile network infrastructure, the concept of NFV proposes to transfer the network functions of commercial appliances, with dedicated hardware and software, to off-the-shelf equipment executing software-based functionality on commodity hardware with virtualization technologies for processing, storage, and networking, as exemplified in Fig. 2.6.

The NFV objectives can be summarized as listed below [ETSI NFV ISG 2014]:

- Reduction of CAPEX when compared to dedicated hardware implementations. This goal is achieved through the adoption of commodity hardware, the use of virtual-

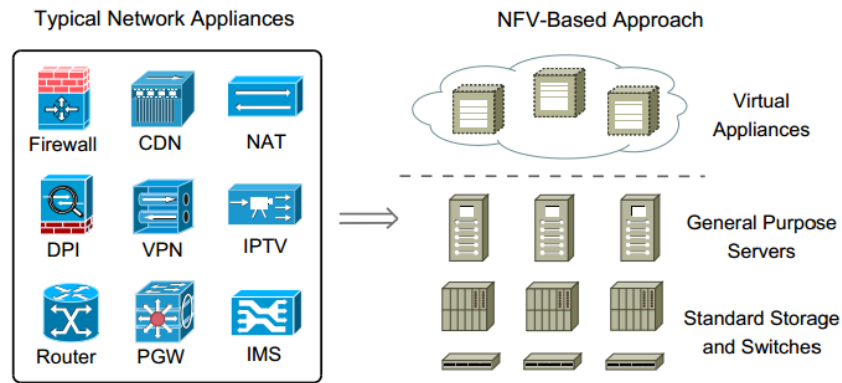


Figure 2.6: NFV implementation of network functions using virtualization techniques over standard hardware. Source: [Han et al. 2015].

ization techniques, the reduction in the number of different hardware architectures, and the resource sharing.

- Scalability and flexibility for instantiating network functions. With NFV, it is possible to decouple location from functionality, and allocate network functions in the most appropriate places according to demands, increasing resiliency through virtualization, and making resource sharing easier.
- Incentive to innovation and time reduction for a new product to reach the market through software-based implementation.
- Reduction of operational expenses (OPEX) through the automation of procedures.
- Reduction of power consumption by migrating workloads and shutting down unused hardware.
- Interoperability through open and standardized interfaces between the network functions, the underlying infrastructure, and the associated management entities. In this way, the elements of the NFV architecture can be implemented by different vendors.

These objectives have great impact on the business model of telecommunications networks. Thus, this sector has invested in the standardization of NFV, as discussed in the next section. However, it is important to emphasize that the paradigm is applicable in both computer networks and telecommunications networks; especially at a time when both converge to a resource-delivery model based on cloud computing.

2.4.2 ETSI NFV reference architecture

The NFV concept gained momentum in 2013 with the creation of the Industry Specification Group (ISG), created within the European Telecommunication Standards Institute (ETSI) to provide guidelines for a new network environment based on modern

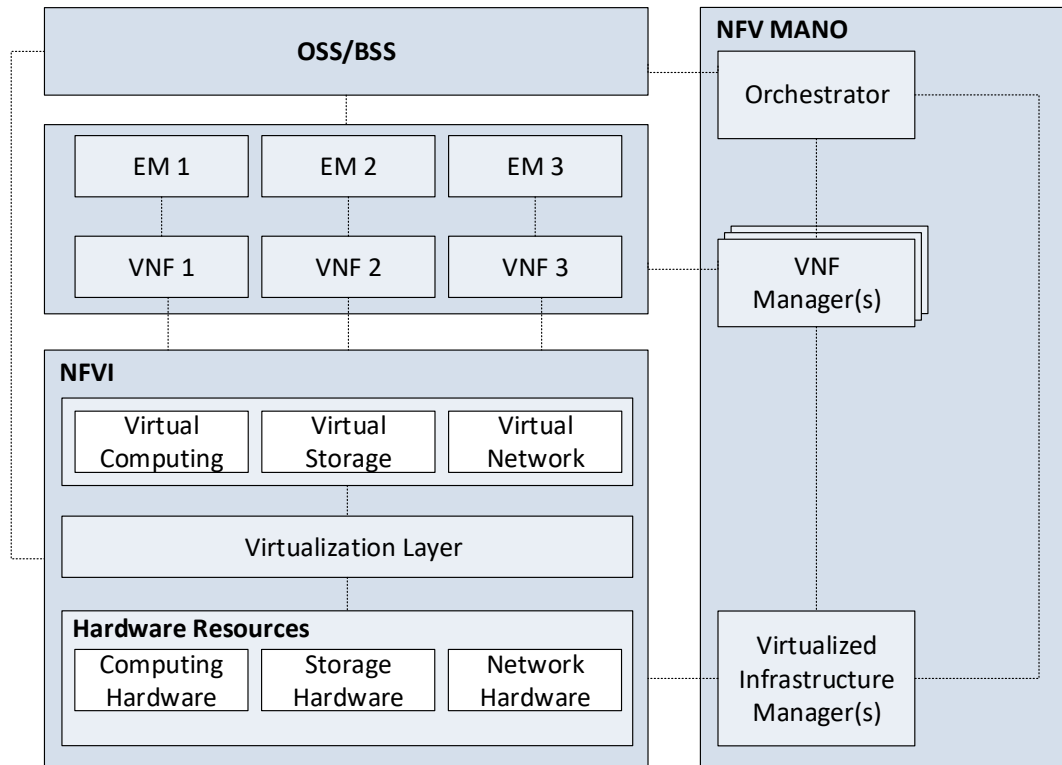


Figure 2.7: ETSI NFV Architecture.

virtualization technologies in order to lower costs and increase efficiency and agility [ETSI NFV ISG 2012]. The NFV ISG group has more than 200 member organizations, including telecommunications service providers and network equipment manufacturers. This section will give an overview of the reference architecture proposed by the ETSI NFV ISG group in its standardization documents [ETSI NFV ISG 2014].

In the reference architecture, one of the main blocks is the Management and Orchestration (NFV MANO), which has the following components: NFV Orchestrator (NFVO), VNF Manager (VNFM), and Virtualized Infrastructure Manager (VIM). The NFV MANO block interacts with an SDN Controller, which controls both physical and software switches.

Fig. 2.7 shows the high-level architecture, divided into the following main components [ETSI NFV ISG 2014, Han et al. 2015, Rosa et al. 2014]:

- Virtualized network function² (VNF): it is the software implementation of a network function that is capable of running on the NFV infrastructure. Examples of network functions include: routers, firewalls, home gateways, MME (Mobility Management

²The Service Function (SF) and Virtualized Network Function (VNF) terms will be used interchangeably in this document, as they represent the same concept for SFC in the terminology used by the Internet Engineering Task Force (IETF) and the European Telecommunications Standards Institute (ETSI), respectively. As defined by the IETF [Quinn and Nadeau 2015], SF is a function that is responsible for specific treatment of received packets, and it can be realized as a virtual element or be embedded in a physical network element. As defined by the ETSI [ETSI NFV ISG 2018], VNF is an implementation of an NF that can be deployed on a Network Function Virtualization Infrastructure.

Entity), PGW (Packet Data Network Gateway), authentication servers, and DHCP servers.

- NFVO: it is responsible for managing and orchestrating software resources and the virtualized hardware infrastructure for enabling network services.
- VNFM: it is responsible for managing the life cycle of a VNF (instantiation, scaling, termination, query, and update).
- VIM: it is responsible for virtualizing and managing computing, network, and storage resources, and controlling their interaction with VNFs. It allocates VMs in hypervisors, and manages their network connectivity.
- NFV Infrastructure (NFVI): the physical hardware resources, including computing, storage and network.
- Virtualization layer: it is responsible for abstracting the hardware resources and connecting the VNFs to the virtualized infrastructure. This layer ensures that the VNF lifecycle is independent of the hardware platforms used.
- Operational support systems (OSS): support the internal processes of the operator such as network inventory, service provisioning, network element configuration, and fault management.
- Business support systems (BSS): handle requests from users, supporting processes such as billing.

2.5 SFC

The SFC is the instantiation of an ordered set of SFs and the subsequent steering of traffic flows through those SFs [Halpern and Pignataro 2015]. SFC steers traffic that would otherwise be routed straight from source to destination to pass through a chain of virtualized nodes. This section explores a set of fundamental concepts required to understand this paradigm shift.

Traditional SFC solutions rely on static wiring of rigid components, and complex routing schemes to steer traffic through network functions [Quinn and Guichard 2014]. Thus, configuration and customization of services usually require significant changes in network configuration, and leads to stretched deployment times and large operational complexity [John et al. 2013]. Because these deployments are tightly coupled to network topology, they also tend to be restricted to a specific service provider (SP) domain, hardly being applicable in different scenarios [Quinn and Nadeau 2015].

SFC has gained momentum in the past years, causing the IETF to publish several drafts and RFCs related to the theme [Quinn and Nadeau 2015, Halpern and Pignataro 2015, Homma et al. 2016, Kumar et al. 2017, Bottorff et al. 2017, Quinn et al. 2018, Clad et al. 2018]. One of the first documents of the series is RFC 7498, a problem statement for SFC defining key areas that working groups could investigate towards new SFC solutions [Quinn and Nadeau 2015].

Another product of such effort is RFC 7665 [Halpern and Pignataro 2015], which proposes a reference architecture for SFC, as illustrated in Figure 2.8. It is composed by the following components:

1. **Classifier**: classify input packets to choose which service chain should be executed;
2. **Service function forwarder (SFF)**: steer packets between SFs in the correct order for each path;
3. **Service functions (SFs)**: perform some computation over received packets and may act at various layers of OSI protocol stack (e.g., at the network layer or application layer);
4. **Proxy**: support SFs that are not aware of the SFC mechanisms (e.g. legacy service functions).

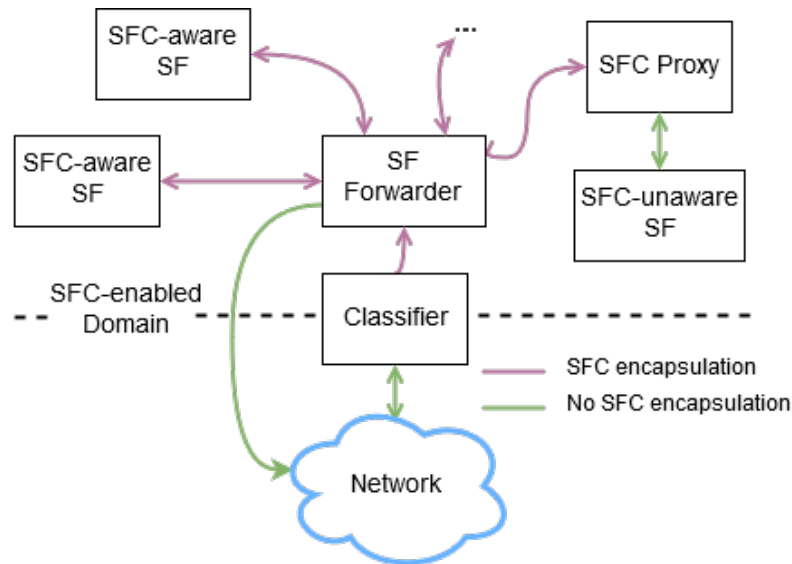


Figure 2.8: Reference architecture from RFC 7665. Source: [Castanho et al. 2018].

An essential concept in this architecture is the **Service Function Path (SFP)**. According to RFC 7665, a SFP may identify the exact network nodes and SFs the packet will visit when it traverses the network, or it can also be less specific using a sequence of abstract SFs.

As shown in Fig 2.8, packets are classified upon arrival to the network by the Classifier. In this step, a packet is matched against a set of pre-configured rules and a SFP is chosen, if one exists for that packet. A SFC encapsulation containing the corresponding path information is added to the packet and it is sent to the next element in the architecture: the SFF. This element is responsible for deciding which SF should be executed at any given time for each path. The SFF also handles service chain termination and final decapsulation of packets.

An SF may be SFC encapsulation aware or unaware. The former receives encapsulated traffic from the SFF and acts on information in the SFC encapsulation. While the latter has no idea that it is part of an SFC environment and receives data without SFC encapsulation. To support SFC-unaware SFs a Proxy is used between the SF and the Forwarder. The Proxy removes the encapsulation prior to sending packets to attached SFs and reapplies the SFC encapsulation when returning them to the SFF.

According to RFC 7498 [Quinn and Nadeau 2015], a SFC solution should address the following elements:

- **Service Overlay:** SFC utilizes a service overlay that creates a service topology to provide service function connectivity, built "on top" of the existing underlay network topology.
- **Service Classification:** Initial classification is used to select which traffic enters in each service function chain. Within a given service function chain, packets may be reclassified to modify the sequence of service functions.
- **SFC Encapsulation:** It carries information for creating the SFC data plane (e.g., chain identification and operation and management status) and for exchanging metadata among classification points and service functions.

2.6 Concluding remarks

This chapter summarized the background concepts that serve as basis for the proposal of this thesis. The next chapter will present a comparison of our proposal with the state of the art.

Chapter 3

Comparison with the state of the art

This chapter describes how this thesis is positioned with respect to the state of the art. Firstly, Section 3.1 explores the impact of the underlay network on SFC, shows the limitations of current SFC solutions, and describes the principles we envision to address these limitations. Afterwards, Section 3.2 compares our solutions with related works in the areas of RNS-based SSR, NFV Orchestration, and SFC.

3.1 Discussion on the SFC underlay layer

Most of the current SFC solutions only support network-centric topologies and table-based routing in the network underlay (more details on Section 3.2). However, as described in Section 1.1, they do not offer adequate mechanisms to dynamically configure underlay paths, and prevent the NFV Orchestrator to fully exploit the network capacity.

To tackle these issues, this section discusses how alternative interconnection topologies and routing methods can be integrated in SFC solutions that offer greater synergy with the network underlay. Firstly, we describe the limitations of current network infrastructures, and how they impact the SFC problem. Then, we explore how to use server-centric topologies (see Section 2.1) and SSR mechanisms with algorithmic forwarding (see Section 2.2) for providing SFC, and discuss how our SFC solutions differ from traditional approaches.

3.1.1 Limitations of current network infrastructures

Traditional DC infrastructures for NFV are based on network-centric topologies, as presented in Fig. 3.1.a, and per-hop table-based routing methods (see Section 2.1 and Section 2.2). For example, in two-tier and three-tier tree DC architectures, the servers hosting the VNFs are the leaf nodes and the network elements are responsible for forwarding NFV traffic between server nodes.

This approach presents some drawbacks: (i) traffic forwarding is managed exclusively

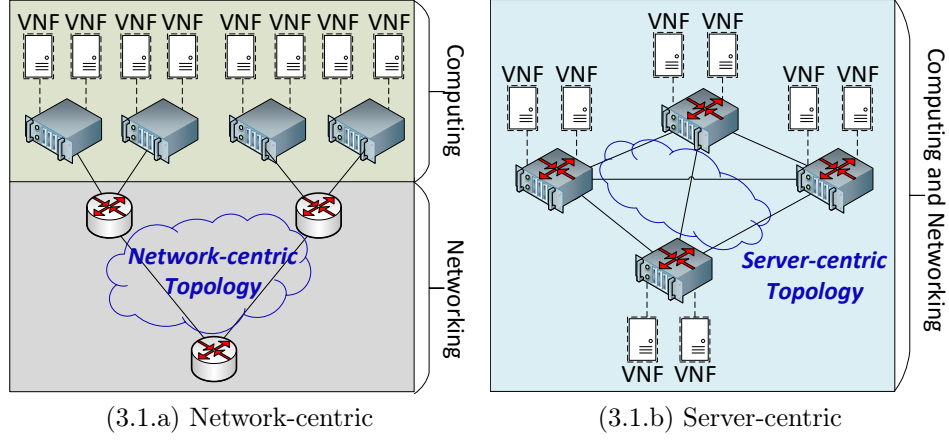


Figure 3.1: Comparison between data center network architectures.

by expensive core, aggregation and edge network elements with high throughput capacity to avoid over-subscription; (ii) network solutions are tied to proprietary hardware and software with limited room for innovation; (iii) physical network infrastructure offers no extra degree of freedom to the NFV Orchestrator, which is tied to costly VNF migration operations when an overload event is detected; (iv) routing is typically done in layer 3 using IP routing tables and ARP flooding, increasing network latency; and (v) spanning tree protocol (STP), used to build a logical topology that avoids loops, limits the bisection bandwidth, impacting on the network performance.

One could argue that some of these problems can be solved by using SDN-enabled switches. This approach allows layer 2 forwarding with the use of flow tables. Besides, the NFV Orchestrator can interact with an SDN controller with a view of the whole network, such that the Orchestrator efficiently decides where to place VNFs in the network and the SDN Controller steers flows through service chains. On the other hand, this management flexibility takes its toll: (i) the available features are restricted to the OpenFlow versions implemented by each equipment and vendor firmware; (ii) the fundamental forwarding mechanism, based on flow tables, cannot be modified; (iii) changes in resource allocation decisions may involve modifications in the flow tables of all the networking elements of a forwarding path, causing control signaling overhead and high latency for path setup convergence; (iv) the size of lookup tables may become a limiting factor when the granularity of flow descriptors increases [Mogul et al. 2010] and restrict the path selection of traffic engineering to one or a small set of shortest paths [Davoli et al. 2015, Bhatia et al. 2015]; and (v) the management of OpenFlow switches across different hierarchical layers, especially in the core layer, is complex [Casado et al. 2012].

This can lead to network overload and prevent the NFV Orchestrator to explore all the capacity of the network underlay when provisioning SFC requests. The next subsections discuss how the exploitation of server-centric topologies (Fig.3.1.b) and SSR can help to address these problems.

3.1.2 Topology: Network-centric vs. server-centric DCNs

Figs. 3.2 and 3.3 show an example of the VNF-FGE problem (discussed in Section 1.1.2) for network-centric and server-centric topologies, respectively.

The problem is divided in three layers: in the service overlay layer (Figs. 3.2.a and 3.3.a), the VNF-FG shows the service view and how the VNF nodes need to be interconnected; in the server layer (Figs. 3.2.b and 3.3.b), the figures show in which physical server each of the VNFs will be hosted and the logical connections that need to be established in order to enable the communication between them; and, in the network underlay layer (Figs. 3.2.c and 3.3.c), the figures show how the logical network will be provisioned in the physical network. If the next hop of the SFC is a VNF in the same physical server (e.g., the link between VNF2 and VNF3), the SFC step will be accomplished by the internal routing mechanism of the physical server; otherwise, the SFC will rely on the routing mechanisms provided by the network.

The comparison of Figs. 3.2 and 3.3 gives an intuition on how the network infrastruc-

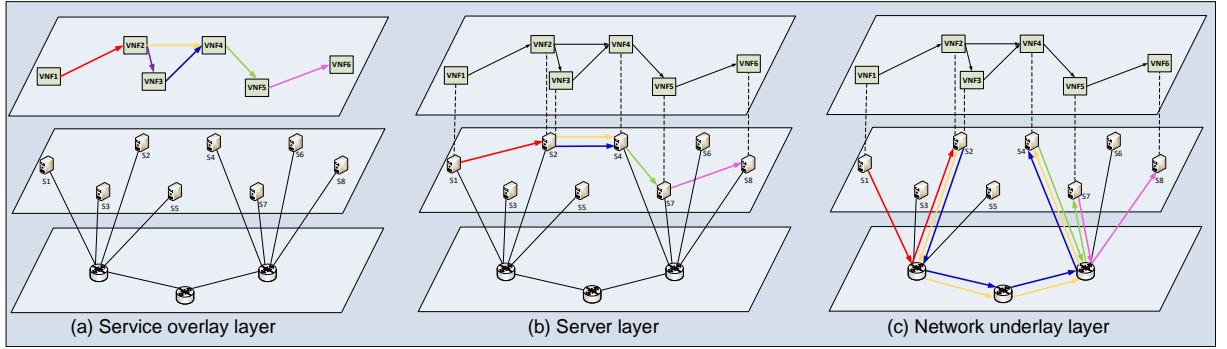


Figure 3.2: VNF-FGE problem in **network-centric topologies**: (a) service overlay layer: VNF forwarding graph; (b) server layer: VNF embedding in physical servers and logical connections between servers; (c) network underlay layer: routing in physical network. Adapted from: [Dominicini et al. 2017].

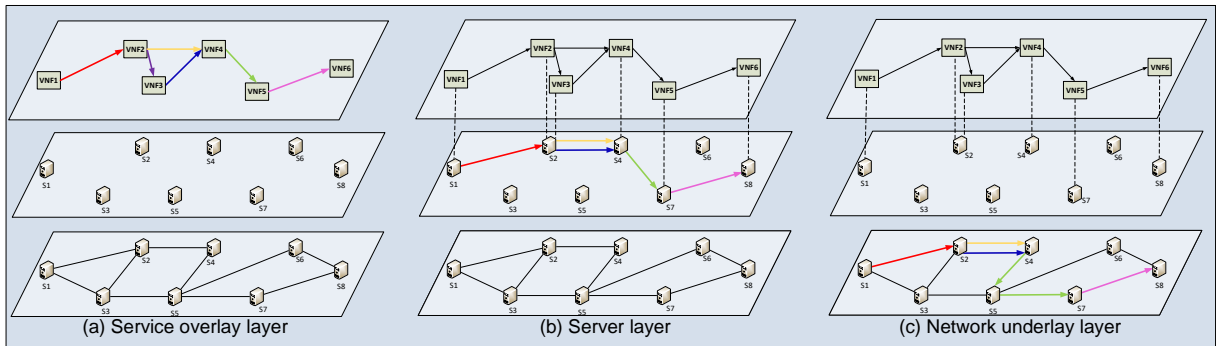


Figure 3.3: VNF-FGE problem in **server-centric topologies**: (a) service overlay layer: VNF forwarding graph; (b) server layer: VNF embedding in physical servers and logical connections between servers; (c) network underlay layer: routing in physical network. Adapted from: [Dominicini et al. 2017].

ture has great impact in the network utilization when embedding SFC requests. Because of the specific nature of the SFC paradigm, the flows must enter and leave the servers that host the VNFs. In server-centric topologies, the orchestrator can take advantage of this characteristic to place the VNFs in servers that are already part of the routing path between source and destination. On the other hand, the orchestration has to consider that CPU capacity of servers is shared among forwarding tasks and VNF processing tasks.

In network-centric topologies, after leaving the physical server, the traffic may need to go up and down in a set of switches until it reaches the next SFC hop. If the VNFs of the SFC can be clustered in the same switch or in neighbor pods, this impact can be minimized. However, the clusterization of VNFs per SFC request may come with the cost of instantiating additional VNFs, even when there is an already existing VNF with spare capacity in a distant pod.

One approach that can be used to provide SFC is based on flow identifiable information [Homma et al. 2016]. In this method, a central entity configures each forwarding element with flow entries to steer packets to the next SF in the chain depending on the headers of the packet being handled. Fig. 3.4(a) shows a scenario, where the network functions are implemented in appliances (physical network functions - PNFs) and connected to a SDN hardware switch that contains the flow entries to enable SFC.

However, the NFV scenario is much more complex, because the network functions are virtualized in a cloud environment. In a typical cloud, each physical server has two main components: a hypervisor that manages the virtualization; and one or more software switches that enable communication with VMs hosted in other servers, internal routing between VMs in the same host, and isolation between different tenants using private networks.

Fig. 3.4(b) shows the equivalent scenario of Fig. 3.4(a) for a network-centric cloud. Now, the SFC flow entries need to be distributed across the software switches of the physical servers and the hardware switches of the network infrastructure. In this case,

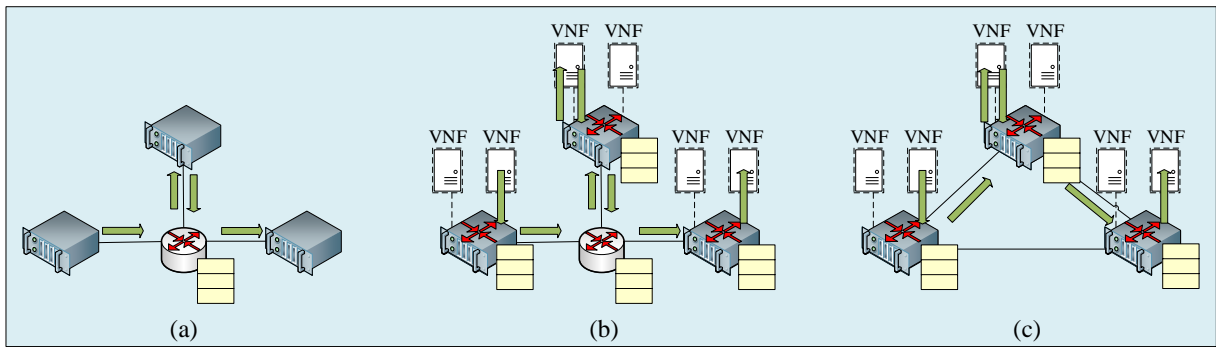


Figure 3.4: SFC scenarios: (a) Legacy networks: PNFs, and SDN hardware switches for SFC. (b) Network-centric cloud: VNFs, and SDN software and hardware switches for SFC. (c) Server-centric cloud: VNFs, and SDN software switches for SFC. Source: [Dominicini et al. 2017].

the routing in the physical network is responsibility of the hardware switches. Fig. 3.4(c) shows the SFC scenario for a server-centric cloud, where there is no hardware switch and flow entries for SFC are distributed in the physical servers, which are also responsible for routing in the physical network.

Furthermore, in a NFV scenario, VNFs are distributed across physical servers and the SFC scheme must cope with various concurrent and dynamic service requests. Therefore, SFC entries need to be managed by the SDN Controller, which has full knowledge of the network topology, the service requests, the VNFs that serve these requests, and the specific virtual ports where each VNF is attached in each physical server.

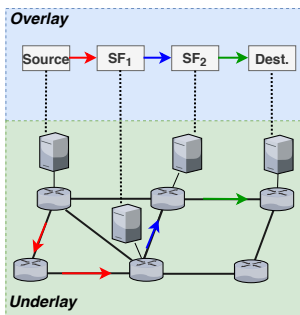
Depending on application requirements, different use cases may benefit of particular characteristics of network-centric, server-centric or hybrid topologies. In this work, we investigate SFC mechanisms that can work with different DC architectures with COTS equipment, extending the infrastructures that can be used to provide SFC in edge DCs.

3.1.3 Routing: per-hop table-based vs. algorithmic SSR

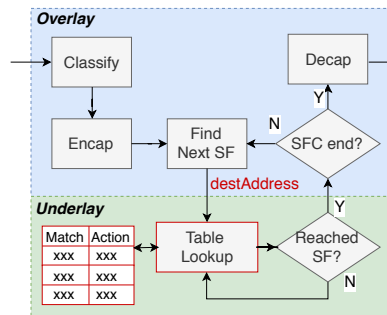
Fig. 3.5 presents the SFC layers for an example chain, and shows how the underlay operations of current SFC approaches differ from the underlay operations of our approach, which is based on algorithmic SSR.

Fig. 3.5.b shows how most of the current SFC solutions perform traffic steering. After classification, the SFC encapsulation carries information used to determine the network address of the SF to be executed in each SFC segment. The SFC encapsulation may create a new SFC header, as in NSH [Quinn et al. 2018] and Segment Routing proposals [Clad et al. 2018], or overload an existing header. To discover the address of the next SF, some methods rely on tables as in the case of NSH that uses a chain identifier to perform a table lookup operation, while other methods encode the SF addresses directly in the header as in the case of Segment Routing. This SF address is often encapsulated in an outermost header that is used for routing to the next SF.

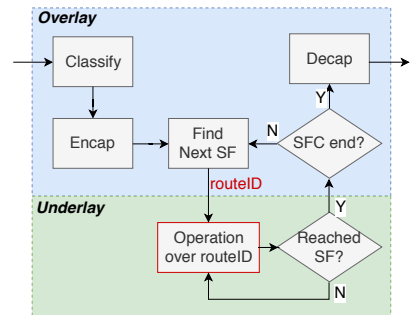
After determining the address of the current SF, the packet is delivered to the un-



(3.5.a) SFC layers.



(3.5.b) Current SFC approaches.



(3.5.c) SFC with algorithmic SSR.

Figure 3.5: SFC problem.

derlying routing mechanism (e.g., MPLS or IP), where the forwarding is executed using per-hop table lookups based on the destination address until it reaches the SF. At this moment, the encapsulation information is checked again to find the address of the next SF and this loop proceeds until the chain terminates, when the packet is decapsulated and delivered to destination.

The problems with this traditional SFC approach are threefold: (i) the commonly adopted underlying routing mechanisms cannot represent all the possible paths between two endpoints due to limited capacity of switch forwarding tables; (ii) as they are based on per-hop table lookup, the overhead to dynamically change a path is high, because it may involve modifications in all the nodes in the path; and (iii) the traffic engineering decisions are made in a decoupled way considering placement, chaining, and routing.

These issues have direct impact on the flexibility offered to the NFVO decision making for optimal resource allocation. This leads to the problem that we tackle in this thesis of designing a programmable, expressive, scalable, and agile SFC solution that enables dynamic and efficient orchestration of the network underlay of edge DCs.

In this work, we advocate that the most appropriate solution is to use SSR with algorithmic forwarding (or algorithmic SSR, see Section 2.2), in the core nodes, and SDN-enabled edge nodes in the servers for SFC classification. Fig. 3.5.c illustrates how our proposal uses algorithmic SSR for determining the specific SFP of each SFC segment. The SFP identifies the exact network nodes and SFs the packet will visit when it traverses the network [Halpern and Pignataro 2015]. The path information is inserted in the encapsulation stage in the form of a *routeID*. This identifier is passed to the underlying routing layer, where each hop can take forwarding decisions on the output port based on a simple operation over the *routeID*.

The algorithmic SSR solution presents several benefits over traditional approaches. Firstly, as already proved by other works [Jyothi et al. 2015], SSR allows traffic engineering to exploit all existing paths, and, consequently, achieve maximum throughput. Secondly, it enables traffic engineering to specify the entire path for each SFC segment. Besides, the elimination of tables in core nodes can reduce the number of forwarding states, and control plane signaling [Martinello et al. 2014, Jyothi et al. 2015]. Finally, if the SFP needs to change to fit any dynamic demand, the source only needs to encapsulate a new *routeID*, and intermediary nodes will continue to operate over the received identifier without any modification.

These benefits are particularly important when we consider the dynamic characteristics of the NFV traffic: SFC requests constantly start and terminate; the sequence of SFs in a chain may change during its life cycle; or traffic engineering may need to migrate paths and SFs to guarantee optimal resource usage.

Fig. 3.6 illustrates an example scenario of dynamic chain migration due to security demands. Initially, the NFVO computes the best routing and placement decisions to

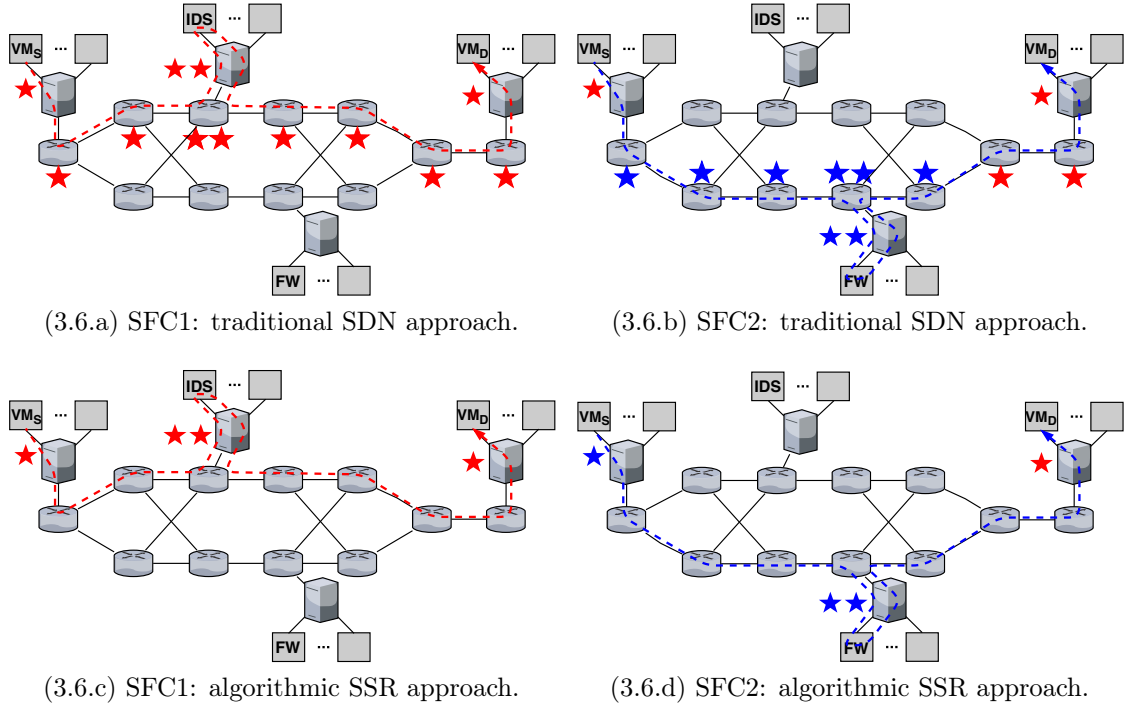


Figure 3.6: Comparison between flow entries (represented by stars) in traditional SDN and algorithmic SSR approaches for dynamic migration from SFC1 ($VM_S \rightarrow IDS \rightarrow VM_D$) to SFC2 ($VM_S \rightarrow FW \rightarrow VM_D$).

embed SFC1 ($VM_S \rightarrow IDS \rightarrow VM_D$) for a specific flow, considering overall resource utilization and the need to steer the traffic through an IDS (intrusion detection system). Then, when an attack is detected, the flow is directed to a firewall (FW) and migrated to SFC2 ($VM_S \rightarrow FW \rightarrow VM_D$).

Fig. 3.6.a shows how a traditional SDN-based approach installs flow entries (represented as red stars) in all the elements in the forwarding path to guarantee the full specification of an SFP for SFC1. When the SFC is migrated, Fig. 3.6.b shows that some of these entries need to be removed and new ones (represented as blue stars) need to be added to steer the traffic through SFC2.

On the other hand, Fig. 3.6.c shows that our algorithmic SSR approach only installs flow entries (represented as red stars) in the SFC segments' endpoints, in order to classify the flow and include the *routeID* for SFC1. When the SFC is migrated, Fig. 3.6.d shows that a small number of flow entries have to be modified (represented as blue stars) to include the *routeID* for SFC2. Thus, our approach requires the installation and modification of fewer flow entries and, consequently, can react faster to path changes. Moreover, as the forwarding states are only maintained in the edges, our solution is more scalable and less subject to inconsistencies than traditional approaches.

Other traditional SFC solutions either do not allow the specification of all elements of the SFP or present limited scalability, as discussed in more details in Section 3.2.3.

3.2 Related works

This section positions our work with respect to related works on RNS-based SSR, NFV orchestration, and SFC.

3.2.1 RNS-based SSR

Port Switching is a traditional method for executing SSR that represents the *routeID* as a stack of ports or addresses and the forwarding operation as a stack pop (see Section 2.2). SecondNet [Guo et al. 2010], Segment Routing [Filsfils et al. 2015], and Sourcey [Jin et al. 2016] are examples of works that explore Port Switching SSR.

RNS-based SSR is an alternative way of performing SSR that defines the output port in each node by a *modulo* operation over the *routeID* (see Section 2.2.1). This thesis argues that RNS-based SSR presents many interesting properties that do not exist in Port Switching, such as:

1. **Packet forwarding without header modifications:** The forwarding operation is the direct result of the *modulo* operation of the *routeID* of the packet by the *nodeID* of the core node.
2. **The order of the nodes in the path is irrelevant:** It can be noticed in Equation 2.2 that the *routeID* does not store the information about the node sequence the packet should traverse. Data from each node (*nodeID* and *portID*) belongs to its own addend of the summation and does not influence the other summation addends. As the finite summation is commutative, the node order is irrelevant to derive the *routeID*.
3. **The path information is not transmitted in the clear:** Given the *routeID* of a packet, it is still necessary to know the *nodeID* to discover the output port of that packet in a specific node.

Therefore, RNS-based SSR can explore these properties to provide networking functionalities that cannot be covered by Port Switching. The following topics exemplify some potential applications of these properties:

1. **Packet forwarding without header modifications:** This property is important for enabling scenarios where header modification is very difficult to implement, like in all-optical switches [Wessing et al. 2002]. Besides, as packet rewriting is a costly operation even for packet-switched networks, this can also be explored by latency-critical applications if the *modulo* operation can be implemented with high performance. Another potential application of this property is route authenticity: since the packet header does not change throughout all the path, the source can sign

the *routeID* information and the nodes could verify this signature before taking the forwarding decision.

2. **The order of the nodes in the path is irrelevant:** This property allows embedding redundant nodes in the *routeID* that are disjoint of the desired route. For example, KAR [Gomes et al. 2016] is a fast failure reaction scheme that uses RNS-based SSR and explores this property. It proactively adds redundant nodes in the *routeID* to create resilient forwarding paths (called protection paths). Packets are deviated from faulty links with routing deflections, and guided to their original destination when they reach nodes that are part of the protection paths.
3. **The path information is not transmitted in the clear:** Public-Key cryptography relies in the practical difficulty of factoring the product of two large prime numbers. In RNS-based SSR, we do not deal with such large primes, but an approach that uses multiple *nodeID* keys per node or changes these keys frequently can make it difficult for an attacker to predict or forge a path if the *routeID* is captured.

RNS-based SSR was firstly explored by [Wessing et al. 2002] and applied to optical packet-switched networks to avoid header rewriting and label distribution protocols. This idea was further integrated with SDN in core packet-switched networks by KeyFlow [Martinello et al. 2014], which builds a fabric model that replaces the table lookup in the forwarding engine by elementary operations relying on RNS. Then, other works, such as KeySet [Ren et al. 2017] and KAR [Gomes et al. 2016] explored additional properties of RNS to extend KeyFlow. KeySet explored techniques to reduce the length of the forwarding label, while KAR deviates packets from faulty links with routing deflections and guides them to their original destination due to resilient paths added to the forwarding label. However, all these works applied RNS to routing in core networks and did not explore how to use these concepts for DCNs where virtualized SFs need to be chained to provide a service. In turn, the works in [Jia 2014, Martinello et al. 2017, Liberato et al. 2018] evaluate the scalability of a RNS-based routing system for DCNs with 2-tier Clos topologies, and demonstrated its ability in fulfilling latency constraints for multicast in DCs. but they did not explore a solution for SFC.

The first work to suggest the application of RNS in SFC was [Zhao and Hu 2017], but it does not provide any details on how to integrate or implement the proposal with virtualized SFs in a DC. In CRT-Chain [Ren et al. 2018], the authors proposed a more elaborate scheme for SFC in DCs using two RNS forwarding labels: one for routing in the physical layer, and one for routing between the SFs in an overlay layer. However, their simulated results focused only on reducing the labels length, and they did not provide any implementation and validation for data and control planes. Moreover, their theoretical proposal presents some critical limitations for real-world scenarios: (i) in contrast to our

approach that performs one modulo operation per core node, their data plane is composed by a loop that performs a modulo operation for each SF attached to each forwarder in the path (even the SFs that are not part of the chain), which will add a huge delay in the end-to-end service latency; (ii) it does not allow loops in the physical layer, which is not acceptable if we consider that a traffic may be steered back and forward in the physical network to traverse different SFs; and (iii) it does not allow more than one SF of the same type in the same forwarder.

To tackle these issues, we propose KeySFC [Dominicini et al. 2019], a SFC scheme that extends the idea of network fabric to intra-DC networking, using programmable edge software switches, and core RNS-based switches. Besides, KeySFC architecture supports network-centric, server-centric, and hybrid DCNs [Li and Yang 2016, Dominicini et al. 2017]. Moreover, we implemented and validated a proof-of-concept of KeySFC in a DC testbed using production technologies, such as OpenStack, OpenFlow and OvS. Thus, to the best of our knowledge, KeySFC is the first work that proposes a complete SFC scheme using RNS that is implementable and efficient for DCNs.

An obstacle for implementing RNS-based SSR is the fact that COTS network hardware does not implement integer *modulo* operations. Therefore, all these works (including KeySFC) either use software switches implementations [Martinello et al. 2014], or depend on synthesizing integer division to ASICs or NetFPGAs [Liberato et al. 2018]. Nevertheless, providing a low cost, high throughput, and low latency implementation is capital for the success of RNS-based SSR.

To this end, we propose PolKA, a novel RNS-based SSR mechanism that explores the Chinese Remainder Theorem in finite fields of two elements [Schroeder 2009]. This shift from integer to polynomial arithmetic can enable performance optimization and reuse of off-the-shelf network hardware, such as CRC (cyclic redundancy check), while portending new network services. In addition, we demonstrate that PolKA can be deployed in commercial programmable switches using P4 architecture (see Section A.1.2), and implemented with similar performance to a traditional Port Switching SSR approach [Jin et al. 2016, Guo et al. 2010]. Furthermore, we integrated the SSR mechanism of PolKA in the KeySFC scheme, replacing its original RNS mechanism based on integers. In this way, we enable the exploitation of RNS-based SSR and SFC in commercial hardware equipment.

3.2.2 NFV Orchestration

NFV orchestration is a very complex problem that involves many different areas and have been widely researched in the last years [Yi et al. 2018]. Many important works have extensively surveyed relevant aspects of NFV orchestration: optimal resource allocation [Herrera and Botero 2016, Tso et al. 2016,

[Laghrissi and Taleb 2019], MANO solutions [Mijumbi et al. 2016], network service orchestration [Rotsos et al. 2017, de Sousa et al. 2019], and SFC [Bhamare et al. 2016, Medhat et al. 2017, Hantouti et al. 2018]. Each of these surveys summarizes more than 100 works.

Most of these works integrate NFV and SDN paradigms as a way to provide dynamic network services, which is also one of the pillars of our proposal. Standardization efforts led by ETSI and by the Open Networking Foundation (ONF) have published several specifications, aiming to point directions to a NFV-SDN architecture evolution [ETSI NFV ISG 2014, ETSI NFV ISG 2015, Open Networking Foundation 2014]. In [Matias et al. 2015], the authors present the shift of NFV from a initial SDN-agnostic initiative to a fully SDN-enabled NFV solution. Besides, they explore how the programmable network infrastructure can implement part of the VNF functionality. In [Masutani et al. 2014], the authors discuss the requirements for integrating NFV and SDN, focusing on the architecture design of NFV-enabled network nodes implemented by commodity servers using OvS and Intel DPDK technologies. In [Zhang et al. 2018], a typical SFC functional framework integrating SDN and NFV is proposed with service modeling, resource allocation, and traffic steering components. Besides, they summarize existing solutions and present challenges and opportunities for each of these components. In [Tso et al. 2016], a converged resource management is proposed to provide optimization across application, network, transport and physical layers. In their solution, the modules communicate events to a SDN controller and receive global decisions back, integrating node-local intelligence with centralized control to shorten the decision-making time.

Besides, a lot of progress has been made, and many commercial and academic solutions were implemented in the MANO area [Mijumbi et al. 2016]. As NFV technology matures, many commercial and open source frameworks were developed to be compliant with ETSI NFV MANO reference architecture [ETSI NFV ISG 2014], such as OPNFV¹, Open Baton², Open Source MANO (OSM)³, Cloudify⁴, and Tacker⁵. Also, there are academic works that implement NFV orchestration architectures as surveyed in [de Sousa et al. 2019], such as T-NOVA [Kourtis et al. 2017], Unify [Sonkoly et al. 2015], and SONATA [Draxler et al. 2017].

However, to the best of our knowledge, most of the existing state-of-the-art frameworks on dynamic NFV Orchestration only consider hardware-based switches to interconnect VNF nodes. Despite being a promising area, the use of programmable server-centric and hybrid infrastructures for NFV has been little explored in the literature. Some works explore resource allocation in server-centric and hybrid topologies [Herker et al. 2015,

¹<https://www.opnfv.org/>

²<https://openbaton.github.io/>

³<https://osm.etsi.org/>

⁴<https://cloudify.co/>

⁵<https://docs.openstack.org/tacker/>

Luizelli et al. 2016], but they only rely on simulations and do not implement a orchestration solution in a DC testbed. The TRIIAD architecture [Vassoler and Ribeiro 2017] explored orchestration of IaaS in server-centric DCNs and served as basis for our proposal, but it did not consider NFV, as explained in details in Section 3.2.2.1.

In the area of NFV orchestration, the contribution of this thesis is to propose a NFV orchestration architecture based on server-centric topologies, called VirtPhy [Dominicini et al. 2017], and specify how commodity servers enabled with specialized software switches can be directly interconnected in a server-centric topology to provide a fully programmable network infrastructure for SFC. In contrast to network-centric approaches, VirtPhy distributes the SFC flow rules on the servers, instead of hardware switches. Furthermore, we implement a prototype of VirtPhy in a DC testbed using production technologies, such as OpenStack, OpenFlow, and OvS.

As previously discussed in this chapter, our main contribution lies on the development of underlay SFC mechanisms that are part of the broader orchestration problem. Section 3.2.3 performs a more detailed analysis about related works in SFC.

3.2.2.1 TRIIAD Architecture

The TRIIAD (TRIPLE-Layered Intelligent and Integrated Architecture for Data Centers) architecture is an approach for cloud orchestration and network control [Vassoler 2015]. As shown in Fig. 3.7, it is composed by three horizontal layers and a vertical plane [Vassoler and Ribeiro 2017]: the top layer offers IaaS (Infrastructure as a Service); the middle layer provides a lightweight forwarding mechanism; the bottom layer works as a distributed photonic switching plane; and the vertical plane is responsible for control, management and orchestration of the other three layers.

TRIIAD was pioneer in unfolding the potential of server-centric networks for providing cloud infrastructure. Also, it showed that it is possible to orchestrate such DCNs using SDN in combination with OpenStack cloud computing platform. Using a cross-braced hypercube prototype with 2x2 photonic switches, it demonstrated its capability of reconfiguring the physical links layer and orchestrating the migration of VMs based on server workloads. Another substantial contribution of TRIIAD was the implementation of an efficient kernel level forwarding mechanism in OvS datapath for the XOR algorithm [Vassoler 2015].

The TRIIAD proposal was developed in the same research group of this work, and it was VirtPhy's predecessor in exploring server-centric DCNs. TRIIAD's objective was to provide cloud and network orchestration in the context of IaaS, and it was not designed to meet specific NFV requirements. For instance, it was not designed to be compliant with ETSI NFV standard, nor provide NFV functionalities, such as a mechanism for executing SFC. In addition, TRIIAD's prototype was deployed in servers with very limited processing power that did not have enough resources for deploying VNFs. Thus, the VirtPhy

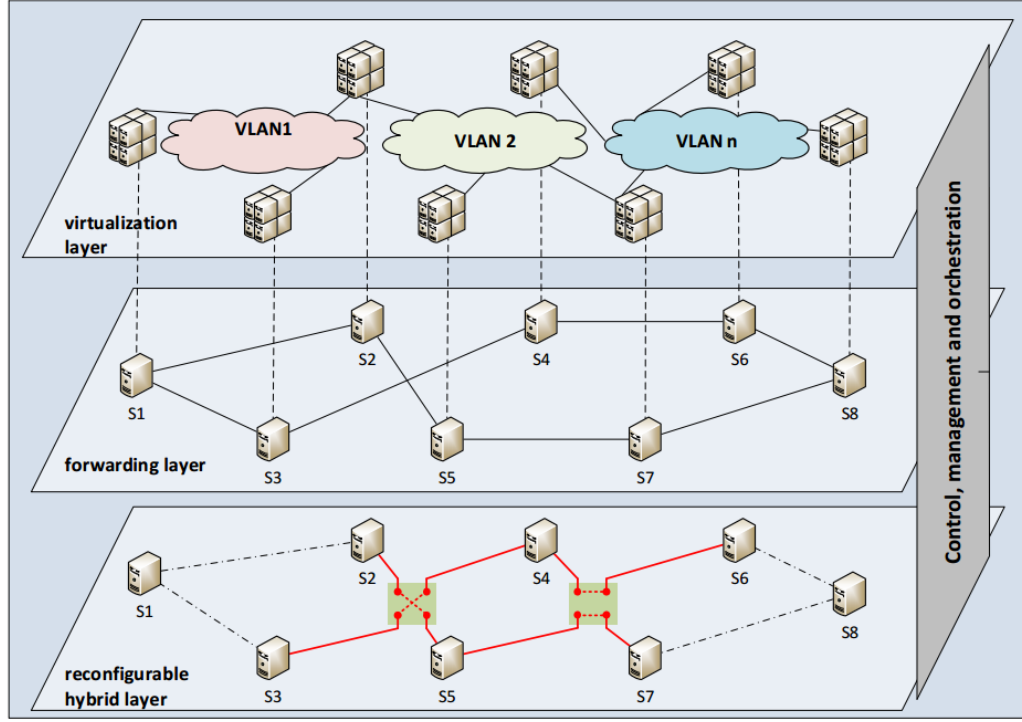


Figure 3.7: Layers of TRIIAD architecture. Source: [Vassoler and Ribeiro 2017].

prototype was built on top of some lessons learned from TRIIAD’s experience on general cloud orchestration to extend its contributions to the NFV environment.

3.2.3 SFC

As described in Fig. 1.2, the SFC problem can be divided in three main phases: service modeling, resource allocation, and traffic steering [Zhang et al. 2018]. In the service modeling phase, a input service request generates a VNF-FG as output. This resulting graph is given as input to the resource allocation phase, which generates the placement and SFC decisions. Then, SFC decisions are deployed in the network infrastructure using traffic steering mechanisms, which are the focus of this thesis. Therefore, the analysis of this section considers the VNF-FGE problem was already solved in a previous resource allocation step, which generated inputs for traffic steering mechanisms.

Based on the RFC 4665 reference architecture [Halpern and Pignataro 2015] (see Section 2.5), many SFC solutions have been proposed. Several of them employ NFV and SDN to enable network services, as surveyed in [Medhat et al. 2017, Hantouti et al. 2018]. In [Homma et al. 2016], the authors classify the different SFC approaches in four types, based on their forwarding methods: (1) flow identifiable information; (2) stacked headers; (3) service chain identifiers with extra header; and (4) service chain identifiers with overload of existing address field.

In this section, we use this classification to compare our work with related works that also present traffic steering schemes. To this end, we select the main representatives

Table 3.1: Comparison between SFC related works.

SFC proposal	Implemented	SFC method	SFC encaps	Routing decision	Forwarding	Topology	Control Plane
StEERING	X	(1) Flow identifiable information	None	Per-hop routing	Table-based	Network-centric	OpenFlow
Segment Routing	X	(2) Stacked headers	MPLS or IPv6	Loose source routing	Table-based and Algorithmic (list pop)	Network-centric	IS-IS, BGP, or OSPF
NSH	X	(3) Chain identifier, extra header	VXLAN, GRE, or Ethernet	Per-hop routing	Table-based	Network-centric	BGP, or NSH Extension for OpenFlow
NetFloc	X	(4) Chain identifier, existing header	MAC address	Per-hop routing	Table-based	Network-centric	OpenFlow
VirtPhy	X	(4) Chain identifier, existing header	MAC address	Per-hop routing	Algorithmic (XOR)	Server-centric	OpenFlow
CRT-Chain		(4) Chain identifier, existing header	MPLS or IPv6	Strict source routing	Algorithmic (modulo)	Network-centric	Not described
KeySFC	X	(4) Chain identifier, existing header	MAC address	Strict source routing	Algorithmic (modulo)	Any	OpenFlow

of each of these four methods to compare the details of these solutions with KeySFC. Table 3.1 summarizes this comparison, classifying them according to the techniques used for SFC, routing, and control. Extensive surveys on traffic steering methods can be found on [Bhamare et al. 2016, Medhat et al. 2017, Hantouti et al. 2018].

Method 1 consists of classifying packets based on flow identifiable information, such as a 5-tuple (e.g., source IP and port, destination IP and port, and L4 protocol), at SFF. A central entity configures each SFF with flow rules to forward packets to the next SF in the chain depending on information contained in packet headers. This method does not demand changes either to the network or the original packet format, because it is based on flow tables and the SDN control plane. On the other hand, when we analyze scalability, it is the least suitable for large networks with a high number of flows, and requires per-hop table lookups. An example of this approach is StEERING [Zhang et al. 2013]: the SDN controller installs flow entries into the switches and handles the service placement, and OpenFlow-enabled switches are responsible for traffic classification and steering. It allows for fine granular application policies and uses multiple forwarding tables to reduce the number of flow rules, but it still suffers from scalability and agility issues because of the large number of flow rules distributed in the switches [Hantouti et al. 2018].

In Method 2, the Classifier analyzes each packet entering the SFC domain, and stacks a series of packet headers over the original packet. Each header includes a network address (e.g., using MPLS or IPv6), and the outermost header addresses the next destination. After receiving the packet, each SF or SFF processes it, removes the outermost header, and uses the next header to find the next SF. This process repeats until all stacked headers are removed and the packet reaches its final destination. This approach may present MTU, fragmentation, and scalability issues when the number of SFs is high.

One of the most important works that uses Method 2 is Segment Routing, which is a flexible source routing method that can enable SFC [Clad et al. 2018]. It can be implemented by using a new type of IPv6 header that encodes segments as a list of 128-bit IPv6 addresses, but this approach presents large overhead [Abdullah et al. 2019]. Also, it can be implemented using MPLS with no modification, since an ordered list of

segments can be encoded as a stack of MPLS labels. The next segment to be process is popped from the top of the stack after the completion of a segment, and a lookup operation in the forwarding table is performed in each hop. Differently to traditional MPLS networks, Segment Routing with MPLS distributes segment labels using simple extensions to current IGP protocols, and LDP and RSVP-TE are no longer required for populating the forwarding tables [Bhatia et al. 2015]. Besides, scalability is improved, because only source nodes maintain state information and the size of forwarding table remains constant regardless of the number of paths [Moreno et al. 2017].

The paths can be derived from a IGP Shortest Path First (SPF) algorithm, which allows a packet to be forwarded along the Equal Cost Multi Path (ECMP)-aware shortest path, or from a Segment Routing Traffic Engineered (SR-TE) path that allows a packet to be steered along an explicit SSR path by using a combination of one or more shortest segments [Abdullah et al. 2019]. However, this flexibility comes at the expense of increased packet overhead and increased label processing in the network [Bhatia et al. 2015]. Moreover, the source node needs to push large segments list in order to realize long optimal explicit paths, but most MPLS equipments can support a limited label stack depth (about 3 to 5 labels), which may lead to inefficient traffic distribution and network congestion [Abdullah et al. 2019]. In this way, the traffic engineering algorithms have to take into account specific constraints of path encoding as additional objective functions [Moreno et al. 2017]. Thus, although Segment Routing can be used for enabling SSR, it commonly uses a loose source routing scheme that specifies a list with some SFFs and SFs that the packet must go through, and delegates the routing between these elements to the underlying network that employs ECMP shortest paths.

In contrast to Segment Routing, KeySFC employs a SSR scheme that does not restrict the path selection by traffic engineering, eliminates forwarding tables, and performs simple operations over service chain identifiers for computing the output port in each forwarding element. KeySFC approach (as any other source routing approach that encodes the path in the packet header) also has to consider the scalability of the routing header, as discussed in Section 6.1.4. To the best of our knowledge, KeySFC is the first work to propose and implement a SFC solution that uses algorithmic SSR, and can be applied in any DCN topology (more details in Section 5.1.2).

In **Methods 3 and 4**, each chain receives a unique service chain identifier from Classifier. Also, SFFs receive forwarding configuration, and consult this information to discover the next hop based on the packet’s chain identifier. In this way, flow entries are defined in a per-chain basis, instead of in a per-flow basis, using less flow entries when compared to Method 1.

This chain identifier can be included in a new specific header, which is the approach adopted in **Method 3**, as implemented in network service headers (NSH) solutions [Quinn et al. 2018]. The NSH header carries two chain identifiers: a 3-byte Service Path

Identifier (SPI), and an 1-byte Service Index (SI), the former indicating the selected chain, and the latter the current position in the chain. The SI is decremented at each step of execution by each SF (or by a Proxy). The SFFs in the path uses the SPI information for performing a table lookup and deciding which SF should be executed at any given time for each chain. Then, the address of the next networking element is encapsulated in an outermost header, relying on the underlying routing methods to deliver the packet between these elements. [Li et al. 2017, Mehmeri et al. 2017] are examples of works that implement SFC solutions using NSH.

Although the NSH protocol presents great flexibility, it is necessary to change all the forwarding elements and control mechanisms in the infrastructure to support it. In addition, attachment of headers expands packet size, causing an increase of traffic, and potential problems with MTU to handle fragmentation. Also, there is a strong separation between the service overlay layer and the underlay network layer. In other words, the SFC mechanism does not give any instruction about how to forward packets in the network (only the order of services that must be traversed) [Quinn et al. 2018]. Therefore, the routing decisions are decoupled from the chaining decisions and executed by different mechanisms. Furthermore, this scheme uses per-hop tables both for discovering the next SF and routing packets in each SFC segment, which leads to scalability and agility issues.

Other alternative, used by **Method 4**, is to include the chain identifier in existing packet fields, like MAC address [Bottorff et al. 2017]. In [Trajkovska et al. 2017], the authors propose a SFC mechanism named Netfloc, where packets entering a service chain invoke new flows in the first and the last bridges of that service chain, which rewrite the original MAC address to a virtual address. Traffic steering relies on L2-based OpenFlow rules, processed by SDN switches that are configured by OpenDayLight SDN controller [Hantouti et al. 2018]. However, their approach relies on various per-hop table lookups and present limited scalability and agility, as discussed before.

KeySFC and CRT-Chain [Ren et al. 2018] [Dominicini et al. 2017] also rewrite existing headers to include chain identification, but they can achieve better scalability and agility by replacing per-hop table lookups with algorithmic SSR using *modulo* operations. VirtPhy [Dominicini et al. 2017] also explores algorithmic forwarding and MAC rewrite for SFC with the use of a high performance forwarding mechanism based on XOR operations, but it does not allow the specification of underlay paths.

Aside from KeySFC and VirtPhy, all the described related works are tailored to network-centric topologies. VirtPhy specifies how commodity servers can be directly interconnected in server-centric topologies to provide SFC. KeySFC, on the other hand, was designed with the objective of supporting any DCN topology.

Regarding the control plane, StEERING, NetFloc, VirtPhy, NSH, and KeySFC support OpenFlow. In Segment Routing, the segments are allocated and signaled by IS-IS, OSPF or BGP. NSH also supports BGP in the control plane.

Table 3.2: Related works and requirements.

Proposal \ Requirement	Programmable	Expressive	Scalable	Agile
StEERING	●	●		
Segment Routing	●	◐	◐	●
NSH	●			
NetFloc	●	●		
VirtPhy	●		●	◐
KeySFC	●	●	●	●

Legend: ● completely meet the requirement ◐ partially meet the requirement

Except for CRT-Chain, all the described works present implementations of their mechanisms. Especially, NSH and Segment Routing received a lot of attention from academia and industry with several real-world deployments.

Table 3.2 summarizes how the described related works meet the requirements defined in Section 1.1.2 for the underlay SFC mechanisms. CRT-Chain was not included, because it does not provide enough information about the implementation of the SFC mechanism.

3.3 Concluding remarks

This chapter discussed how our work differs from related works and its contributions. The next chapters will present the three solutions (VirtPhy, KeySFC, and PolKA) that compose the proposal of this thesis.

Chapter 4

VirtPhy: NFV Orchestration Architecture

VirtPhy is a novel NFV orchestration architecture based on server-centric topologies, where server nodes represent both computing and networking resources, and hardware switches are replaced by SDN software switches that can be directly configured by a NFV Orchestrator integrated with a SDN Controller, as shown in Fig. 4.1.

This chapter describes the design principles, the architecture, and the prototype of the VirtPhy orchestration architecture.

4.1 Proposal

This section describes the potential limitations, the design principles, and the architecture of the VirtPhy proposal.

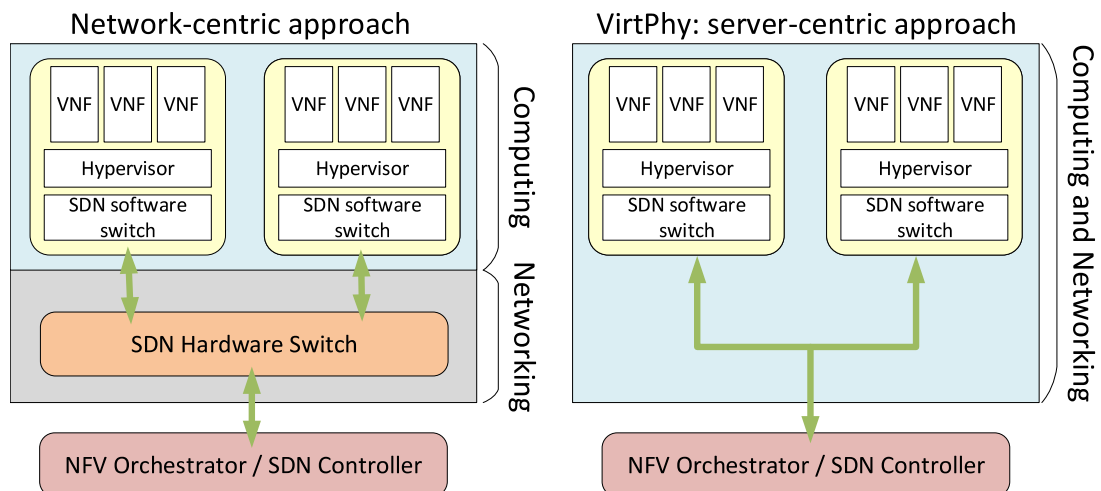


Figure 4.1: Network-centric approach vs. VirtPhy. Source: [Dominicini et al. 2017].

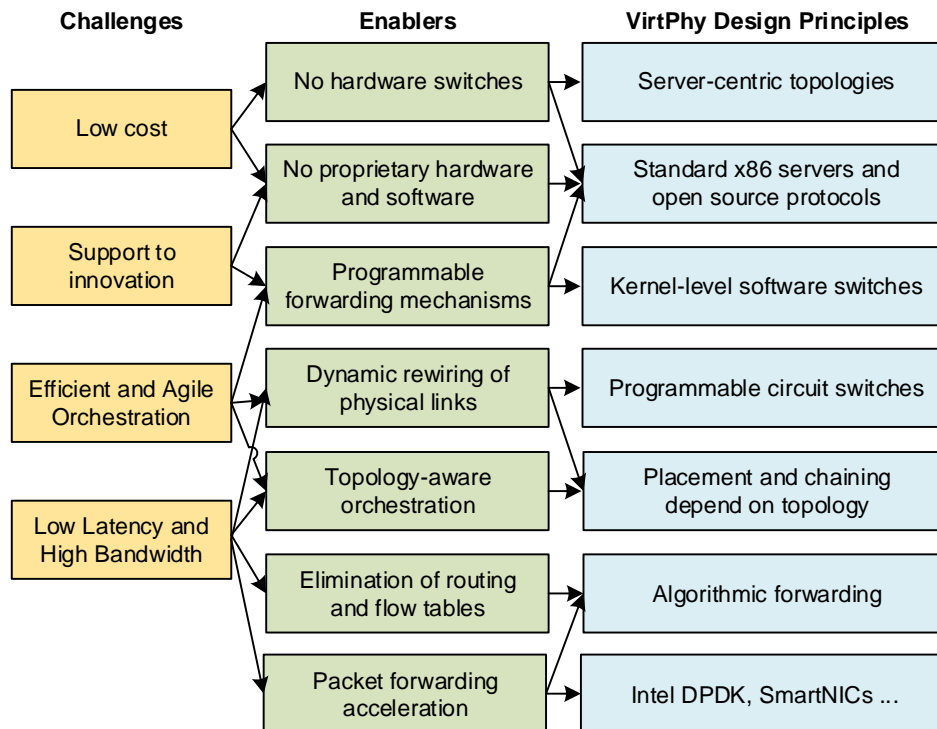


Figure 4.2: Challenges, enablers, and design principles. Source: [Dominicini et al. 2017].

4.1.1 Challenges, enablers, and design principles

The design principles of the VirtPhy architecture were driven by the following NFV challenges in edge DCs: low cost; support to innovation; efficient and agile orchestration; and low latency and high bandwidth.

Fig. 4.2 presents the enablers we envision to address the described challenges. Based on these enablers, VirtPhy adopts the following design principles:

1. **No hardware switches:** The selection of a server-centric topology eliminates the need for acquiring expensive switches, reducing CAPEX.
2. **No proprietary hardware and software:** Our solution is based on standard x86 servers and open source protocols. Thus, it supports innovation and reduces both CAPEX and OPEX, since it eliminates proprietary interfaces that demand specialized professionals and cuts time to market at the deployment phase. This also tackles problems such as: vendor lock-in, firmwares with partial implementation of SDN functionalities, and innovation barriers for new forwarding/routing methods.
3. **Programmable forwarding mechanisms:** The server nodes have a kernel-level software switch that enables the SDN Controller to program the forwarding mechanism [Vencioneck et al. 2014]. This principle enables innovation, rapid deployment of new concepts, and agile network orchestration.

4. **Dynamic rewiring of physical links:** Programmable circuit switching mechanisms (e.g., optical switches and FPGAs, depending on the network interface) allow a SDN controller to dynamically reconfigure the network and create shortcuts to heavy flows, reducing network latency and traffic on overloaded nodes.
5. **Topology-aware orchestration:** The NFV Orchestrator can make efficient decisions based on server monitoring information and knowledge of the topology, reducing end-to-end latency as VNFs are placed and chained in a more optimized way. For example, the Orchestrator can place VNFs in a service chain based on the less overloaded neighbors, or place VNFs in servers that are part of the shortest path between source and destination.
6. **Elimination of routing and flow tables:** To reduce latency on packet forwarding, the interconnection topology should present a lightweight routing algorithm that eliminates flow tables, enables layer 2 forwarding and reduces flow level signaling to the SDN Controller.
7. **Packet forwarding acceleration:** Mitigation of operating system bottlenecks can improve the performance of packet processing, increasing the throughput and reducing the latency of VNFs (e.g, Intel DPDK, and offload to SmartNICs or FPGAs). Another way to accelerate packet forwarding is the adoption of algorithmic forwarding.

4.1.2 Architecture design

Fig. 4.3 shows how VirtPhy fits in the ETSI NFV standard architecture [ETSI NFV ISG 2014]. Our focus lies on the NFVI, and on the NFV MANO components. Besides, specification [ETSI NFV ISG 2015] describes possible locations of an SDN Controller in a NFV framework. In our case, the SDN Controller is part of the NFVI and exchanges information about the network with the NFV Orchestrator via an orchestration interface [ETSI NFV ISG 2015]. In addition, to support persistence of infrastructure information, a new communication interface between the VIM and the Data Repositories was created.

The main components of VirtPhy architecture are shown in Fig. 4.3 and explained as follows:

1. **VIM:** It manages the interaction of VNFs with hardware resources, as well as their virtualization, by interacting directly with hypervisors. Also, it sets the VNF address during its creation (or migration) according with the physical host. Besides, it manages virtual networks.

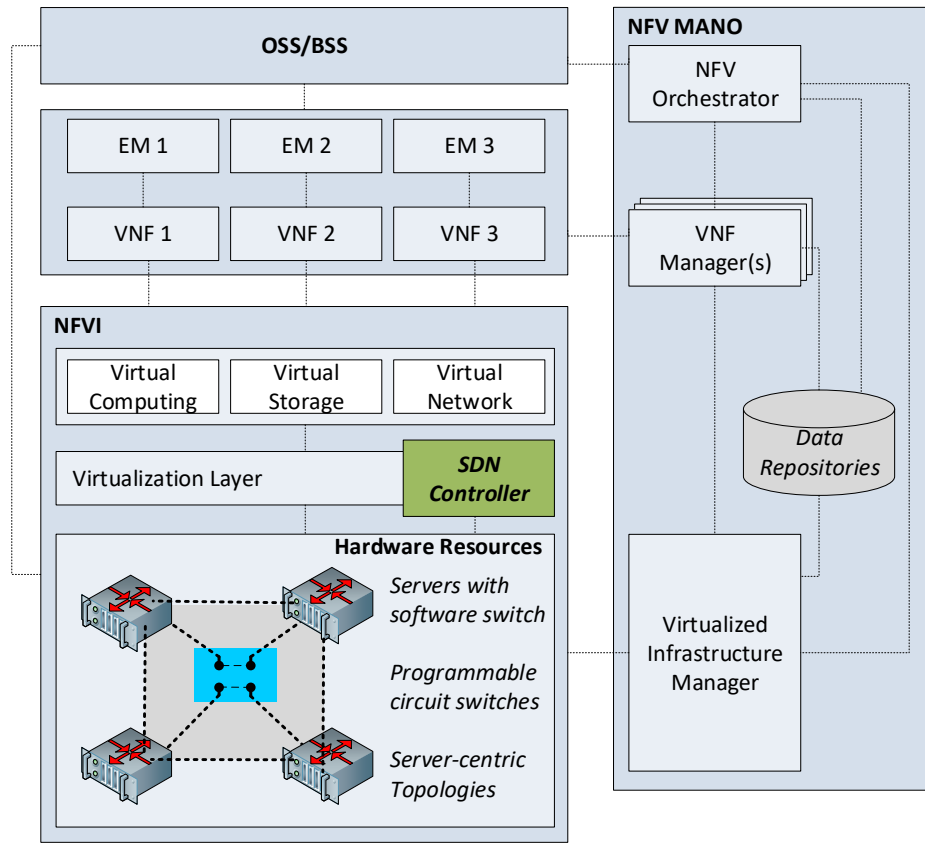


Figure 4.3: VirtPhy and ETSI NFV standard. Source: [Dominicini et al. 2017].

2. **NFV Orchestrator (NFVO)**: It centralizes the orchestration of NFVI resources and management of network services. It shares a common information base about VNF instances and resources with the other components, such as the VIM and the SDN controller.
3. **VNF Manager**: It is responsible for the life cycle management of VNF instances (e.g., instantiation, monitoring, scaling, and termination).
4. **SDN Controller**: It is responsible for configuring software switches of server nodes, for dynamically rewiring physical links using programmable circuit switches, and for installing flow rules related to SFC.
5. **Data Repositories**: It stores data about service requests (e.g., VNF-FGs, and users), VNFs (e.g., Ethernet MAC address, IP address, and host), physical hosts workload (e.g., CPU, and memory), network traffic, and topology.
6. **Hardware resources**: The servers are interconnected in a server-centric topology enabled with software switches and programmable circuit switches (e.g., optical switches or FPGAs), which enable physical link rewiring.

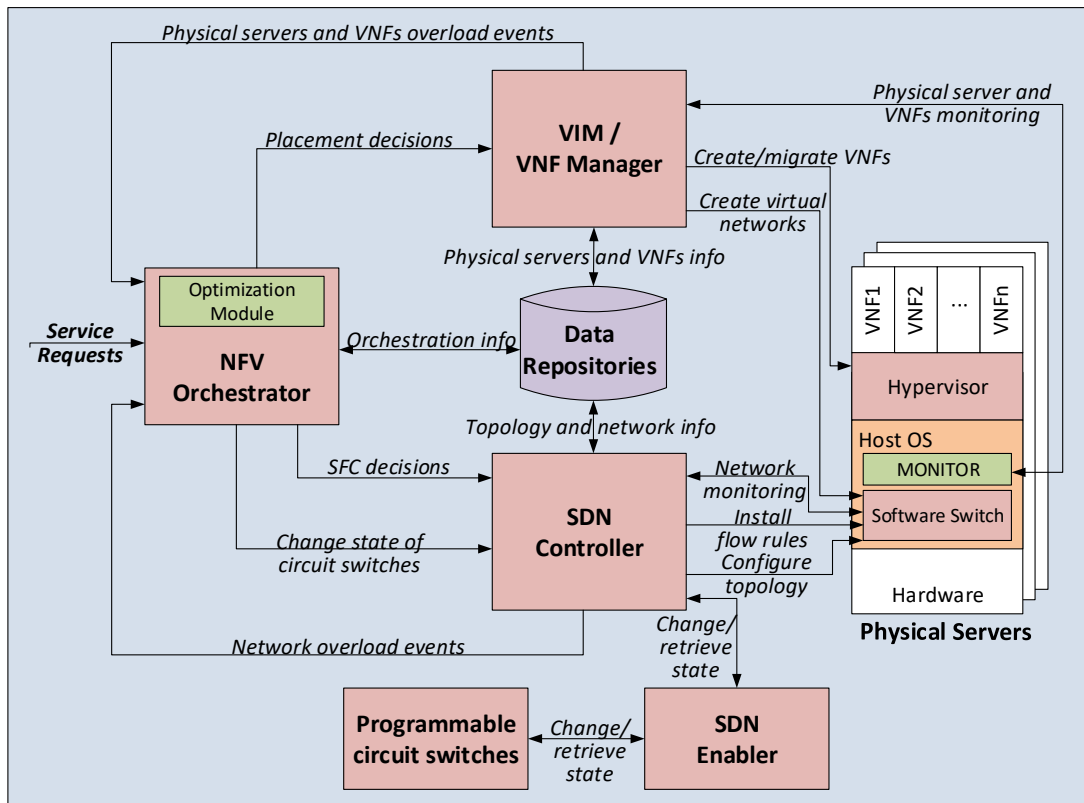


Figure 4.4: NFV orchestration in VirtPhy. Source: [Dominicini et al. 2017].

4.1.3 NFV orchestration architecture

Fig. 4.4 shows how VirtPhy enables NFV orchestration by the interaction of the main components described in Section 4.1.2. The Data Repositories receive information about physical servers and VNFs from the VIM and the VNF Manager, and information about the network status and topology from the SDN Controller. This information helps the Orchestrator to define the overall and available capacity from processing nodes and networking elements, and also what are the already existent VNF instances and where they are located.

When new service requests arrive, the NFV Orchestrator accesses the Data Repositories, computes the status of the physical infrastructure and runs an Optimization Module to take placement and SFC decisions, as described in Section 1.1.2. The user can specify service requests and corresponding VNF-FGs in TOSCA language¹ using a REST API to communicate with the Orchestrator, which stores a mapping in the Data Repositories between each VNF from the VNF-FGs and the corresponding identifiers from the VIM and SDN Controller. If the service request is updated, the NFV Orchestrator takes new decisions for placement and SFC.

Since one VNF can serve one or more service requests, depending on VNF capacity and security settings, the NFV Orchestrator may decide to serve a new request by assigning

¹<http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html>

it to an existing VNF, by creating a new VNF in a specific host or by migrating an existing VNF to a more suitable host. Based on its centralized view of the physical infrastructure, the NFV Orchestrator sends the placement decisions to the VIM, which is responsible to send migration and creation commands to the hypervisors in the selected physical servers, and to create the virtual networks that establishes connectivity between VNFs. Note that physical topology information is used for routing purposes, but this information is transparent for the VNFs, which have their own virtual networks and IP addressing, enabled by technologies such as VLANs, VXLAN or GRE tunnels. VirtPhy also enables an extra degree of freedom to the orchestration, since it is possible to modify the topology by changing the state of programmable circuit switches, such as optical switches or FPGAs. To this end, the NFV Orchestrator sends the commands to the SDN controller that, in turn, sends a message to a SDN Enabler component, which translates messages to the programmable circuit switches.

With the VNFs allocated, the NFV Orchestrator sends the SFC decisions to the SDN Controller, which communicates with the software switches of physical servers in order to create flow rules in these switches. The main idea behind our proposed scheme is that any traffic flow that traverses these switches and matches the flow rules for a given service request will be redirected to the next VNF in the ordered SFC. In this way, source and destination can communicate transparently, even when their traffic is steered through a long SFC sequence. To be able to create the flow rules, the SDN Controller needs to know information about the SFC requests and the VNFs that serve these requests. Thus, the NFV Orchestrator needs to share this information with the SDN Controller by saving all these details in the Data Repositories. Moreover, as VirtPhy adopts underlying server-centric topologies, the flow rules are distributed in the software switches of several physical servers instead of in few centralized hardware switches, reducing flow tables size. More details about the SFC scheme will be given in Section 4.2.4.

Finally, the servers contain a monitor module responsible for reporting to the VIM information about CPU usage, memory usage, and overload events. The SDN Controller also receives network monitoring information and overload events from the software switches. These overload events are sent to the NFV Orchestrator, which has to reevaluate placement and SFC decisions and apply changes in the infrastructure in order to meet the requirements of service requests.

4.1.4 NFV in a server-centric infrastructure

The fundamental problem of server-centric approaches is that server CPU is shared among processing tasks (i.e., VNFs) and network tasks (i.e., routing/forwarding). This can lead either to poor VNF performance or limited bandwidth bisection between servers due to busy CPUs in intermediate elements. This implies more complex orchestration when

compared to network-centric approaches whereas those tasks are decoupled.

Although VirtPhy already uses a lightweight kernel-level routing/forwarding, it takes precedence over user-level processes (i.e., VNF processing). Therefore, mechanisms should be available to avoid CPU saturation on server nodes due to transit traffic (traffic originated when the node is not the final destination, but it is used for forwarding purposes). As a corrective mechanism, circuit switches can be used for reconfiguring physical links in order to bypass overloaded intermediate nodes. Note, however, that changing the physical topology may affect the routing mechanism for the whole network.

This issue could be solved by limiting the number of circuit switching elements and their placement in the topology. Orchestrator must be aware of those constraints to avoid route loops and other routing problems. One preventive mechanism is to monitor an adaptive CPU threshold that limits the amount of transit traffic in a server node: the higher the CPU usage of a node, the lower is the acceptable limit of traffic transit for that node. If at any time this threshold is exceeded, the Orchestrator actuates on the network.

In practice, these mechanisms can be well coordinated by a Orchestrator with a centralized view, either proactively, when making placement and SFC decisions for new services, or reactively, when receiving overload events from server nodes. If the Orchestrator receives CPU, network, or memory overload events, one possible action is to migrate VNFs from the overload server node to another one with lower load or to scale the VNFs to meet the increasing demand. If the event is transit traffic overload, the Orchestrator could also check the transit traffic over the neighbors of that server node. If it is possible to redirect the transit traffic to another server, it could reconfigure the physical link.

Another way to avoid the consumption of CPU cores with forwarding tasks is to offload networking processing tasks to network co-processors, such as SmartNICs ². In this way, the resources of the software switch datapath are liberated for VNF processing tasks.

Despite the mechanism selected by the Orchestrator, it has to consider the impact of each action on the network as whole. In fact, the stability of the entire network depends on the continuous monitoring of all the network nodes to support informed decisions on how the Orchestrator can actuate to efficiently use the infrastructure to enable existing and new VNF requests. In [Dominicini et al. 2016], the work investigated the effects of the forwarding tasks on user processes performance at server nodes in a server-centric hypercube topology and demonstrated VirtPhy's ability to redistribute transit traffic to save CPU for user processes. In the proposed scheme, it was also possible to reconfigure the connections to reduce the average number of hops for some flows, without affecting the node degree and without imposing modifications in the original routing strategy.

Appendix B performs a preliminary investigation, using a ILP model, if server-centric topologies can efficiently provision diverse NFV workloads for edge DCs when compared to network-centric topologies.

²<https://www.netronome.com/products/smartnic/overview/>

4.2 Proof-of-concept

This section covers the practical implementation aspects of applying the VirtPhy reference architecture presented in the previous section to a specific topology in a real cloud environment. As a proof-of-concept, we chose the OpenStack cloud platform in a DC where servers are interconnected in a hypercube server-centric topology. Other topologies could be used, such as mesh or torus [Mirza-Aghatabar et al. 2007], as long as it is possible to implement tableless algorithmic forwarding. The hypercube topology was selected because it has a native routing algorithm, allows insertion of 2X2 optical switches in its faces, and presents multiple paths between pairs of nodes.

4.2.1 The hypercube topology and routing mechanism

The routing in hypercube is based on a algorithmic mechanism, in which a server uses a single XOR operation over its neighbors to find the next hop. This scheme is simple, efficient, and free of lookups in route tables. Besides, one important property of this mechanism is the intrinsic fault tolerant capability. If the closest neighbor is not available, the routing protocol can send the packet to another neighbor.

We use the addressing scheme from TRIIAD [Vassoler 2015], which is inspired in Portland [Niranjan et al. 2009], where a global identifier is replaced by a positional identifier. Particularly, the MAC address is divided in two parts: the first most significant 32 bits represent the hypercube address of the physical server, and the other remaining 16 bits identify the specific VNF in that physical server (VNF ID), as shown in Fig. 4.5.

Fig. 4.5 shows an example of how a packet is routed when VNF2 at server 1 sends packets to VNF1 at server 3. Initially, VNF2 sends a ARP request broadcast message with the IP of VNF1. This message is intercepted by the software switch of server 1 and forwarded to the SDN controller that resolves the address and returns an ARP reply message to the software switch, which, in turn, delivers the message to VNF2. Then, VNF2 can normally send data to VNF1 using the IP address. When a server node receives a packet, the software switch extracts the hypercube address from the 32 more significant bits of the MAC destination field and executes a XOR over this address and its own hypercube address to find the neighbor closest to the packet destination, and forwards the packet to it. The MAC destination field is preserved during all the path from VNF2 at server 1 to VNF1 at server 3. When it reaches the destination physical server (server 3), the second part of the MAC destination address (the VNF ID) is used to identify VNF1 in that server. Note that intermediate nodes forward packets directly in the software switch layer as there is no VM involved in the intermediary hops.

All ARP information is centralized in the Data Repositories, managed by the VIM and accessed by the SDN Controller. In this way, the NFV Orchestrator can migrate a VM without changing its IP address, transparently to layer 3. Nevertheless, in the migration,

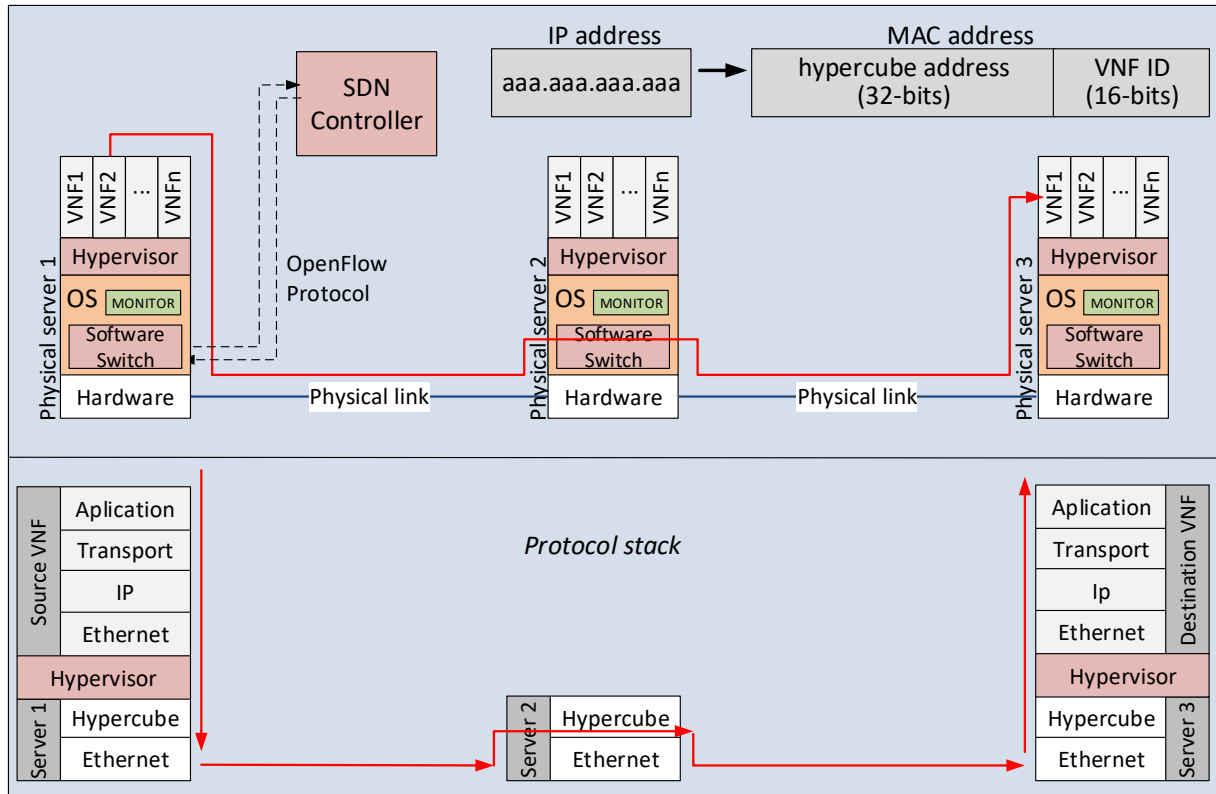


Figure 4.5: Forwarding mechanism using MAC to locate and identify a VNF. Source: [Dominicini et al. 2017].

the NFV Orchestrator communicates with the VIM to update the MAC address of the VM to reflect its new position in hypercube. Moreover, the proposed solution has the following advantages over traditional SDN solutions:

- The forwarding mechanism only needs to read the MAC destination address from the Ethernet frame, instead of analyzing Ethernet, IP and TCP headers.
- Considering routing, the SDN Controller is only used to map IP/MAC pairs, since servers execute forwarding using hypercube routing mechanism in a decentralized way without communicating with the SDN Controller and overloading the network with control messages;
- As each physical server has its own software switch that supports OpenFlow protocol, they can be directly and individually managed by the SDN Controller.

4.2.2 The testbed

The testbed is a 3D-hypercube [Saad and Schultz 1989] composed by eight server nodes. We used twelve servers with Linux Ubuntu: eight physical servers, one NFV Orchestrator, one OpenStack Controller node, one OpenStack Network node, and one SDN Controller. Each physical server has five Ethernet NICs: three are used to create the 3D-hypercube

topology, one is used for a data network that is connected to the OpenStack Network node, and one is used for a management network that is connected to the OpenStack Controller node. We used 1Gbps links and servers with 16GB memory and Intel Xeon E5-2620 2.4 GHz processors. Our testbed does not include optical switches, because their effect was already investigated in [Dominicini et al. 2016].

In Fig. 4.5, we show the logical organization of a server node: at the top, there are the VNFs; in the middle, there is the host operating system that integrates the hypervisor, based on libvirt, to the OvS software switch and the monitor module; and on the bottom, there are the actual hardware resources. The monitor module is implemented as a daemon running in user space. We use the OvS implementation from TRIIAD [Vassoler 2015], which modifies the original OvS in such way that the SDN controller can set the node forwarding mechanism (e.g., XOR or flow tables), using OpenFlow messages. To perform routing and forwarding in kernel level, this OvS version implements the hypercube routing algorithm in the OvS datapath module.

The VIM and the VNF Manager components were implemented using OpenStack [Openstack 2017], with some modifications to enable VirtPhy features. For instance, the functions to create and to migrate VMs were modified to setup the MAC addresses according with the proposal. The NFV Orchestrator was implemented in Python and has the following main functionalities: (i) to receive service requests; (ii) to run the Optimization Module to produce placement and SFC decisions; (iii) to communicate placement decisions to VIM and VNF Manager; (iv) to communicate SFC decisions to the SDN Controller; (v) to take actions when receiving overload events; and (vi) to manage Data Repositories.

The SDN Controller was implemented using the Ryu platform [Ryu 2015] and has the following main functionalities: (i) to discover and monitor available nodes and links; (ii) to build the virtual hypercube topology; (iii) to install SFC rules; and (iv) to configure server nodes according with their position in the hypercube. This last task depends on two custom OpenFlow messages exchanged between the SDN Controller and the OvS in each server node, as shown in Fig. 4.5. The first message sends the hypercube node address and defines the XOR as the routing mechanism. The second message sends the node neighbors and the respective physical port of each neighbor. When our modified version of OvS receives these messages, it reconfigures its datapath module to forward the packets using the hypercube routing function, which chooses the next hop without exchanging information with the SDN controller.

4.2.3 SFC using SDN

In [Vacaro et al. 2016], the authors present a scheme that uses OpenFlow switches, legacy network functions appliances, and virtual MACs (VMACs) to enable dynamic SFC. The

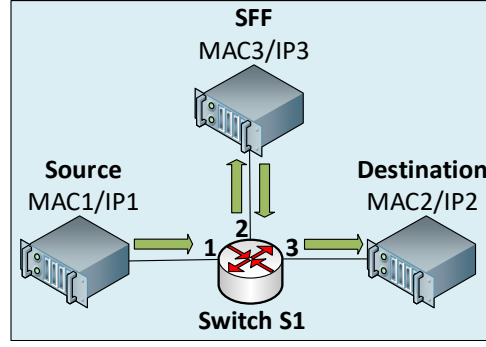


Figure 4.6: Basic SFC strategy using SDN. Source: [Dominicini et al. 2017].

Table 4.1: Flow table at switch S1.

Rule	Match	Action
1	SourceIP=IP1	Modify Dest_MAC to VMAC1; Output to port 2.
2	Dest_MAC=VMAC1	Modify Dest_MAC to MAC2; Output to port 3.

main idea is that, for each hop, you replace the destination MAC by a VMAC, which will be used as a match condition in a flow rule that steers the traffic to the next VNF in a SFC. Thus, the VMAC is a path indicator and can receive any value as long as there is no real MAC with the same value in the network domain.

Fig. 4.6 exemplifies this idea: source and destination are not aware of the chaining between them, and all nodes are connected to a OpenFlow switch S1. SFF is a network function that only receives packets and forwards them back to receiving interface. The traffic is originated in source node (switch port 1), pass through SFF node (switch port 2), and, finally, reaches the destination node (switch port 3). The strategy installs flow rules in the switch S1 according to Table 4.1, and the SFC will happen as follows: (i) source node sends a packet to destination node; (ii) packet reaches switch port 1 and matches rule 1, which changes Destination MAC to VMAC1 and redirects the packet to SFF (switch port 2); (iii) SFF node receives the packet and forwards it back to the same interface; (iv) packet reaches switch port 2 and matches rule 2, which changes Destination MAC to MAC2 and redirects the packet to the destination (switch port 3); (iv) destination node receives exactly the same packet sent by source node.

This strategy has some benefits as it is simple, scalable, low cost and compliant with legacy network appliances, as long as the appliances are connected to SDN switches [Vacaro et al. 2016]. However, as already explained in Section 3.1, the NFV scenario is more complex, and involves software switches in the physical servers to interconnect VMs and isolate tenants. Section 4.2.4 will show how this basic strategy was extended in VirtPhy.

4.2.4 SFC in OpenStack and Hypercube

This section explains how we designed a SFC approach that is compliant with VirtPhy reference architecture and works in a OpenStack DC where servers are interconnected in a hypercube server-centric topology. Our proposal decouples SFC from routing and considers the following aspects:

- As explained in Section [A.2.1](#), server nodes in OpenStack present a networking structure based on two OvS bridges: a br-int bridge that is responsible for distributing the packets to the correct VMs when they reach the physical server; and a br-eth bridge that handles communication between VMs that reside in different physical servers. As a implementation decision in the physical servers, we split the functionality of the software switch from VirtPhy reference architecture in two software switches: the br-int and the br-eth. The first (br-int) will be responsible for intercepting traffic and for directing it to the right VNF in the SFC sequence. Thus, the OpenFlow rules for executing SFC will be installed in the br-int bridge of each physical server. The later (br-eth) will be responsible for executing the hypercube routing based on XOR operation between physical servers (see Section [4.2.1](#)).
- We take benefit of our proposed hypercube addressing scheme (see Section [4.2.1](#)) to extend the SFC strategy presented in Section [4.2.3](#) and create VMACs that already represent the address of the physical server that hosts the next VNF in the chaining. For instance, if the next hop in the chaining is a VNF that is hosted by a server node with hypercube address 3, the VMAC will necessarily have the format 00:00:00:03:XX:XX, where XX:XX can be any hexadecimal digit as long as there is no VM in that physical server with this VNF ID (e.g., 00:00:00:03:11:3B). In that way, the SFC strategy is transparent to the br-eth bridges of the hosts in the routing path, which only check the hypercube address (first 32 bits of MAC destination field) for routing. When the packets reach the host where the VNF is actually hosted, the full VMAC will match an existing rule that was pre-installed by the SDN controller in the br-int, and the traffic will be redirected to the correct VNF.
- The NFV Orchestrator runs the Optimization Module to generate placement and SFC decisions. The SDN controller is responsible to convert these SFC decisions in flow rules for each one of the br-int bridges of the physical servers. As already explained, our strategy does not install any flow rule in br-eth bridges.
- OpenStack automatically installs a set of iptables rules in the Linux bridge connected to each VM to provide a protection against ARP spoofing (e.g., a host can only send packets to the network with its own MAC source and IP source pair). To enable some

NFV use cases with forwarding functions that would be blocked by this protection (e.g., router, firewall, and NAT), our implementation disables the related rules for the specific ports that are connected to the VNFs.

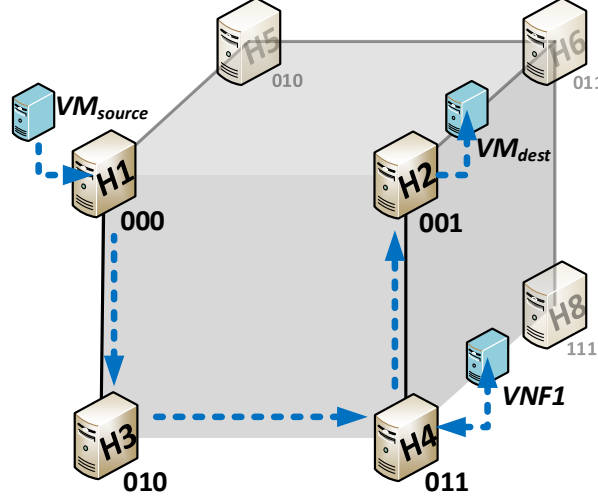


Figure 4.7: SFC example in a hypercube with degree 3. Source: [Dominicini et al. 2017].

To explain our strategy, consider the example of Fig. 4.7, in which all the traffic from VM_{source} to VM_{dest} must be steered to a $VNF1$ of type t_1 . In our example, this function receives the packets, do some processing, and send the packets back to the same interface. In the initial scenario, VM_{source} is already created in host $H1$ (hypercube address 000_{BIN} or 0_{DEC}) at br-int port 5 with MAC $00:00:00:00:e4:7d$, and VM_{dest} is already created in host $H2$ (hypercube address 001_{BIN} or 1_{DEC}) at br-int port 5 with MAC $00:00:00:01:54:1b$, and there is no VNF of type t_1 in the DC. When the orchestrator receives the service request, the Optimization module will analyze the physical infrastructure status to decide where to place the $VNF1$. Consider it decides to allocate $VNF1$ at host $H4$ (hypercube address 011_{BIN} or 3_{DEC}) at br-int port 5 with MAC $00:00:00:03:bb:02$. Because of hypercube inherit source routing mechanism, there is no need to decide the physical paths between each node in the SFC, but it is still necessary to force the traffic that would normally flow from VM_{source} to VM_{dest} via link $H1$ - $H2$ to go through $VNF1$.

Table 4.2 shows the SFC rules that are installed in each of the physical servers for this example. When VM_{source} sends a flow traffic to VM_{dest} the following events will happen:

1. **At H1:** The packet leaves VM_{source} , and, reaches the br-int bridge. There, the flow matches **rule 1**, causing the destination MAC field to be changed to $VMAC1$ ($00:00:00:03:00:0a$) and the packet to be forwarded to br-eth according to NORMAL flow action. When the packet reaches br-eth, the forwarding mechanism implemented in OvS applies the XOR operation over the hypercube address of $H1$ (000_{BIN}) with the hypercube address portion of the destination MAC address ($00:00:00:03:00:0a \rightarrow 011_{BIN}$), which gives $000 \text{ XOR } 011 = 011$, meaning there

Table 4.2: Flow rules for SFC example.

Rule	Node	MAC_Src	MAC_Dst	In_Port	Action
1	H1 (000)	H1_MAC	*	*	Modify Dst_MAC to VMAC1; Output: Normal.
2	H4 (011)	*	VMAC1	*	Output: Port 5.
3	H4 (011)	*	VMAC1	5	Modify Dst_MAC to VMAC2; Output: Normal.
4	H2 (001)	*	VMAC2	*	Modify Dst_MAC to H2_MAC; Output: Normal.

VMAC1 = 00:00:00:03:00:0a, VMAC2 = 00:00:00:01:00:0b, H1_MAC= 00:00:00:00:e4:7d, H2_MAC= 00:00:00:01:54:1b.

are two possible moves: **010** (change second bit of 000) or **001** (change third bit of 000). Consider first option is chosen and the packet is forwarded to host *H3* (hypercube address 010_{BIN} or 2_{DEC}).

2. **At H3:** When the packet reaches br-eth, the forwarding mechanism in OvS applies the XOR operation over the hypercube address of *H3* (010_{BIN}) with the hypercube address portion of the destination MAC address ($00:00:00:03:00:0a \rightarrow 011_{BIN}$), which gives $010 \text{ XOR } 011 = 001$, meaning there is only one possible move: **011** (change third bit of 010). Therefore, packet is forwarded to host *H4* (hypercube address 011_{BIN} or 3_{DEC}). Note the packet does not go up to br-int, because *H3* is only forwarding traffic.
3. **At H4:** When the packet reaches br-eth, the forwarding mechanism in OvS verifies that the destination MAC address ($00:00:00:03:00:0a$) is a VM in that same host, and forwards the packet to br-int. When the packet reaches br-int, the flow matches **rule 2**, causing the packet to be forwarded to port 5, where *VNF1* is connected. *VNF1* processes the packet and forwards it back to the same interface. The returning packet again reaches br-int and matches **rule 3**, causing the destination MAC field to be changed to VMAC2 ($00:00:00:01:00:0b$) and the packet to be forwarded to br-eth according to NORMAL flow action. When the packet reaches br-eth, the forwarding mechanism in OvS applies the XOR operation over the hypercube address of *H4* (011_{BIN}) with the hypercube address portion of the destination MAC address ($00:00:00:01:00:0b \rightarrow 001_{BIN}$), which gives $011 \text{ XOR } 001 = 010$, meaning there is only one possible move: **001** (change second bit of 011). Therefore, packet is forwarded to host *H2* (hypercube address 001_{BIN} or 1_{DEC}).
4. **At H2:** When the packet reaches br-eth, the forwarding mechanism in OvS verifies that the destination MAC address ($00:00:00:01:00:0b$) points to a VM in that same host, and forwards the packet to br-int. When the packet reaches br-int, the

flow matches **rule 4**, causing the destination MAC field to be changed to VM_{dest} MAC (00:00:00:01:54:1b), and the packet to be forwarded to port 5, where VM_{dest} is connected. Finally, the packet that reaches destination is exactly equal to the packet that left source and, therefore, the SFC happens transparently to VM_{source} and VM_{dest} .

4.3 Evaluation

In order to evaluate the proposed orchestration architecture (Section 4.1) and its SFC strategy (Section 4.2.4), three set of experiments were carried out in the testbed described in Section 4.2.2. The first set aims to demonstrate that VirtPhy orchestration architecture can provision a service chaining request. The second set aims to analyze the impact of the proposed SFC strategy in the service performance, considering latency and jitter. Finally, we present a throughput analysis of the proposed architecture. Each test was run 30 times and the graphs present the mean of the different runs. Variance, standard deviation, and confidence intervals are not shown because they were not significant.

The test scenario is illustrated in Fig. 4.8. Initially, virtual machines VM_{source} and VM_{dest} are hosted by servers $H1$ and $H2$, respectively. As shown in Fig. 4.8.a, the NFV Orchestrator receives a service request $r1 = \langle VM_{source}, VM_{dest} : 5000, [t_1, t_2], 200Mbps \rangle$, i.e., a 200Mbps traffic originated in VM_{source} and destined to VM_{dest} , port 5000, must pass through a VNF of type t_1 and VNF of type t_2 , in this order. For testing purposes, we consider types t_1 and t_2 implement a socket application that receives the packets and send them back to the same network interface.

From the service request, the NFV Orchestrator decides to place VNF $SFF1$ of type t_1 and $SFF2$ of type t_2 at servers $H3$ and $H4$, respectively. This placement decision

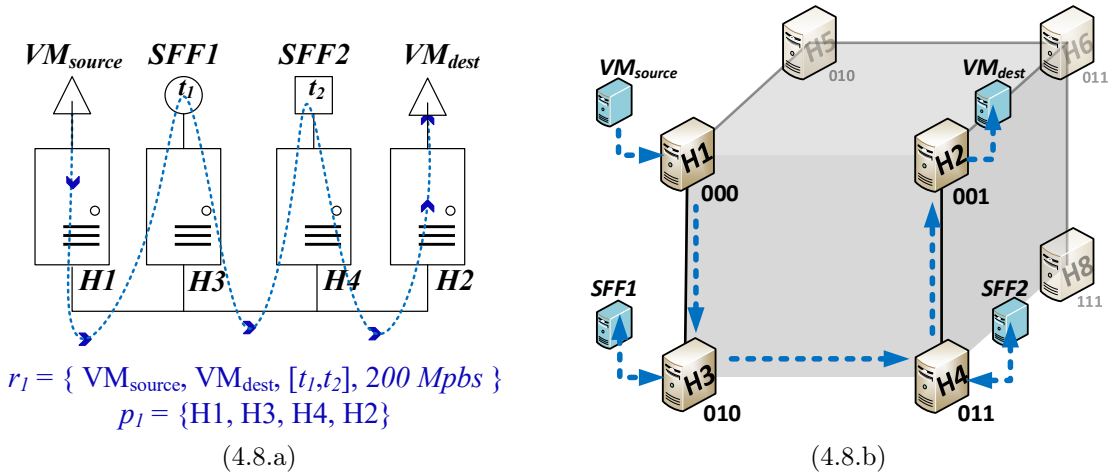


Figure 4.8: SFC test: (a) Service request. (b) SFC scenario: $VM_{source} \rightarrow SFF1 \rightarrow SFF2 \rightarrow VM_{dest}$. Source: [Dominicini et al. 2017].

is passed to the VIM component, implemented in OpenStack as described in Section 4.2.2, that creates the virtual machines for $SFF1$ and $SFF2$ in the assigned host (with the correct MAC addresses) and attaches them to a virtual network. All the VMs and VNFs are connected to the same private network, which means they do not need the intermediation of the Network node from OpenStack to communicate with each other and use the hypercube connections for that purpose.

After the placement, the NFV Orchestrator communicates with the SDN Controller that installs flow rules as shown in Table 4.3 according to our proposed SFC strategy (Section 4.2.4): rules number 1, 3, 4, 7, 8, and 11 are used to steer traffic in the direction $VM_{source} \rightarrow SFF1 \rightarrow SFF2 \rightarrow VM_{dest}$, and rules number 2, 4, 6, 8, 10, and 12 are used to steer traffic in the opposite direction ($VM_{dest} \rightarrow SFF2 \rightarrow SFF1 \rightarrow VM_{source}$). At this point, placement and SFC decisions are already applied in the testbed using our orchestration architecture, as shown in Fig 4.8.b, and tests can be executed.

Table 4.3: Flow Rules for SFC Test.

Rule	Node	Priority	In_Port	MAC_Dst	IP_Src	UDP_Port_Dest	UDP_Port_Src	Action
1	H1 (000)	100	*	*	H1_IP	5000	*	Modify MAC_Dst to VMAC1; Output: Normal.
2	H1 (000)	101	*	VMAC6	*	*	*	Modify MAC_Dst to H1_MAC; Output: Port 5.
3	H3 (010)	100	*	VMAC1	*	*	*	Output: Port 5.
4	H3 (010)	101	5	VMAC1	*	*	*	Modify MAC_Dst to VMAC2; Output: Normal.
5	H3 (010)	102	*	VMAC5	*	*	*	Output: Port 5.
6	H3 (010)	103	5	VMAC5	*	*	*	Modify MAC_Dst to VMAC6; Output: Normal.
7	H4 (011)	100	*	VMAC2	*	*	*	Output: Port 5.
8	H4 (011)	101	5	VMAC2	*	*	*	Modify MAC_Dst to VMAC3; Output: Normal.
9	H4 (011)	102	*	VMAC4	*	*	*	Output: Port 5.
10	H4 (011)	103	5	VMAC4	*	*	*	Modify MAC_Dst to VMAC5; Output: Normal.
11	H2 (001)	100	*	VMAC3	*	*	*	Modify MAC_Dst to H2_MAC; Output: Port 5.
12	H2 (001)	101	*	*	H2_IP	*	5000	Modify MAC_Dst to VMAC4; Output: Normal.

VMAC1 = 00:00:00:02:00:0a, VMAC2 = 00:00:00:03:00:0b, VMAC3 = 00:00:00:01:00:0e, VMAC4 = 00:00:00:03:00:0c, VMAC5 = 00:00:00:02:00:0d, VMAC6 = 00:00:00:00:00:0f, H1_MAC = 00:00:00:00:e4:7d, H2_MAC = 00:00:00:01:54:1b.

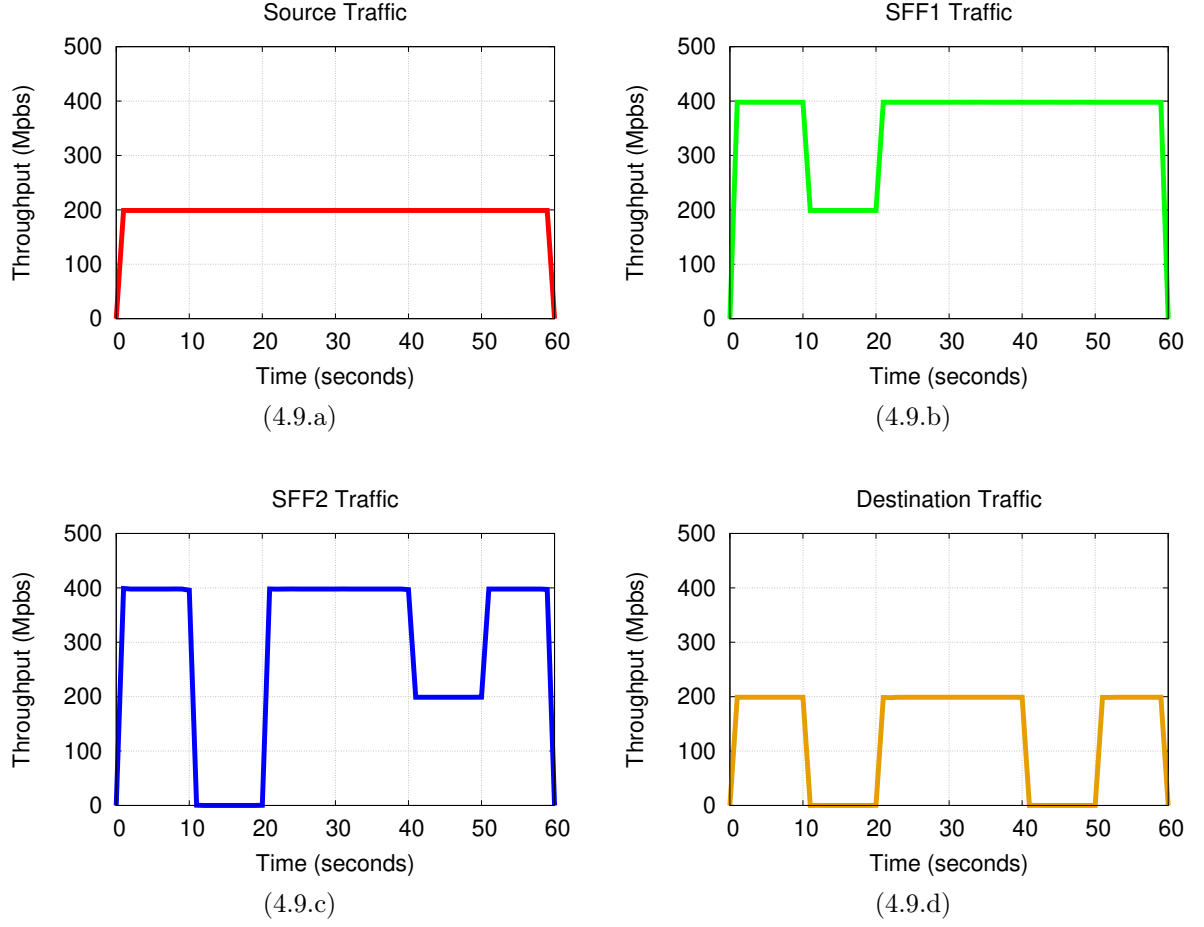


Figure 4.9: SFC Test 1: iperf traffic for SFC $VM_{source} \rightarrow SFF1 \rightarrow SFF2 \rightarrow VM_{dest}$: (a) Traffic at source. (b) Traffic at SFF1. (c) Traffic at SFF2. (d) Traffic at destination. Source: [Dominicini et al. 2017].

In the first test (Fig. 4.9), we consider only the one way traffic originated in VM_{source} and terminated in VM_{dest} . During the test duration, VM_{source} sends a 200Mbps UDP traffic using the iperf tool to VM_{dest} , port 5000, and this traffic must be steered to VNFs $SFF1$ and $SFF2$, in this order. Initially, as shown in the interval from 0 to 10s in Fig. 4.9, $SFF1$ and $SFF2$ are forwarding traffic. Because the VNFs receive the traffic and send it back to the same interface, it is possible to see the total bandwidth (400Mbps) perceived by them is twice the bandwidth sent by VM_{source} (200Mbps), as shown in Figs. 4.9.b and 4.9.c. VM_{dest} receives 200Mbps of traffic during this interval (Figs. 4.9.d).

From 10s to 20s, the forwarding function at $SFF1$ is turned off and, as expected, we can see the following consequences: traffic drops from 400 Mbps to 200 Mbps at $SFF1$ (Fig. 4.9.b), because traffic enters in network interface, reaches $SFF1$, but does not return; and traffic drops to zero in $SFF2$ and VM_{dest} (Fig. 4.9.c and 4.9.d), because it was interrupted at $SFF1$. From 20s to 40s, $SFF1$ is turned on again and the traffic returns to initial conditions. From 40s to 50s, the forwarding function at $SFF2$ is now turned off and, as expected, we can see the following consequences: traffic continues at

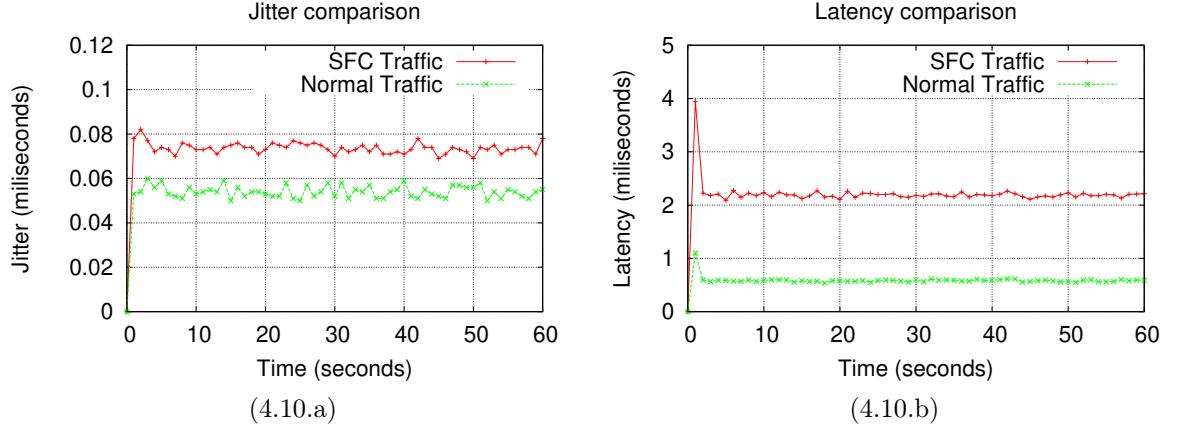


Figure 4.10: SFC tests for traffic passing through SFC and normal traffic going directly from source to destination. (a) Test 2: Jitter over time. (b) Test 3: Latency over time. Source: [Dominicini et al. 2017].

400 Mbps $SFF1$ (Fig. 4.9.b), because the interruption happens after $SFF1$; traffic drops from 400 Mbps to 200 Mbps at $SFF2$ (Fig. 4.9.c), because traffic enters in network interface, reaches $SFF2$, but does not return; and traffic drops to zero in VM_{dest} (Fig. 4.9.d), because it was interrupted at $SFF2$. From 50s to 60s, $SFF2$ is turned on again and the traffic returns to initial conditions.

In the second test, we compare jitter in the SFC scenario with three hops (Fig. 4.8.b, direction $VM_{source} \rightarrow SFF1 \rightarrow SFF2 \rightarrow VM_{dest}$) with the normal scenario when traffic goes directly from VM_{source} to VM_{dest} without SFC (one single hop). During the test duration, VM_{source} sends a 200Mbps UDP traffic using the iperf tool to VM_{dest} , and $SFF1$ and $SFF2$ are forwarding traffic without interruption. Fig 4.10.a shows that the jitter has increased from approximately 0.06ms to approximately 0.08ms when performing SFC. Thus, even with addition of two extra hops, the SFC didn't cause great degradation in the jitter, which is small in both scenarios.

To understand SFC impact over latency, a third test was performed using the ping tool (Fig. 4.10.b). Similarly to the jitter experiment, we considered a scenario with the same SFC (Fig. 4.8) and another scenario with direct communication between VM_{source} and VM_{dest} . This test is important, because, differently from tests 1 and 2, it needs rules for both directions: $VM_{source} \rightarrow SFF1 \rightarrow SFF2 \rightarrow VM_{dest}$ (ping request), and $VM_{dest} \rightarrow SFF2 \rightarrow SFF1 \rightarrow VM_{source}$ (ping reply). The results in Fig. 4.10.b show that latency increases from approximately 1ms to approximately 2ms in the scenario with SFC when compared with the scenario without SFC. This latency growth is acceptable and expectable, because we added two more hops with SFC and the forwarding functions were implemented in application layer using sockets.

The fourth test (Fig. 4.11) compares the throughput sent by VM_{source} with the throughput received in VM_{dest} , in the SFC scenario (Fig. 4.8) and in the normal sce-

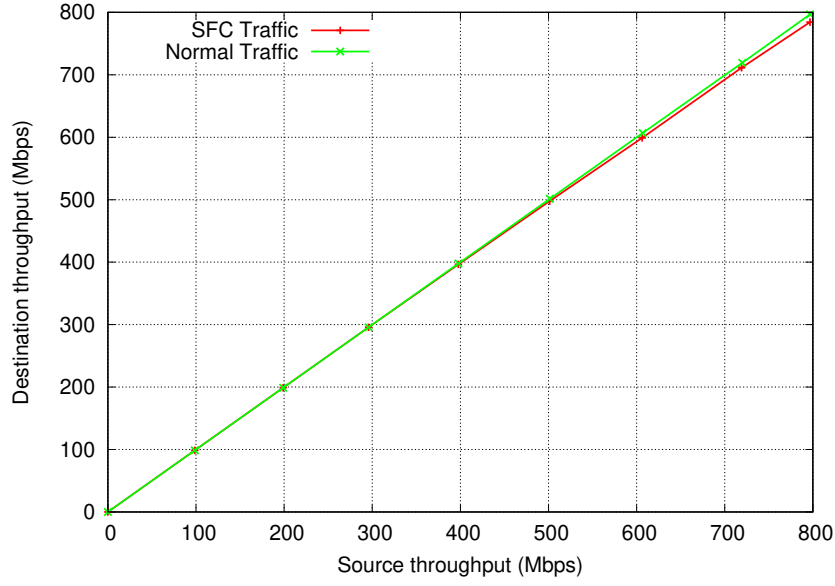


Figure 4.11: SFC Test 4: Source throughput vs. Destination throughput for SFC traffic and normal traffic going directly from source to destination. Source: [Dominicini et al. 2017].

nario without SFC. We incremented the source throughput from 100Mbps to 800Mbps, in a total of eight iterations. For each iteration, we sent a UDP traffic from VM_{source} to VM_{dest} during 60s, using the iperf tool. For each source throughput value, Fig 4.11 shows the mean of all the destination throughput measurements considering the duration of each iteration (60s). As expected, in the normal scenario without SFC (only one hop), the packet loss is negligible, i.e., the destination throughput is approximately equal to the source throughput. However, this graph also shows that even with the addition of two more hops in the SFC scenario, the throughput perceived by the destination is very close to the throughput sent by the source and, in the worst case, when the source throughput is 800Mbps, the packet loss is at most 1.5%.

Since network functions (SFF1 and SFF2) use an application layer forwarding based on sockets, results from Tests 3 and 4 could be further improved if we use lower layer network functions or technologies that bypass the conventional Operating System network stack, such as Intel DPDK.

4.4 Concluding remarks

With the proposal of VirtPhy, a fully programmable NFV orchestration architecture based on server-centric topologies, we showed that DC network infrastructures and orchestration mechanisms based on server-centric DCNs can meet the requirements of NFV services in edge DCs. Moreover, we presented a distributed SFC scheme, which integrates NFV and SDN to take benefit of the physical network topology and enable SFC in DC environments based on software switches.

Compared to traditional approaches, VirtPhy provides more information about infrastructure and topology, and also, more orchestration mechanisms to actuate in the infrastructure in real time. Therefore, the NFV Orchestrator is able to make more efficient and agile orchestration decisions. In addition, server-centric topologies along with software switches and open source protocols allow VirtPhy to support innovation with low cost, which is specially important for edge DCs. Moreover, the integration between the NFV Orchestrator and the SDN Controller improves placement and SFC decisions, while the adopted algorithmic routing scheme reduces control plane overload, allowing VirtPhy to meet throughput and latency constraints for NFV services.

To demonstrate VirtPhy, we implemented a testbed in a real DC with a production cloud platform based on the hypercube topology. The experiments showed the feasibility of using VirtPhy orchestration mechanisms to provision NFV services that traverse a chain of VNFs. Besides, results demonstrated that our distributed SFC solution is able to minimize performance impacts on latency, jitter, and throughput, when steering a traffic through a SFC.

As future work, we plan to investigate other routing schemes and server-centric topologies. In addition, we plan to tackle performance issues of software switches by implementing Kernel-Bypass features with Intel DPDK, and to explore other technologies that promise programmable packet processing at line rate, such as SmartNICs and NetFPGAs.

VirtPhy exploits specific properties of the underlying topology to perform SFC. This can be considered an advantage for gaining performance in some cases. However, this may also restrict the infrastructures where it can be deployed, especially because some topologies do not provide a efficient routing algorithm, and some algorithms may not work properly in the event of link or node failures. Moreover, for some use cases the use hybrid or network-centric DCNs may be more adequate than server-centric DCNs. Besides, the path selection may be limited by the algorithmic routing of the underlay.

The next chapter will extend the ideas developed in VirtPhy to present KeySFC, a SFC scheme with topology-independent data plane implementation that can be deployed in any DCN. It is based on RNS-based SSR and simple *modulo* operations for taking forwarding decisions. In this way, KeySFC can cover broader SFC scenarios and achieve improved expressiveness, while delivering good performance.

Chapter 5

KeySFC: SFC Scheme

This chapter proposes KeySFC [Dominicini et al. 2019], a new SFC scheme that explores algorithmic SSR. We argue that to make optimal use of network capacity, it is crucial to replace tables on the routing stage by an efficient algorithmic forwarding mechanism, using tables only to classify flows for SFC. To that purpose, we extend the idea of network fabric [Casado et al. 2012, Martinello et al. 2014] to intra-DC server-based networking, creating a clear separation between (i) programmable edge elements that provide SFC classification using software switches, and (ii) core elements that are only devoted to efficient packet transport using a RNS-based SSR mechanism [Martinello et al. 2014] (see Section 2.2.1). Also, in contrast to traditional SFC solutions that focus only on network-centric DCNs, KeySFC supports server-centric and hybrid DCNs. Moreover, we implement and validate KeySFC in a proof-of-concept testbed orchestrated by OpenStack, demonstrating its feasibility in production DCs.

Section 5.1 presents the KeySFC proposal. Afterwards, we describe our proof-of-concept implementation and its evaluation in Section 5.2. Finally, we outline the conclusions and future works in Section 5.4.

5.1 Proposal

5.1.1 Architecture

The KeySFC architecture, shown in Fig. 5.1, complies with the ETSI NFV standard [ETSI NFV ISG 2014], where hardware resources are managed by a MANO block, formed by a VIM, VNFM, and a NFVO (see Section 2.4.2). The NFV Infrastructure block is composed by COTS servers and switches, both programmable by an SDN Controller. In addition, there is a data repository shared by these blocks.

The NFVO is responsible for traffic engineering, and generates placement and SFC decisions based on service requests. When new requests arrive, the NFVO interacts with the data repository, and computes the status of the NFV infrastructure to take placement

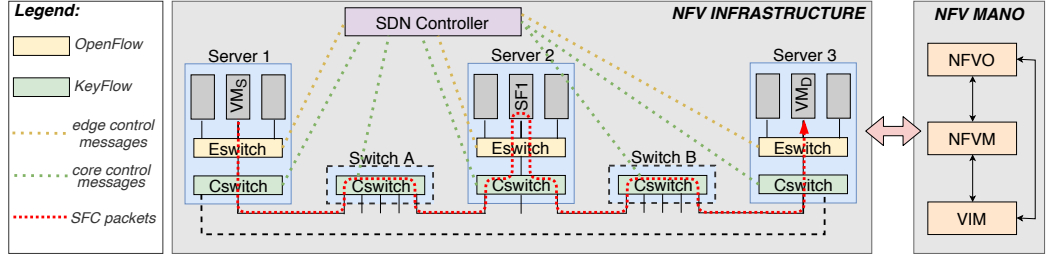


Figure 5.1: KeySFC architecture, and example for chain $VM_S \rightarrow SF1 \rightarrow VM_D$.

and SFC decisions. The placement decisions are sent to the VIM and to the VNFM, and the SFC decisions are sent to the SDN Controller. Then, the VIM interacts with hypervisors in the servers to create or migrate SFs, and to establish virtual networks between them. The VNFM manages the life cycle of SFs. The SDN controller is responsible for converting the SFC decisions into forwarding rules installed in the software switches of the servers. A KeySFC-enabled server is composed by four elements (Fig. 5.1):

1. VMs that act as sources and destinations of packets.
2. VMs that host SFs.
3. An edge switch (in yellow), called *Eswitch*: a software switch that serves as both ingress and egress element for traffic entering and leaving VM instances. It supports the OpenFlow protocol, and is managed by the SDN Controller, which installs forwarding rules to tag and steer SFC traffic.
4. A core switch (in green), named *Cswitch*, which connects to physical network interfaces, and handles communication between VMs that reside in different servers by using a RNS-based SSR mechanism inspired in KeyFlow [Martinello et al. 2014]. It can be implemented as a software or a hardware switch.

Fig. 5.1 also shows that KeySFC fits various DCN architectures designs. For architectures based on hardware switches (i.e., network-centric, and hybrid DCNs), these switches are implemented as core elements (e.g. Switches A and B). On the other hand, in server-centric architectures, servers are directly interconnected, and there is no need to have hardware switches as the server itself contains a *Cswitch* element for routing.

5.1.2 KeySFC underlay routing design

KeySFC replaces table lookup operations by a *modulo* operation using RNS-based SSR, as described in Section 2.2.1. The adopted scheme can be applied in any topology, and uses SSR to explicitly define all nodes in the SFP.

The routing in KeySFC relies on two identifiers: a node identifier, called *nodeID*, and a route identifier, called *routeID*. The set of *nodeIDs* is composed by pairwise co-prime

numbers, which are assigned by the SDN Controller to each *Cswitch* of the topology in a network configuration phase. The *routeIDs* are tagged in the Ethernet header of packets, as a design choice, when they traverse *Eswitches* (more details in Section 5.1.3.1). The forwarding in each node is defined by the remainder of the division of the *routeID* of the packet by the *nodeID* of the *Cswitch*, which gives the output port. Thus, forwarding nodes perform a simple operation over L2 headers to find the output port.

The implementation of *Cswitches* can be done by modifying the datapath of devices based on Open vSwitch (OvS), by using NetFPGAs [Martinello et al. 2017], or by exploring devices that support the P4 language. Besides, KeySFC allows performance optimization in the *Cswitches* of servers. Although the basic approach uses software switches (as implemented in the prototype of Section 5.2), it can be extended by offloading forwarding tasks to specialized devices (e.g., a networking co-processor SmartNIC).

5.1.3 How does KeySFC work?

5.1.3.1 KeySFC protocol

Since SFC is transparent to source and destination, the network needs mechanisms to classify the traffic when it leaves the source, and steer the flow through the SFC. The main idea behind our proposal is that any traffic flow that traverses *Eswitches*, and is part of a chain, will be classified by matching pre-installed flow rules. Such rules are installed by the SDN Controller and rewrite Ethernet MAC addresses in order to give a new meaning for that set of bits, called virtual MAC (VMAC) in our approach.

The VMAC can use the 48 bits of the Destination MAC (DstMAC) address in the Ethernet frame, or combine the 96 bits of DstMAC and Source MAC (SrcMAC) addresses. The choice of whether to use one or two MAC addresses is coupled with the maximum bit length of the *routeID* (as explained in Section 5.1.2), and depends on design choices when deploying KeySFC, such as number of nodes, number of chains, and number of segments per chain.

Fig. 5.2 shows the format of a VMAC address. It is divided in two parts: the most significant 16 bits represent the *segID*, and the remaining bits (32, if using only DstMAC, or 80, if using DstMAC and SrcMAC) represent the *routeID*. The *segID* uniquely identifies the current chain segment, and, when the segment ends, it is used to match a flow entry that defines the next segment. The *routeID* is used by each hop in the forwarding path to define the next hop using RNS-based SSR.

It is important to highlight that the use of the MAC address field to encapsulate the SFC information was a *design choice*, and the same approach could be adapted to use other kinds of encapsulation, such as NSH or MPLS headers, or even a new header. The main advantages of using existing headers are the reduction of header overhead and the compatibility with legacy forwarders and SFs. On the other hand, some SFs may require



Figure 5.2: VMAC address format.

to read or modify the MAC addresses. This can be solved with the addition of a Proxy between the SFFs and SFs, as defined in RFC 7665 [Halpern and Pignataro 2015].

When receiving SFC decisions from the NFVO, the SDN Controller splits each chain in segments, and calculates the VMAC for each segment. Then, it installs flow entries in *Eswitches* to tag packets with the corresponding VMACs. Initially, the packet leaves the source VM, and reaches the *Eswitch* of the host server. There, it matches a flow entry that performs classification based on 5-tuple, and tags the first VMAC into the packet. If the next SF in the chain is located in the same server, the packet is directly sent to that SF. Otherwise, the packet is sent to the *Cswitch*. At each hop, the forwarding mechanism in the *Cswitches* performs a *modulo* operation over the packet's *routeID* to find the next hop.

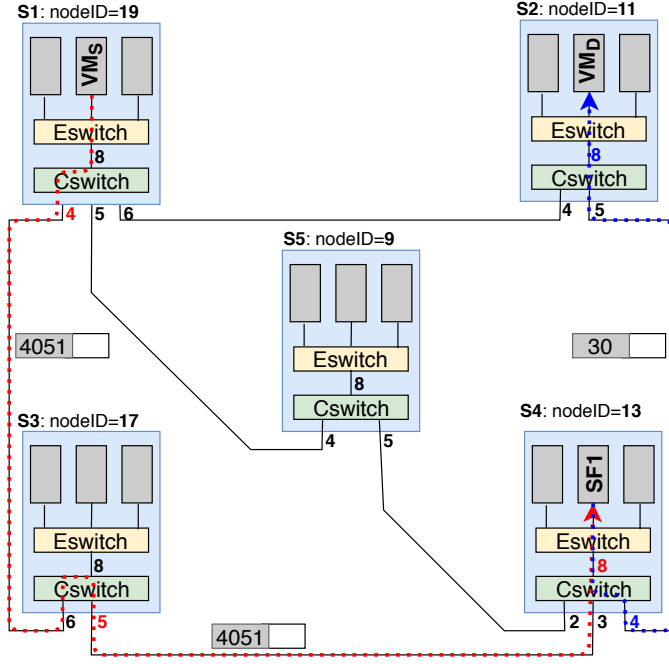
These stateless operations take place along the path until the packet reaches the server hosting the endpoint SF of that SFC segment, and is forwarded to the *Eswitch* of that server. There, the packet's VMAC matches a flow entry that redirects the packet to the SF. After processing, the SF sends the packet back to the *Eswitch*, where the current VMAC matches another flow entry that tags the VMAC of next segment into the packet.

These steps are repeated for all segments, until the packet arrives at the host of the destination VM. Finally, a flow entry at the *Eswitch* changes the VMAC field to the original MAC addresses, and the packet is forwarded to the destination VM. It is important to note that the packet that reaches destination is exactly equal to the packet that left source. Moreover, only the endpoints of each SFC segment have to maintain states, making our approach more scalable and agile than traditional table-based approaches.

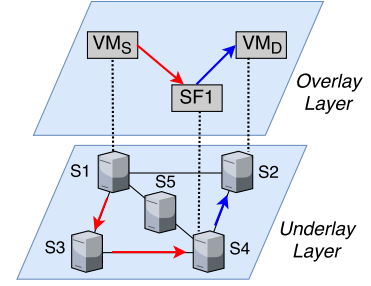
5.1.3.2 A day in the life of a SFC request

This section provides a step-by-step example of how KeySFC executes a SFC request. Consider the illustrative scenario shown in Fig. 5.3: a 5-node server-centric topology that provisions the SFC $VM_S \rightarrow SF1 \rightarrow VM_D$. We have two segments: $VM_S \rightarrow SF1$, and $SF1 \rightarrow VM_D$. The first segment is mapped to two hops in the underlay ($S1 \rightarrow S3$, and $S3 \rightarrow S4$), and the second segment is mapped to a single hop ($S4 \rightarrow S2$).

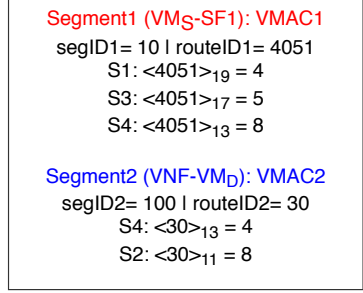
Initially, the controller assigns the *nodeIDs* to servers (9, 11, 13, 17, 19), splits the chain in two segments, and calculates the identifiers: VMAC1 ($routeID1 = 4051$, $segID1 = 10$) for segment $VM_S \rightarrow SF1$, and VMAC2 ($routeID2 = 30$, $segID2 = 100$) for segment $SF1 \rightarrow VM_D$. Then, it installs flow entries at *Eswitches*, as shown in Table 5.1. When VM_S sends traffic to VM_D the following events happen:



(5.3.a) Detailed scenario.



(5.3.b) Overlay and underlay.



(5.3.c) Forwarding operations.

Figure 5.3: KeySFC example (VM_S → SF1 → VM_D).Table 5.1: Simplified flow tables at *Eswitches*.

#	Server	Match	Action
1	S1	SrcIP=IP VM _S ; L4DstPort=P1	DstMAC=VMAC1; Out to <i>Cswitch</i>
2	S4	DstMAC=VMAC1	Out to SF1
3	S4	DstMAC=VMAC1; InPort=SF1 port	DstMAC=VMAC2; Out to <i>Cswitch</i>
4	S2	DstMAC=VMAC2	DstMAC=MAC VM _D ; Out to VM _D

- At S1:** A packet leaves VM_S and reaches the *Eswitch*, where the flow matches **rule 1**, causing the DstMAC to be changed to VMAC1 and the packet to be forwarded to the *Cswitch*. In the *Cswitch*, the *modulo* operation of *routeID1* by the *nodeID* of S1 gives $\langle 4051 \rangle_{19} = 4$. Thus, the output port is 4, and the packet is forwarded to host S3.
- At S3:** When the packet reaches the *Cswitch*, the *modulo* operation of *routeID1* by the *nodeID* of S3 gives $\langle 4051 \rangle_{17} = 5$. Therefore, the packet is forwarded to port 5, which is connected to S4. The packet does not go up to the *Eswitch*, because S3 only forwards traffic.
- At S4:** When the packet reaches the *Cswitch*, the *modulo* operation of *routeID1* by the *nodeID* of S4 gives $\langle 4051 \rangle_{13} = 8$. Thus, it forwards the packet to the *Eswitch*, where it matches **rule 2**, causing the packet to be sent to SF1. SF1 processes the packet and forwards it back to the same interface. The returning packet reaches the *Eswitch* and matches **rule 3**, causing the DstMAC to be changed to VMAC2, with

the packet being sent to the *Cswitch*. There, computing the *modulo* of *routeID2* by the *nodeID* of S4 gives $< 30 >_{13} = 4$. Therefore, the packet is sent to S2 via port 4.

4. **At S2:** When the packet reaches the *Cswitch*, the *modulo* operation of the *routeID2* by the *nodeID* of S2 gives $< 30 >_{11} = 8$. Thus, the packet is sent to the *Eswitch*, where the flow matches **rule 4**, causing the DstMAC to be changed to the MAC VM_D , and the packet to be delivered to VM_D .

5.1.4 KeySFC control plane

The control plane has five fundamental roles in KeySFC: (i) topology discovery and monitoring; (ii) configuration of *nodeIDs* for *Cswitches*; (iii) management of chain segments; (iv) installation of flow rules in *Eswitches* according to SFC decisions; and (v) ARP proxy.

To perform these tasks, the SDN Controller has to have full knowledge of the network topology, the SFC requests, the SFs that serve these requests, and the specific virtual ports where each SF is attached in each physical server. The information about the SFCs and the SFs is obtained via communication with the NFVO or queries to the data repository. The discovery and monitoring of the network topology uses the LLDP protocol.

Based on its centralized view of the infrastructure, the SDN Controller is responsible for a network configuration phase, where it assigns a set of co-prime numbers to the *nodeIDs* of servers and hardware switches. This is accomplished by communicating with *Cswitches* via OpenFlow protocol.

Afterwards, the Controller calculates two identifiers for all the segments of all active chains: the *routeID* and the *segID*. Then, the Controller installs forwarding rules in the *Eswitches* using the OpenFlow protocol. These rules are responsible to classify the flows and guarantee that, when entering in each chain segment, the flow is tagged with its respective identifiers. The correlation between each chain and its respective segment identifiers is persisted in the shared data repository.

Finally, the SDN Controller also acts as an ARP Proxy in order to reduce the impact of broadcast. When a VM_S at Server 1 wants to send packets to a VM_D hosted in other server, VM_S sends an ARP request broadcast message with the IP of VM_D . This message is intercepted by the *Eswitch* of Server 1 and forwarded to the SDN controller that resolves the address and sends an ARP reply message to the *Eswitch*, which, in turn, delivers the message to VM_S . Then, VM_S can send data to VM_D using the discovered MAC.

One advantage of KeySFC is the significant reduction of control plane signaling. Considering routing, the SDN Controller only answers ARP queries, since *Cswitches* execute forwarding in a decentralized way without control plane communication. Other steps, such as topology management, configuration of *nodeIDs*, and installation of flow entries can happen in a prior network configuration phase in a proactive manner. So, considering SFC, only update or creation of chains generate control plane communication.

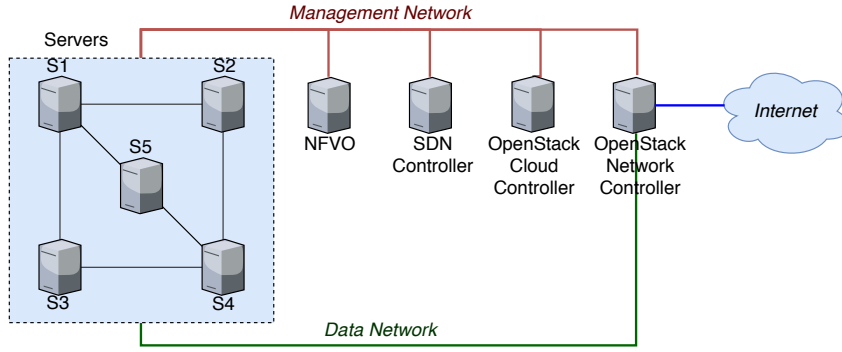


Figure 5.4: KeySFC prototype testbed.

5.2 Proof-of-concept and evaluation

This section evaluates a proof-of-concept implementation of the KeySFC scheme, which is described in Section 5.2.1. Firstly, we present reference SFC scenarios that can capture some trade-offs for SF placement and chaining in Section 5.2.2. Then, in Section 5.2.3, we execute functional tests to check the correct behavior of our traffic steering scheme. Afterwards, Section 5.2.4 evaluates the performance of KeySFC by measuring latency and jitter on an end-to-end service. Finally, in Section 5.2.5, we test some traffic engineering scenarios that explore steering functionalities offered by KeySFC.

5.2.1 Prototype

We implemented a proof-of-concept prototype of the KeySFC scheme for the 5-node server-centric topology of Fig. 5.4. It implements all the KeySFC architectural components, as presented in Fig. 5.1: servers, NFVO, NFVM, VIM, and SDN Controller.

The OpenStack platform was selected as our VIM and VNFM because its server networking architecture is organized in two layers, implemented as OvS bridges, that could be adapted to our scheme. To this end, the KeySFC RNS forwarding algorithm was implemented in the OvS datapath module of *Cswitches*. The OpenStack deployment includes a Cloud and a Network Controller [Openstack 2017]. The SDN Controller was implemented using the Ryu framework. The NFVO is a Python application that has a CLI interface to receive SFC requests, and integrates with the OpenStack Controllers and the SDN Controller.

The testbed is composed by 9 physical servers running Linux Ubuntu, as shown in Fig. 5.4: 5 server nodes (S1-S5), one NFVO, one SDN Controller, one OpenStack Cloud Controller, and one OpenStack Network Controller. Each server node has one Intel Xeon E5-2620 2.4 GHz processor, 16GB of memory, and four or five 1Gbps Ethernet NICs: one is connected to the OpenStack data network, one is connected to the OpenStack management network, and the remaining are connected to other servers to build the server-centric topology of Fig. 5.4.

The basic behavior of a SF is to receive packets, to perform some processing that may or not modify packets, and to send packets to the next hop. The time a SF spends processing packets and the nature of the modification it performs in the packets is deeply related to the specific application. In our proof-of-concept, to isolate the effects of the SFC mechanism itself from the performance limitations of specific SFs, all the SFs in our test scenarios implement a *forwarding function*. In this way, we can state that the performance evaluation conducted in this paper is influenced only by the KeySFC scheme and not caused by the behavior of one or more specific SF. To implement this forward function, we create an OvS bridge inside the VM of the SF and install a single flow entry to redirect the traffic back to *Eswitch* when it reaches the SF.

5.2.2 Reference SFC scenarios

One SFC segment is the path between two VMs, which can be of three types: source, SF, or destination. When installing SFC rules, it is necessary to verify the following cases for each segment:

1. Source and destination endpoints of that segment are SFs;
2. Source endpoint of that segment is also the source of the chain; or
3. Destination endpoint of that segment is also the destination of the chain.

For each case, it is necessary to verify the following subcases: (a) VMs are allocated in the same server, and (b) VMs are allocated in different servers.

When the two endpoints of a segment are in different servers, a *routeID* must be generated for routing in the physical network. Also, a flow entry in the *Eswitch* of the source endpoint should include this identifier in the packet header, and send it to the *Cswitch*. If the endpoints are in the same server, the flow remains in the *Eswitch*, but is redirected to the input port of the VM that is the destination of that segment. Moreover, for Case 3, where the VM is also the destination of the chain it is necessary to replace the VMAC by the real MAC of the destination to terminate the chain.

To cover different combinations between these cases and subcases, we designed four reference scenarios presented in Fig. 5.5. Scenario 1 (SC1) and Scenario 2 (SC2) have one single SF, but the later covers the case where the SF is placed in the same server of destination. Besides, SC1 involves segments with 1 and 2 hops in the physical network, while SC2 involves segments with 0 and 3 hops in the physical network. On the other hand, both Scenario 3 (SC3) and Scenario 4 (SC4) have two SFs, but, in SC4, both SFs are placed in the same server. In addition, SC3 only involves segments with 1 hop in the physical network, while SC4 involves segments with 0, 1, and 2 hops in the physical network, diversifying the routing combinations to test KeySFC. The case where the source

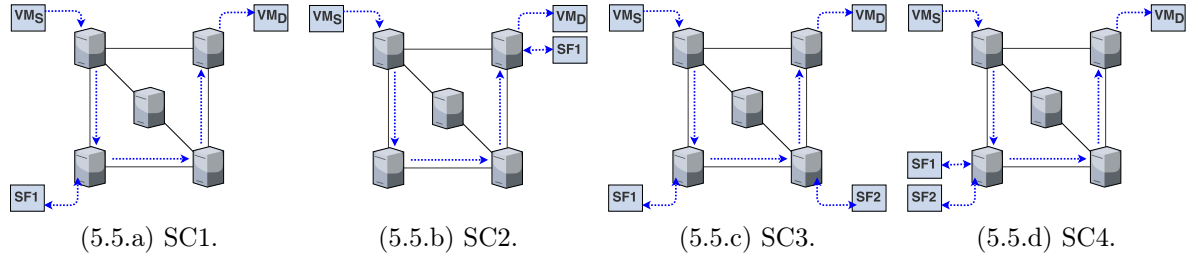


Figure 5.5: Reference scenarios exploring different placement strategies: (a) in **SC1**, 1 SF and all VMs in different servers; (b) in **SC2**, 1 SF allocated with VM_D ; (c) in **SC3**, 2 SFs and all VMs in different servers; and (d) in **SC4**, 2 SFs allocated in the same server.

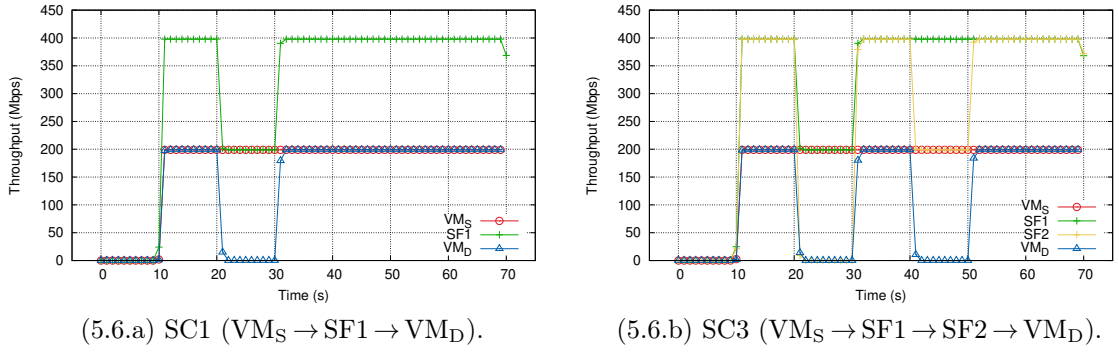


Figure 5.6: Throughput results for functional tests: (a) in **SC1**, SF1 is off from 20s to 30s; and (b) in **SC3**, SF1 is off from 20s to 30s and SF2 is off from 40s to 50s.

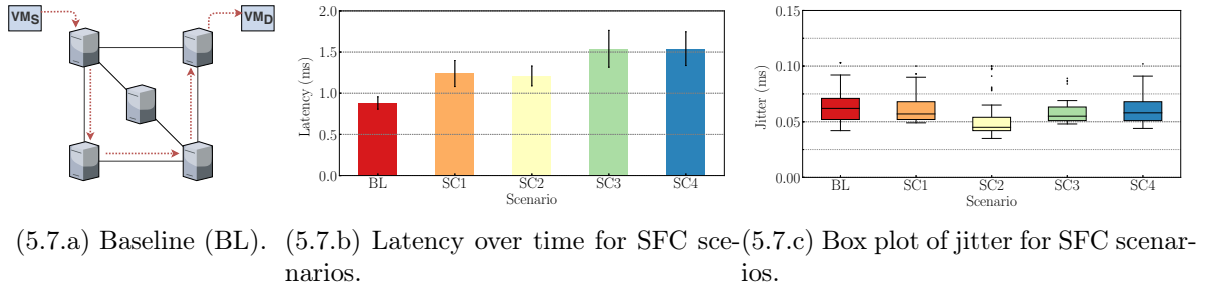


Figure 5.7: Jitter and latency results comparing reference scenarios with baseline (BL).

VM is allocated in the same server as a SF was also tested, but results were omitted because they were similar to the case where the destination is allocated with a SF.

We compare the results of the reference scenarios with a baseline scenario (BL) presented in Fig. 5.7.a, when traffic goes from VM_S to VM_D without going through any SF. To keep consistency, the end-to-end path in the underlay is always the same for all the reference scenarios, only the SFs used in the overlay change.

5.2.3 Functional test

This first test is intended to verify the correct operation of KeySFC in our reference scenarios (Fig. 5.5). We want to show that all traffic that leaves the source is intercepted

and steered through the specified SFs in the right order, and arrives at destination. To this end, we turn off one SF of the chain at a time, and check that nodes positioned after such SF stop to receive traffic while the SF is not forwarding.

Fig. 5.6 shows the functional tests for scenarios SC1 and SC3, as specified in Fig. 5.5. Results for SC2 and SC4 were omitted, because they follow the same behavior as SC1 and SC3, respectively. In Fig. 5.6, at instant 10s, VM_S starts to send a 200Mbps UDP traffic using the `iperf` tool to VM_D, and this traffic must be steered through one or two SFs, depending on the scenario. From 10 to 20s, all the SFs are forwarding traffic. SFs perceive a total bandwidth that is twice the bandwidth sent by VM_S (200Mbps), because they receive traffic and send it back to the same network interface. VM_D receives 200Mbps of traffic during this interval. From 20s to 30s, the forwarding function at SF1 is turned off, so traffic drops from 400Mbps to 200Mbps at SF1, because traffic enters in network interface, reaches SF1, but does not return. Besides, traffic drops to zero in VM_D and SF2 (for scenarios with two SFs), because it was interrupted at SF1. At instant 30s, SF1 is turned on again, and traffic returns to initial conditions, proving that it was indeed passing through SF1.

In scenarios with two SFs, we repeat the same procedure for SF2, turning it off during a 10s interval, and we see that VM_D stops receiving traffic, but nodes in a previous position in the chain (i.e., VM_S and SF1) continue to receive traffic normally. These tests show that, for all scenarios, packets were indeed steered through each SF in the correct order, and that VM_D correctly receives traffic sent by VM_S after the SFC.

5.2.4 Performance tests

5.2.4.1 Latency and jitter on reference scenarios

In this first set of performance tests, we measure end-to-end latency and jitter in the reference scenarios (Fig. 5.5). Each test was run for 60s, graphs present the mean of the values in the interval, and bars represent standard deviation.

The first test (Fig. 5.7.b) aims to understand SFC impact on latency, and uses `ping` to send an ICMP message from VM_S to VM_D. It needs SFC rules in both directions to guarantee that both ICMP request and reply messages pass through the same SFs, but in reverse order. The results show that latency increases from approximately 0.8ms in the baseline scenario to an average of 1.2ms in scenarios with one SF, and to an average of 1.5ms in scenarios with two SFs. This impact in latency is expected, because SFC scenarios add extra hops when passing through the SFs.

The second test, shown in Fig. 5.7.c, compares jitter in the reference scenarios to the baseline. VM_S sends a 200Mbps UDP traffic using `iperf` to VM_D, and SFs forward traffic without interruption. Our first observation is that there are small differences in the median: 0.05ms for the baseline against approximately up to 0.06ms in all SFC scenarios.

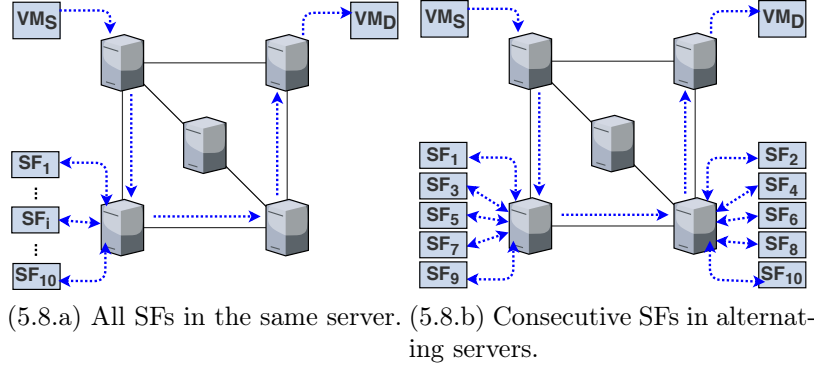


Figure 5.8: Scenarios for evaluating the impact of SFC length.

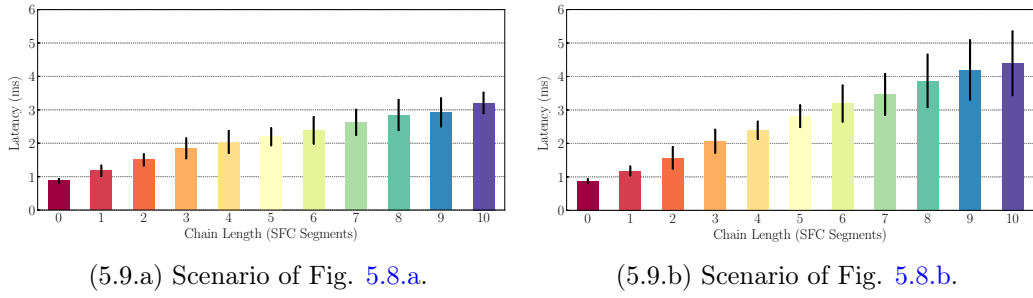


Figure 5.9: Latency results when chain length varies from 1 to 10.

Even when one or two hops are added in the chain, there is no considerable degradation in jitter, keeping it quite small. Also, the box plot shows that the distributions are well behaved, varying from 0.04 to 0.07ms for the 75 percentis, with jitter reaching 0.1ms only in the worst case of SC1. Thus, these results indicate that the KeySFC scheme has no strong impact on jitter, which remains very small independently of the number of SFs for the reference scenarios. Also, the distributions can be considered statistically equivalent for all scenarios, which can be later explored by jitter sensitive applications.

5.2.4.2 The impact of chain length in latency

This test investigates how the chain length impacts end-to-end latency in KeySFC. To this end, we analyze latency growth by increasing chain length from 1 ($VM_S \rightarrow SF_1 \rightarrow VM_D$) to 10 ($VM_S \rightarrow SF_1 \rightarrow SF_2 \rightarrow SF_3 \rightarrow SF_4 \rightarrow SF_5 \rightarrow SF_6 \rightarrow SF_7 \rightarrow SF_8 \rightarrow SF_9 \rightarrow SF_{10} \rightarrow VM_D$). In each of these steps, we add one extra SF between VM_S and VM_D .

We analyze two placement scenarios: when all SFs are allocated in the same server (Fig. 5.8.a), and when consecutive SFs are allocated in alternating servers (Fig. 5.8.b). In the first scenario, the hops between SFs are executed in the *Eswitch* of the same server, so packets do not need to be routed in the physical network. On the other hand, in the second scenario, each segment in the overlay is mapped to one hop in the physical layer, passing through the *Eswitches* and *Cswitches* of two servers. The second scenario is also relevant to prove that KeySFC is able to represent loops in the physical network,

considering the end-to-end chain.

Fig. 5.9 shows average latency values measured using ICMP in different chain lengths for both scenarios. The bars represent the standard deviation of our measurements, and each test was run 60 times. Both charts show that latency grows in a close-to-linear fashion for up to 10 SFs in the chain. As the second scenario (Fig. 5.8.b) involves routing in the physical network, we note an overall increase in the latency measurement. Nevertheless, this growth goes only from $3.2ms$ in the first scenario to $4.3ms$ in the second scenario, even in the worst case when the chain length is 10, showing the routing in the physical network is efficient.

5.2.5 Traffic engineering enabled by KeySFC

One of the main contributions of KeySFC is to provide the appropriate SFC mechanisms to the NFVO, so traffic engineering can select any path and take optimal placement and chaining decisions. This section explores some traffic engineering scenarios that can be enabled by KeySFC.

5.2.5.1 Allocation of maximum bandwidth with path migration

Consider the NFVO has to embed the following SFC request in our test topology: $VM_S \rightarrow SF1 \rightarrow VM_D$. The goal is to dynamically provide the maximum bandwidth to a TCP flow in a SFC.

Fig. 5.10.a shows VM_S and VM_D are allocated in servers S1 and S2, respectively. Besides, there are two different paths with length 2 that connect VM_S to SF1. Initially, these paths present the following background traffic:

- Path 1: $S1 \rightarrow S5 \rightarrow S4$: with $400Mbps$ UDP traffic.
- Path 2: $S1 \rightarrow S3 \rightarrow S4$: with $800Mbps$ UDP traffic.

At the beginning of this experiment, the SFC request is embedded in Path 1, as shown in Fig. 5.10.a, because it is the best choice. We send a TCP flow using `iperf` that will use all the available bandwidth in the SFC. Fig. 5.10.c shows that the throughput at VM_D starts at approximately $560Mbps$.

As NFV demands are very dynamic in a DCN, at any moment, the NFVO may discover there is an alternative that may improve the bandwidth utilization for the TCP flow under evaluation. This happens at instant 60s, when Path 2 becomes idle and the NFVO migrates this TCP flow from Path 1 to Path 2, causing the throughput at VM_D to increase to $980Mbps$, as shown in Figs. 5.10.b and 5.10.c.

To perform the described path migration, the NFVO communicates with the SDN Controller, which only have to execute a very simple `flow mod` operation to modify a

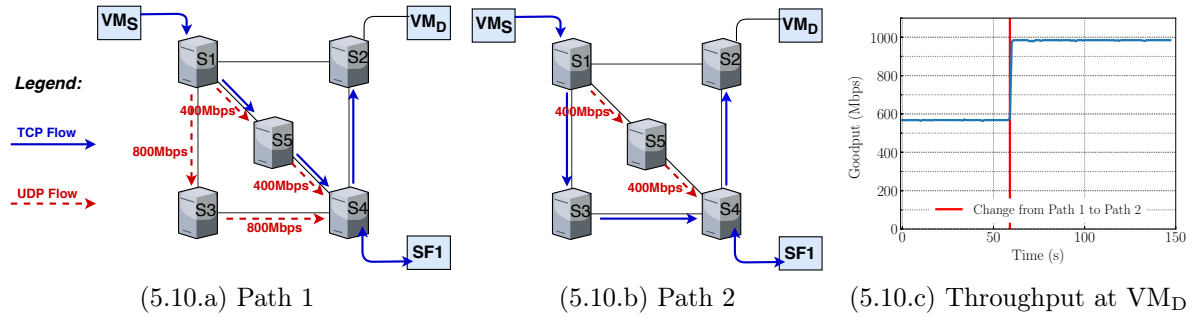


Figure 5.10: Traffic engineering for migration from Path 1 to Path 2: (a) Path 1 with concurrent 400Mbps UDP traffic, (b) Path 2 with no concurrent traffic, and (c) Throughput results at VM_D during migration test.

Table 5.2: Flow entries at S1 for paths 1 and 2.

Path	Server	Match	Action
1	S1	SrcIP=IP VM _S ; L4DstPort=P1 DstMAC=VMAC1; Output to Cswitch	
2	S1	SrcIP=IP VM _S ; L4DstPort=P1 DstMAC=VMAC2; Output to Cswitch	

single flow entry at the *Eswitch* of S1. Table 5.2 shows the original flow entry that embeds VMAC1 into the packet, which contains the *routeID* for Path 1. For selecting Path 2, the only field that has to change is the action “DstMAC=VMAC2”, shown in the second line of Table 5.2, as VMAC2 embeds the new route through Path 2. Even for longer paths, no other changes would be required in the hops along the path, thanks to strict SR. Once this single flow mod operation is executed, all the packets that leave VM_S will be tagged with the *routeID* of the new path.

Comparing KeySFC with traditional SFC schemes based on SDN approaches, the path migration would involve changing flow entries in all the hops along the old and in the new paths. Depending on the path length, this can represent a long delay to update a large number of flow entries in switches that might be distributed, leading to consistency problems and packet loss in a TCP flow. This time to reconfigure a path becomes even more significant for chains with a large number of segments.

5.2.5.2 Agile path selection with replication of SFs

Apart from seamless path migration, the KeySFC mechanism allows to use replication of distributed SFs instances that might be selected on demand. In this way, the NFVO component can offer an extra degree of freedom by allowing the migration to a path that uses a replicated SF instance. The goal is still to dynamically achieve the maximum available bandwidth for a TCP flow that crosses VM_S → SF1 → VM_D.

Fig. 5.11.a shows VM_S and VM_D are allocated in servers S1 and S4, respectively. Besides, there are three instances of the SF of type SF1 allocated in servers S2, S3, and S5. Also, there are three different paths with length 3 that connect VM_S to VM_D passing

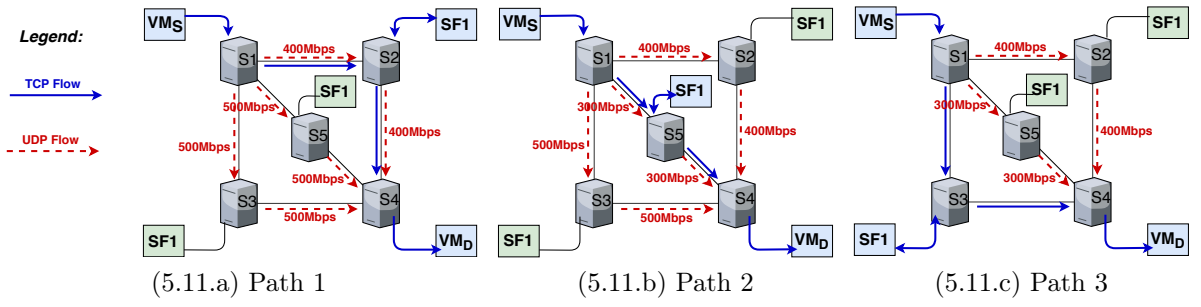


Figure 5.11: Traffic engineering scenarios for different paths using SF redundancy: (a) Path 1 with 600Mbps UDP traffic, (b) Path 2 with 400Mbps UDP traffic, and (c) Path 3 with no concurrent traffic.

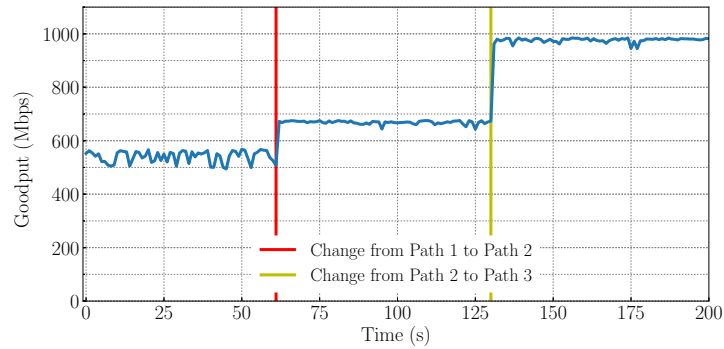


Figure 5.12: Throughput results when SFC migrates from Path 1 to Path 2, and from Path 2 to Path 3 (see Figure 5.11).

through one instance of SF1. In the initial condition, these paths present the following background traffic:

- Path 1: $S1 \rightarrow S2 \rightarrow S4$: with 400Mbps UDP traffic.
- Path 2: $S1 \rightarrow S5 \rightarrow S4$: with 500Mbps UDP traffic.
- Path 3: $S1 \rightarrow S3 \rightarrow S4$: with 500Mbps UDP traffic.

Initially, the SFC request is embedded in Path 1, as shown in Fig. 5.11.a, and, as in the previous experiment, we send a TCP flow using `iperf` to use all the available bandwidth. Fig. 5.12 shows the flow is able to achieve approximately 530Mbps at VM_D .

At instant 60s, the NFVO detects the utilization of Path 2 decreased to 300Mbps, and decides to migrate the SFC from Path 1 to Path 2 using an extra SF1 instance that was already provisioned, as shown in Fig. 5.11.b. In a similar way, at instant 130s, Path 3 becomes idle, and the SFC migrates from Path 2 to Path 3, as shown in Fig. 5.11.c. Fig. 5.12 presents throughput results when NFVO migrates the SFC from Path 1 to Path 2, and from Path 2 to Path 3. In this test, we can see that the first migration was able to increase the bandwidth from about 530Mbps to about 660Mbps, and the second migration was able to increase the bandwidth to about 980Mbps.

The cost of performing these operations in terms of flow entry modifications is the same as the previous example: a *flow mod* operation. We highlight here one of the main

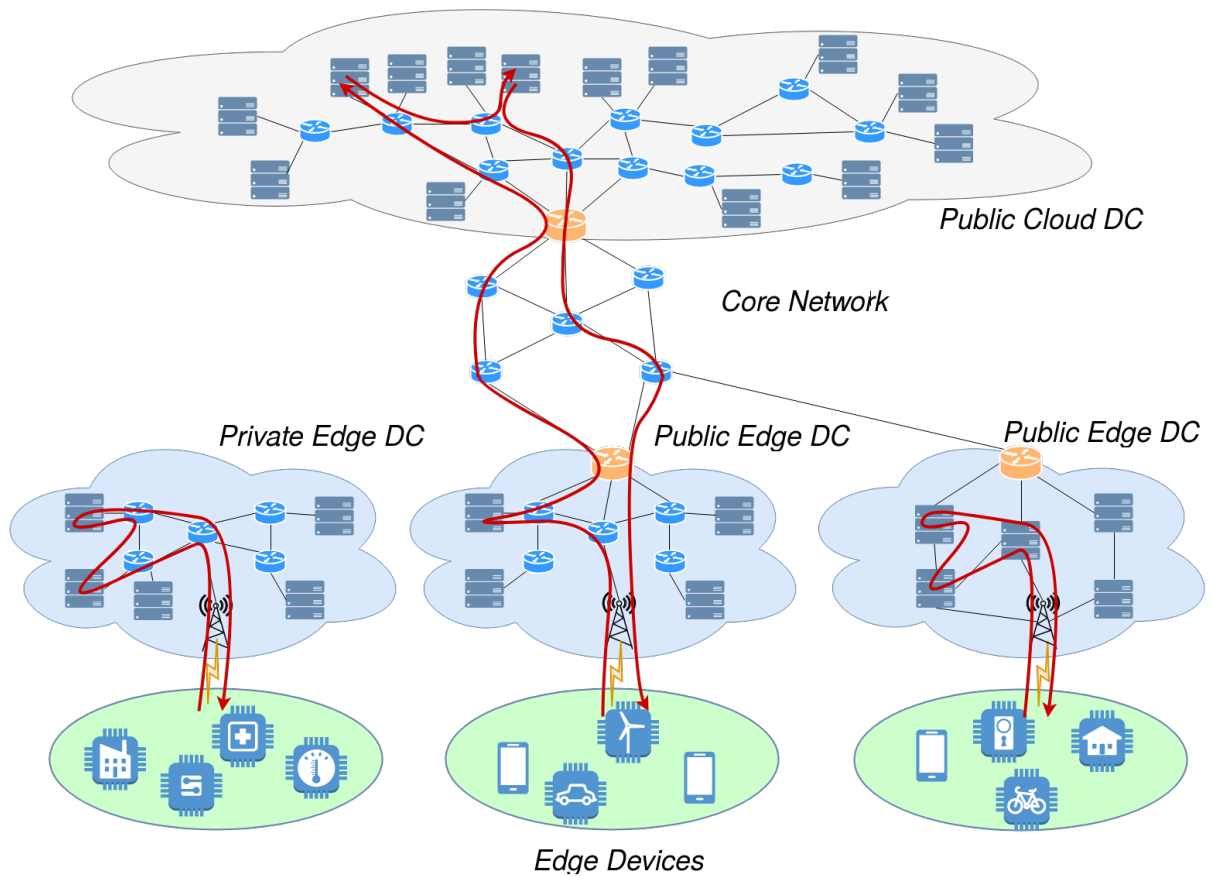


Figure 5.13: Example of Multi-domain SFC.

contributions of our KeySFC scheme: NFVO takes placement, routing, and chaining decisions in an integrated way. This integration allows traffic engineering tasks to make optimal use of networking and processing resources. In most of current SFC approaches, routing is a separate responsibility of the data plane, so SFC decisions are not integrated to routing decisions. Also, data plane routing frequently restricts the NFVO to choose one from a set of shortest paths, and, consequently, prevents traffic engineering to find an optimal path for a SFC.

5.3 Multi-domain SFC

The proposals of this thesis focus on a single SFC domain. However, KeySFC scheme could be extended to solve the SFC problem when SFs are distributed in multiple DCs and administrative domains. Fig. 5.13 shows examples of SFC in single and multiple domains.

The solution to the multi-domain problem does not require major adaptations in the KeySFC scheme. Each domain can manage its own keys, and the switches/routers that interconnect the domains (represented in orange in the figure) can act as SFC gateways. When a flow crosses a domain, the SFC gateways have to match a flow entry containing

the *segID* of the outgoing flow, and rewrite the VMAC according to the new *segID* and *routeID* of the particular domain the flow is going to enter. A global orchestrator have to manage these identifiers in SFC gateways across the different domains, and synchronize this information with local orchestrators.

5.4 Concluding remarks

This chapter proposed, implemented, and evaluated KeySFC: a SFC scheme built up over algorithmic SSR designed as a fabric network, separating edge and core switching elements. With our proof-of-concept prototype orchestrated by OpenStack we demonstrated that KeySFC is a feasible solution to the dynamic SFC problem in production DCs. While current solutions are tailored to network centric DCNs, we showed that it is possible to deploy SFC in any DCN topology, extending the infrastructures that can be used to provide SFC.

In addition, performance results showed that the selected SSR mechanism based on RNS can provide efficient packet transport with low latency and low jitter. We showed also that our fabric SFC scheme is efficient in steering the SFC traffic by evaluating different SFC scenarios. Finally, our PoC indicates KeySFC can provide programmable, expressive, agile, and scalable mechanisms for the NFV orchestration to select and modify paths for SFC according to the traffic engineering needs.

The dataplane implementation of KeySFC needs to be able to perform the *modulo* operation over the *routeID* bits. However, the *modulo* and division operations do not map to the instruction sets of most of the available network hardware. Although it is possible to modify the datapath of devices based on OvS, as done in the prototype of KeySFC, or develop a new implementation using NetFPGAs [Martinello et al. 2017], both alternatives require a steep learning curve, and deep understand on how their internal structures work. In addition, the former presents limited performance, and the later needs specialized hardware that presents high cost per unit performance when compared to traditional ASIC approach.

To tackle these issues, we present in the next chapter a new source routing mechanism, named PolKA, to replace KeyFlow in the KeySFC scheme. This new mechanism is also based on RNS, but it is implementable over COTS network hardware.

Chapter 6

PolKA: SSR Mechanism

PolKA (Polynomial Key-based Architecture) is a RNS-based SSR mechanism that applies the Chinese Remainder Theorem (CRT) to finite fields of two elements [Schroeder 2009]. This shift from integer to polynomial binary arithmetic can enable performance optimization and reuse of off-the-shelf network hardware, such as CRC, while portending new network services. Furthermore, we investigate if PolKA can be deployed in commercial programmable switches using P4 architecture and implemented with similar performance to a Port Switching SSR approach. In this way, we can enable the exploitation of RNS-based SSR in commercial hardware equipment.

This chapter is structured as follows. Section 6.1 proposes PolKA SSR mechanism, demonstrates how it enables unicast and multicast SSR, and analyzes the scalability of the *routeID*. Section 6.2 presents the design of PolKA and Port Switching according to P4 Architecture. Section 6.3 describes and evaluates emulated and hardware prototypes of PolKA and Port Switching. Finally, Section 6.4 discusses conclusions.

6.1 Proposal

This section proposes PolKA for unicast and multicast SSR, providing examples on how to use the mechanism, and a scalability analysis.

6.1.1 Mathematical background

This section describes the mathematical foundation of PolKA mechanism. More information can be found in [Herstein 1975, Shoup 2009].

Galois Field (GF): Let $GF(2) = \{0,1\}$ be the GF of order 2. Bear in mind that the elements of $GF(2)$ are residue classes modulo 2, and $a = b$ in $GF(2)$ means that $a \equiv b \pmod{2}$, i.e., $a - b$ is a multiple of 2. The arithmetic operations of addition and multiplication in $GF(2)$ are defined modulo 2.

Polynomial Ring over GF(2): The set of all polynomials in one variable t with coefficients defined over GF(2) is a ring considering the arithmetic operations of addition and multiplication modulo 2. If $f(t) = a_n t^n + a_{n-1} t^{n-1} + \dots + a_1 t + a_0$ is a polynomial over GF(2), where $a_n \neq 0$, n is defined as the degree of $f(t)$, denoted by $\deg(f)$. The length of $f(t)$, denoted by $\text{len}(f)$, is defined by $\text{len}(f) = \deg(f) + 1$.

Euclidean Division Theorem for Polynomials: Let $f(t)$ and $g(t)$ be polynomials in one variable t over GF(2), where $g(t) \neq 0$. Then there exist unique polynomials $q(t)$ and $r(t)$ over GF(2) such that $f(t) = g(t) \cdot q(t) + r(t)$, where either $r(t) = 0$ or $\deg(r) < \deg(g)$. The polynomial $r(t)$ is called the remainder of the division of $f(t)$ by $g(t)$, and will be denoted by $\langle f(t) \rangle_{g(t)}$.

Polynomial congruence: Given $f(t)$, $g(t)$, and $h(t)$ polynomials over GF(2), we write $f(t) \equiv h(t) \pmod{g(t)}$, and say that $f(t)$ is congruent to $h(t)$ modulo $g(t)$, if $f(t) = a(t) \cdot g(t) + h(t)$ holds for some polynomial $a(t)$ over GF(2).

Irreducible Polynomials: A non-zero polynomial $g(t)$, is called a divisor of $f(t)$ over GF(2) if $f(t) = a(t) \cdot g(t)$, for some polynomial $a(t)$ over GF(2). Two polynomials $f(t)$ and $g(t)$ over GF(2) are coprime if their only common divisor is 1. A non-constant polynomial $f(t)$ over GF(2) is called irreducible over GF(2) if its only divisors are possibly a constant polynomial and itself.

Chinese Remainder Theorem (CRT) for polynomials: Let $s_1(t), s_2(t), \dots, s_N(t)$ be monic pairwise coprime polynomials over GF(2) and let $M(t) = \prod_{i=1}^N s_i(t)$. There exists a unique polynomial $R(t)$ over GF(2) with $\deg(R) < \deg(M)$, satisfying $R(t) \equiv o_i(t) \pmod{s_i(t)}$, for $i = 1, 2, \dots, N$, where:

$$R(t) = \langle \tilde{R}(t) \rangle_{M(t)} \quad (6.1)$$

$$\tilde{R}(t) = \sum_{i=1}^N o_i(t) \cdot m_i(t) \cdot n_i(t) \quad (6.2)$$

$$m_i(t) = M(t) / s_i(t) \quad (6.3)$$

$$n_i(t) \cdot m_i(t) \equiv 1 \pmod{s_i(t)} \quad (6.4)$$

The computation of the polynomials $n_i(t)$ is based on the definition of polynomial congruence given above, and can be implemented using the Extended Euclidean Algorithm for polynomials, which basically consists of Euclidean Division Theorem applied several times.

6.1.2 Unicast source routing

PolKA proposes a source routing unicast mechanism that uses CRT for polynomials over GF(2). In this way, the RNS computation of the *routeID* relies on finite field arithmetic [Chang et al. 2015, Shoup 2009, Bajard 2007]. From now on, all the polynomials in this paper will be considered as polynomials over GF(2).

The forwarding in PolKA relies on three polynomial identifiers: a route identifier, called *routeID*, a node identifier, called *nodeID*, and a output port identifier, called *portID*. In each *core node*, the forwarding operation is defined by the remainder of the division of the *routeID* of the packet by the *nodeID* of the *core node*, which gives the *portID* of the output port.

In order to do this, note that a polynomial $f(t) = a_n t^n + a_{n-1} t^{n-1} + \dots + a_1 t^1 + a_0 t^0$ can be represented by the bit string $a_n a_{n-1} \dots a_1 a_0$. Thus, an identifier is represented by a bit string formed by the coefficients of a polynomial, which are either 0 or 1. Also, the bit length of the identifier is $\text{len}(f)$.

Let $S = \{s_1(t), s_2(t), \dots, s_N(t)\}$ be a multiset of the N polynomials to be assigned to the *nodeIDs* of the *core nodes* on the desired path by the *controller* of the topology, in a network configuration phase. The set S must be composed by pairwise co-prime polynomials. In this work, we assume that the elements of S are irreducible polynomials. Since the output ports are the remainder of the division of the *routeID* by the *nodeID*, we must have $\deg(s_i(t)) \geq \lceil \log_2(nports) \rceil$, where *nports* denotes the number of node ports, so we have enough polynomials to assign to all ports.

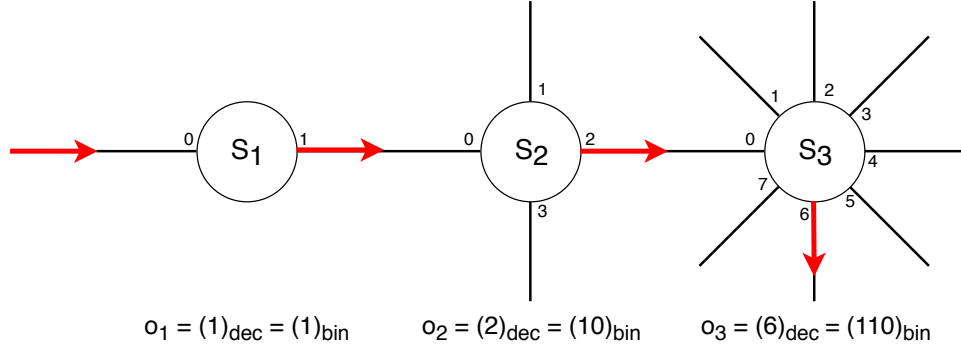
Besides, let $O = \{o_1(t), o_2(t), \dots, o_N(t)\}$ be a multiset of N polynomials satisfying the condition that $\deg(s_i) > \deg(o_i)$, where $o_i(t)$ is the polynomial assigned to the output port of the packet at the *core node* $s_i(t)$, for $i = 1, 2, \dots, N$. Note that the coefficients of the output port polynomial map the bits of the port labels. For instance, if the output port polynomial is $o_i(t) = 1 \cdot t^2 + 1 \cdot t$, it maps the port 110 and the packet is forwarded to port label 110.

To calculate the *routeID* for packets that traverse this path, we compute $R(t)$ that satisfies the following condition:

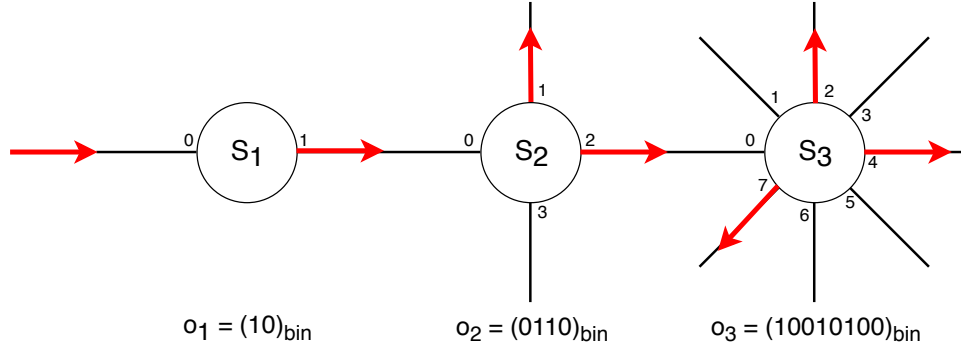
$$R(t) \equiv o_i(t) \pmod{s_i(t)}, \quad \text{for } i = 1, 2, \dots, N \quad (6.5)$$

For each node, the *modulo* of the *routeID* polynomial, $R(t)$, of a packet by its *nodeID* polynomial, $s_i(t)$, gives the *portID* polynomial, $o_i(t)$. Such $R(t)$ is calculated using the CRT (see Section 6.1.1) and the algorithm complexity is $\mathcal{O}(\text{len}(M)^2)$ [Shoup 2009]. Note that $R(t)$ is computed by the controller, while nodes only execute one simple *modulo* operation per packet.

Example 1: Fig. 6.1.a shows an example of how to apply PolKA to unicast source routing in a topology composed by nodes s_1 , s_2 , and s_3 , which have 2, 4, and 8 ports,



(6.1.a) **Unicast**: polynomial o_i directly represents label of the transmitting port.



(6.1.b) **Multicast**: polynomial o_i represents transmitting states of ports.

Figure 6.1: Example of source routing using PolKA.

respectively. The degrees of the polynomials assigned to s_1 , s_2 , and s_3 must be equal or greater than 1, 2, and 3, respectively, in order to encode 2 ports at s_1 , 2^2 ports at s_2 , and 2^3 ports at s_3 . Let the following irreducible polynomials be assigned to s_i nodes:

$$\begin{aligned}
 s_1(t) &= t + 1 = 11 \\
 s_2(t) &= t^2 + t + 1 = 111 \\
 s_3(t) &= t^3 + t + 1 = 1011
 \end{aligned}$$

Considering the path $(s_1 \rightarrow s_2 \rightarrow s_3)$, the output port set O is composed by the polynomials:

$$o_1(t) = 1, \quad o_2(t) = t = 10, \quad o_3(t) = t^2 + t = 110$$

The polynomial $R(t)$ has to satisfy the conditions of Eq. (6.5):

$$\begin{aligned}
 R(t) &\equiv 1 \pmod{(t + 1)} \\
 R(t) &\equiv t \pmod{(t^2 + t + 1)} \\
 R(t) &\equiv (t^2 + t) \pmod{(t^3 + t + 1)}
 \end{aligned}$$

Using the CRT for polynomials in Section 6.1.1, we calculate:

$$\begin{aligned} M(t) &= (t+1) \cdot (t^2+t+1) \cdot (t^3+t+1) \\ m_1(t) &= s_2(t) \cdot s_3(t) = (t^2+t+1) \cdot (t^3+t+1) \\ m_2(t) &= s_1(t) \cdot s_3(t) = (t+1) \cdot (t^3+t+1) \\ m_3(t) &= s_1(t) \cdot s_2(t) = (t+1) \cdot (t^2+t+1) \end{aligned}$$

From Eq. (6.4), there are polynomials $x_1(t)$, $x_2(t)$ and $x_3(t)$ that satisfy the following conditions:

$$\begin{aligned} x_1(t) \cdot s_1(t) + n_1(t) \cdot m_1(t) &= 1 \\ x_2(t) \cdot s_2(t) + n_2(t) \cdot m_2(t) &= 1 \\ x_3(t) \cdot s_3(t) + n_3(t) \cdot m_3(t) &= 1 \end{aligned}$$

Solving these equations, we find the polynomials $n_i(t)$:

$$n_1(t) = 1, \quad n_2(t) = 1, \quad n_3(t) = t^2 + 1$$

Finally, we can calculate $R(t)$ according to Eq. (6.1):

$$\begin{aligned} \tilde{R}(t) &= (t^2+t+1)(t^3+t+1) + t(t+1)(t^3+t+1) + \\ &\quad (t^2+t)(t+1)(t^2+t+1)(t^2+1) = t^7 + t^6 + t^5 + t^2 + 1 \end{aligned}$$

$$R(t) = \langle \tilde{R}(t) \rangle_{M(t)} = t^4 = 10000$$

Therefore, each packet that traverses the path defined in Fig. 6.1 should embed the *routeID* 10000 in its header, so each switch in the path can discover the *portID* by dividing this *routeID* by its own *nodeID*. For example, the remainder of the division of $R(t) = 10000$ by $s_3(t) = 1011$ is $o_3(t) = 110$.

6.1.3 Multicast source routing

Although multicast is not the focus of this work, the mechanism presented for unicast can be easily extended to multicast routing, where a single sender can send a copy of the packet to multiple receivers. To this end, the coefficients of the $o_i(t)$ polynomial now represent the transmitting state of the ports (i.e., 0 do not transmit, 1 transmit a packet copy), and we must have $\deg(s_i) \geq nports$. Thus, all the formulation of Section 6.1.2 remains the same, except the meaning of $o_i(t)$.

For instance, the polynomial $o_i(t) = a_2t^2 + a_1t + a_0$ maps the state $a_2a_1a_0$, which

means that there are 3 ports in the node s_i , and each coefficient represents the state of one port. If the output port polynomial is $o_i(t) = t^2 + t = 110$ then $a_2 = 1$, $a_1 = 1$, and $a_0 = 0$. This means that port2 is transmitting, port1 is transmitting, and port0 is not transmitting.

Example 2: Fig. 6.1.b shows an example of how to apply PolKA to multicast SSR in the same topology of example 1. Now, the degrees of the polynomials assigned to s_1 , s_2 , and s_3 must be equal or greater than 2, 4, and 8, respectively, to represent the necessary number of transmitting states at each node. Consider the following irreducible polynomials are assigned to the s_i nodes:

$$\begin{aligned} s_1(t) &= t^2 + t + 1 = 111 \\ s_2(t) &= t^4 + t + 1 = 10011 \\ s_3(t) &= t^8 + t^4 + t^3 + t + 1 = 100011011 \end{aligned}$$

Considering the path ($s_1 \rightarrow s_2 \rightarrow s_3$) and the transmitting states of Fig. 6.1.b, the output ports set O is composed by:

$$\begin{aligned} o_1(t) &= t = 10 \\ o_2(t) &= t^2 + t = 0110 \\ o_3(t) &= t^7 + t^4 + t^2 = 10010100 \end{aligned}$$

Following the same calculations of the example 1, it is possible to compute the *routeID* polynomial for multicast:

$$R(t) = t^{13} + t^{11} + t^9 + t^8 + t^5 + t^3 + t^2 = 10101100101100$$

6.1.4 Scalability of the bit length of the *routeID*

The bit length of $R(t)$, $len(R)$, for both unicast and multicast depends on the path length N and the degree of the polynomials assigned to the nodes in the path. Since $M(t) = \prod_{i=1}^N s_i(t)$, where $s_1(t), s_2(t), \dots, s_N(t)$ are the *nodeID* polynomials, $len(R)$ is given by:

$$len(R) = len(< \tilde{R}(t) >_{M(t)}) \leq \sum_{i=1}^N deg(s_i) \quad (6.6)$$

Algorithm 1 shows a pseudo-code for investigating the scalability of $len(R)$. For the sake of simplicity, we consider all nodes have the same number of ports. The function MAXLEN (*line 1*) computes the maximum $len(R)$, given: the number of ports in each node (*nports*), the number of nodes (*size*), and the topology diameter (*diameter*). More-

Algorithm 1 Computation of the maximum $\text{len}(R)$.

```

1: function MAXLEN( $nports, diameter, size, is\_multicast$ )
2:   if ( $is\_multicast$ ) then
3:      $mindeg \leftarrow nports$  ▷ Multicast
4:   else
5:      $mindeg \leftarrow \lceil \log_2(nports) \rceil$  ▷ Unicast
6:    $nodelist \leftarrow generate\_irred\_poly\_list(mindeg, size)$ 
7:    $pathlist \leftarrow get\_last\_itens(nodelist, diameter)$ 
8:    $length \leftarrow 0$ 
9:   for  $elem \in pathlist$  do
10:     $length \leftarrow length + deg(elem);$ 
11:   return  $length$  ▷ Maximum  $\text{len}(R)$ 

```

Table 6.1: Maximum $\text{len}(R)$ for example topologies.

Topology	$nports$	$diam.$	$size$	Bits for unicast	Bits for multicast
2-tier S=16 L=16*	24	3	32	21	72
Fat-tree 16 pods	16	5	320	55	80
ARPANET	4	7	20	42	44
GEANT2	8	7	30	49	56
Internet2	3	21	56	164	165

* Two-tier topology with 16 spine switches and 16 leaf switches.

over, it is necessary to specify if the computation is for multicast. Note that, for multicast, $\deg(s_i) \geq nports$ (line 3), while, for unicast, $\deg(s_i) \geq \lceil \log_2(nports) \rceil$ (line 5). So the representation of $R(t)$ for multicast may require more bits than for unicast.

A list of *nodeID* polynomials (*nodelist*) is generated (line 6), which consists of *size* irreducible polynomials with degree greater than or equal to the minimum degree (*mindeg*). Note that we select polynomials with the lowest possible degree (e.g., if $mindeg = 5$, we start assigning one of the 6 existing irreducible polynomials of degree 5 to nodes, and, if necessary, we use the 9 existing irreducible polynomials of degree 6, and so forth). Thus, *nodelist* is already ordered by degree. Finally, we select the very worst case scenario in which the polynomials in *nodelist* with the greatest degrees are assigned to the nodes in the longest possible path (i.e., the diameter). To this end, we pick the x last elements of *nodelist* (line 7), where $x = diameter$, and calculate the maximum $\text{len}(R)$ according to Eq.(6.6) (lines 9 and 10).

Table 6.1 shows the result of the presented scalability analysis for two common DC topologies (fat-tree and two-tier), and three well-known continental backbone topologies [Routray et al. 2013] (ARPANET, GEANT2, and Internet2). For backbone topologies, as the number of ports varies for each node, we considered $nports$ as the maximum number of ports of any node. DC networks usually present switches with high number of ports and small diameter, while backbone networks usually present switches with few ports and

Table 6.2: Maximum $len(R)$ for different unicast SSR mechanisms in DC topologies.

Topology	nports	diam.	size	servers	PolKA	KeyFlow	Port Swit.
2-tier S=06 L=06	24	3	12	108	18	19	15
	48	3	12	252	21	20	18
	96	3	12	540	21	22	21
2-tier S=12 L=12	24	3	24	144	21	22	15
	48	3	24	432	21	23	18
	96	3	24	1008	24	24	21
2-tier S=16 L=16	24	3	32	128	21	23	15
	48	3	32	512	24	23	18
	96	3	32	1280	24	25	21
2-tier S=24 L=24	48	3	48	576	24	25	18
	96	3	48	1728	24	26	21
2-tier S=36 L=36	48	3	72	432	27	27	18
	96	3	72	2160	27	27	21
2-tier S=48 L=48	96	3	96	2304	27	29	21
Fat-tree	4	5	20	16	30	31	10
	8	5	80	128	45	44	15
	16	5	320	1024	55	56	20
	24	5	720	3456	60	63	25
	32	5	1280	8192	65	67	25
Hypercube	3	3	8	8	14	13	6
	4	4	16	16	24	24	8
	5	5	32	32	35	36	15
	6	6	64	64	48	50	18
	7	7	128	128	67	67	21
	8	8	256	256	88	86	24
	9	9	512	512	108	107	36
	10	10	1024	1024	130	130	40

larger diameter. Therefore, our topology set covers diverse and realistic use cases.

From a practical perspective, there are two design choices for embedding the *routeID*: (i) add extra headers, which causes transmission overhead; or (ii) reuse existing headers. For the topologies under worst case analysis in Table 6.2, the maximum $len(R)$ results show that PolKA fits existing packet headers for unicast and multicast (e.g., 96 bits of Ethernet source and destination addresses, 256 bits of IPv6 source and destination addresses, or a stack of MPLS labels with 20 bits per label). The atypically large diameter (21) of Internet2 may impose an extra header depending on the protocol stack.

As one of the main goals of this chapter is to develop a unicast SSR mechanism that can replace KeyFlow (integer RNS-based SSR mechanism) in the KeySFC scheme, we extended the previous scalability analysis focusing on unicast SSR in DC topologies. The results of PolKA are compared with KeyFlow, and also with Port Switching, which is the most traditional mechanism used for unicast SSR. In Table 6.2, we consider different configurations of two well-known network-centric topologies (two-tier and fat-tree) and one server-centric topology (hypercube).

It is important to note that in network-centric topologies the core nodes are represented by switches, while in server-centric networks the core nodes are represented by the servers itself, since servers are responsible for both computing and forwarding in this kind of architecture. A direct consequence of this consideration is that server-centric networks use more core nodes to interconnect an equivalent number of servers when compared to network-centric networks, and, consequently, demand more bits to encode the *routeID*.

The results for the maximum $len(R)$ are very close for PolKA and KeyFlow approaches. PolKA is better or equal to KeyFlow in almost all the cases, except in the ones that are highlighted in the table. This means that replacing KeyFlow by PolKA within the KeySFC scheme does not incur in reserving extra bits for the *routeID* header field.

On the other hand, the Port Switching method consumes less bits for representing the *routeID* than the RNS-based SSR approaches, specially in the cases where the size of the topology is high and the number of ports per node is low (fat-tree and hypercube). This happens because these cases demand a large number of irreducible polynomials (for PolKA) or prime number (for KeyFlow), causing the selection of polynomials of high degree or large integer numbers even when the node does not demand many ports.

When deploying RNS-based SSR, it is important to understand that, depending on the topology, the cost to exploit the special features from RNS may be to reserve more bits for representing the *routeID* in the packet header. Nevertheless, our scalability analysis considers the worst case scenario, and there are known techniques to make optimal assignment of *nodeIDs* and reduce the bit length of the *routeID* [Ren et al. 2017, Liberato et al. 2018].

6.1.5 Control plane

The control plane functionalities in PolKA are very similar to KeyFlow, with the difference that, instead of dealing with integers, the *nodeIDs* and *routeIDs* are the binary coefficients of GF(2) polynomials, as described in Section 6.1.2.

One of the main roles of the control plane is topology discovery and monitoring, since other steps depend on full knowledge of the network. The SDN Controller is also responsible for a network configuration phase, where it assigns unique *nodeIDs* to each core node. With the knowledge of the number of nodes and the number of ports of each node, the controller has to calculate the N irreducible polynomials for the *nodeIDs* while ensuring the degree of each polynomial supports the number of ports of that node (for unicast: $deg(s_i) \geq \lceil \log_2(nports) \rceil$).

When a new flow reaches an edge switch, the packet is forwarded to the SDN controller, which defines an end-to-end path across the core network for that flow. Then, the controller calculates the *routeID*, and installs a new table entry in the edge switch, which

is responsible to include the *routeID* in the packet headers before forwarding the flows to the core network. This procedure of installing the entries in the edge switches can also be done proactively by the SDN Controller.

The calculation of the *routeID* depends on the nodes in the path, the output port of each of these nodes, and the interconnection topology. Based on that, the computation involves a series of basic arithmetic operations over $GF(2)$, and the calculation of the multiplicative inverses, as explained in Section 6.1.2. This last step uses the Extended Euclidean Algorithm for GF polynomials.

When the packet is forwarded in the core network, the *modulo* operation replaces table lookup by performing the remainder of the division between the *routeID* embedded in the packet with the *nodeID* of each node. This is a great advantage, because there is significant reduction in control plane signaling when compared to traditional approaches, since *core switches* execute forwarding in a decentralized way without communicating with the Controller.

6.2 Design based on P4 architecture

This section studies how to design PolKA and a Port Switching SSR method according to P4 Architecture (see Section A.1.2). From now on, we will refer to the Port Switching method as Sourcey [Jin et al. 2016], since it is a prominent example of a Port Switching SSR method. In Section 6.3, this design will be used to build a Proof-of-Concept of PolKA and Sourcey in emulated and physical testbeds.

It is important to note that some of the design choices can be changed according to application requirements and target platform features. For example, the SSR information could be embedded in a new header or in an existing header (e.g., MAC addresses), as done in Section 5. In addition, the number of bits required to represent the port depends on the maximum number of ports of the switches in the network, and the maximum length of SSR headers depends on the network diameter. Moreover, limitations of the selected target may impose restrictions that affect the design choices.

6.2.1 Sourcey pipeline

With Port Switching SSR, the source adds a stack of output ports in the packet, which will be used by each switch in the network to forward the packet. We adapted and extended the source routing solution provided in P4.org tutorials ¹. This solution includes the port stack as a new header after the Ethernet header and selects a special etherType: `TYPE_SR=0x1234`. Each item of the source routing stack has a `bos` (bottom of stack) bit and a port number. The `bos` bit is 1 only for the last entry of stack. At ingress,

¹https://github.com/p4lang/tutorials/tree/master/exercises/source_routing

Ethernet	bos	port	bos	port	...	bos	port	IP	data
----------	-----	------	-----	------	-----	-----	------	----	------

(6.2.a) Sourcey header.

Ethernet	routeID						IP	data
----------	---------	--	--	--	--	--	----	------

(6.2.b) PolKA header with fixed length.

Ethernet	length	routeID				IP	data
----------	--------	---------	--	--	--	----	------

(6.2.c) PolKA header with variable length.

Figure 6.2: Comparison between SSR headers for Sourcey and PolKA.

each switch must pop an item from the stack and set the egress port according to the specified port number. Then, at ingress, it should pop an entry from the stack and set the egress port accordingly. The last hop should also revert back the etherType to `TYPE_IPv4=0x800`. Figure 6.2.a shows the format of Sourcey header.

The original solution has two main shortcomings: (i) it is not transparent to end-hosts, which must encapsulate SSR headers; and (ii) it unnecessarily parses the whole SSR header in each hop, producing a loop. To overcome these limitations, we designed a new pipeline in which the switches are responsible for encapsulating, decapsulating, and forwarding the packets according to the SSR header, and each hop only parses a single SSR header, as shown in Fig. 6.3.a.

When the packet reaches the switch, the first step is to parse the Ethernet header and check the etherType. If it is `TYPE_IPv4` (left side), the packet came from the end-host and the switch must parse the IPv4 header and encapsulate the SSR header. Each switch has a table, which was previously populated by the SDN Controller and provides the mapping between the destination IP address (DestIP) and the routing path, represented by the list of output ports. The result of this table lookup is an action that will set the first output port and call a function that will encapsulated n SSR headers, where n is the number of hosts in the path. Also, the etherType is changed to `TYPE_SR`, so the next switches can detect they must use SSR.

If the etherType is `TYPE_SR` (right side), the switch parses the first SSR header, get the output port, and pops this header from the stack. If the `bos` bit is set to one, it means this is the last hop and the etherType is changed to `TYPE_IPv4`. Instead of the pop operation, another possible implementation for progressing in the list is to have an index to the current position, which is incremented in each hop. Nevertheless, both options involve a packet rewrite in each hop.

6.2.2 PolKA pipeline

PolKA explores RNS to encode the list of output ports for SSR. Thus, instead of a list of ports, the PolKA header contains a *routeID*. Fig. 6.2.b shows the format of PolKA

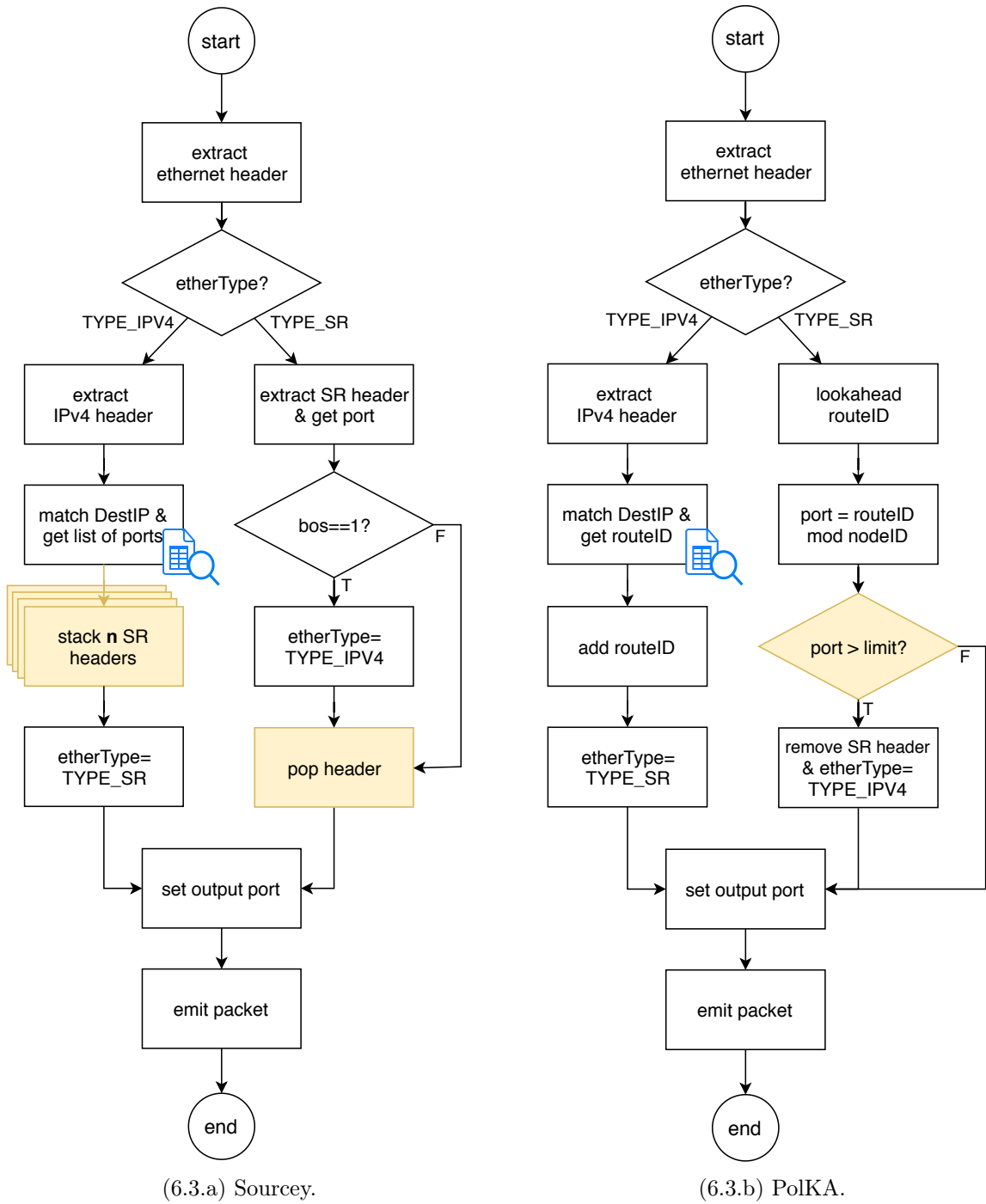


Figure 6.3: Comparison between Sourcey and PolKA complete pipelines.

header with a fixed field for storing the *routeID*, after the Ethernet header. This length will depend on some topology characteristics like the diameter and the path length, as explained in 6.1.4. Another possible implementation is to use an extra field that contains the length of the *routeID*, which would have variable length, as shown in Fig. 6.2.c. As a design choice, we selected the fixed length header, because the tests of Section 6.3 show this approach is more efficient.

As shown in Fig. 6.3.b, the left side of PolKA pipeline for encapsulating the SSR

header is similar to Sourcey, but the result of the table lookup is an action that sets the first output port and calls a function that encapsulates a single *routeID*, representing the routing path.

In the right side of the pipeline, when *etherType* is `TYPE_SR`, as PolKA only needs read access to the packet headers, it uses the `lookahead` method provided by P4 language that evaluates a set of bits from the input packet without advancing the packet index pointer. To discover the output port, the switch has to perform a *modulo* operation between the *routeID* in the packet and its own *nodeID*. Since this binary *modulo* operation with variables is not supported by P4 16 language, one contribution of this thesis is to enable it in P4 switches by reusing CRC operations, as presented in Section 6.2.5.

Besides, PolKA does not provide the notion of progression in the list, so there is no information in the header to identify the last hop. This is not a problem in fabric networks (as discussed in Section 6.2.4), because the packet delivery to an edge switch represents the end of the SSR path. However, in other cases, it may be necessary to separate a range of ports (e.g., the higher port numbers) that represent the switch ports connected to end-hosts. If the output port is one of these ports, the SSR header containing the *routeID* must be removed and the *etherType* is changed to `TYPE_IPv4`.

6.2.3 Headers and lookup tables

This section discusses the design using P4 of the headers and lookup tables used by the pipelines of Fig. 6.3. More information about P4 architecture can be found in Section A.1.2.

Code 6.1 and Code 6.2 shows the P4 code for defining the headers for Sourcey and PolKA according to the design of Fig. 6.2.

```

1 #define MAX_HOPS 10
2 typedef bit<48> macAddr_t;
3 typedef bit<32> ip4Addr_t;
4
5 header ethernet_t {
6     macAddr_t dstAddr;
7     macAddr_t srcAddr;
8     bit<16> etherType;
9 }
10
11 header srcRoute_t {
12     bit<1> bos;
13     bit<15> port;
14 }
15
16 header ipv4_t {
17     bit<4> version;

```

```

18  bit<4>      ihl;
19  bit<8>      diffserv;
20  bit<16>     totalLen;
21  bit<16>     identification;
22  bit<3>      flags;
23  bit<13>     fragOffset;
24  bit<8>      ttl;
25  bit<8>      protocol;
26  bit<16>     hdrChecksum;
27  ip4Addr_t  srcAddr;
28  ip4Addr_t  dstAddr;
29 }
30
31 struct headers {
32     ethernet_t      ethernet;
33     srcRoute_t [MAX_HOPS]  srcRoutes;
34     ipv4_t          ipv4;
35 }

```

Code 6.1: Sourcey: example P4 code for defining headers.

```

1  typedef bit<48> macAddr_t;
2  typedef bit<32> ip4Addr_t;
3
4  header ethernet_t {
5     macAddr_t dstAddr;
6     macAddr_t srcAddr;
7     bit<16>   etherType;
8 }
9
10 header srcRoute_t {
11     bit<160>   routeId;
12 }
13
14 header ipv4_t {
15     bit<4>     version;
16     bit<4>     ihl;
17     bit<8>     diffserv;
18     bit<16>    totalLen;
19     bit<16>    identification;
20     bit<3>     flags;
21     bit<13>    fragOffset;
22     bit<8>     ttl;
23     bit<8>     protocol;
24     bit<16>    hdrChecksum;
25     ip4Addr_t  srcAddr;
26     ip4Addr_t  dstAddr;
27 }

```

28
29
30
31
32
33

Code 6.2: PolKA: example P4 code for defining headers.

Our implementation of the Sourcey pipeline (Fig. 6.3.a) uses variable number of SSR headers depending on the number of hops in the path stack. Although P4 language allows to extract, assign and deparse variable-length headers, it does not directly support the encapsulation of variable size headers. One solution is to create one encapsulating action for each stack size, as shown in the examples of Table 6.3 and Code 6.3. Code 6.4 shows an example of an action that adds two SSR headers to the packet.

Table 6.3: Sourcey: example lookup table.

Key	Action	Action Data
10.0.1.2/32	add_header_1hop	eport=3, dstAddr=00:00:00:00:02:02, bos1=1, p1=1
10.0.1.3/32	add_header_2hops	eport=3, dstAddr=00:00:00:00:03:03, bos1=0, p1=3, bos2=1, p2=1
10.0.1.4/32	add_header_3hops	eport=3, dstAddr=00:00:00:00:04:04, bos1=0, p1=3, bos2=0, p2=3, bos3=1, p3=1

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

Code 6.3: Sourcey: example P4 code for declaration of lookup table.

1
2

```

3         bit<1> bos1, bit<15> p1){
4     standard_metadata.egress_spec = eport;
5     hdr.ethernet.dstAddr = dmac;
6     hdr.srcRoutes[0].setValid();
7     hdr.srcRoutes[0].bos = bos0;
8     hdr.srcRoutes[0].port = p0;
9     hdr.srcRoutes[1].setValid();
10    hdr.srcRoutes[1].bos = bos1;
11    hdr.srcRoutes[1].port = p1;
12 }

```

Code 6.4: Sourcey: example P4 code for declaration of an action.

On the other hand, the pipeline of PolKA (Fig. 6.3.b) uses a fixed size header to encapsulate the *routeID*. Thus, a single action can be used to add the SSR header regardless of the number of hops, as shown in the examples of Table 6.4, Code 6.5, and Code 6.4.

Table 6.4: PolKA: example lookup table.

Key	Action	Action Data
10.0.1.2/32	add_sr_header	eport=3, dstAddr=00:00:00:00:02:02, <i>routeID</i> =4294771599
10.0.1.3/32	add_sr_header	eport=3, dstAddr=00:00:00:00:03:03, <i>routeID</i> =159022805856541
10.0.1.4/32	add_sr_header	eport=3, dstAddr=00:00:00:00:04:04, <i>routeID</i> =17263697437380439085

```

1 table sr_encap {
2     key = {
3         hdr.ipv4.dstAddr: lpm;
4     }
5     actions = {
6         add_sr_header;
7         tdrop;
8     }
9     size = 1024;
10    default_action = tdrop();
11 }

```

Code 6.5: PolKA: example P4 code for declaration of a lookup table.

```

1 action add_sr_header(egressSpec_t eport, macAddr_t dmac, bit<160> routeID){
2     standard_metadata.egress_spec = eport;
3     hdr.ethernet.dstAddr = dmac;
4     hdr.srcRoute.setValid();
5     hdr.srcRoute.routeID = routeID;
6 }

```

Code 6.6: PolKA: example P4 code for declaration of an action.

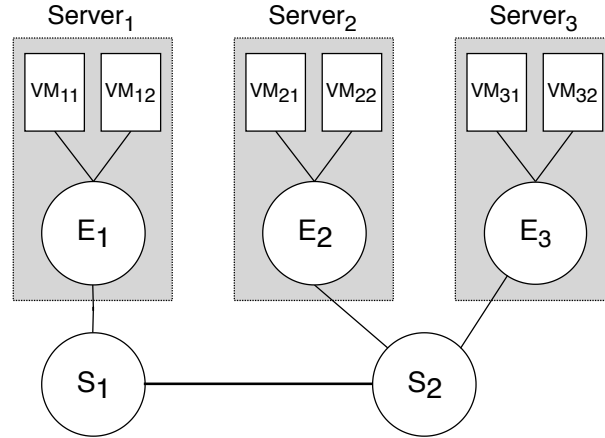


Figure 6.4: Example fabric network in DCs.

6.2.4 Discussion on fabric networks and pipeline differences

In Chapter 5, we argued that the solution for efficient SFC lies on the use of fabric networks, where core switches execute efficient packet forwarding and edge switches perform complex tasks (e.g., encapsulation and SFC classification). Also, this fabric network structure is commonly adopted in DC networks, which already have a clear distinction between software edge switches in the servers that host VMs and core hardware switches, as shown Fig. 6.4

In accordance with this approach, Sourcey and PolKA pipelines can be divided into an edge pipeline and a core pipeline. In this way, edge switches perform encapsulation and decapsulation of SSR headers, and core switches can be very simple and only execute forwarding without tables.

Fig. 6.5 shows the core pipelines for Sourcey and PolKA. Note that these core pipelines execute forwarding based on an operation over the SSR header (e.g., *pop* or *modulo* operation) and do not depend on any table lookup. Section 6.3 will present a prototype that compares the original pipelines with their fabric versions.

It is important to highlight the following characteristics of our Sourcey pipeline implementation that differ from our PolKA pipeline implementation (see Fig. 6.2, Fig. 6.3, and Fig. 6.5):

- Sourcey presents a variable number of SSR headers, depending on the number of hops in the routing path and the switch position in the path. Each hop performs a pop operation in the header stack, decreasing the stack size until the last hop removes the last SSR header. On the other hand, PolKA header has a fixed length and stores the *routeID*.
- Sourcey needs to create one encapsulating action for each stack size depending on the path length (e.g., `add_header_1hop`, `add_header_2hops`, `add_header_3hops`, ...), which increases the number of code lines and memory for deploying the edge



Figure 6.5: Comparison between Sourcey and PolKA core pipelines.

pipeline code. More information in Section 6.3.

- In Sourcey, each core switch performs a packet rewrite of the packet header when performing the pop operation (yellow in Fig. 6.5.a) to get the current output port and update the stack. On the other hand, in the PolKA core pipeline (Fig. 6.5.b), the packet remains unchanged for the entire routing path as the *routeID* is the same for all hops.
- In Sourcey, the output port is directly available in the SSR header, while PolKA requires an arithmetic operation over the *routeID* to calculate the output port.

Considering these differences, our hypothesis is that, if we can perform the *modulo* operation of PolKA with equivalent performance to the rewrite operation of Sourcey, we could take advantage of RNS special properties discussed in Section 3.2.1 without losing

performance when compared to traditional Port Switching methods. To this end, the next section presents a proposal for enabling the *modulo* operation in P4-enabled hardware.

6.2.5 Reuse of CRC hardware for implementing *modulo*

The binary polynomial and integer *modulo* operations with non-constant operands are not natively supported by commodity network hardware and are not available in standard P4 language. However, PolKA enables the exploitation of binary polynomial arithmetic operations that could be more easily mapped to network hardware than integer arithmetic.

For instance, CRC hardware provides wire-speed implementations of a binary *modulo* operation with a fixed polynomial base [Peterson and Brown 1961], and is evolving for supporting configurable polynomials [Grymel and Furber 2011, Microchip 2018]. In fact, we developed a technique that allows the execution of the binary *modulo* by reusing common CRC operations, which are supported by P4 standard architectures. In this way, PolKA can be deployed in P4-enabled switches and the *modulo* operation can be executed in CRC hardware with better performance.

6.2.5.1 Technique description

CRC codes are an important error-detection technique commonly employed in today's networking equipment. Similar to PolKA approach, in the CRC technique, the bit string coefficients are 0 and 1 values, manipulated using polynomial arithmetic operations to represent a quotient remainder.

Firstly, sender and receiver have to agree on a fix polynomial (G), called generator, which is a $r + 1$ bit pattern used to generate the error-detection code. The idea is that, for any binary data to be protected (D), with length of d bits, the sender will calculate additional r bits (R), and append them to D in such a way that the result is a polynomial with $d + r$ bits that is exactly divisible by G using modulo-2 arithmetic [Kurose and Ross 2013]. Fig. 6.6 shows an example of the CRC format.



Figure 6.6: Example of CRC format.

Thus, the computation of the CRC code depends on the remainder of the data shifted left by r bits, divided by the generator polynomial: $R = \langle D \cdot 2^r \rangle_G$.

Fig. 6.7 shows an example of the CRC calculation for $G = 1001$, $D = 101110$, $r = 3$, and $d = 6$. The result for the CRC code is $R = 011$. The quotient is not relevant, in this

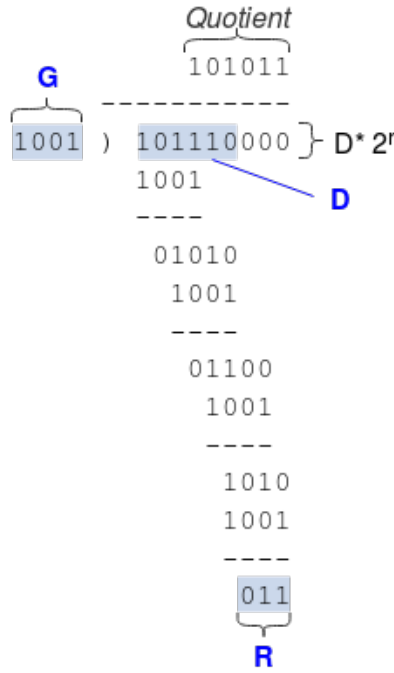


Figure 6.7: Example of CRC calculation. Adapted from [Kurose and Ross 2013].

case.

Since most of the network commodity hardware already supports CRC operations, we could try to map the *routeID* as *D*, the *nodeID* as *G*, and the output port as *R* in order to calculate the output port in each core node as: $R = portID = \langle routeID \rangle_{nodeID}$. However, PolKA does not perform a shift left operation over the *routeID* bits as done in the CRC strategy.

As the degree of *G* is *r*, this problem can be solved if we separate the *routeID* in two parts: $routeID = D * 2^r + dif$, where *dif* is the *r* least significant bits of the *routeID*. Firstly, we shift right the *routeID* by *r* bits to produce the data *D*, which will be the input of the CRC function. Then, the bits that were lost with the shift right operation (*dif*) can be added back to the calculated CRC remainder in the end of the computation to produce the output port. Since the degree of the *portID* obtained is less than *r*, the unicity property in division algorithm for polynomials assures that the outcome polynomial coincides with the remainder obtained by direct division of *routeID* by the *nodeID*. Also, in binary arithmetic, both the addition and subtraction operations are identical to the logical XOR operation. These steps are described as follows:

1. $G = nodeID, r = degree(G)$
2. $D = routeID \div 2^r$ **(SHIFT RIGHT)**
3. $dif = routeID - D * 2^r$ **(SHIFT LEFT, XOR)**
4. $R = \langle D * 2^r \rangle_G$ **(CRC)**

$$5. \text{ portID} = \text{dif} + R \quad (\text{XOR})$$

The example below illustrates how to use this method to calculate the output port when $\text{routeID} = 101110001$, and $\text{nodeID} = 1001$.

1. $G = 1001, r = 3$
2. $D = \text{routeID} \div 2^r = 101110001 \div 2^3 = 101110$
3. $\text{dif} = \text{routeID} - D * 2^r = 101110001 \oplus 101110000 = 001$
4. $R = \langle D * 2^r \rangle_G = \langle 101110000 \rangle_{1001} = 011$
5. $\text{portID} = \text{dif} + R = 001 \oplus 011 = 010$

Indeed, the direct *modulo* operation produces the same result:

$$\text{portID} = \langle \text{routeID} \rangle_{\text{nodeID}} = \langle 101110001 \rangle_{1001} = 010$$

Using this method, the switch can calculate the output port by using two *SHIFT* (steps 2 and 3), one CRC (step 4), and two *XOR* operations (steps 3 and 5), which is more computationally efficient in terms of clock cycles than executing a integer division or modulo-2 division.

Besides the basic *modulo* operation, the CRC technique may execute extra operations for the CRC standard codes (e.g., CRC32 or CRC16) and some parameters have to be configured, such as "Reversed", "Init-Value", and "XOR-out" ². PolKA sets these parameters to default values in order to enable basic quotient remainder: Reversed=False, Init-Value=0x0, XOR-out=0x0.

6.2.5.2 CRC support in P4

P4 language supports CRC operations through the use of external libraries, called externs, which are architecture-specific code that can be manipulated by P4 programs through APIs [P4.org 2017, da Silva et al. 2018]. The configuration of customized generator polynomials is a requirement for reusing the CRC operation for the *modulo* operation and depends on the selected P4 architecture model and its respective externs. The PSA [P4.org 2019c, P4.org 2019b] and v1model architectures [P4.org 2018b, P4.org 2019a] support the specification of an arbitrary CRC polynomial of 16 and 32 bits. Code 6.7 and Code 6.8 show the definition of the hash externs and the supported hash algorithms for PSA and v1model architectures, respectively.

```
1 enum PSA_HashAlgorithm_t {
2     IDENTITY,
3     CRC32,
```

²<https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code>

```

4     CRC32_CUSTOM,
5     CRC16,
6     CRC16_CUSTOM,
7     ONES_COMPLEMENT16,
8     TARGET_DEFAULT
9 }
10
11 extern Hash<O> {
12     Hash(PSA_HashAlgorithm_t algo);
13     O get_hash<D>(in D data);
14     O get_hash<T, D>(in T base, in D data, in T max);
15 }

```

Code 6.7: PSA: hash extern and supported hash algorithms. Source: [P4.org 2019b]

```

1 enum HashAlgorithm {
2     crc32,
3     crc32_custom,
4     crc16,
5     crc16_custom,
6     random,
7     identity,
8     csum16,
9     xor16
10 }
11
12 extern void hash<O, T, D, M>(out O result, in HashAlgorithm algo, in T base
    , in D data, in M max);

```

Code 6.8: v1model: hash extern and supported hash algorithms. Source:[P4.org 2019a]

The support on specific targets will depend on how they implement the architecture models. For instance, the software switch `bmw2_simple_switch` and the hardware switch Tofino from Barefoot support customized CRC polynomials, while Netronome SmartNICs implement some of the functionalities of *v1model* and only support fixed CRC polynomials.

Code 6.9 shows a example P4 code that implements the *modulo* operation with CRC, using the technique described in Section 6.2.5.1. It receives the *routeID*, which was previously parsed from the input packet, and executes the steps for calculating the output port using one CRC operation (implemented in the hash extern of *v1model*), two shift operations and two *XOR* operations.

```

1 action srcRoute_nhop() {
2     bit<16> nbase=0;
3     bit<64> ncount=4294967296*2;
4     bit<16> nresult;
5     bit<16> nport;
6
7     bit<160> routeid = meta.routeId;

```

```

8   bit<160> ndata = routeid >> 16;
9   bit<16> dif = (bit<16>) (routeid ^ (ndata << 16));
10  hash(nresult, HashAlgorithm.crc16_custom, nbase, {ndata}, ncount);
11  nport = nresult ^ dif;
12  meta.port = (bit<9>) nport;
13 }

```

Code 6.9: Example P4 code for calculating the output port using CRC operation.

6.3 Proof-of-concept and evaluation

The proof-of-concept (PoC) aims to demonstrate and evaluate the main functionalities of PolKA in comparison to a traditional Port Switching approach, named Sourcey. To this end, we built two prototypes of the solution proposed in Section 6.1 and designed in Section 6.2 using P4 16 language: (i) an emulated setup on Mininet based on *bmv2* (behavioral model)[P4.org 2019d] software switch to evaluate end-to-end scenarios, and (ii) a physical setup that uses Netronome SmartNICs³ to evaluate forwarding in a single hop scenario. Thus, this section addresses two main goals of this thesis: (i) to prove PolKA can be deployed in commercial programmable switches that support P4; and (ii) to prove PolKA can be implemented with similar performance to a Port Switching SSR approach.

6.3.1 Emulated setup with Mininet and bmv2

6.3.1.1 P4 target and architecture

The second version of the behavioral model framework, nicknamed *bmv2*, lets developers implement their own P4-programmable architecture as a software switch [P4.org 2019d]. The selected target was *simple_switch*, which is a software switch that runs on a general purpose CPU (e.g., Intel, and AMD) [P4.org 2019e, P4.org 2018a]. It was selected because it is currently the most complete implementation of the functionalities specified by P4 14 and P4 16 languages. Other target implementations have restrictions on specific features that were necessary for implementing PolKA, such as configuration of CRC polynomials.

The *v1model* architecture is the de-facto architecture for most P4 developers [P4.org 2019e], because it was designed to match the architecture defined in the P4 14 language specification [P4.org 2018b]. Besides, it is included with the *p4c* compiler and supports both P4 14 and P4 16 programs. The P4 16 language [P4.org 2017] also has a Portable Switch Architecture (PSA) [P4.org 2019c], but the implementation in *bmv2* using the target *psa_switch* is not yet complete. Since the *psa_switch* still does not have a mature implementation and the *v1model* architecture with *bmv2 simple_switch*

³<https://www.netronome.com/>

target supports P4 16 language with all the pipeline blocks and functionalities required for PolKA implementation, `v1model` was selected for this prototype.

It is important to highlight that `bmv2 simple_switch` is a user space implementation with focus on functionality and feature testing rather than production performance⁴. There are other high performance open source implementations of P4 software switches and compilers (e.g., PISCES⁵, P4ELTE⁶, and MACSAD⁷), but to the best of our knowledge they do not yet cover all the features required by PolKA. As these implementations evolve, it will be possible to test our prototype with higher loads. For the time being, the solution was to limit the link rates in our emulated prototype to avoid reaching the processing capacity limits of the `bmv2 simple_switch` implementation. In addition, we compiled `bmv2` with the appropriate compile time and runtime flags for improving performance: `'CXXFLAGS=-O2' -disable-logging-macros -disable-elogger`.

6.3.1.2 Setup description

The physical setup consists of one server Dell PowerEdge T430, with one Intel Xeon E5-2620 v3 2.40GHz processor, and 64GB of RAM.

To build our emulated environment in this server, we used `p4app`⁸, which is a tool that can build, run, debug, and test P4 programs using a Docker image containing the P4 compiler, `bmv2`, and `Mininet`. It creates a container with an emulated network using `Mininet`, where it is possible to execute experiments with special functionality like compiling P4 programs and configuring `bmv2` switches. However, this original design presents limited flexibility, since it exposes a JSON configuration file with fixed options to the user and only allows the execution of a single P4 program. Recently, a branch of `p4app`⁹ was released that converts `p4app` into a Python library that wraps-up `Mininet`, providing more flexible ways to configure the emulated topology and run different P4 programs on different switches. This functionality is crucial to emulate fabric networks, which require different P4 programs for edge and core elements.

Table 6.5 shows the versions of the software used in this prototype.

6.3.1.3 P4 control plane implementation

The control plane in PolKA has the following main functionalities: (i) compute the *nodeIDs* and configure the core switches with their respective identifiers; and (ii) compute the routing paths for the traffic flows, calculate the *routeIDs* for the paths, and configure

⁴http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2016-July/000412.html

⁵<http://pisc.es.princeton.edu/>

⁶<http://p4.elte.hu/>

⁷<https://github.com/intrig-unicamp/macsad/>

⁸<https://github.com/p4lang/p4app/tree/master/>

⁹<https://github.com/p4lang/p4app/tree/rc-2.0.0/>

Table 6.5: Emulated setup: software versions.

Software	Name	Version
Operating System	Ubuntu server	16.04.6 LTS
Kernel	Linux	4.4.0-148-generic
P4 compiler	p4c	1.1.0-rc1
Software switch	bmv2 simple_switch	1.13.0
Emulation tool	Mininet	2.3.0d5

the table entries in the edge switches that will be responsible for the encapsulation and decapsulation of *routeIDs*.

For the RNS computation of *nodeIDs* and *routeIDs*, as described in Section 6.1.2, we developed a Python library that uses the package *sympy.polys.galoistools* of the *sympy* library¹⁰ for GF(2) arithmetic operations. For the prototype, the following main functions were implemented for GF(2) polynomials:

- Generate k irreducible polynomials of minimum degree n:
`create_list_irrpoly_mindeg(k, n)`
- Calculate the *routeID* polynomial for a routing path, given the list of *nodeID* polynomials (nodelst) and the list of *portID* polynomials (portlst):
`calculate_routeid(nodelst, portlst)`

The topology discovery and path computation were considered static inputs for the control plane application, because it is not the focus of this work.

The `bmv2 simple_switch` accepts TCP connections from a controller, which can be used to program the runtime behavior of each switch device [P4.org 2019d]. The format of control messages is defined by a Thrift API and a CLI (`simple_switch_CLI`) connects to the Thrift RPC server running in each switch process [P4.org 2019e]. The `bmv2 simple_switch` also supports P4 Runtime¹¹, a silicon-independent and protocol-independent API for control plane applications that can be auto-generated from an unambiguous definition of a packet processing pipeline in P4. However, it was not possible to use it in this prototype, because it does not yet support the configuration of CRC parameters.

For the configuration of core and edge switches, we developed a control plane application in Python that communicates with the switches using the CLI commands. Code 6.10 shows usage examples of the CLI commands used in this prototype:

```

1 #Usage: set_crc16_parameters <name> <polynomial> <initial remainder> <final
   xor value> <reflect data?> <reflect remainder?>
2 #Example: Set CRC generator polynomial 0x1002b with default parameters.
3 set_crc16_parameters calc 0x002b 0x0 0x0 false false

```

¹⁰<https://www.sympy.org/>

¹¹<https://p4.org/p4-runtime/>

```

4 #Usage: table_add <table name> <action name> <match fields> => <action
   parameters> [priority]
5 #Example: Add table entry that encapsulates a PolKA header
6 table_add sr_encap add_sr_header 10.0.2.1/32 => 1 00:00:00:00:02:01
   179902035595884
7 #Usage: table_delete <table name> <entry handle>
8 #Example: Delete table entry
9 table_delete sr_encap 1
10 #Usage: table_modify <table name> <action name> <entry handle> [action
    parameters]
11 #Example: Modify table entry that encapsulates a PolKA header
12 table_modify sr_encap add_sr_header 1 => 1 00:00:00:00:02:01
   179902035595884

```

Code 6.10: Commands of switch CLI used by control plane application.

6.3.1.4 P4 data plane implementation

The following P4 16 programs were developed using `v1model`:

- **sourcey.p4**: implements all edge and core functionalities of Sourcey pipeline described in Fig. 6.3.a;
- **sourcey-edge.p4**: implements only encapsulation and decapsulation functionalities of Sourcey pipeline described in Fig. 6.3.a;
- **sourcey-core.p4**: implements only core functionalities of Sourcey pipeline described in Fig. 6.5.a;
- **polka.p4**: implements all edge and core functionalities of PolKA pipeline described in Fig. 6.3.b;
- **polka-edge.p4**: implements only encapsulation and decapsulation functionalities of PolKA pipeline described in Fig. 6.3.b;
- **polka-core.p4**: implements only core functionalities of PolKA pipeline described in Fig. 6.5.b;
- **polka-varheader.p4**: implements all edge and core functionalities of PolKA pipeline, but uses a different PolKA header with variable length as described in Fig. 6.2.c (instead of fixed length header);

Table 6.6 shows the source lines of code (SLOC) metric for these programs, which were used for implementing the following solutions:

- **Sourcey**: all switches deploy `sourcey.p4`.

Table 6.6: P4 programs: Source lines of code (SLOC).

Program	SLOC
sourcey.p4	821
polka.p4	192
polka-var.p4	746
sourcey-core.p4	105
sourcey-edge.p4	871
polka-core.p4	121
polka-edge.p4	147

- **Sourcey Fabric:** edge switches deploy `sourcey-edge.p4` and core switches deploy `sourcey-core.p4`.
- **PolKA:** all switches deploy `polka.p4`.
- **PolKA Fabric:** edge switches deploy `polka-edge.p4` and core switches deploy `polka-core.p4`.
- **PolKA-Var:** all switches deploy `polka-varheader.p4`.

As explained in Section 6.1.4 the number of bits that are necessary to represent the *routeID* and, consequently, the size of the PolKA header depends on some topologies characteristics. However, `bm2 simple_switch` only supports the specification of CRC polynomials of 16 and 32 bits (see Section 6.2.5.2). Therefore, although, in theory, we could use polynomials of much smaller degree to implement our tests, in practice, our PoC has to adopt polynomials of degree 16. As our test topologies have a maximum diameter of 10, we defined the size of the PolKA header as 160 bits (worst case scenario). In order to get a fair comparison, we defined the Sourcey header as 16 bits (1 bit for *bos* and 15 bits for representing the ports), so, in the worst case of 10 nodes in the path, the array of Sourcey headers would achieve 160 bits.

6.3.1.5 Test scenarios and experiments

The following test scenarios were designed to test the developed solutions:

1. **Linear:** The scenario of Fig. 6.8 contains 10 switches connected in a linear topology, each switch is connected to one host and runs both core and edge functionalities. The objective is to compare PolKA and Sourcey solutions with increasing number of hops in the core network (e.g., from 0 for path $H_1 \rightarrow H_{11}$ to 9 for path $H_1 \rightarrow H_{10}$). We also compare two different PolKA solutions with fixed and variable-length headers.
2. **Fabric:** The scenario of Fig. 6.9 is similar to the previous scenario, but each switch was split into a core and edge switch. In this way, we can measure the impact of the fabric solutions.

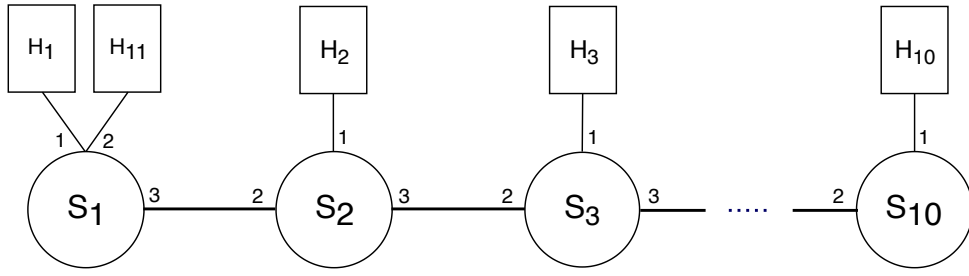


Figure 6.8: Linear topology.

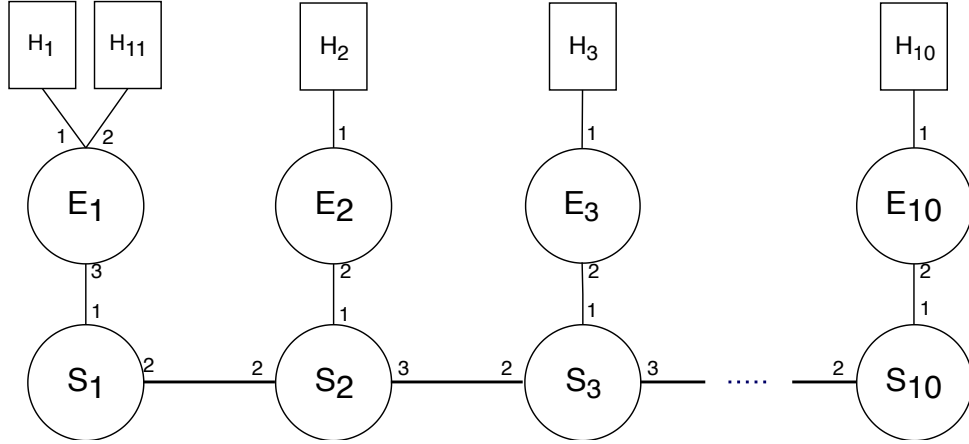


Figure 6.9: Linear fabric topology.

3. **Two-tier:** This scenario represents a two-tier DC topology with fabric separation (Fig. 6.12.a), where spine switches and leaf switches run core functionalities and servers contain an edge switch that interconnects VMs. The objective is to demonstrate how PolKA and Sourcey, as SSR solutions, can provide agile path migration.

As discussed in Section 6.3.1.1, the link rates were limited to 10Mbps to avoid reaching the processing capacity limits of the `bm2 simple_switch` implementation. For the tests, we considered small packets as Ethernet frames of 98 bytes (IP payload of 64 bytes), and big packets as Ethernet frames of 1242 bytes (IP payload of 1208 bytes).

The following experiments were designed for the test scenarios:

1. **Round Trip Time (RTT):** host H_1 sends 1 ICMP packet/s during 65s to each of the other hosts using `ping` tool. First 5 samples are discarded. This test was executed with small and big packets. Also, in one test a background UDP traffic of 5Mbps (half of link capacity) with same destination of the ICMP traffic was generated in parallel.
2. **Jitter:** host H_1 sends a UDP traffic of 5Mbps (half of link capacity) to each of the other hosts during 70s using `iperf` tool. First 5 samples are discarded. This test was executed with big packets.
3. **Flow completion time (FCT):** host H_1 transmits a file of 100Mb over a TCP

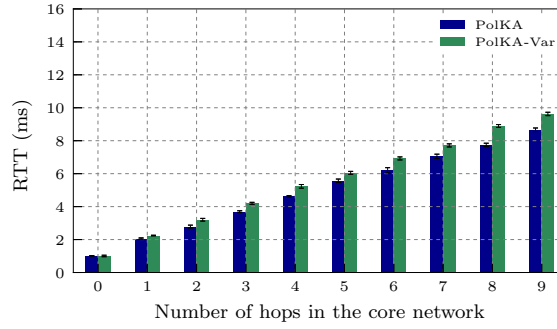


Figure 6.10: Linear scenario: comparison between PolKA and PolKA-Var solutions.

connection to each of the other hosts using `iperf` tool. The test was repeated 3 times. This test was executed with big packets.

All the graphs show average and standard deviation for the related measurements.

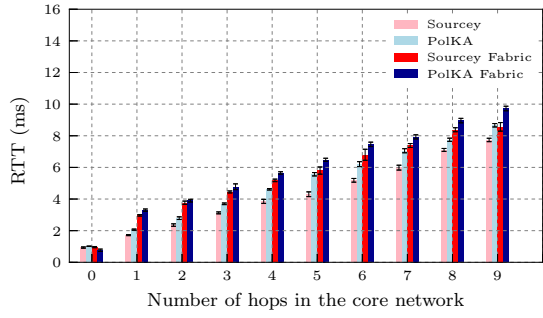
6.3.1.6 Linear and fabric scenarios

Firstly, we compared PolKA and PolKA-Var solutions as described in Sec. 6.3.1.4 for the linear topology of Fig. 6.8. The RTT experiment in Fig. 6.10 shows the PolKA solution with fixed header presents lower latency when compared with PolKA-Var solution with variable header size. Moreover, the PolKA-Var implementation is more complex than PolKA implementation (see Table 6.6 for SLOC comparison). Thus, the PolKA solution with fixed header was adopted for comparison with Sourcey.

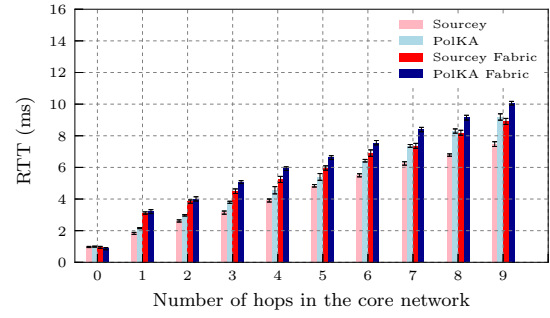
Fig. 6.11 shows the comparison between PolKA and Sourcey solutions (including their fabric versions) for the RTT, Jitter, and FCT experiments for the topologies of Fig. 6.8 and Fig. 6.9.

In the RTT small packet (Fig. 6.11.a), RTT big packet (Fig. 6.11.b), and RTT background traffic (Fig. 6.11.c) experiments, it is possible to observe that some behaviors are repeated: (i) the latency grows linearly with the increase of the number of hops for all the solutions; (ii) fabric solutions add a bit more latency, because they use the linear fabric topology, which have two extra hops in the path between two end hosts compared to the non-fabric solutions that use the linear topology; (iii) Sourcey fabric and non-fabric solutions present better latency performance than their respective PolKA solutions; and (iv) the standard deviation is small and in the same order of magnitude for all the solutions.

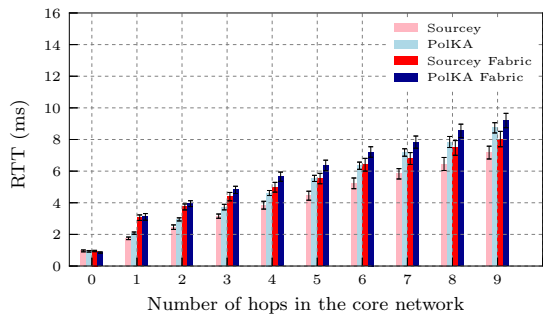
Besides, there is no significant difference in the results for different packet sizes in the RTT experiments. This is because Mininet does not consider transmission time in the emulation. Also, the background traffic did not significantly affected the RTT results. In addition, the jitter (Fig. 6.11.d) is small and equivalent for all the solutions. Finally, the FCT experiments (Fig. 6.11.e) show that all the solutions require approximately the same time to transfer the file and the standard deviation is small.



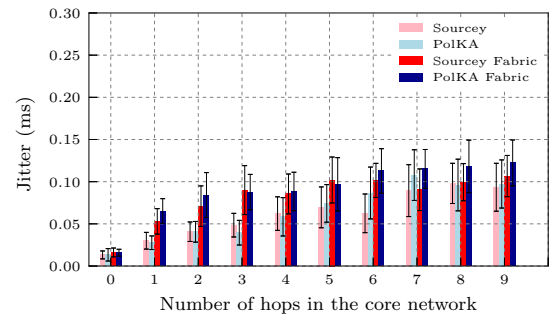
(6.11.a) RTT small packet.



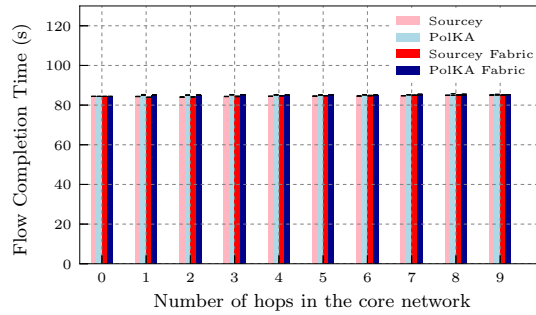
(6.11.b) RTT big packet.



(6.11.c) RTT background traffic.



(6.11.d) Jitter.



(6.11.e) FCT.

Figure 6.11: Linear and fabric scenarios: comparison between Sourcey and PolKA solutions.

The fact that Sourcey has a better latency performance than PolKA is related to two facts: Sourcey loses one SSR header per hop, so the average packet header size is smaller than the fixed header used by PolKA; and the CRC operation for PolKA in this emulated prototype is executed in software. Nevertheless, the difference between the two solutions is small and can decrease if the CRC operation is performed in hardware, as we show in Section 6.3.2.

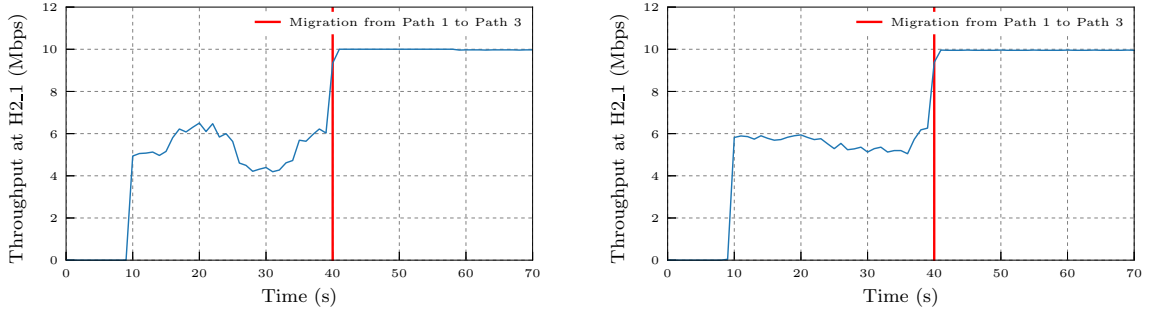
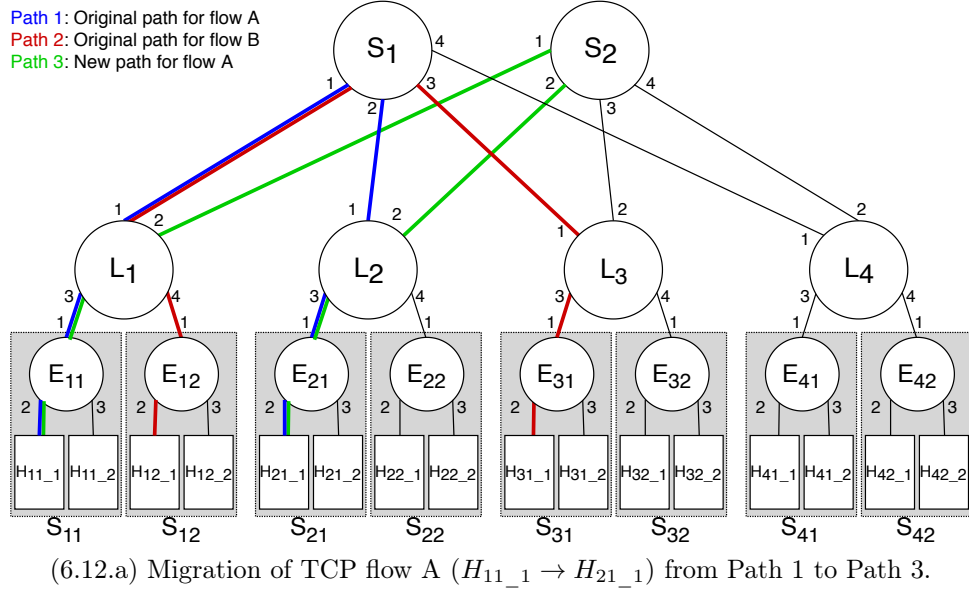


Figure 6.12: Two-tier scenario: agile path migration of TCP flow for allocation of maximum bandwidth in Sourcey and PolKA.

6.3.1.7 2-Tier scenario

Fig. 6.12.a shows a two-tier DC topology, which contains two spine switches (S_1 and S_2) and four leaf switches (L_1, L_2, L_3 , and L_4), all running core functionalities. Each leaf switch is connected to two servers (S_{ij}), which have one edge switch (E_{ij}) to interconnect VMs (H_{ij_k}).

We run the same experiments for Sourcey and PolKA using the core and edge P4 programs described in Section 6.3.1.4. The first experiment shows how traffic engineering can benefit from SSR mechanisms for allocation of maximum bandwidth with agile path migration.

At instant 10s, flow A ($H_{11_1} \rightarrow H_{21_1}$) and flow B ($H_{12_1} \rightarrow H_{31_1}$) start TCP traffics with big packets using `iperf` tool. Initially, flow A is allocated to Path 1 ($H_{11_1} - E_{11} - L_1 - S_1 - L_2 - E_{21} - H_{21_1}$) and flow B is allocated to Path 2 ($H_{12_1} - E_{12} - L_1 - S_1 - L_3 - E_{31} - H_{31_1}$), as shown in Fig. 6.12.a.

Thus, these flows compete for bandwidth at link $L_1 - S_1$ in the interval of 10s to 40s.

The effects of the TCP congestion control mechanism for fair share of total bandwidth can be seen in the graphs of Fig. 6.12.b and Fig. 6.12.c that show the throughput at the destination of flow A (H_{21_1}) for Sourcey and PolKA, respectively.

At instant 40s, the traffic engineering decides to exploit alternative idle links and migrates flow A from Path 1 to Path 3 ($H_{11_1} - E_{11} - L_1 - S_2 - L_2 - E_{21} - H_{21_1}$). From this moment, there is no more competition between flow A and flow B and the graphs show that flow A reaches destination consuming the total link bandwidth of 10Gbps.

To perform the described path migration, the SDN Controller only has to modify a single flow entry at the edge switch E_{11} for destination H_{21_1} (IP address 10.0.2.1/32). Table 6.7 and Table 6.8 show the original entry that embeds the SSR header for Path 1 for Sourcey and PolKA, respectively. For selecting Path 3, the only action data field that has to change in this example is the $p1$ field in Sourcey, as shown in the second line of Table 6.7. For PolKA, the only field that has to be modified is the *routeID* to embed the new route through Path 3, as shown in Table 6.8. Code 6.11 shows the `simple_switch_CLI` commands for dynamically modifying this entry for path migration in Sourcey and PolKA. Once this single `table_modify` operation is executed, all the packets of flow A that leave H_{11_1} will be tagged with the route information of the new path.

Table 6.7: Sourcey: Table entries at edge switch E_{11} for destination H_{21_1} .

Path	Key	Action	Action Data
1	10.0.2.1/32	add_header_3hops	ipport=1, dstAddr=00:00:00:00:02:01, bos1=0, p1=1 , bos2=0, p2=2, bos3=1, p3=3
3	10.0.2.1/32	add_header_3hops	ipport=1, dstAddr=00:00:00:00:02:01, bos1=0, p1=2 , bos2=0, p2=2, bos3=1, p3=3

Table 6.8: PolKA: Table entries at edge switch E_{11} for destination H_{21_1} .

Path	Key	Action	Action Data
1	10.0.2.1/32	add_sr_header	ipport=1, dstAddr=00:00:00:00:02:01, routeID=71955628459531
3	10.0.2.1/32	add_sr_header	ipport=1, dstAddr=00:00:00:00:02:01, routeID=179902035595884

```

1 #Command usage: table_modify <table name> <action name> <entry handle> =>
2 #[action parameters]
3 #Migration of Flow A to Path 3 in Sourcey:
4 table_modify tunnel_encap_process_sr add_header_3hops 1 =>
5     1 00:00:00:00:02:01 0 2 0 2 1 3
6 #Migration of Flow A to Path 3 in PolKA:
7 table_modify tunnel_encap_process_sr add_sr_header 1 =>
8     1 00:00:00:00:02:01 179902035595884

```

Code 6.11: Usage example of `simple_switch_CLI` command for modifying table entry.

The second experiment investigates the effect of path reconfiguration in the packet loss. To that end, we use the same topology of the previous experiment (Fig. 6.12.a). At instant 10s, we start a single 5Mbps UDP flow with big packets from H_{11_1} to H_{21_1} ,

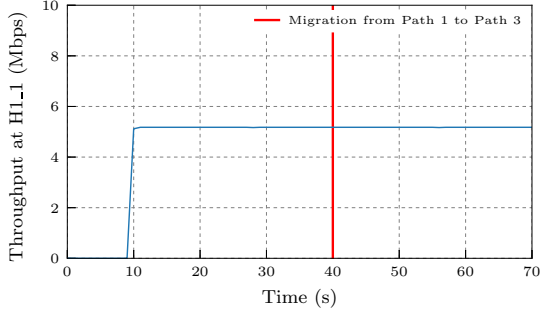
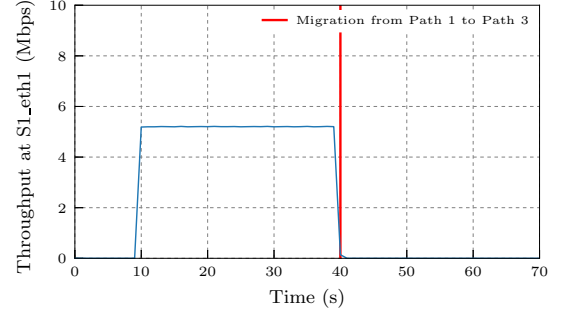
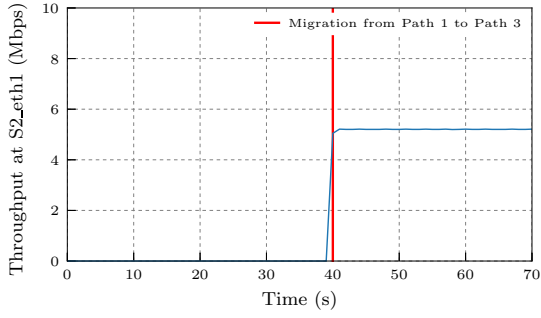
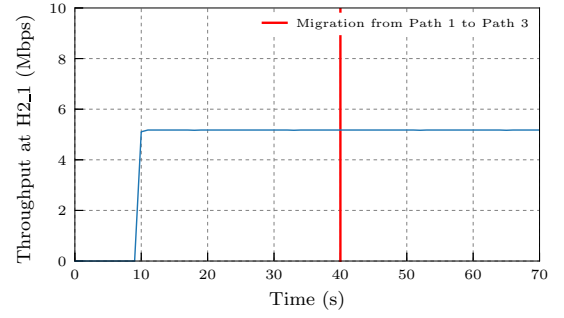
(6.13.a) PolKA: throughput at source (H_{11_1}).(6.13.b) PolKA: throughput at S_1 .(6.13.c) PolKA: throughput at S_2 .(6.13.d) PolKA: throughput at destination (H_{21_1}).

Figure 6.13: Migration of UDP flow from Path 1 to Path 3 with no concurrent traffic in PolKA.

with no concurrent traffic. Initially, this UDP flow is allocated to Path 1 ($H_{11_1} - E_{11} - L_1 - S_1 - L_2 - E_{21} - H_{21_1}$). Then, at instant 40s, we migrate this flow to Path 3 ($H_{11_1} - E_{11} - L_1 - S_2 - L_2 - E_{21} - H_{21_1}$), changing one single flow entry at edge switch E_{11} , as explained in the previous experiment.

Fig. 6.13 shows the throughput measurements of this second experiment for PolKA using `bwm-ng` tool at source (Fig. 6.13.a), spine switch S_1 (Fig. 6.13.b), spine switch S_2 (Fig. 6.13.c), and destination (Fig. 6.13.d). The graphs show the traffic correctly pass through S_1 when Path 1 is selected (10s to 40s) and through S_2 when Path 3 is selected (40s to 70s).

Considering the limits of throughput and packet per second rate of our setup with `bwm2 simple_switch`, this experiment was executed with throughputs of 1Mbps, 2Mbps, 3Mbps, 4Mbps, 5Mbps, 6Mbps, 7Mbps, and 8Mbps, and repeated 30 times for each throughput rate. Comparing the total traffic sent by source with the total traffic received at destination, no packet loss was registered. The same behavior was observed for tests with Sourcey.

6.3.2 Physical setup with SmartNICs

In the `bm2` prototype, the CRC operation is executed in software using a table lookup implementation¹². However, one of the main benefits of exploring the CRC operation in PolKA is the possibility to implement the modulo operation in hardware with better performance.

A P4 hardware target can be a switching ASIC, a FPGA, or a generic compute, while a hardware platform is a switch (using ASIC), a smartNIC (using FPGA), or a server machine¹³. To support P4, a target has to support a P4 compiler backend. This section evaluates PolKA and Sourcey solutions in a hardware prototype developed using Netronome SmartNICs.

6.3.2.1 SmartNICs features

The SmartNICs give access to the following timestamps to P4 programs :

- *`intrinsic_metadata.ingress_global_timestamp`*: 64-bit value representing the time at packet ingress within the MAC chip component.
- *`intrinsic_metadata.current_global_timestamp`*: 64-bit value representing the current time with respect to the time within the MAC block.

The timestamps are composed of a 32-bit seconds value in the top 32-bits and a 32-bit nanoseconds value in the lower 32-bits. This prototype uses these hardware timestamps to measure forwarding latency.

The Netronome SmartNIC partially implements the functionalities of *v1model* for P4 16, but it currently supports only fixed CRC polynomials, such as standard CRC-32 and CRC-CCITT (CRC-16). This restricts the use of its CRC hardware, because PolKA requires to set different polynomials with default parameters for each core node. Nevertheless, it is possible to measure forwarding latency in one core node as proposed in the next section.

6.3.2.2 Setup description

The physical setup is illustrated by Fig. 6.14 and consists of two servers: (i) S0: a device under test (DUT), running the core functionalities; and (ii) S1: a traffic generator (TG), running the edge functionalities, and transmitter (TX) and receiver (RX) functionalities in separate network namespaces. Both servers are Dell PowerEdge T430, with one Intel Xeon E5-2620 v3 2.40GHz processor, 16GB of RAM, and one Netronome Agilio CX 2x10GbE SmartNIC (Fig. 6.15). Table 6.9 shows the versions of the software used in this prototype.

¹²https://github.com/p4lang/behavioral-model/blob/master/src/bm_sim/crc_map.cpp

¹³<https://github.com/hesingh/p4-info>

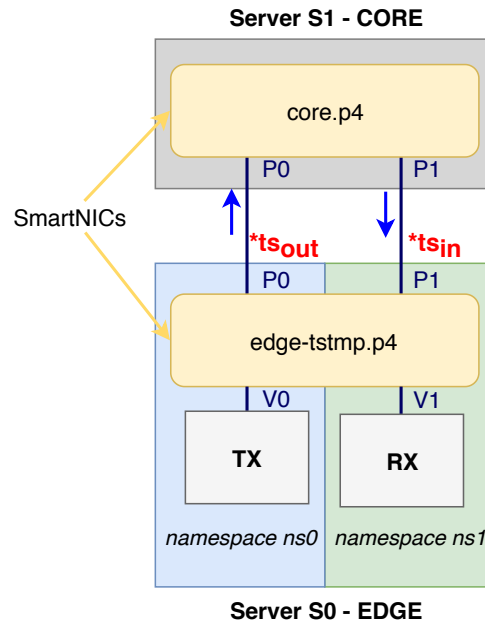
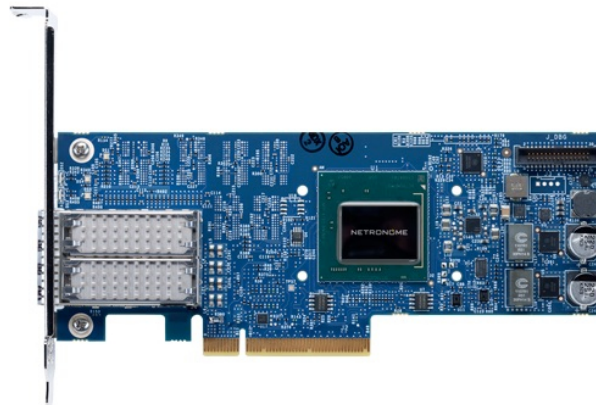


Figure 6.14: SmartNIC setup.

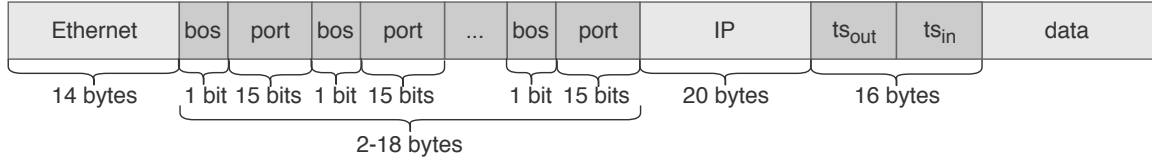
Figure 6.15: Netronome Agilio CX 2x10GbE SmartNIC. Source: <https://www.netronome.com/products/agilio-cx/>

The P4 programs deployed on the SmartNICs are adaptations of the core and edge codes for Sourcey and PolKA described in Section 6.3.1.4. The adaptations of the core codes (**polka-core.p4** and **sourcey-core.p4**) only ensure compatibility with some bit lengths supported by the SmartNICs. The new edge codes extend (**polka-edge-tstmp.p4** and **sourcey-edge-tstmp.p4**) the original codes to encapsulate both the SSR headers and a timestamp header, which is composed by an egress (ts_{out}) and an ingress timestamp (ts_{in}) and inserted after the IP header. Fig 6.16 shows the new format of the packet with the timestamp header for executing latency measurements.

At namespace **ns0** in server **S0**, the packets are generated at **TX** using an IPv4 traffic generator tool (e.g., **pktgen** or **ping**) and forwarded to virtual interface **V0**, which is attached to the SmartNIC. The P4 program that runs on the SmartNIC receives these packets from **TX** and encapsulates the SSR header (PolKA or Sourcey) and the timestamp

Table 6.9: Physical setup: software versions.

Software	Name	Version
Operating System	Ubuntu server	18.04.1 LTS
Kernel	Linux	4.15.0-29-generic
P4 compiler	nfp4c	1.0.1-766d9d3cd66
Software Development Kit	Network Flow Processor SDK	6.1-preview



(6.16.a) Sourcey.



(6.16.b) PolKA.

Figure 6.16: Timestamps header.

header.

When the packet leaves the ingress block in the edge pipeline, the value of *intrinsic_metadata.current_global_timestamp* is assigned to *ts_{out}*. Then, it goes to the deparser block and is transmitted over physical interface P0.

The packet reaches physical interface P0 at server S1 and is processed by the core code at the SmartNIC, which is responsible for parsing the SSR header and computing the output port. Since it is not possible to configure customized CRC polynomials in the SmartNICs, we use a standard CRC operation with a fixed CRC-16 polynomial. In this way, we execute all the PolKA pipeline steps (including the CRC operation) to measure their contribution in the total latency, but, instead of using the result of the CRC operation to select the output port, we select a fixed output port. As a result, the packet is delivered to physical interface P1 at server S1, which is connected to physical interface P1 at server S0.

The edge code at S0 parses the packet and assigns the value of *intrinsic_metadata.ingress_global_timestamp* to *ts_{in}*. Then, it removes the SSR header and delivers the packet to virtual interface V1 at namespace ns1. At ns1, we capture all the packets that arrive at V1 with `tcpdump` tool and save the output to a *.pcap file. Finally, we parse the files offline with `scapy` to extract the forwarding latency in the core for each packet as: $ts_{in} - ts_{out}$.

Table 6.10: SSR headers based on destination IP address

DestIP	SSR Header in Sourcey	SSR Header in PolKA
10.0.100.1	(bos1, p1)	routeID=1
10.0.100.2	(bos1, p1),(bos2, p2)	routeID=4294771599
10.0.100.3	(bos1, p1),(bos2, p2),(bos3, p3)	routeID=159022805856541
10.0.100.4	(bos1, p1),(bos2, p2),...(bos4, p4)	routeID=17263697437380439085
10.0.100.5	(bos1, p1),(bos2, p2),...(bos5, p5)	routeID=1149398238047081127332954
10.0.100.6	(bos1, p1),(bos2, p2),...(bos6, p6)	routeID=59723885083156140294227912283
10.0.100.7	(bos1, p1),(bos2, p2),...(bos7, p7)	routeID=2194656173762523641939709652656780
10.0.100.8	(bos1, p1),(bos2, p2),...(bos8, p8)	routeID=32763471027773366297233451711039667216
10.0.100.9	(bos1, p1),(bos2, p2),...(bos9, p9)	routeID=16050698998725239657676330566116710828499122

Table 6.11: SmartNIC test scenarios: deployed P4 programs.

#	Test scenarios	Core	Edge
1	PolKA	polka-core.p4	polka-edge-tstmp.p4
2	PolKA Baseline	wire.p4	polka-edge-tstmp.p4
3	Sourcey	sourcey-core.p4	sourcey-edge-tstmp.p4
4	Sourcey Baseline	wire.p4	sourcey-edge-tstmp.p4

6.3.2.3 Test scenarios and cases

The objective is to measure the average core forwarding latency for a single hop in PolKA and Sourcey when the path lengths increases. Here, we consider the path length as the number of core nodes that must be considered in the SSR header to reach the destination. For each test execution, the traffic generator tool at TX varies the IP destination address. The last digit of the IP destination address represents the number of core nodes. For example, if IP destination is 10.0.100.1, the number of core nodes to the destination is 1, while IP destination 10.0.100.9 represents 9 core nodes to the destination. Table 6.10 shows the SSR headers that are converted to table entries in the edge to match the respective IP destination address and encapsulate the appropriate headers.

Table 6.11 shows the four test scenarios and the respective edge and core P4 programs. The `wire.p4` code is a simple P4 program that receives packets and forward them to port physical P1 without any further processing. We use this program to create baseline to compare PolKA and Sourcey results.

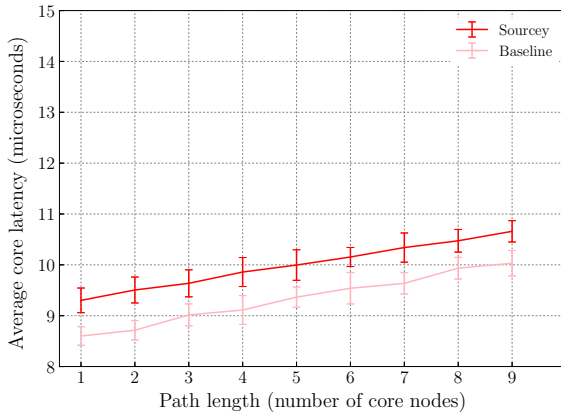
We investigate these scenarios considering the variation of two parameters:

- **Packet size:**

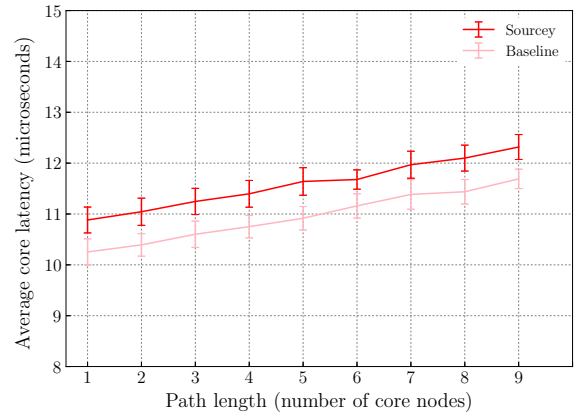
- **small:** Ethernet frame of 98 bytes (IP payload of 64 bytes).
- **large:** Ethernet frame of 1242 bytes (IP payload of 1208 bytes).

- **Throughput:**

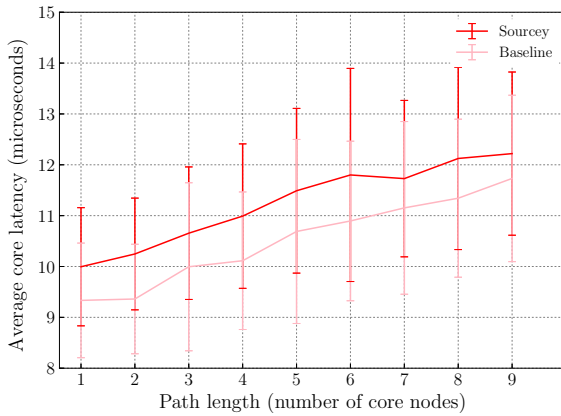
- **low:** one ICMP packet/second, 100 packets in total, generated with *ping* tool.
- **high:** 1Gbps UDP packets, 1000 packets in total, generated with *pktgen* tool.



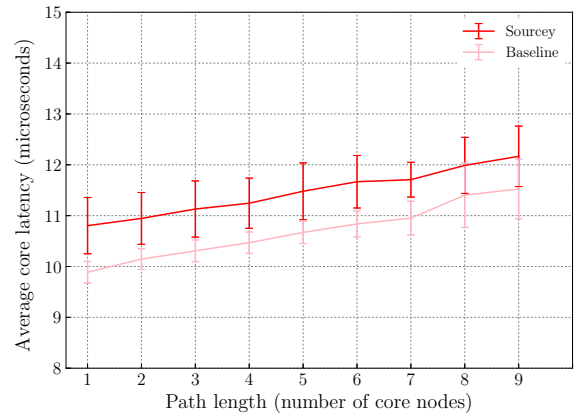
(6.17.a) Low throughput and small packets.



(6.17.b) Low throughput and big packets.



(6.17.c) High throughput and small packets.



(6.17.d) High throughput and big packets.

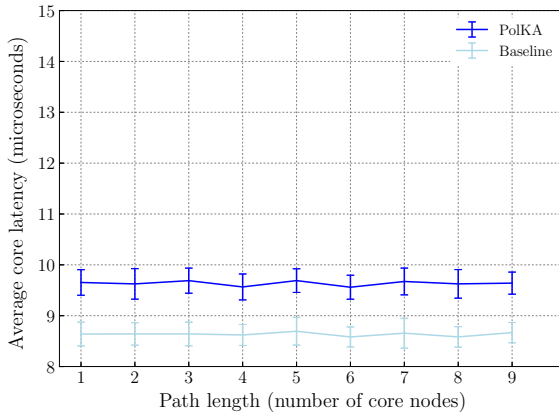
Figure 6.17: Comparison of test cases for Sourcey and Sourcey Baseline scenarios.

Thus, for each of the test scenarios, four test cases were measured:

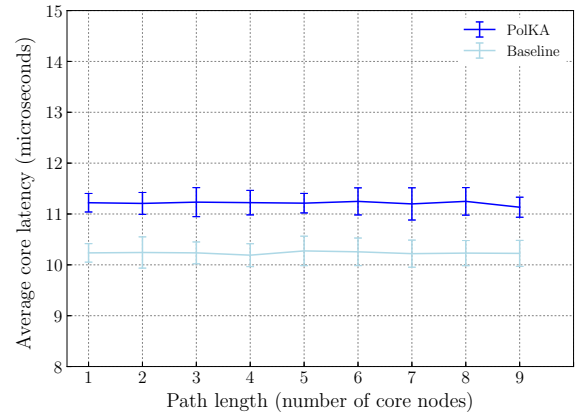
1. Small packet and low throughput.
2. Big packet and low throughput.
3. Small packet and high throughput.
4. Big packet and high throughput.

6.3.2.4 Test results

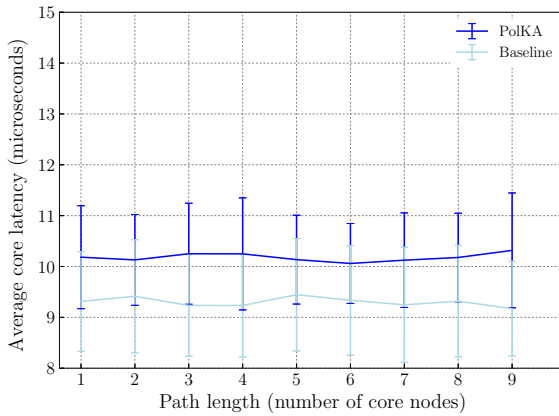
Fig 6.17 and Fig 6.18 show the average core latency and standard deviation results for Sourcey and PolKA test scenarios in comparison with their baselines. In all the four test cases, the results from Sourcey and PolKA scenarios follow the same behavior of the baseline scenarios and add a fixed value to the latency as expected, because of the extra processing to calculate the output port.



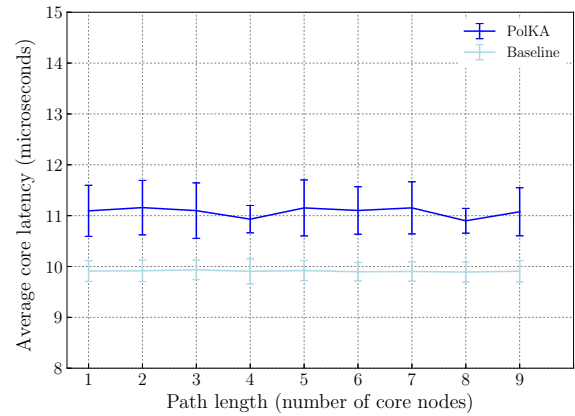
(6.18.a) Low throughput and small packets.



(6.18.b) Low throughput and big packets.



(6.18.c) High throughput and small packets.



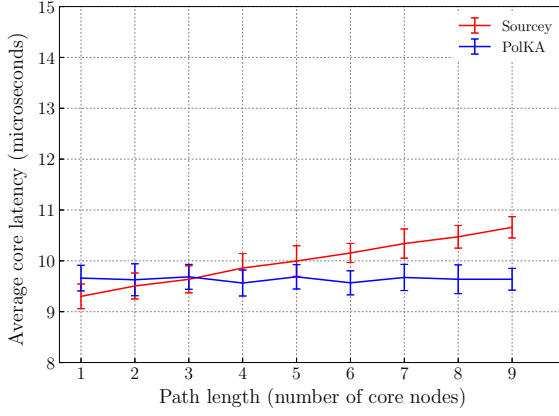
(6.18.d) High throughput and big packets.

Figure 6.18: Comparison of PolKA and PolKA Baseline test cases.

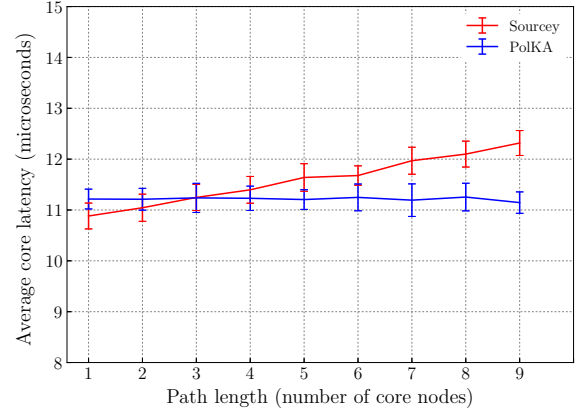
Also, in the test cases with low throughput for Sourcey (Fig 6.17.a and Fig 6.17.b) and PolKA (Fig 6.18.a and Fig 6.18.b), it is possible to observe that the increase in the packet size causes an upward vertical shift in the latency, because of the increased transmission time. Comparing the test cases with big packets for Sourcey (Fig 6.17.b and Fig 6.17.d) and PolKA (Fig 6.18.b and Fig 6.18.d), it is possible to observe that the increase in the throughput has little impact when the packet size is big.

On the other hand, the scenario with small packets and high throughput (Fig 6.17.c for Sourcey and Fig 6.18.a for PolKA) has much higher packet per second (pps) rate, which causes an increase in the average latency and in the related standard deviation.

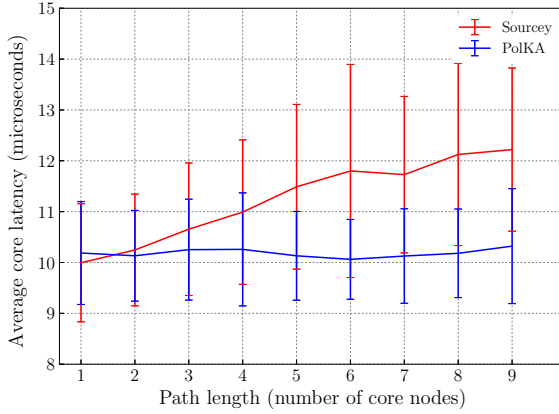
Fig. 6.19 compares the results in the four test cases for Sourcey and PolKA scenarios, without showing the baselines. Within each test case, the average latency and standard deviation in PolKA virtually do not vary when the path length increases, while in Sourcey the average latency grows linear when the path length increases. This linear increase in the latency measurements for Sourcey is emphasized in the test case with high pps (Fig. 6.19.c), when the standard deviation is high for both PolKA and Sourcey. This is



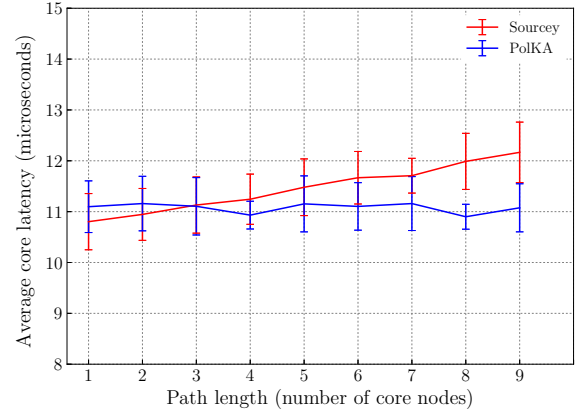
(6.19.a) Low throughput and small packets.



(6.19.b) Low throughput and big packets.



(6.19.c) High throughput and small packets.



(6.19.d) High throughput and big packets.

Figure 6.19: Comparison of Sourcey and PolKA test cases.

due to the stress caused both in the edge and core components to process high pps rates.

However, even in the best case for Sourcey, when the path length is one and the SSR header in Sourcey is much smaller (16 bits in Sourcey against 144 bits in PolKA), the average latency in the PolKA implementation is almost equivalent to the Sourcey implementation. More investigation needs to be carried out in a hardware prototype that allows tests in multi-hop topologies to understand how the contribution in each hop will be combined in end-to-end scenarios, but the results collected so far indicate that the PolKA implementation using CRC hardware is promising and can offer at least equivalent performance to the Sourcey implementation.

6.4 Concluding remarks

Herein, a binary polynomial representation of a RNS-based SSR mechanism, called PolKA, was proposed, implemented and tested for unicast and multicast SSR. To the best of our knowledge, this is the first work to apply the CRT theorem in conjunction with finite

fields polynomials to solve routing problems.

Moreover, our P4-based emulated and hardware prototypes demonstrated it is possible to deploy RNS-based SSR in commercial network equipment by reusing CRC hardware, with performance equivalent to traditional Port Switching approaches. As discussed in Section 2.2.1, this achievement has the potential to enable a new range of complex network applications that explore RNS intrinsic features, such as fast failure reaction, optical switching, and route authenticity. In Chapter 7, we apply the results obtained in this chapter to propose a solution for integrating PolKA with KeySFC, enabling the implementation of SFC with RNS-based SSR in commodity equipment.

Future works include extensions of our hardware prototype for supporting end-to-end test scenarios using Tofino switch and a software implementation of CRC in the SmartNICs. Furthermore, we envision that PolKA is more than a way of supporting RNS-based SSR implementations in commodity network hardware platforms. Chapter 8.2 will discuss how we envision to extend PolKA.

Chapter 7

Integration of PolKA and KeySFC

In Chapter 5, the proposal of KeySFC used an integer RNS-based SR mechanism, which is based on RNS over integer arithmetic. There, the implementation of the prototype was done in OvS. Now, this section explores how PolKA can be integrated as the SR mechanism of KeySFC with a P4 implementation of RNS over GF(2) polynomials.

7.1 Design

The tests in Section 6.3 evaluated the application of PolKA in unicast SR scenarios, where source and destination have no VNF inserted between them. KeySFC enables the steering of a certain flow through a list of VNFs.

The architecture of KeySFC requires a fabric network, composed by core and edge switches, that is already supported by PolKA. In KeySFC, the traffic flows that traverse edge switches match pre-installed flow entries that rewrite Ethernet MAC addresses to a VMAC. The VMAC is divided in two parts (see Fig. 5.2): the most significant 16 bits represent the *segID*, and the remaining bits (32, if using only DstMAC, or 80, if using DstMAC and SrcMAC) represent the *routeID*. The *segID* uniquely identifies the current chain segment and the *routeID* defines the next hop using RNS-based SR.

However, since our prototype in `bmw2` is limited to CRC-16 polynomials, the bit length of the *routeID* does not fit the Ethernet frame. Thus, as a design choice, we decided to embed the *routeID* in a new header between Ethernet and IP headers, as done for unicast SR in Section 6.3, and embed the *segID* in the Ethernet DstMAC as a VMAC, as done in Chapter 5.

The core pipeline (Fig. 6.5.b) used for unicast SR does not need any change, but the edge functionalities (Fig. 6.3.b) need to change to enable traffic steering according to Fig. 7.1. This new edge pipeline for SFC includes the original table for encapsulating SR headers (table *sr_encap*, green diamond in Fig. 7.1) and also a new table for performing SFC (table *sfc_process*, blue diamond in Fig. 7.1), as shown in Code 7.1 for

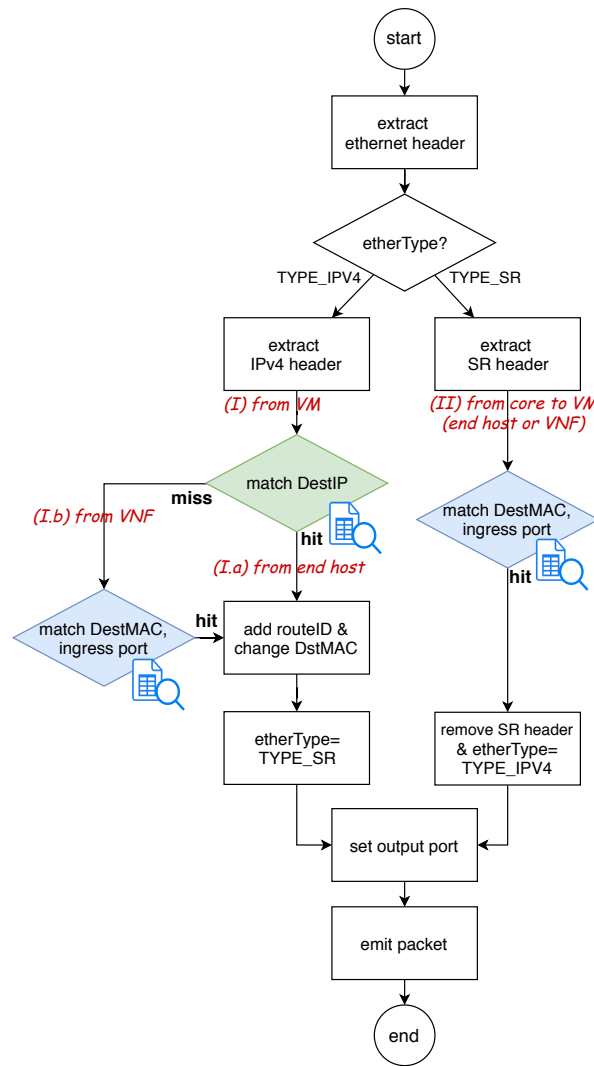


Figure 7.1: SFC edge pipeline.

table declaration and Code 7.2 for action declaration. It considers two cases for a ingress packet:

- (I) It came from a VM: encapsulate SR header and VMAC.
 - (I.a) From end host.
 - (I.b) From VNF.
- (II) It came from core with destination to a VM (VNF or end host): decapsulate SR header.

```

1 table sr_encap {
2   key = {
3     hdr.ipv4.dstAddr: lpm;
4   }
5   actions = {

```

Table 7.1: Table entries in edge switches for example SFC.

Sw.	#	Table	Key	Action	Action Data
E_1	1	sr_encap	10.0.4.4/32	add_sr_header	eport=3, dstAddr=fe:00:00:01:04:04, routeID =2147713608
E_2	2	sfc_process	fe:00:00:01:04:04, iport=3	sfc_to_vm	eport=2
	3	sfc_process	fe:00:00:01:04:04, iport=2	add_sr_header	eport=3, dstAddr=fe:00:00:02:04:04, routeID =715686143
E_3	4	sfc_process	fe:00:00:02:04:04, iport=3	sfc_to_vm	eport=2
	5	sfc_process	fe:00:00:02:04:04, iport=2	add_sr_header	eport=3, dstAddr=00:00:00:00:04:04, routeID =4294771545
E_4	6	sfc_process	00:00:00:00:04:04, iport=3	sfc_to_vm	eport=1

```

6 add_sr_header;
7 tdrop;
8 }
9 size = 1024;
10 default_action = tdrop();
11 }
12
13 table sfc_process {
14 key = {
15   hdr.ethernet.dstAddr : exact;
16   standard_metadata.ingress_port : exact;
17 }
18 actions = {
19   add_sr_header;
20   sfc_to_vm;
21   tdrop;
22 }
23 size = 1024;
24 default_action = tdrop();
25 }

```

Code 7.1: P4 code for table declaration.

```

1 action sfc_to_vm (egressSpec_t port){
2   standard_metadata.egress_spec = port;
3   srcRoute_finish();
4 }
5
6 action add_sr_header (egressSpec_t port, macAddr_t dmac, bit<160> routeID){
7   standard_metadata.egress_spec = port;
8   hdr.ethernet.dstAddr = dmac;
9   hdr.srcRoute.setValid();
10  hdr.srcRoute.routeID = routeID;
11  hdr.ethernet.etherType = TYPE_SR;
12 }

```

Code 7.2: P4 code for action declaration.

Fig 7.2 shows an example SFC with two VNFs: $H_1 \rightarrow VNF_2 \rightarrow VNF_3 \rightarrow H_4$. Initially, the SDN Controller assigns *nodeIDs* to S_1 (0x1002b), S_2 (0x1002d), S_3 (0x10039), and S_4 (0x1003f), and installs the table entries for steering the traffic. For this example,

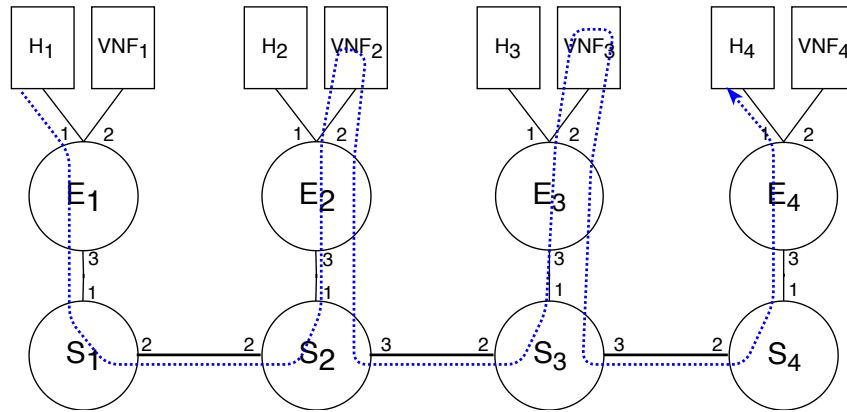


Figure 7.2: KeySFC with PolKA: example SFC.

Table 7.1 shows the table entries in the edge switches. When H_1 sends traffic to H_4 the following events happen:

1. **At E_1 :** The packet leaves H_1 and reaches E_1 , where it matches **entry 1**, receiving a SR header with $routeID1 = 2147713608$, and causing the DstMAC to be changed to $VMAC1$ (fe:00:00:01:04:04), and the packet to be forwarded to S_1 at port 3. In S_1 , the *modulo* operation of $routeID1$ by the *nodeID* of S_1 gives port=2 and the packet is forwarded to S_2 . In S_2 , the *modulo* operation of $routeID1$ by the *nodeID* of S_2 gives port=1 and the packet is forwarded to E_2 .
2. **At E_2 :** the packet matches **entry 2**, which removes the SR header and sends the packet to port 2. Then, VNF_2 processes the packet and forwards it back to the same interface. The returning packet reaches E_2 and matches **entry 3**, receiving a SR header with $routeID2 = 715686143$, and causing the DstMAC to be changed to $VMAC2$ (fe:00:00:02:04:04), with the packet being sent to S_2 at port 3. There, computing the *modulo* of $routeID2$ by the *nodeID* of S_2 , gives port=3 and the packet is forwarded to S_3 . In S_3 , the *modulo* operation of $routeID2$ by the *nodeID* of S_3 gives port=1 and the packet is forwarded to E_3 .
3. **At E_3 :** the packet matches **entry 4**, which removes the SR header and sends the packet to port 2. VNF_3 processes the packet and forwards it back to the same interface. The returning packet reaches E_3 and matches **entry 5**, receiving a SR header with $routeID3 = 4294771545$, and causing the DstMAC to be changed to the original MAC of the destination H_4 (00:00:00:00:04:04), with the packet being sent to S_3 at port 3. There, computing the *modulo* of $routeID3$ by the *nodeID* of S_3 gives port=3 and the packet is forwarded to S_4 . In S_4 , the *modulo* operation of $routeID3$ by the *nodeID* of S_4 gives port=1 and the packet is forwarded to E_4 .
4. **At E_4 :** the packet matches **entry 6**, which removes the SR header and delivers the packet to H_4 .

Equivalent modifications in the pipeline and tables were implemented for Sourcery to enable the comparison between the solutions.

7.2 Proof-of-concept and evaluation

We extended the emulated setup of Section 6.3.1 to test the integration of PolKA with KeySFC according to the design of the previous section.

The first set of experiments performs SFC tests on a linear topology, similarly to the tests of Section 6.3.1.6. Then, the second experiment explores a SFC migration scenario. Moreover, KeySFC is a powerful SFC scheme that can be extended to support features that explore RNS properties. So, the third experiment exemplifies a potential feature of fast failure reaction.

7.2.1 Linear topology

In this experiment, the test topology is similar to the linear fabric topology, but each edge switch is connected to one host and one VNF. The experiments are the same described in Section 6.3.1.5.

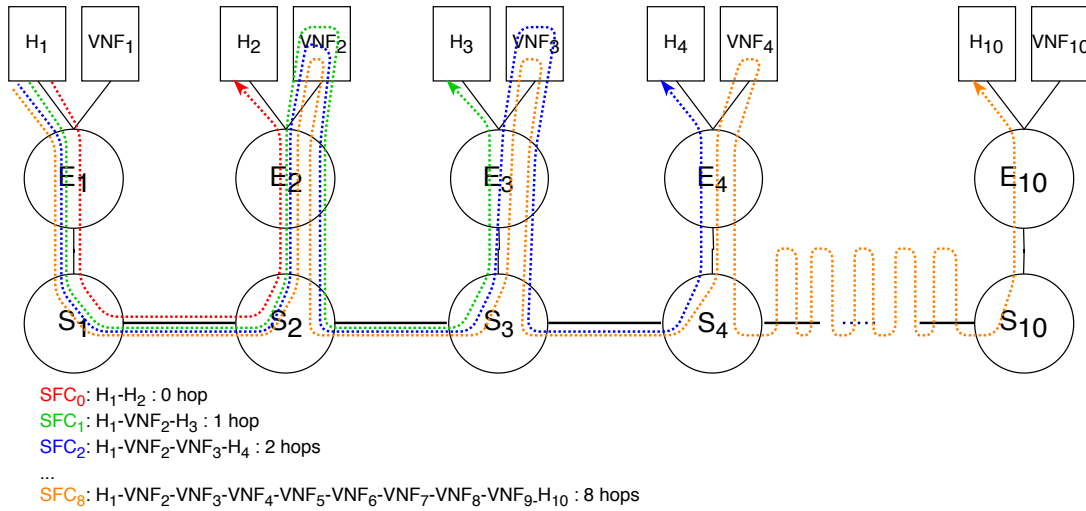
Before the experiments start, the SDN controller installs the table entries for all the chains and configures the *nodeIDs* of core switches. Similar to the KeySFC prototype of Chapter 5 the VNFs run a simple forwarding function. To implement this function, a Python script with raw sockets receives packets from one network interface and return then to the same interface without further processing or modification.

The first objective is to demonstrate that PolKA and Sourcery solutions can be used by KeySFC to enable traffic steering. The second objective is to evaluate PolKA and Sourcery solutions in two scenarios: (i) increasing the number of VNFs in the chain (chain length) (Fig. 7.3.a), and (ii) increasing number of hops per SFC segment (Fig. 7.4.a).

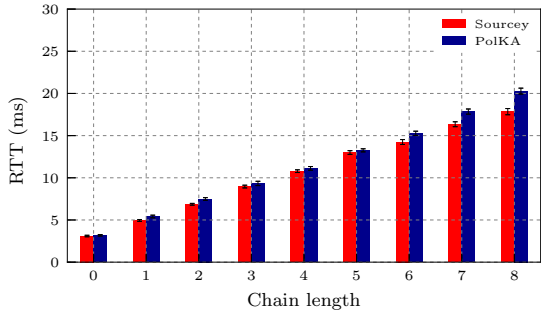
Fig. 7.3 shows the comparison between PolKA and Sourcery solutions for the RTT (Fig. 7.3.b), Jitter (Fig. 7.3.c), and FCT (Fig. 7.3.d) experiments in the example of Fig. 7.3.a with increasing chain length (e.g., from 0 for SFC $H_1 \rightarrow H_2$ to 8 for SFC $H_1 \rightarrow VNF_2 \rightarrow VNF_3 \rightarrow VNF_4 \rightarrow VNF_5 \rightarrow VNF_6 \rightarrow VNF_7 \rightarrow VNF_8 \rightarrow VNF_9 \rightarrow H_{10}$).

Fig. 7.4 shows the comparison between PolKA and Sourcery solutions for the RTT (Fig. 7.4.b), Jitter (Fig. 7.4.c), and FCT (Fig. 7.4.d) experiments in the example of Fig. 7.4.a with increasing number of hops per SFC segment (e.g., 1 for SFC $H_1 \rightarrow VNF_2 \rightarrow H_3$, 2 for SFC $H_1 \rightarrow VNF_3 \rightarrow H_5$, 3 for SFC $H_1 \rightarrow VNF_4 \rightarrow H_7$, and 4 for SFC $H_1 \rightarrow VNF_5 \rightarrow H_9$).

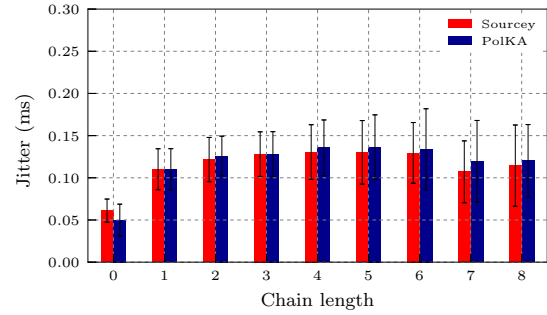
For both test scenarios, it is possible to observe that some behaviors are repeated: (i) the latency grows linearly with the increase of the chain length and the number of hops per segment for Sourcery and PolKA solutions, (ii) Sourcery solution has a slightly better



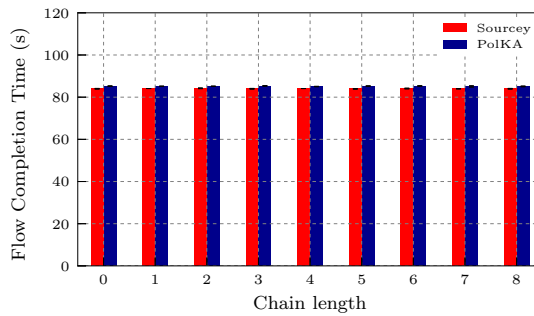
(7.3.a) SFC description.



(7.3.b) RTT.



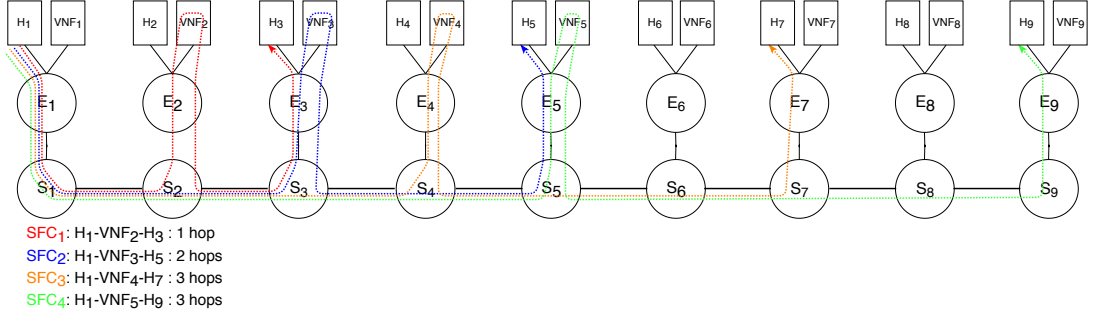
(7.3.c) Jitter.



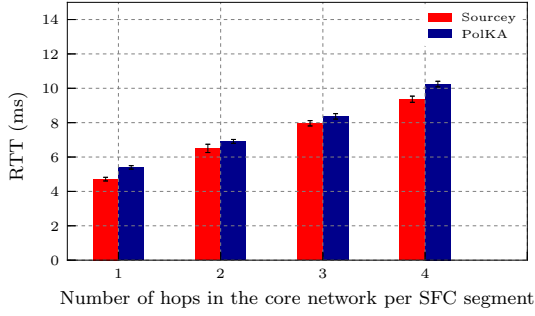
(7.3.d) FCT.

Figure 7.3: SFC tests for increasing chain length: comparison between Sourcey and PolKA solutions.

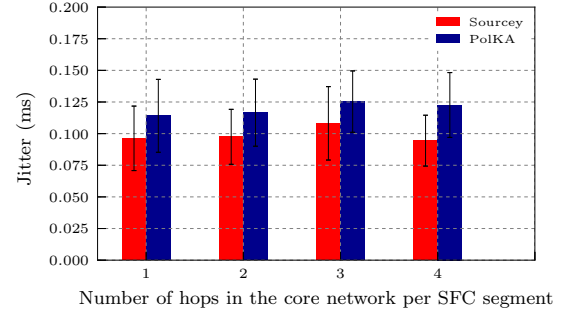
RTT latency than PolKA solution, (iii) the standard deviation for RTT is small and in the same order of magnitude for PolKA and Sourcey solutions, (iv) the jitter is small and equivalent for both solutions, and (v) the FCT experiments show that all the solutions require approximately the same time to transfer the file and the standard deviation was small. As discussed in Section 6.3.1.6, the performance of the PolKA solution can be improved if the CRC operation is performed in hardware.



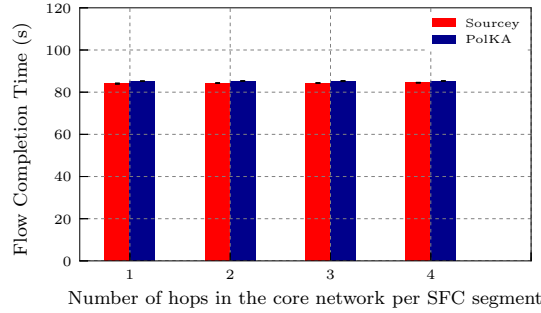
(7.4.a) SFC description.



(7.4.b) RTT.



(7.4.c) Jitter.



(7.4.d) FCT.

Figure 7.4: SFC tests for increasing number of hops per SFC segment: comparison between Sourcey and PolKA solutions.

7.2.2 Programmable, expressive, scalable, and agile migration

The objective of this experiment is to show a practical example that explores how the combination of KeySFC and PolKA can provide the ability to migrate paths of the SFC segments in a programmable, expressive, scalable, and agile way.

To this end, we explore the example discussed in Fig. 3.6 of Section 3.1.3, which illustrates an example scenario of dynamic chain migration due to security demands. In the example, the SDN-based approach needs to install flow entries in all the elements in the forwarding path, while our algorithmic SSR approach only installs flow entries in the SFC segments' endpoints.

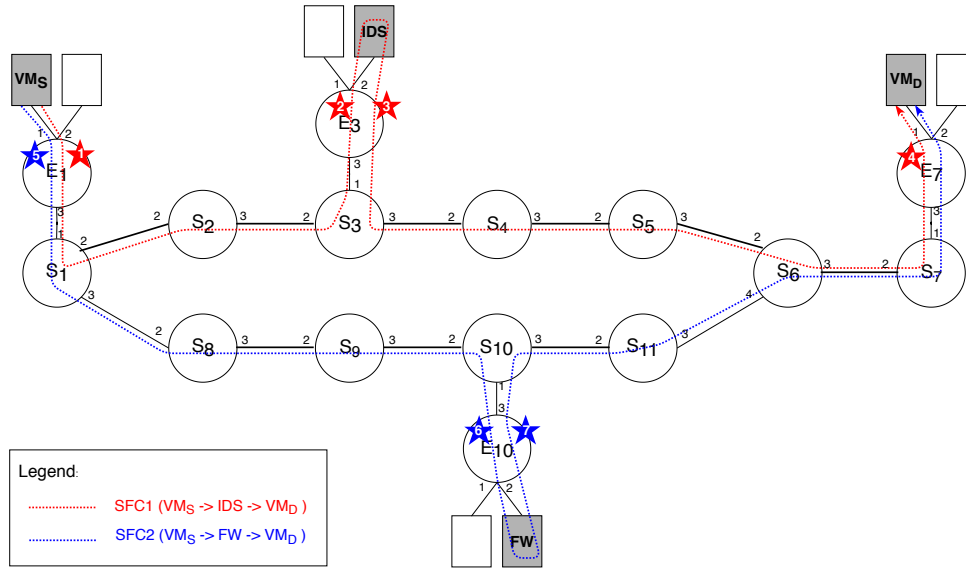


Figure 7.5: KeySFC with PolKA: SFC migration example.

Table 7.2: Table entries in edge switches before SFC migration.

Sw.	#	Table	Key	Action	Action Data
E_1	1	sr_encap	10.0.7.7/32	add_sr_header	eport=3, dstAddr=fe:00:00:01:07:07, routeID = 103941321831683
E_3	2	sfc_process	fe:00:00:01:07:07, iport=3	sfc_to_vm	eport=2
	3	sfc_process	fe:00:00:01:07:07, iport=2	add_sr_header	eport=3, dstAddr=00:00:00:00:07:07, routeID = 1187790491324897019619195
E_7	4	sfc_process	00:00:00:00:07:07, iport=3	sfc_to_vm	eport=1

Table 7.3: Table entries in edge switches after SFC migration.

Sw.	#	Table	Key	Action	Action Data
E_1	5	sr_encap	10.0.7.7/32	add_sr_header	eport=3, dstAddr=fe:00:00:01:07:07, routeID = 10417762529131015975
E_{10}	6	sfc_process	fe:00:00:01:07:07, iport=3	sfc_to_vm	eport=2
	7	sfc_process	fe:00:00:01:07:07, iport=2	add_sr_header	eport=3, dstAddr=00:00:00:00:07:07, routeID = 18289869304896468450
E_7	4	sfc_process	00:00:00:00:07:07, iport=3	sfc_to_vm	eport=1

Fig. 7.5 shows the equivalent topology of the example of Fig. 3.6, which is emulated in our bmv2 prototype. Initially, the flow is steered through an IDS function (SFC1 in red). The paths are represented by the set of servers and their respective output ports, which are used to calculate the *routeIDs*. For the first SFC segment, the selected physical path in the core network is represented by nodes $S = \{S_1, S_2, S_3\}$ and ports $O = \{2, 3, 1\}$. For the second SFC segment, it is represented by nodes $S = \{S_3, S_4, S_5, S_6, S_7\}$ and ports $O = \{3, 3, 3, 3, 1\}$.

Then, when an attack is detected, the flow needs to be redirected to a firewall function (SFC2 in blue). For the first SFC segment, the selected physical path in the core network is represented by $S = \{S_1, S_8, S_9, S_{10}\}$ and $O = \{3, 3, 3, 1\}$. For the second SFC segment, it is represented by $S = \{S_{10}, S_{11}, S_6, S_7\}$ and $O = \{3, 3, 3, 1\}$.

In Fig. 7.5, the red stars represent the flow entries installed for SFC1. The blue stars represent the new flow entries that are necessary to be installed for the migration to SFC2. Table 7.2 and Table 7.3 show the table entries in the edge switches before and after the SFC migration, respectively. The *routeIDs* in the flow entries 1, 3, 5, and 7 specify the physical paths of the SFC segments.

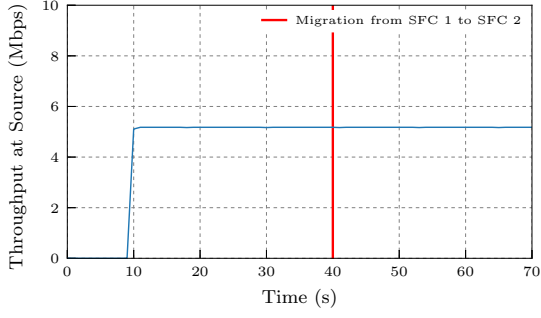
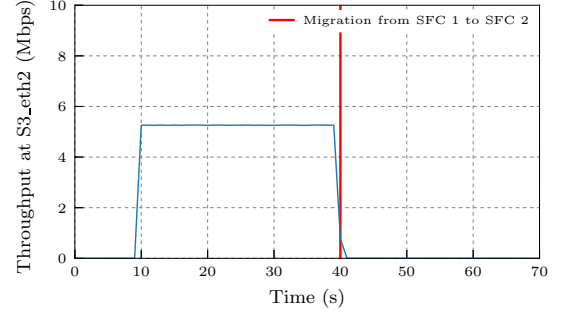
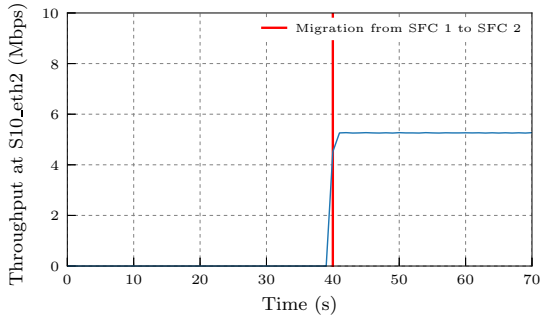
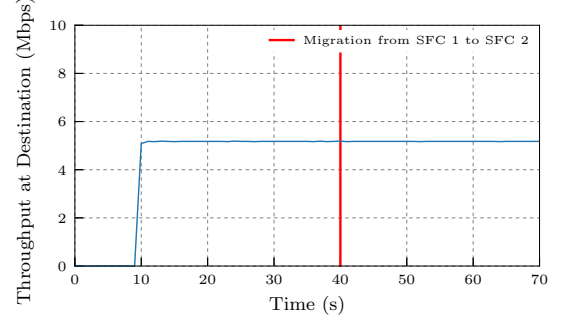
(7.6.a) Throughput at source (VM_S).(7.6.b) Throughput at S_3 .(7.6.c) Throughput at S_{10} .(7.6.d) Throughput at destination (VM_D).

Figure 7.6: Migration of UDP flow from SFC 1 to SFC 2 in KeySFC with PolKA.

At instant 10s, we start a 5Mbps UDP flow from VM_S to VM_D , with no concurrent traffic. Initially, this UDP flow is allocated to SFC1. Then, at instant 40s, we migrate this flow to SFC2, changing one single flow entry at edge switch E_1 , and creating two new flow entries at edge switch E_{10} . The two flow entries at edge switch E_3 are deleted and the flow entry at edge switch E_7 is maintained.

Fig. 7.6 shows the throughput measurements of this experiment using **bwm-ng** tool at source (Fig. 7.6.a), core switch S_3 (Fig. 7.6.b), core switch S_{10} (Fig. 7.6.c), and destination (Fig. 7.6.d). The graphs show the traffic correctly pass through S_3 when SFC1 is selected (10s to 40s) and through S_{10} when SFC2 is selected (40s to 70s).

7.2.3 Exploitation of RNS properties: fast failure reaction

The goal of this experiment is to show an example of how KeySFC with PolKA can take benefit of the special properties offered by RNS-based SR, as discussed in Section 3.2.1. More specifically, it explores a property that states that the order of the nodes in the path is irrelevant. Based on this property, it integrates a fast failure reaction mechanism proposed by KAR [Gomes et al. 2016] to our SFC scheme.

KAR [Gomes et al. 2016] proposes the concept of resilient forwarding paths, called protection paths. The main idea is to proactively add redundant nodes in the *routeID*

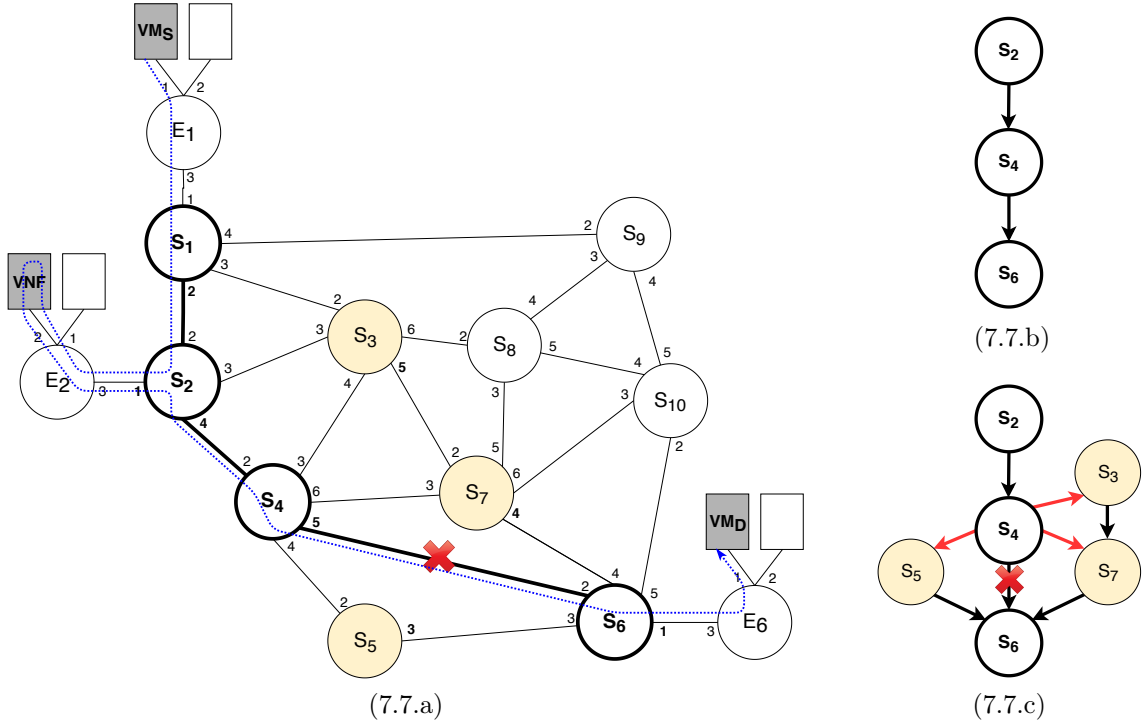


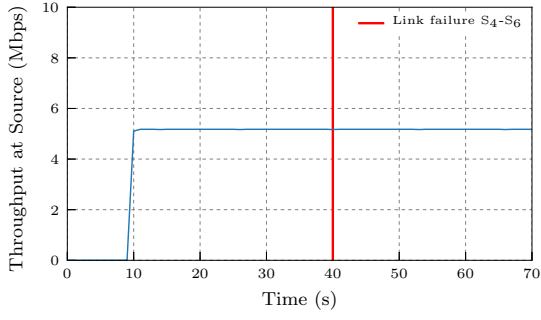
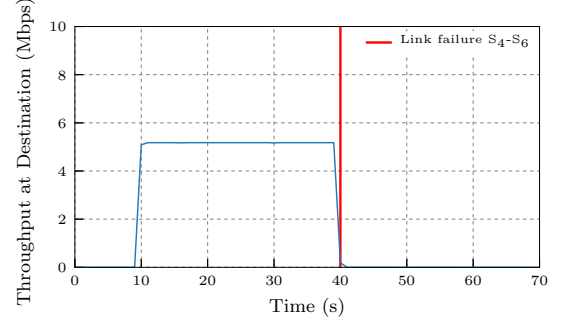
Figure 7.7: Fast failure reaction for SFC $VM_S \rightarrow VNF \rightarrow VM_D$: (a) topology, (b) unprotected path for segment $VNF \rightarrow VM_D$, and (c) protected path for segment $VNF \rightarrow VM_D$.

that are not part of the original route. When there is a link failure, packets are deviated from faulty links with routing deflections and may occasionally reach these redundant nodes, which are responsible to guide the packets back to their original route. In this way, there is no need to communicate with a Controller or the source node to compute an alternative path, because as soon as the forwarding node detects a failure it randomly deflects the packet to one of its other healthy links.

Fig. 7.7.a shows an example scenario for SFC $VM_S \rightarrow VNF \rightarrow VM_D$. The path in the core switches for the first SFC segment ($VM_S \rightarrow VNF$) is represented by nodes $S = \{S_1, S_2\}$ and ports $O = \{2, 1\}$. The path for the second SFC segment ($VNF \rightarrow VM_D$) is represented by nodes $S = \{S_2, S_4, S_6\}$ and ports $O = \{4, 5, 1\}$, as shown in Fig. 7.7.b. These paths are called unprotected paths, because they do not add any redundant node for failure protection. Therefore, if any link of these paths fails, the packets will be dropped.

In the first part of this experiment, we emulate the test scenario of Fig. 7.7.a without any protection and generate a failure in the link S_4-S_6 at instant 40s. Fig. 7.8 shows the throughput measurements of this experiment using **bwm-ng** tool at source (Fig. 7.8.a), and destination (Fig. 7.8.b). From 40s to 70s, the link S_4-S_6 is down and no traffic reaches the destination.

In the second part of the experiment, we apply the protection mechanism based on KAR for fast failure reaction. To generate the *routeID* of the second SFC segment ($VNF \rightarrow VM_D$), we add the extra nodes S_3, S_5 , and S_7 with ports 5, 3, and 4, respectively. In this way, when the link S_4-S_6 fails, S_4 can deflect the packets to any of its other links

(7.8.a) Throughput at source (VM_S).

(7.8.b) Throughput at destination.

Figure 7.8: UDP test for unprotected path: results for failure of link S_4 - S_6 .

and the packets will be driven back to S_6 , as shown in Fig. 7.7.c. Thus, the protected path for the second segment is represented by nodes $S = \{S_2, S_3, S_4, S_5, S_6, S_7\}$ and ports $O = \{4, 5, 5, 3, 1, 4\}$. Note that the protected path already contains the redundant nodes, so no change is necessary in the *routeID* when the failure happens.

The KAR mechanism proposed in [Gomes et al. 2016] implements three fundamental features: RNS-based SSR, failure detection, and deflection. It uses an emulated OpenFlow prototype in Mininet to implement these features. However, the `v1model` architecture with `bmv2 simple_switch` target does not offer a standard way to access port status information directly in the P4 program¹, which is necessary for failure detection.

To enable the integration of KAR’s fast failure reaction mechanism in our SFC prototype, we developed a simple control plane application that causes link failures and makes port failure information available to the dataplane by populating a table of faulty ports in affected switches. In addition, we modified the program of the core nodes to check this table before transmitting the packet to the output port that was obtained via the *modulo* operation. If there is a hit, the packet is randomly deflected to one of the other healthy ports. Otherwise, the packet is transmitted normally. The generation of a random value within an interval is provided by `v1model` and could also be replaced by a hash function if the objective is to always select the same port per flow. More sophisticated mechanisms for failure detection and other P4 targets can be explored in future works [Cascone et al. 2017], as this is not the scope of this thesis.

Fig. 7.9 shows the throughput measurements of this second part of the experiment using `bwm-ng` tool at source (Fig. 7.9.a), destination (Fig. 7.9.b), core switch S_3 (Fig. 7.9.c), core switch S_5 (Fig. 7.9.d), and core switch S_7 (Fig. 7.9.e). At instant 10s, we start a 5Mbps UDP traffic from VM_S to VM_D . At instant 40s, the link S_4 - S_6 is disconnected and maintained down until the end of the experiment. The graphs show the traffic is uniformly deflected through S_3 , S_5 , and S_7 after the failure. At the destination, the

¹http://lists.p4.org/pipermail/p4-dev_lists.p4.org/2016-May/000290.html

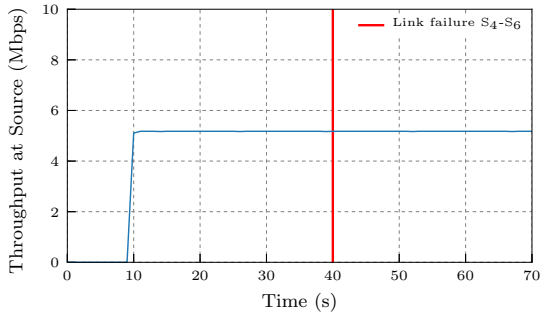
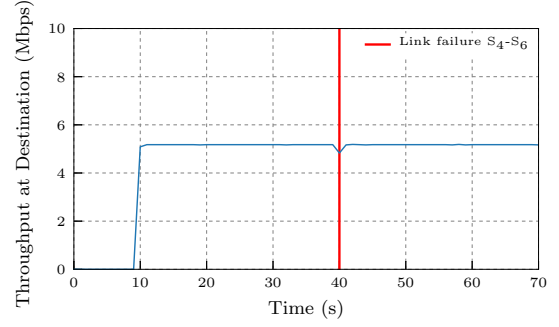
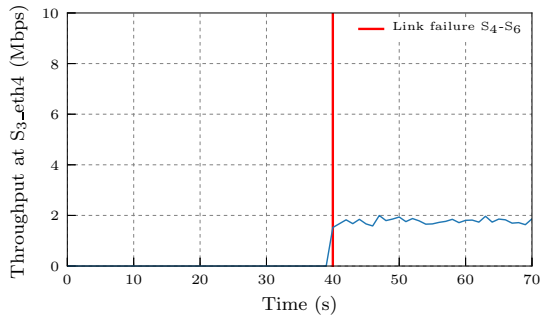
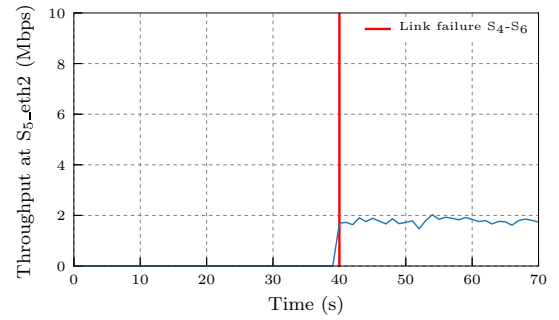
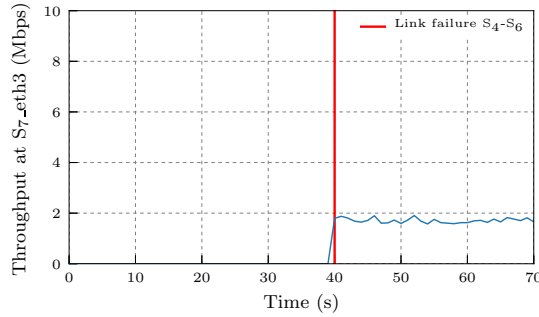
(7.9.a) Throughput at source (VM_S).(7.9.b) Throughput at destination (VM_D).(7.9.c) Throughput at S_3 (Port 4).(7.9.d) Throughput at S_5 (Port 2).(7.9.e) Throughput at S_7 (Port 3).

Figure 7.9: UDP test for protected path: results for failure of link S_4 - S_6 with the deflections of the fast failure reaction mechanism.

received traffic perceives a small loss until the failure is signalized by the control plane and deflections start. Therefore, our scheme was able to react to the failure and deliver packets to destination without any modification of the packets.

Then, we repeated the same test for a TCP traffic. Fig. 7.10 shows the throughput measurements of this experiment using **bwm-ng** tool at source (Fig. 7.10.a), destination (Fig. 7.10.b), core switch S_3 (Fig. 7.10.c), core switch S_5 (Fig. 7.10.d), and core switch S_7 (Fig. 7.10.e). At instant 10s, we start a TCP traffic from VM_S to VM_D . At instant 40s, the link S_4 - S_6 is disconnected and maintained down until the end of the experiment. The graphs show the traffic is uniformly deflected through S_3 , S_5 , and S_7 after the failure. At

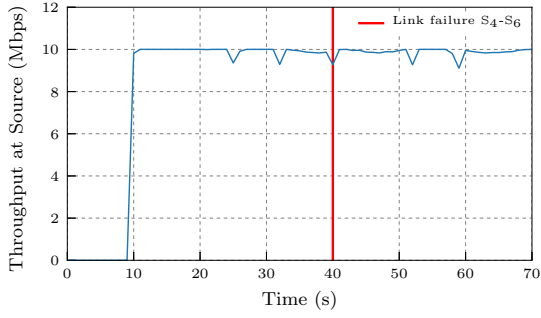
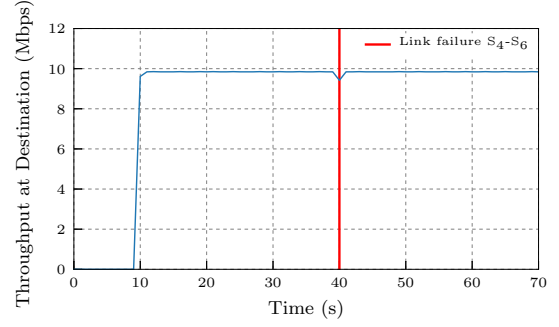
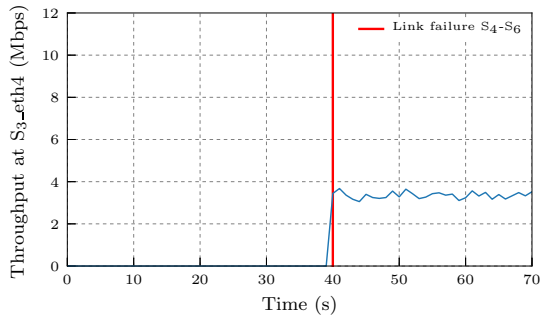
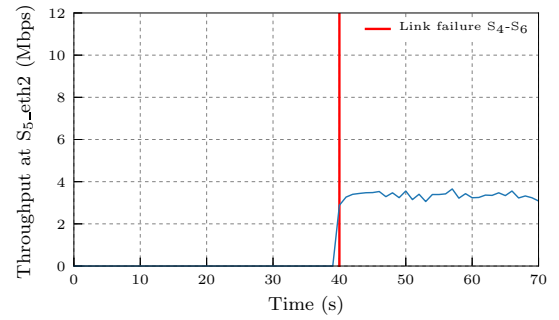
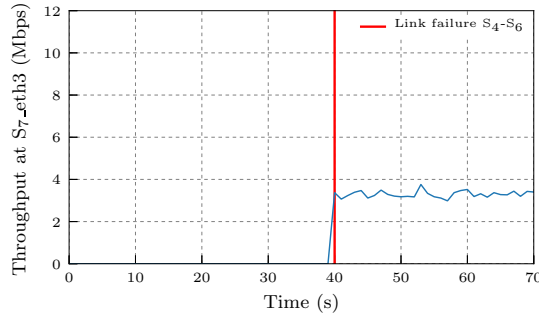
(7.10.a) Throughput at source (VM_S).(7.10.b) Throughput at destination (VM_D).(7.10.c) Throughput at S_3 (Port 4).(7.10.d) Throughput at S_5 (Port 2).(7.10.e) Throughput at S_7 (Port 3).

Figure 7.10: TCP test for protected path: results for failure of link S_4 - S_6 with the deflections of the fast failure reaction mechanism.

the destination, the received traffic perceives a small loss until the failure is signaled by the control plane and deflections start.

No effect of TCP packet reordering was observed for this particular case. It is important to outline that, depending on the scenario and the difference on the number of hops of the alternative paths, the deflections may impact the TCP throughput due to packet disordering. However, even in these cases, the ability to keep the network connectivity during failures without any reconfiguration on the network nodes justifies the use of KAR to enable network resiliency. Furthermore, as soon as the link connection is reestablished, the deflections stop and the original path is used.

7.3 Concluding remarks

This chapter presented the integration between PolKA and KeySFC to provide a programmable, expressive, scalable, and agile SFC solution that can be implemented over COTS equipment. As PolKA can be implemented in P4 programmable switches, it can be used as the SSR mechanism of KeySFC scheme to improve performance and make hardware implementation more feasible. Furthermore, we demonstrated KeySFC can be extended to exploit RNS features, such as fast failure reaction. The next chapter will summarize the contributions of this thesis and point out future works.

Chapter 8

Conclusion

This chapter describes the final considerations and highlights the contributions and future works of this thesis.

8.1 Conclusions

Emerging trends such as Internet of Things, Smart Cities, and Industry 4.0 require the dynamic composition of end-to-end services in edge DCs that can meet stringent performance, cost, and flexibility requirements. In this context, NFV challenges network operators to compose services by steering dynamic flows across a set of SFs.

However, traditional SFC solutions lack flexibility and efficiency due to the fact that they are adaptations of the classical source-to-destination routing paradigm. As a result, one may not be able to use all the existing paths for chaining specific segments, either because the SFC decisions are completely decoupled from the routing decisions, or because the path choice is constrained by the routing mechanism to a small set of shortest paths. In addition, dynamic modification of chain paths may impact a large set of nodes, leading to large operational complexities even for simple service modification tasks.

To address these issues, this thesis proposed a framework composed by three interrelated solutions (VirtPhy, KeySFC, and PolKA) for providing programmable, expressive, scalable, and agile SFC solution that enables dynamic and efficient orchestration of the network underlay of edge data centers with COTS equipment. The proposal builds upon the following enablers: NFV, SDN, SSR, algorithmic forwarding, RNS, support for any DCN, topology-aware orchestration, and fabric networks.

Chapter 4 presented VirtPhy, a NFV orchestration architecture based on server-centric topologies. The main contributions of VirtPhy are: to demonstrate how server-centric DCNs using COTS equipment can fit in NFV ETSI orchestration architecture; to propose topology-aware orchestration and the exploitation of algorithmic routing mechanisms provided by the network underlay; and to demonstrate how table lookup can be replaced by algorithmic forwarding in a efficient and programmable manner using SDN.

Chapter 5 presented KeySFC, a topology-independent SFC scheme that uses RNS-based SSR. It advances the contributions of VirtPhy, and proposes a SFC scheme that can be used by any topology (server-centric, network-centric, or hybrid). Moreover, it reduces network overhead by applying the concepts of fabric networks. To this end, it defines edge and core elements that provide SDN-based SFC and algorithmic SSR functionalities, respectively. Furthermore, it provides a more expressive scheme to allow the traffic engineering to specify any path for SFC segments and perform agile migration between paths.

To test VirtPhy and KeySFC proposals we implemented and tested a proof-of-concept prototype in a DC testbed that uses OpenStack, OpenFlow, and OvS. The results showed our framework has the potential to meet performance, cost, and flexibility requirements in production DC environments.

Chapter 6 presented PolKA, a novel RNS-based SSR mechanism based on binary arithmetic. Its novelty lies in applying finite fields arithmetic and RNS for solving routing problems. This new representation allowed the development of a technique that allows computationally efficient implementations of the *modulo* operation by reusing CRC hardware. The proposal was implemented in P4 emulated and hardware prototypes that demonstrated it can achieve similar performance to traditional Port Switching methods.

Chapter 7 demonstrated that, by integrating PolKA with KeySFC, we can provide a programmable, expressive, scalable, and agile SFC solution, which can be implemented over COTS equipment. In addition, our integrated SFC solution tested all the hypothesis stated in Section 1.1.5 (SSR, algorithmic forwarding, RNS, support to any DCN, SDN, and fabric networks) to tackle the research question of Section 1.1.4.

Using our expressive SFC solution, the traffic engineering can specify any path between two endpoints of a SFC segment, and agilely migrate between paths using SDN in a programmable manner. Besides our solution only needs to install flow rules in the SFC segments' endpoints, significantly improving scalability and agility when compared to traditional table-based approaches. Thus, we provide a SFC solution that enables dynamic and efficient orchestration of the network underlay of edge data centers and does not restrict the traffic engineering on the selection of paths for the SFC segments.

This work also paves the way for exploring RNS-based SSR properties in SFC schemes (as described in Section 3.2.1). In particular, we demonstrated an example of the integration of a fast failure reaction mechanism to the KeySFC scheme.

Finally, it is important to highlight that we implemented prototypes of all the proposed solutions. Considering the four years time frame of this thesis, we followed the evolution of cutting edge technologies for edge DCs during this period.

Section 8.2 details the next steps of this research.

8.2 Future works

As described in Section 5.3, KeySFC scheme has the potential to offer an efficient and modular solution to the SFC problem in multi-provider and multi-tenant scenarios [Dietrich et al. 2017]. Besides, the integration of MANO solutions and optimized resource allocation strategies from related works [Mijumbi et al. 2016], [Herrera and Botero 2016], [Bari et al. 2015], [Luizelli et al. 2015] is an important extension of this work.

Offload of forwarding tasks and SFs to a networking co-processor, such as SmartNICs, is also planned as future work. Performance tests in large DCN topologies with realistic SFs are also in our roadmap. As carrier-grade services can benefit from our dynamic mechanisms, the integration of KeySFC with operation and management (OAM) systems will be addressed. Besides, we want to explore the use of metadata and network context for policy enforcement strategies.

In addition, our solution has the advantage of enabling the traffic engineering to specify the complete forwarding paths in the network underlay. This can be explored by load balancing mechanisms, which need to be able to classify the existing flows, isolate flows with conflicting network requirements, and assign paths in a way that meets the flow requirements without compromising other flows. As SSR solutions reduce the number of flow entries installed in the network core it can offer scalable solutions to traffic engineering. We started to explore this approach for unicast routing [Valentim et al. 2019], and plan to extend these studies for SFC scenarios.

Future works include two extensions of our hardware prototype of PolKA for supporting end-to-end test scenarios: (i) with Tofino switch, since it supports the customization of CRC polynomials; and (ii) with SmartNICs by implementing CRC operations with customized polynomials in software using a custom C plugin with a CRC lookup table in local memory, which would cost few instructions per data byte. Besides, we plan to exploit the use of alternative commodity modules, such as forward error correction (FEC), DDR-DRAM, random number generators, and cryptographic hardware, whose arithmetic is also based on GFs.

Furthermore, by bringing routing expressiveness closer to hardware operations, PolKA opens the possibility to explore binary polynomial arithmetic in hardware description languages. Therefore, RNS-based SR elements have the potential to be synthesized in a smaller chip area with reduced power consumption. This also creates an opportunity to replace the complex routing mechanisms that are commonly used by network on chip (NoC) designs in many-cores systems [Ruaro et al. 2018] by PolKA, reducing costs and gaining flexibility.

Moreover, we also envision the expressiveness of our polynomial mechanism may be extended using GFs of higher orders for complex routing problems, like wavelength switching, multilayer networks, and network slicing. In particular, we plan to support chains

with branches with the exploration of multicast forwarding trees in PolKA, according to the mechanism proposed in Section 6.1.3.

Finally, this work can be extended with the exploitation of RNS properties to support security and resilience use cases, as discussed in Section 3.2.1. An interesting application of these properties is route authenticity: as the *routeID* remains the same throughout the path, the source can sign the *routeID* information and embed this signature in the packet header; then, each node can verify this signature before using the *routeID* to forward packets.

8.3 Publications

Contributions directly related to the proposal are listed in reverse chronological order through the following publications:

1. **DOMINICINI, C. K.** ; MARTINELLO, MAGNOS ; RIBEIRO, M. R. N. ; VASSOLER, G. L. ; VILLACA, R. S. ; VALENTIM, R. ; ZAMBON, E. . KeySFC: Agile Traffic Steering using Strict Source Routing for Enabling Efficient Traffic Engineering. **Submitted to Elsevier Computer Networks Journal (in major review phase).**
2. **DOMINICINI, C. K.** ; MARTINELLO, MAGNOS ; RIBEIRO, M. R. N. ; VASSOLER, G. L. ; VILLACA, R. S. ; VALENTIM, R. ; ZAMBON, E. . KeySFC: Agile Traffic Steering using Strict Source Routing. In: Symposium on SDN Research (SOSR), 2019, São Diego. Symposium on SDN Research (SOSR), 2019.
3. VALENTIM, R. ; **DOMINICINI, C. K.** ; VILLACA, R. ; RIBEIRO, M. R. N. ; MARTINELLO, MAGNOS ; MAFIOLETTI, D. . RDNA Balance: Balanceamento de Carga por Isolamento de Fluxos Elefante em Data Centers com Roteamento na Origem. In: SBRC, 2019, Gramado. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2019.
4. CASTANHO, M. S. ; VIEIRA, MARCOS A. M. ; **DOMINICINI, C. K.** Cadeia-Aberta: Arquitetura para SFC em Kernel Usando eBPF. In: XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2019, Gramado. XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), 2019.
5. CASTANHO, M. ; **DOMINICINI, C. K.** ; VILLACA, R. S. ; MARTINELLO, M. ; RIBEIRO, M. R. N. . PhantomSFC: A Fully Virtualized and Agnostic Service Function Chaining Architecture. In: IEEE Symposium on Computers and Communications, 2018, Natal. IEEE ISCC, 2018.

6. **DOMINICINI, C. K.** ; VASSOLER, G. L. ; MENESES, L. F. ; VILLACA, R. S. ; RIBEIRO, M. R. N. ; MARTINELLO, M. . VirtPhy: Fully Programmable NFV Orchestration Architecture for Edge Data Centers. *IEEE Transactions on Network and Service Management*, v. 14, p. 1-1, 2017.
7. **DOMINICINI, C. K.**; VASSOLER, G. L.; RIBEIRO, M. R.; MARTINELLO, M.. VirtPhy: A Fully Programmable Infrastructure for Efficient NFV in Small Data Centers. *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, November 2016.
8. MAFIOLETTI, D. R., LIBERATTO, A. B., **DOMINICINI, C. K.**, VILLACA, R. S., MARTINELLO, M., RIBEIRO, M. R.. Latency Measurement as a Virtualized Network Function using Metherxis. *ACM SIGCOMM Computer Communication Review*, 46(4), 2016.
9. GOMES, R. R. ; **DOMINICINI, C. K.** ; LIBERATO, A. ; MARTINELLO, M.; RIBEIRO, M. R. N. . Analytical Modeling Approach of Routing Deflection for Intra-domain Networks. In: *WPerformance - Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, 2016, Porto Alegre-RS. *Anais do CSBC*, 2016.
10. MAFIOLETTI, D. ; LIBERATO, A. ; VILLACA, R. ; **DOMINICINI, C. K.** ; RIBEIRO, M. R. N. ; MARTINELLO, M. . Metherxis: Virtualized Network Functions for Micro-second Grade Latency Measurements. In: *ACM SIGCOMM Workshop on Fostering Latin-American Research in Data Communication Networks (LANCOMM 2016)*, 2016, Florianópolis-SC. *Anais do LANCOMM*, 2016.
11. GOMES, R. R. ; LIBERATO, A. ; **DOMINICINI, C. K.** ; RIBEIRO, M. R. N. ; MARTINELLO, M. . KAR: Key-for-Any-Route Resilient Routing System. In: *Workshop on Dependability Issues on SDN and NFV (DISN)*, 2016, Toulouse. *Workshop on Dependability Issues on SDN and NFV (DISN)*, 2016.

The following publications (listed in reverse chronological order) were developed during the PhD, and are not directly linked to this proposal. However, they helped to gain important knowledge in the matter and, therefore, should also be considered relevant contributions.

1. BOTH, CRISTIANO et al. FUTEBOL Control Framework: Enabling Experimentation in Convergent Optical, Wireless, and Cloud Infrastructures. **Accepted at IEEE Communications Magazine (to appear)**.
2. DO CARMO, ALEXANDRE P. ; VASSALO, R. ; DE QUEIROZ, FELIPPE M. ; PICORETTI, R. ; FERNANDES, M. R. ; GUIMARAES, R. S. ; **DOMINICINI, C. K.** ; MARTINELLO, M. ; M. R. N. Ribeiro ; GARCIA, A. S. ; SIMEONIDOU,

- DIMITRA . Programmable intelligent spaces for Industry 4.0: Indoor visual localization driving attocell networks. *Transactions on Emerging Telecommunications Technologies*, v. 3, p. 20-40, 2019.
3. FRASCOLLA, VALERIO ; **DOMINICINI, C. K.** ; PAIVA, MARCIA ; CAPOROSI, GILLES ; MAROTTA, MARCELO ; RIBEIRO, MOISES ; SEGATTO, MARCELO ; MARTINELLO, MAGNOS ; MONTEIRO, MAXWELL ; BOTH, CRISTIANO . Optimizing C-RAN Backhaul Topologies: A Resilience-Oriented Approach Using Graph Invariants. *Applied Sciences-Basel*, v. 9, p. 136, 2019.
 4. MARQUES, P. et al. Optical and wireless network convergence in 5G systems - an experimental approach, 2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Barcelona, pp. 1-5, 2018.
 5. **DOMINICINI, C. K.** ; MARTINELLO, M. ; WICKBOLDT, J. ; BOTH, C. ; NEJABATI, R. ; MARQUEZ-BARJA, J. M. ; SILVA, L. . Enabling Experimental Research Through Converged Orchestration of Optical Wireless and Cloud Domains. In: *EUCNC 5G and Beyond*, 2018, Ljubljana, Slovenia. EUCNC, 2018.
 6. CERAVOLO, I. ; CARDOSO, D. ; **DOMINICINI, C. K.** ; HASSE, P. ; VILLACA, R. S. ; RIBEIRO, M. R. N. ; MARTINELLO, M. ; NEJABATI, R. ; SIMEONIDOU, D. . O2CMF: Experiment-as-a-Service for Agile Fed4Fire Deployment of Programmable NFV. In: *Optical Fiber Communication Conference (OFC) 2018*, 2018, San Diego. SDN/NFV Demonstration Zone, 2018.
 7. GOMES, R. L. ; MARTINELLO, M. ; **DOMINICINI, C. K.** ; HASSE, P. ; VILLACA, R. S. ; VASSALO, R. F. ; DO CARMO, A. P. ; DE QUEIROZ, F. M. ; PICORETI, R. ; GARCIA, A. S. ; RIBEIRO, M. R. N. ; ESPIN, J. G. ; HAMMAD, A. ; NEJABATI, R. ; SIMEONIDOU, D. . How can emerging applications benefit from EaaS in open programmable infrastructures?. *IEEE International Summer School on Smart Cities (IEEE S3C)*, Natal-RN, 2017.
 8. CERAVOLO, I. ; CARDOSO, D. ; **DOMINICINI, C. K.** ; MARTINELLO, M. ; VILLACA, R. S. ; RIBEIRO, M. R. N. . Proposta e Implementação de um Framework de Controle para Testbeds Federados que Integram Nuvem e SDN. In: *SBRC 2017 - WPEIF*, 2017, Belem. SBRC 2017 - WPEIF, 2017.
 9. GUIMARAES, R. S. ; MARTINELLO, M. ; **DOMINICINI, C. K.** ; LIMA, D. S. A. ; VILLACA, R. ; RIBEIRO, M. R. N. . ROUTEOPS: Orquestração de Rotas Centrada na Operação de Sistemas Autônomos. In: *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2016, Salvador-BA. Anais do SBRC, 2016.

Bibliography

- [Abdullah et al. 2019] Abdullah, Z. N., Ahmad, I., and Hussain, I. (2019). Segment routing in software defined networks: A survey. *IEEE Communications Surveys Tutorials*, 21(1):464–486.
- [Abts and Kim 2011] Abts, D. and Kim, J. (2011). High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities. *Synthesis Lectures on Computer Architecture*, 6(1):1–115.
- [Al-Fares et al. 2008] Al-Fares, M., Loukissas, A., and Vahdat, A. (2008). A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM '08*, volume 38, page 63, New York, New York, USA. ACM Press.
- [Bajard 2007] Bajard, J. C. (2007). A residue approach of the finite fields arithmetics. In *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, pages 358–362.
- [Bari et al. 2016] Bari, F., Chowdhury, S. R., Ahmed, R., Boutaba, R., and Duarte, O. C. M. B. (2016). Orchestrating virtualized network functions. *IEEE Transactions on Network and Service Management*, 13(4):725–739.
- [Bari et al. 2015] Bari, M. et al. (2015). On orchestrating virtual network functions. In *CNSM 2015*, pages 50–56.
- [Barroso et al. 2013] Barroso, L. A., Clidaras, J., and Hölzle, U. (2013). The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154.
- [Beck and Botero 2017] Beck, M. T. and Botero, J. F. (2017). Scalable and coordinated allocation of service function chains. *Computer Communications*, 102:78 – 88.
- [Bhamare et al. 2016] Bhamare, D., Jain, R., Samaka, M., and Erbad, A. (2016). A survey on service function chaining. *Journal of Network and Computer Applications*, 75:138 – 155.

- [Bhamare et al. 2017] Bhamare, D., Samaka, M., Erbad, A., Jain, R., Gupta, L., and Chan, H. A. (2017). Optimal virtual network function placement in multi-cloud service function chaining architecture. *Computer Communications*, 102:1 – 16.
- [Bhatia et al. 2015] Bhatia, R., Hao, F., Kodialam, M., and Lakshman, T. V. (2015). Optimized network traffic engineering using segment routing. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 657–665.
- [Bilal et al. 2014] Bilal, K., Malik, S., Khan, S., and Zomaya, A. (2014). Trends and challenges in cloud datacenters. *Cloud Computing, IEEE*, 1(1):10–20.
- [Bosshart et al. 2014] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.
- [Bottorff et al. 2017] Bottorff, P., Fedyk, D., and Assarpour, H. (2017). Ethernet mac chaining. Internet-draft, Internet Engineering Task Force.
- [Casado et al. 2012] Casado, M. et al. (2012). Fabric: a retrospective on evolving sdn. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 85–90. ACM.
- [Casado et al. 2010] Casado, M., Koponen, T., Ramanathan, R., and Shenker, S. (2010). Virtualizing the Network Forwarding Plane. PRESTO '10, pages 8:1–8:6, New York, NY, USA. ACM.
- [Cascone et al. 2017] Cascone, C., Sanvito, D., Pollini, L., Capone, A., and Sansò, B. (2017). Fast failure detection and recovery in sdn with stateful data plane. *International Journal of Network Management*, 27(2):e1957.
- [Castanho et al. 2018] Castanho, M., Dominicini, C. K., Villaça, R. S., Martinello, M., and Ribeiro, M. R. N. (2018). Phantomsfc: A fully virtualized and agnostic service function chaining architecture. In *IEEE ISCC 2018*, Natal, RN, Brazil.
- [Chang et al. 2015] Chang, C. et al. (2015). Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. *IEEE Circuits and Systems Magazine*, 15(4):26–44.
- [Chen et al. 2011] Chen, K., Hu, C., Zhang, X., Zheng, K., Chen, Y., and Vasilakos, A. (2011). Survey on routing in data centers: insights and future directions. *IEEE Network*, 25(4):6–10.
- [Clad et al. 2018] Clad, F. et al. (2018). Segment Routing for Service Chaining. Internet-draft, IETF.

- [Cui et al. 2012] Cui, H. et al. (2012). Optically Cross-Braced Hypercube: a Reconfigurable Physical Layer for Interconnects and Server-Centric Datacenters. *Optical Fiber Communication Conference*, 1:OW3J.1.
- [da Silva et al. 2018] da Silva, J. S., Boyer, F., Chiquette, L., and Langlois, J. M. P. (2018). Extern objects in p4: an rohc header compression scheme case study. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pages 517–522.
- [Davoli et al. 2015] Davoli, L., Veltri, L., Ventre, P. L., Siracusano, G., and Salsano, S. (2015). Traffic engineering with segment routing: Sdn-based architectural design and open source implementation. In *2015 Fourth European Workshop on Software Defined Networks*, pages 111–112.
- [de Sousa et al. 2019] de Sousa, N. F. S., Perez, D. A. L., Rosa, R. V., Santos, M. A., and Rothenberg, C. E. (2019). Network service orchestration: A survey. *Computer Communications*, 142-143:69 – 94.
- [Dietrich et al. 2017] Dietrich, D., Abujoda, A., Rizk, A., and Papadimitriou, P. (2017). Multi-provider service chain embedding with nestor. *IEEE Transactions on Network and Service Management*, 14(1):91–105.
- [Dominicini et al. 2016] Dominicini, C. et al. (2016). VirtPhy: a fully programmable infrastructure for efficient NFV in small data centers. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN) (NFV-SDN'16)*, Palo Alto, USA.
- [Dominicini et al. 2017] Dominicini, C. K. et al. (2017). Virtphy: Fully programmable nfv orchestration architecture for edge data centers. *IEEE Transactions on Network and Service Management*, 14(4):817–830.
- [Dominicini et al. 2019] Dominicini, C. K., Vassoler, G., Valentim, R., Villaça, R., Ribeiro, M. R. N., Martinello, M., and Zambon, E. (2019). Keysfc: Agile traffic steering using strict source routing. In *Proceedings of the 2019 ACM Symposium on SDN Research, SOSR '19*, pages 154–155, New York, NY, USA. ACM.
- [Draxler et al. 2018] Draxler, S. et al. (2018). Jasper: Joint optimization of scaling, placement, and routing of virtual network services. *IEEE Transactions on Network and Service Management*, 15(3):946–960.
- [Draxler et al. 2017] Draxler, S., Karl, H., Peuster, M., Kouchaksaraei, H. R., Bredel, M., Lessmann, J., Soenen, T., Tavernier, W., Mendel-Brin, S., and Xilouris, G. (2017). SONATA: Service programming and orchestration for virtualized software networks. In

- 2017 *IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 973–978. IEEE.
- [ETSI MEC ISG 2014] ETSI MEC ISG (2014). Mobile edge computing - introductory technical white paper.
- [ETSI NFV ISG 2012] ETSI NFV ISG (2012). Network functions virtualization, an introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow SDN and OpenFlow World Congress*.
- [ETSI NFV ISG 2014] ETSI NFV ISG (2014). NFV 002 V1.2.1. Network Functions Virtualisation (NFV); Architectural Framework.
- [ETSI NFV ISG 2015] ETSI NFV ISG (2015). NFV-EVE 005 V1.1.1. Network Functions Virtualisation (NFV); Report on SDN Usage in NFV Architectural Framework.
- [ETSI NFV ISG 2018] ETSI NFV ISG (2018). Nfv 003 v1.3.1 network functions virtualisation (nfv); terminology for main concepts in nfv.
- [Farhady et al. 2015] Farhady, H., Lee, H., and Nakao, A. (2015). Software-Defined Networking: A survey. *Computer Networks*, 81:79–95.
- [Filsfils et al. 2015] Filsfils, C., Nainar, N. K., Pignataro, C., Cardona, J. C., and Francois, P. (2015). The Segment Routing Architecture. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE.
- [Filsfils et al. 2018] Filsfils, C., Previdi, S., Ginsberg, L., Decraene, B., Litkowski, S., and Shakir, R. (2018). Segment Routing Architecture. Internet-draft.
- [Garner 1959] Garner, H. L. (1959). The residue number system. *Transactions on Electronic Computers*, pages 140 – 147.
- [Gomes et al. 2016] Gomes, R. R. et al. (2016). Kar: Key-for-any-route, a resilient routing system. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, pages 120–127.
- [Greenberg et al. 2009] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. (2009). V12: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA. ACM.
- [Grymel and Furber 2011] Grymel, M. and Furber, S. B. (2011). A novel programmable parallel crc circuit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(10):1898–1902.

- [Guedes et al. 2012] Guedes, D., Vieira, L., Vieira, M., Rodrigues, H., and Nunes, R. (2012). Redes Definidas por Software: uma abordagem sistêmica para o desenvolvimento de pesquisas em Redes de Computadores. *Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC 2012*, 30(4):160–210.
- [Guo et al. 2009] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. (2009). Bcube: A high performance, server-centric network architecture for modular data centers. In *In SIGCOMM*.
- [Guo et al. 2010] Guo, C., Lu, G., Wang, H. J., Yang, S., Kong, C., Sun, P., Wu, W., and Zhang, Y. (2010). Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference, Co-NEXT '10*, pages 15:1–15:12, New York, NY, USA. ACM.
- [Guo et al. 2008] Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., and Lu, S. (2008). Dcell: A scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 75–86, New York, NY, USA. ACM.
- [Halpern and Pignataro 2015] Halpern, J. and Pignataro, C. (2015). Service function chaining (sfc) architecture. Technical report.
- [Han et al. 2015] Han, B., Gopalakrishnan, V., Ji, L., and Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97.
- [Hantouti et al. 2018] Hantouti, H., Benamar, N., Taleb, T., and Laghrissi, A. (2018). Traffic steering for service function chaining. *IEEE Communications Surveys Tutorials*, pages 1–1.
- [Herker et al. 2015] Herker, S. et al. (2015). Evaluation of data-center architectures for virtualized network functions. In *ICCW 2015*, pages 1852–1858.
- [Herrera and Botero 2016] Herrera, J. G. and Botero, J. F. (2016). Resource allocation in nfv: A comprehensive survey. *IEEE Transactions on Network and Service Management*, 13(3):518–532.
- [Herstein 1975] Herstein, I. (1975). *Topics in Algebra*. John Wiley & Sons, 2nd edition edition.
- [Homma et al. 2016] Homma, S. et al. (2016). Analysis on Forwarding Methods for Service Chaining. Internet-draft, IETF.
- [Jia 2014] Jia, W. (2014). A scalable multicast source routing architecture for data center networks. *IEEE Journal on Selected Areas in Communications*, 32(1):116–123.

- [Jin et al. 2016] Jin, X. et al. (2016). Your data center switch is trying too hard. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research*, pages 12:1–12:6, New York, NY, USA. ACM.
- [John et al. 2013] John, W. et al. (2013). Research Directions in Network Service Chaining. In *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, pages 1–7. IEEE.
- [Jyothi et al. 2015] Jyothi, S. A. et al. (2015). Towards a flexible data center fabric with source routing. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research*, pages 10:1–10:8, New York, NY, USA. ACM.
- [Kasey Panetta 2017] Kasey Panetta (2017). Top Trends in the Gartner Hype Cycle for Emerging Technologies, 2017 - Smarter With Gartner.
- [Kotronis et al. 2012] Kotronis, V., Dimitropoulos, X., and Ager, B. (2012). Outsourcing the routing control logic: Better Internet routing based on SDN principles. pages 55–60. ACM.
- [Kourtis et al. 2017] Kourtis, M.-A., McGrath, M. J., Gardikis, G., Xilouris, G., Riccobene, V., Papadimitriou, P., Trouva, E., Liberati, F., Trubian, M., Batalle, J., Koumaras, H., Dietrich, D., Ramos, A., Ferrer Riera, J., Bonnet, J., Pietrabissa, A., Ceselli, A., and Petrini, A. (2017). T-NOVA: An Open-Source MANO Stack for NFV Infrastructures. *IEEE Transactions on Network and Service Management*, 14(3):586–602.
- [Kumar et al. 2017] Kumar, S., Tufail, M., Majee, S., Captari, C., and Homma, S. (2017). Service function chaining use cases in data centers. Internet-draft, Internet Engineering Task Force.
- [Kurose and Ross 2013] Kurose, J. F. and Ross, K. W. (2013). *Computer networking: a top-down approach*. Pearson.
- [Laghrissi and Taleb 2019] Laghrissi, A. and Taleb, T. (2019). A survey on the placement of virtual resources and virtual network functions. *IEEE Communications Surveys Tutorials*, 21(2):1409–1434.
- [Lantz et al. 2010] Lantz, B., Heller, B., and McKeown, N. (2010). A Network in a Laptop: Rapid Prototyping for Software-defined Networks. *Hotnets-IX*, pages 19:1–19:6, New York, NY, USA. ACM.
- [Leivadeas et al. 2017] Leivadeas, A., Falkner, M., Lambadaris, I., and Kesidis, G. (2017). Optimal virtualized network function allocation for an sdn enabled cloud. *Computer Standards and Interfaces*, 54:266 – 278. SI: Standardization SDN and NFV.

- [Li et al. 2017] Li, G., Zhou, H., Feng, B., Li, G., Li, T., Xu, Q., and Quan, W. (2017). Fuzzy theory based security service chaining for sustainable mobile-edge computing. *Mobile Information Systems*, 2017.
- [Li et al. 2016] Li, Z., Guo, Z., and Yang, Y. (2016). BCCC: An Expandable Network for Data Centers. *IEEE/ACM Transactions on Networking*, 24(6):3740–3755.
- [Li and Yang 2016] Li, Z. and Yang, Y. (2016). GBC3: A Versatile Cube-Based Server-Centric Network for Data Centers. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2895–2910.
- [Liberato et al. 2018] Liberato, A. et al. (2018). RDNA: Residue-Defined Networking Architecture Enabling Ultra-Reliable Low-Latency Datacenters. *IEEE TNSM*.
- [Lin et al. 2016] Lin, T., Zhou, Z., Tornatore, M., and Mukherjee, B. (2016). Demand-aware network function placement. *Journal of Lightwave Technology*, 34(11):2590–2600.
- [Luizelli et al. 2016] Luizelli, M. C., Bays, L. R., Buriol, L. S., Barcellos, M. P., and Gasparry, L. P. (2016). How physical network topologies affect virtual network embedding quality: A characterization study based on isp and datacenter networks. *Journal of Network and Computer Applications*, 70:1 – 16.
- [Luizelli et al. 2017] Luizelli, M. C., da Costa Cordeiro, W. L., Buriol, L. S., and Gasparry, L. P. (2017). A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining. *Computer Communications*, 102:67 – 77.
- [Luizelli et al. 2015] Luizelli, M. C. et al. (2015). Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions. In *2015 IFIP/IEEE International Symposium on Integrated Network Management*, pages 98–106, Ottawa.
- [Mao et al. 2017] Mao, Y., You, C., Zhang, J., Huang, K., and Letaief, K. B. (2017). A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358.
- [Martinello et al. 2014] Martinello, M. et al. (2014). Keyflow: a prototype for evolving sdn toward core network fabrics. *IEEE Network*, 28(2):12–19.
- [Martinello et al. 2017] Martinello, M. et al. (2017). Programmable residues defined networks for edge data centres. In *CNSM*, pages 1–9.
- [Martini et al. 2015] Martini, B. et al. (2015). Latency-aware composition of virtual functions in 5G. In *NetSoft 2015*, pages 1–6.

- [Martins et al. 2014] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. (2014). ClickOS and the Art of Network Function Virtualization. NSDI'14, pages 459–473, Berkeley, CA, USA. USENIX Association.
- [Masutani et al. 2014] Masutani, H. et al. (2014). Requirements and design of flexible nfv network infrastructure node leveraging sdn/openflow. In *Optical Network Design and Modeling, Intl. Conf.*, pages 258–263. IEEE.
- [Matias et al. 2015] Matias, J. et al. (2015). Toward an sdn-enabled nfv architecture. *Communications Magazine, IEEE*, 53(4):187–193.
- [McCauley et al. 2019] McCauley, J., Panda, A., Krishnamurthy, A., and Shenker, S. (2019). Thoughts on load distribution and the role of programmable switches. *SIGCOMM Comput. Commun. Rev.*, 49(1):18–23.
- [Mckeown et al. 2008] Mckeown, N., Anderson, T., Peterson, L., Rexford, J., Shenker, S., and Louis, S. (2008). OpenFlow : Enabling Innovation in Campus Networks.
- [Mechtri et al. 2016] Mechtri, M., Ghribi, C., and Zeghlache, D. (2016). A scalable algorithm for the placement of service function chains. *IEEE Transactions on Network and Service Management*, 13(3):533–546.
- [Medhat et al. 2017] Medhat, A. M., Taleb, T., Elmangoush, A., Carella, G. A., Covaci, S., and Magedanz, T. (2017). Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges. *IEEE Communications Magazine*, 55(2):216–223.
- [Mehmeri et al. 2017] Mehmeri, V., Wang, X., Zhang, Q., Palacharla, P., Ikeuchi, T., and Monroy, I. T. (2017). Optical network as a service for service function chaining across datacenters. In *2017 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3.
- [Microchip 2018] Microchip (2018). 32-bit programmable cyclic redundancy check (crc). [http://ww1.microchip.com/downloads/en/DeviceDoc/dsPIC33_PIC24-FRM,-32-Bit-Programmable-Cyclic-Redundancy-Check-\(CRC\)-DS30009729C.pdf](http://ww1.microchip.com/downloads/en/DeviceDoc/dsPIC33_PIC24-FRM,-32-Bit-Programmable-Cyclic-Redundancy-Check-(CRC)-DS30009729C.pdf). Accessed: 2019-01-13.
- [Mijumbi et al. 2016] Mijumbi, R., , et al. (2016). Management and orchestration challenges in network functions virtualization. *IEEE Communications Magazine*, 54:98–105.
- [Mininet 2018] Mininet (2018). An Instant Virtual Network on your Laptop (or other PC). Available at: <http://mininet.org>.

- [Miotto et al. 2019] Miotto, G., Luizelli, M. C., Cordeiro, W. L. d. C., and Gaspar, L. P. (2019). Adaptive placement & chaining of virtual network functions with nfv-pear. *Journal of Internet Services and Applications*, 10(1):3.
- [Mirza-Aghatabar et al. 2007] Mirza-Aghatabar, M. et al. (2007). An empirical investigation of mesh and torus noc topologies under different routing algorithms and traffic models. *17th Euromicro Conf. on Digital System Design*, 0:19–26.
- [Mogul et al. 2010] Mogul, J. et al. (2010). DevoFlow: cost-effective flow management for high performance enterprise networks. In *Proc. of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*, pages 1–6.
- [Moreno et al. 2017] Moreno, E., Beghelli, A., and Cugini, F. (2017). Traffic engineering in segment routing networks. *Computer Networks*, 114:23 – 31.
- [Niranjan et al. 2009] Niranjan, R. et al. (2009). PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication - SIGCOMM '09*, page 39.
- [Open Networking Foundation 2014] Open Networking Foundation (2014). OpenFlow-enabled SDN and Network Functions Virtualization.
- [Openstack 2017] Openstack (2017). OpenStack Open Source Cloud Computing Software. Available at: <https://www.openstack.org/software/>.
- [OpenStack Foundation 2015] OpenStack Foundation (2015). Openstack cloud administrator guide. Technical report. Available at: <http://projects.sigma-orionis.com/eciao/wp-content/uploads/2015/05/admin-guide-cloud.pdf>.
- [OpenStack.org 2015] OpenStack.org (2015). OpenStack: The Open Source Cloud Operating System. Available at: <https://www.openstack.org/software/>.
- [P4.org 2017] P4.org (2017). P4 16 Language Specification version 1.0.0. Available at: <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>.
- [P4.org 2018a] P4.org (2018a). Behavioral model targets. Available at: <https://github.com/p4lang/behavioral-model/blob/master/targets/README.m>.
- [P4.org 2018b] P4.org (2018b). P4 14 Language Specification version 1.0.5. Available at: <https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf>.
- [P4.org 2019a] P4.org (2019a). P4-16 declaration of the P4 v1.0 switch model (v1model). Available at: <https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4>.

- [P4.org 2019b] P4.org (2019b). P4-16 declaration of the Portable Switch Architecture (PSA). Available at: <https://github.com/p4lang/p4c/blob/master/p4include/psa.p4>.
- [P4.org 2019c] P4.org (2019c). P416 Portable Switch Architecture (PSA) (working draft). Available at: <https://p4.org/p4-spec/docs/PSA.html>.
- [P4.org 2019d] P4.org (2019d). The behavioral model framework. Available at: <https://github.com/p4lang/behavioral-model>.
- [P4.org 2019e] P4.org (2019e). The BMv2 Simple Switch target. Available at: https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md.
- [Peterson and Brown 1961] Peterson, W. W. and Brown, D. T. (1961). Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235.
- [Quinn et al. 2018] Quinn, P., Elzur, U., and Pignataro, C. (2018). Network service header (nsh). RFC 8300, RFC Editor.
- [Quinn and Guichard 2014] Quinn, P. and Guichard, J. (2014). Service Function Chaining: Creating a Service Plane via Network Service Headers. *Computer*, 47(11):38–44.
- [Quinn and Nadeau 2015] Quinn, P. and Nadeau, T. (2015). Problem Statement for Service Function Chaining. RFC 7498, RFC Editor.
- [Ren et al. 2017] Ren, Y. et al. (2017). Flowtable-free routing for data center networks: A software-defined approach. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6.
- [Ren et al. 2018] Ren, Y., Huang, T., Lin, K. C., and Tseng, Y. (2018). On scalable service function chaining with $\mathcal{O}(1)$ flowtable entries. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 702–710.
- [Rosa et al. 2014] Rosa, R., Siqueira, M., Barea, E., Marcondes, C., and Rothenberg, C. E. (2014). Network Function Virtualization: Perspectivas, Realidades e Desafios. *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos-Florianópolis-SC*.
- [Rosen et al. 2001] Rosen, E., Viswanathan, A., and Callon, R. (2001). RFC 3031: Multiprotocol Label Switching Architecture. Technical report, IETF.
- [Rotsos et al. 2017] Rotsos, C., King, D., Farshad, A., Bird, J., Fawcett, L., Georgalas, N., Gunkel, M., Shiimoto, K., Wang, A., Mauthe, A., Race, N., and Hutchison, D. (2017). Network service orchestration standardization: A technology survey. *Computer Standards and Interfaces*, 54:203 – 215. SI: Standardization SDN and NFV.

- [Routray et al. 2013] Routray, S. K. et al. (2013). Statistical model for link lengths in optical transport networks. *J. Opt. Commun. Netw.*, 5(7):762–773.
- [Ruaro et al. 2018] Ruaro, M. et al. (2018). Software-defined networking architecture for noc-based many-cores. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.
- [Ryu 2015] Ryu (2015). Ryu SDN Framework Community. Available at: <http://osrg.github.io/ryu/>.
- [Saad and Schultz 1989] Saad, Y. and Schultz, M. H. (1989). Data communication in hypercubes. *J. of Parallel and Distributed Computing*, 6(1):115–135.
- [Satyanarayanan 2017] Satyanarayanan, M. (2017). The Emergence of Edge Computing. *Computer*, 50(1):30–39.
- [Schroeder 2009] Schroeder, M. (2009). *Number theory in science and communication: with applications in cryptography, physics, digital information, computing, and self-similarity*. Springer, Berlin, Heidelberg.
- [Sherry et al. 2012] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. (2012). Making Middleboxes Someone else’s Problem: Network Processing As a Cloud Service. SIGCOMM ’12, pages 13–24, New York, NY, USA. ACM.
- [Sherwood et al. 2009] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. (2009). FlowVisor: A Network Virtualization Layer. *OpenFlow Switch Consortium, Tech. Rep.*
- [Shi and Dustdar 2016] Shi, W. and Dustdar, S. (2016). The Promise of Edge Computing. *Computer*, 49(5):78–81.
- [Shoup 2009] Shoup, V. (2009). *A computational introduction to number theory and algebra*. Cambridge university press.
- [Soliman et al. 2012] Soliman, M., Nandy, B., Lambadaris, I., and Ashwood-Smith, P. (2012). Source routed forwarding with software defined control, considerations and implications. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 43–44. ACM.
- [Sonkoly et al. 2015] Sonkoly, B., Szabo, R., Jocha, D., Czentye, J., Kind, M., and Westphal, F.-J. (2015). UNIFYing Cloud and Carrier Network Resources: An Architectural View. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE.

- [Stephens et al. 2011] Stephens, B., Cox, A. L., Rixner, S., and Ng, T. S. E. (2011). A scalability study of enterprise network architectures. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, ANCS '11, pages 111–121, Washington, DC, USA. IEEE Computer Society.
- [Sun et al. 2019] Sun, G., Li, Y., Yu, H., Vasilakos, A. V., Du, X., and Guizani, M. (2019). Energy-efficient and traffic-aware service function chaining orchestration in multi-domain networks. *Future Generation Computer Systems*, 91:347 – 360.
- [Sunshine 1977] Sunshine, C. A. (1977). Source routing in computer networks. *SIGCOMM Comput. Commun. Rev.*, 7(1):29–33.
- [Trajkovska et al. 2017] Trajkovska, I. et al. (2017). Sdn-based service function chaining mechanism and service prototype implementation in nvf scenario. In *Computer Standards and Interfaces*. Elsevier.
- [Tso et al. 2016] Tso, F. P., Jouet, S., and Pezaros, D. P. (2016). Network and server resource management strategies for data centre infrastructures: A survey. *Computer Networks*, 106:209 – 225.
- [Vacaro et al. 2016] Vacaro, J. et al. (2016). High performance service chaining for ethernet transport networks. ACM SIGCOMM Industrial Demos '16.
- [Valentim et al. 2019] Valentim, R. et al. (2019). RDNA balance: Balanceamento de carga por isolamento de fluxos elefante em data centers com roteamento na origem. In *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Gramado, RS, Brazil.
- [Vassoler et al. 2014] Vassoler, G. et al. (2014). Twin datacenter interconnection topology. *Micro, IEEE*, 34(5):8–17.
- [Vassoler 2015] Vassoler, G. L. (2015). *TRIIAD: Uma Arquitetura para Orquestração Automônica de Redes de Data Center Centrado em Servidor*. PhD thesis, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal do Espírito Santo (PPGEE/UFES), Vitória/ES, Brasil.
- [Vassoler and Ribeiro 2017] Vassoler, G. L. and Ribeiro, M. R. N. (2017). An intelligent and integrated architecture for data centers with distributed photonic switching. In *2017 SBMO/IEEE MTT-S International Microwave and Optoelectronics Conference (IMOC)*, pages 1–5. IEEE.
- [Vencioneck et al. 2014] Vencioneck, R. et al. (2014). FlexForward: Enabling an SDN manageable forwarding engine in Open vSwitch. In *10th Int. Conf. on Network and Service Management (CNSM)*, pages 296–299.

- [Verdi et al. 2010] Verdi, F. L., Rothenberg, C. E., Pasquini, R., and Magalhães, M. (2010). Novas arquiteturas de data center para cloud computing. *XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos-Gramado-RS*.
- [Wessing et al. 2002] Wessing, H. et al. (2002). Novel scheme for packet forwarding without header modifications in optical networks. *Journal of Lightwave Technology*, 20(8):1277–1283.
- [Yen 1971] Yen, J. Y. (1971). Finding the k-shortest loopless paths in a network. *Management Science*, 17(11):712–716.
- [Yi et al. 2018] Yi, B., Wang, X., Li, K., k. Das, S., and Huang, M. (2018). A comprehensive survey of network function virtualization. *Computer Networks*, 133:212 – 262.
- [Zhang et al. 2018] Zhang, J., Wang, Z., Ma, N., Huang, T., and Liu, Y. (2018). Enabling efficient service function chaining by integrating nfv and sdn : Architecture, challenges and opportunities. *IEEE Network*, pages 1–8.
- [Zhang et al. 2013] Zhang, Y. et al. (2013). StEERING: A software-defined networking for inline service chaining. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE.
- [Zhao and Hu 2017] Zhao, M. and Hu, Z. (2017). An RNS-based forwarding approach to implement SFC. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 679–682. IEEE.

Appendix A

Enabling technologies

This appendix aims to describe the main enabling technologies used by the solutions proposed in this thesis. Firstly, we present concepts related to SDN, including OpenFlow protocol, P4 language, OvS software switch, and Mininet emulation tool. Then, we describe the concepts related to the OpenStack cloud platform.

A.1 SDN

SDN is a paradigm based on the following properties: (i) separation between control plane and data plane, (ii) uniform and vendor-independent interface to the forwarding engine, (iii) logically centralized control plane, and (iv) capability of virtualizing the underlying physical network [[Martinello et al. 2014](#)].

In SDN, the control plane acts as a networking operating system, which observes the entire state of the network from a central point and controls the forwarding plane, or data plane, to define the behavior of each forwarding element (switch, router, access point, or base station), hosting features such as routing protocols, access control, network virtualization, and energy management [[Lantz et al. 2010](#)]. The idea is that a controller can centralize communication with all programmable network elements, and offer a unified view of the state of the network [[Casado et al. 2010](#)].

One of the great benefits of SDN is the ability to use this centralized view to perform detailed analysis and make decisions about the overall system operation and the configuration of each network element. It is important to note that the controller is logically centralized, but it can be implemented in a distributed way by dividing the control elements between different domains or by implementing a truly distributed controller, which, for example, uses consensus algorithms to consolidate a view among its parts [[Guedes et al. 2012](#)].

Another advantage of SDN is that network functionalities can be changed or added after network deployment without modifying the hardware, allowing the network to evolve at the same speed as software, rather than waiting for the development of standards and

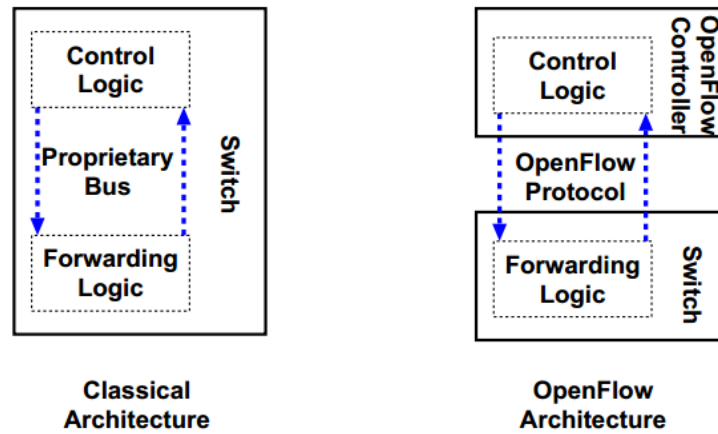


Figure A.1: Traditional networking architecture versus OpenFlow architecture. Source: [Sherwood et al. 2009].

new hardware. Another particularly interesting benefit to DCNs is that high-level packet manipulation decisions are made by a centralized controller, so that the data plane can be implemented in low-cost commodity switches [Verdi et al. 2010].

A.1.1 OpenFlow

The OpenFlow standard is the most successful and widely adopted implementation of the SDN paradigm. Its primary purpose is to provide a standard and open programming interface that allows remote control of routing tables of forwarding devices such as switches, routers and access points.

[Sherwood et al. 2009] compares a classical network architecture, where control and data forwarding logics are located at the same device (for example, a switch) and communicate through an internal proprietary bus, with the OpenFlow architecture, where control logic is transferred to an external controller that communicates with the device responsible for forwarding logic using the OpenFlow protocol. This comparison is shown in more detail in Fig. A.1.

According to the OpenFlow architecture, an OpenFlow switch has at least three components [Mckeown et al. 2008], as shown in Fig. A.2:

- The Flow Table, which associates an action with each flow entry, to tell the switch how to process each flow.
- The Secure Channel that connects the switch to the controller, allowing commands and packets to be sent in order to protect data confidentiality and integrity.
- The OpenFlow Protocol, which provides an open, standard way for a controller to communicate with a switch, and allows entries in the flow table to be defined externally.

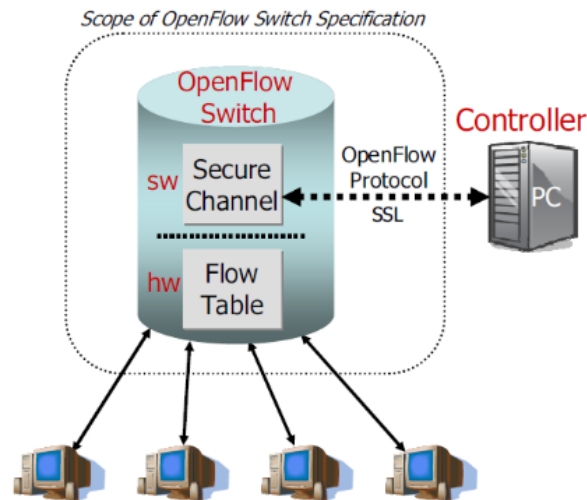


Figure A.2: OpenFlow architecture. Source: [Mckeown et al. 2008].

The OpenFlow protocol abstracts forwarding/routing rules as flow entries, where the collection of flow entries in a packet forwarder (a switch or router) is called a flow table [Sherwood et al. 2009]. Each flow table entry associates a rule with a flow, and, if a packet fits into a specific rule, the defined actions are taken, such as: forward the packet to a specific port, change part of its headers, or forward it for inspection in the network controller [Guedes et al. 2012]. If there is no rule defined for a given packet, a message is sent to the controller, which defines a new flow entry. In this way, these rules allow remote control of how packets that are part of a specific flow must be forwarded and processed in each network element, according to the interests identified by the centralized view of the controller.

A.1.1.1 OpenFlow Controllers

As discussed earlier, an OpenFlow controller manipulates the flow table entries in the network elements through the OpenFlow protocol, and can act in a static or dynamic way [Mckeown et al. 2008]. In the case of a static controller, the controller may be, for example, a simple application running on a PC that statically establishes a set of flow entries to enable the interconnection of some test computers during an experiment. In the case of more sophisticated controllers, the controller can add and remove flow entries dynamically during network operation and even allow different users and applications to interfere with control decisions. There are currently a number of controllers that support the OpenFlow protocol, such as: Floodlight¹, Ryu², ONOS³, and OpenDaylight⁴.

¹<http://www.projectfloodlight.org/floodlight/>

²<https://osrg.github.io/ryu/>

³<https://onosproject.org/>

⁴<https://www.opendaylight.org/>

A.1.2 P4 language

P4 is a high-level language for programming the data plane of network devices in a protocol-independent manner [Bosshart et al. 2014]. Differently from traditional switches, the data plane functionality in P4 is not defined beforehand (e.g., match-action tables with fixed protocol columns), but is described by a program. This ability to define the data plane according to application needs is important to enable many innovative SDN packet processing algorithms. For instance, these applications may need to perform table lookups over new fields, or to perform general-purpose operations over the packet headers, as the case of the solutions we propose in this thesis.

The first version of the P4 language, P4 14, was released in 2014 [Bosshart et al. 2014], and assumes specific device capabilities that covers only a subset of programmable networking targets. It is based on an abstract forwarding model of a programmable switch architecture, called PISA (Protocol-Independent Switch Architecture), consisting of a parser and a set of match+action table resources, divided between ingress and egress, as shown in Fig. A.3. The parser identifies the headers of incoming packets, which are used by each match+action table to perform lookup on header fields and apply the corresponding actions [P4.org 2018b]. In contrast to OpenFlow, P4 supports a programmable parser with the definition of new headers, match+action stages can be in parallel or in series, and actions are composed from protocol-independent primitives supported by the switch [Bosshart et al. 2014].

In 2016, the second version of P4 language, P4 16 [P4.org 2017], was released as a more mature and stable language definition. It does not assume device capabilities, which instead are defined by target manufacturer via external libraries, called externs. In this way, it can work for many targets, which can be any packet-processing system capable of executing a P4 program, such as switches and NICs. Fig. A.4 shows a typical P4

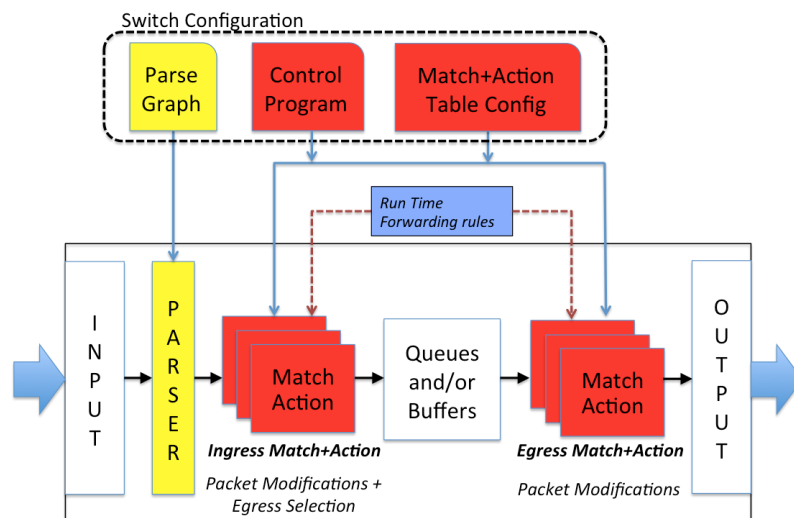


Figure A.3: PISA architecture. Source: [P4.org 2018b].

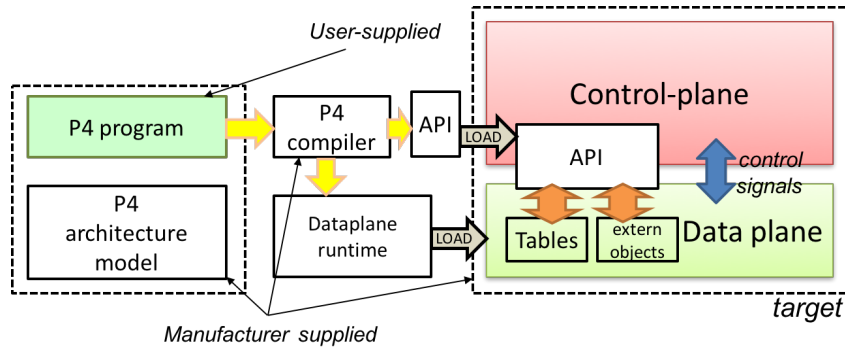


Figure A.4: Example of P4 workflow. Source: [P4.org 2017].

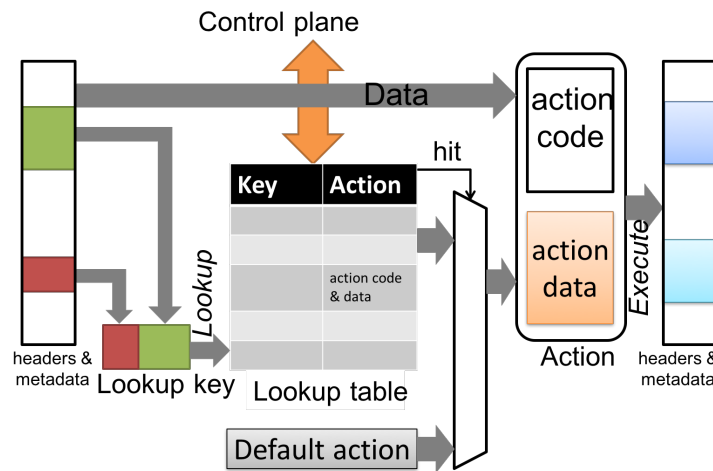


Figure A.5: Lookup table in P4. Source: [P4.org 2017].

workflow when programming a specific target, which provides: the hardware or software implementation framework, an architecture definition, and a P4 compiler for that target.

P4 programs describe the data plane functionality that is configured at initialization time. The compilation of a set of P4 programs produces two artifacts [P4.org 2017]: a data plane configuration that implements the forwarding logic described in the program, and an API that allows the control plane to manage the data plane objects.

A table in a P4 program describes a match-action unit that is composed by three fields: key, action, and action data. Processing a packet using a match-action table executes the following steps, shown in Fig. A.5 [P4.org 2017]: the key lookup in the table (the “match” step) results in the execution of either a “miss” or action indication over the input data (the “action step”), which may produce mutations of the input packet and associated metadata. The control plane is responsible for populating the tables entries with specific information based on static configuration, automatic discovery, or protocol calculations.

A.1.3 Open vSwitch (OvS)

Open vSwitch (OvS) is an open source software switch with support for leading protocols and management interfaces such as OpenFlow, NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, and 802.1ag. It has features comparable to physical switches, but with the flexibility and speed of development of a software implementation. Thus, it facilitates the network management in virtual environments, as well as offers additional functionalities not available hardware switches [Verdi et al. 2010]. The OvS follows the OpenFlow architecture and presents good performance, because it separates the data plane into kernel from the Linux operating system, while the control plane is accessed from the user space [Guedes et al. 2012]. In this way, OvS is a powerful tool for implementing network virtualization using the OpenFlow protocol.

A.1.4 Mininet

Mininet is a network emulation tool that provides light virtualization, extensible CLI, and API, providing an environment for rapid prototyping in which it is possible to create, interact and customize parts of a software-defined network [Lantz et al. 2010]. Using Mininet, it is possible to create an OpenFlow or P4 virtual network, running in real kernel, with switches, hosts, links and application code, in a single real machine or virtual machine [Mininet 2018]. Thus, Mininet appears as a great alternative to develop, share and experiment with SDN systems.

A.2 OpenStack

OpenStack⁵ is an open source software platform for public and private clouds that allows the management of large pools of computing, storage, and network resources within a DC. The resources are managed through a dedicated interface that enables administrators to control and provision resources for multiple users. Currently more than 200 hardware, software and service companies support the development of the platform, which is maintained by a global community of developers.

One of the main goals of OpenStack is to build a cloud computing service that can be installed on commodity hardware. As shown in Fig. A.6, the components responsible for controlling groups of computers, storage, and network resources act on the hardware layer to provide interfaces to upper layers. At the top of this structure, there are the application layer and administration layer, which is represented by a web interface called Dashboard.

The OpenStack platform has been used in various proof-of-concepts and use cases for NFV. Despite still having limitations in relation to NFV requirements, it has been consid-

⁵<https://www.openstack.org/>

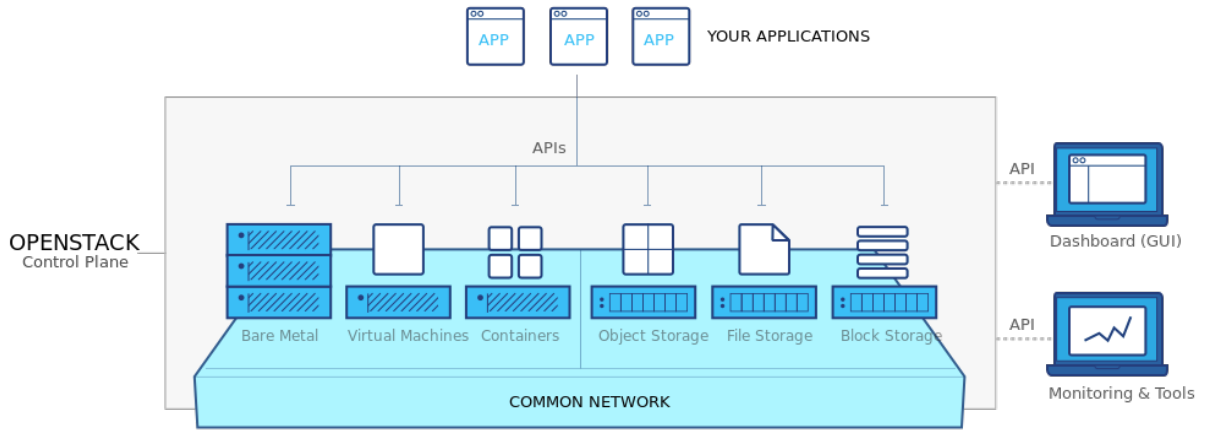


Figure A.6: OpenStack Diagram. Source: <https://www.openstack.org/software/>

ered one of the main technologies to enable resource orchestration in NFV [Rosa et al. 2014]. This work uses the OpenStack platform as a enabling technology for implementing our orchestration and SFC solutions.

A.2.1 Networking

Some of the prototypes of this thesis were implemented using OpenStack. Therefore, understanding the inner working of OpenStack networking is crucial. In OpenStack, the physical servers are called compute nodes and their internal networking structure is described by Fig. A.7.

In this structure, VMs appear in the upper layer, connected to a Linux bridge, called qbr, that implements security groups using iptables. These bridges are connected to the integration bridge, br-int, which is an OvS bridge that interconnects the VMs running on the compute node. The br-int is, then, connected to the br-eth OvS bridge, which provides connectivity to the physical network interface cards. In the example of Fig. A.7, when virtual machine VM1 sends an Ethernet frame to the physical network, it must pass through nine devices inside of the host: TAP eth0, Linux bridge qbrxxx, veth pair (qvbxxx, qvovxxx), OvS bridge br-int, veth pair (intbr-eth, phy-br-eth), OvS bridge br-eth, and, finally, to one of the physical network interface cards [OpenStack Foundation 2015].

When a virtual private network needs to be created, the VIM configures the OvS of server nodes, using VLAN tags. Thus, a tenant gets a subnet and range of private IPs that are only accessible for a specific VLAN, where all his VMs are bridged. In the example of Fig. A.7, the physical network supports VLAN IDs 101 and 102 and there are two private virtual networks, net01 and net02, which have VLAN ids of 1 and 2. Flow rules in br-int and br-eth execute VLAN translation between physical and private VLANs.

The OpenStack have various deployment options. In the prototypes of this thesis, we used a deployment option composed by three types of nodes: a Controller node, which

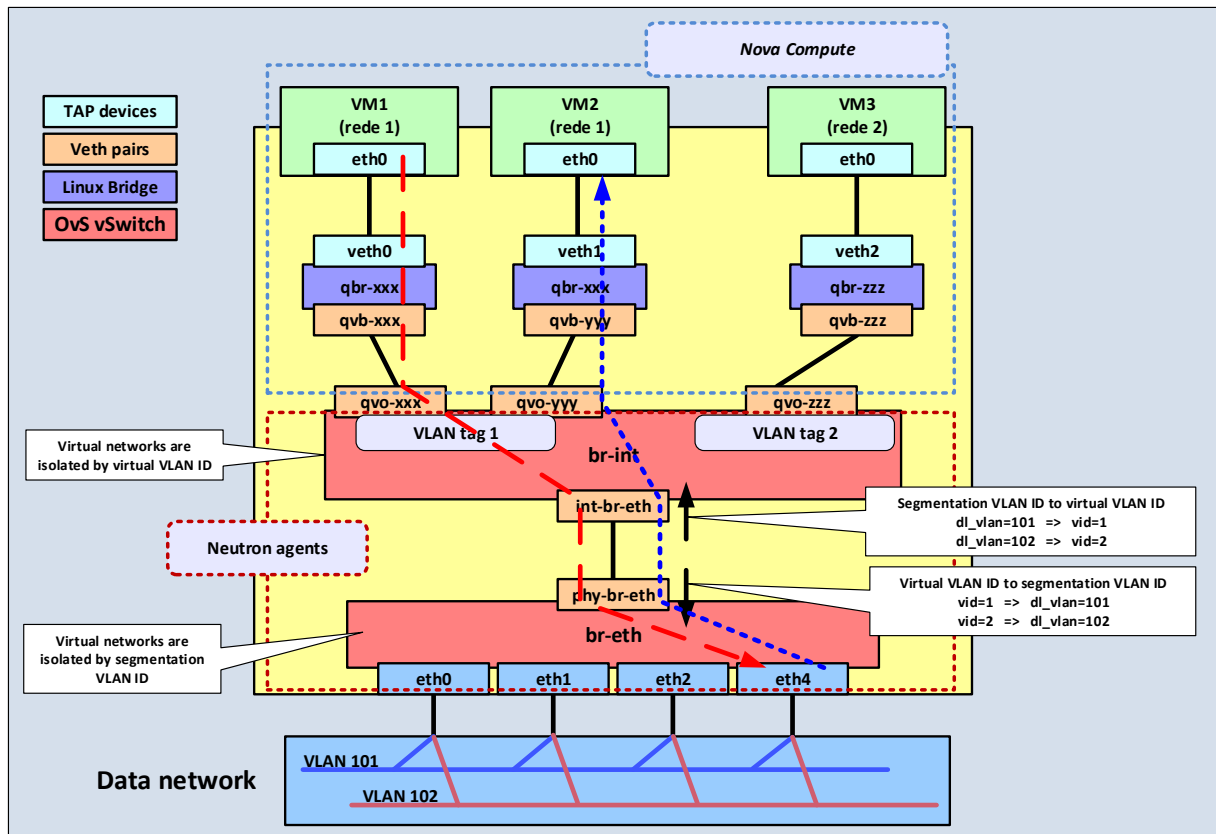


Figure A.7: Networking of compute node in OpenStack. Source [Dominicini et al. 2017].

provides management and provisioning services; a Network node, which is responsible for L3 routing, NAT, and DHCP functionalities; and a set of compute nodes. VM instances that are in the same virtual network and are hosted in the same server node, as VM1 and VM2 in Fig. A.7, communicate directly via br-int bridge. On the other hand, if two VM instances are hosted in the same server node, but are not in the same private network, as VM1 and VM3 in Fig. A.7, their communication is intermediated by the Network node, which acts as a L3 router. As another example, if two VMs are in the same virtual network, but are hosted in different servers, the communication will pass through all the bridge layers, up and down, but without the intermediation of the Network node, as the communication does not involve L3 routing.

Appendix B

Comparison between the Fat-Tree and the Hypercube topologies

This appendix aims to execute a preliminary investigation if server-centric DCNs can efficiently provision diverse NFV workloads when compared to traditional network-centric DCNs, considering edge DCs. To this end, we present a ILP model (See Section B.1) and analyze two data center network topologies: the Fat-Tree, a traditional network-centric topology where servers host VNFs and switches forward traffic, and the Hypercube, a well known server-centric topology where servers play both forwarding and virtualization roles [Vassoler et al. 2014]. In the next sections, we explore how the potential tradeoffs of each topology affect NFV orchestration requirements in edge DCs.

This work was developed in conjunction with Leandro C. Resendo, Gilmar L. Vassoler, Moises R. N. Ribeiro, Magnos Martinello, Eduardo Zambon, and Renato de Moraes.

B.1 The optimization model

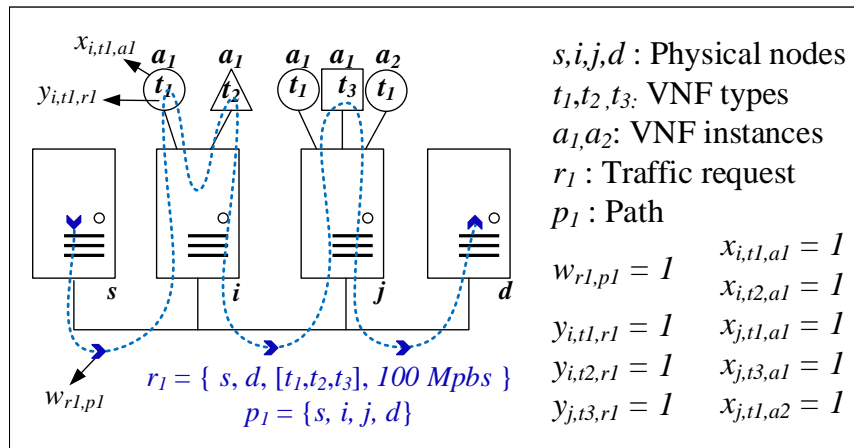


Figure B.1: VNF-FGE model example.

Herrera and Botero [Herrera and Botero 2016] present a survey of the VNF-FGE problem and a classification based on the following aspects: optimization objectives (e.g., minimize OPEX [Bari et al. 2015], the number of VNF instances [Luizelli et al. 2015], or deployment costs), solution strategy (e.g., exact and heuristic [Bari et al. 2015, Lin et al. 2016, Luizelli et al. 2015]), and application domain (e.g., TSP [Bari et al. 2015], NFV [Luizelli et al. 2015] and optical networks [Lin et al. 2016]). This survey shows that several works have proposed optimization models to solve the VNF-FGE problem. However, they usually overlook the fact that VNF-FGE and interconnection topologies are intertwined with each other.

To tackle this issue, this section proposes a novel modeling approach that takes into account the influence of interconnects on NFV orchestration. It also outlines a common ground for comparing different aspects, such as CPU and link usage, from interconnection schemes in network-centric and server-centric approaches. We develop a novel optimization model that is able to represent both server-centric and network-centric datacenter network topologies. More specifically, we formulate the VNF-FGE problem [Herrera and Botero 2016] as an ILP implemented in CPLEX.

The following notation is used in our mathematical model: i and j are physical nodes; $t \in \mathbf{T}$ is a VNF type; a is a VNF instance of a VNF type; $r = \langle s(r), d(r), \mathbf{T}(r), b(r) \rangle$ is a traffic request; and $p \in \mathbf{P}(r)$ is a path (route) in the set of possible routes to provision a request r . A traffic request r contains the source node $s(r)$, the destination node $d(r)$, the bandwidth demand $b(r)$, and a VNF service chaining containing the sequence of VNF types through which the traffic must pass, $\mathbf{T}(r)$ (e.g., $\mathbf{T}(r) = t_1 \rightarrow t_2 \rightarrow t_3$).

Given: $E[i][j]$: topology adjacency matrix; N : number of nodes in the network; \overline{CPU} : number of CPU cores of a physical node (equal for all nodes); $CPU[t]$: number of CPU cores required by each VNF type; \overline{Proc} : processing capacity of a physical node (equal for all nodes); $Proc[t]$: processing capacity of each VNF type; \overline{Link} : link capacity (equal for all links); $\delta_{i,j}^{r,p}$: 1 if request r is provisioned by link i,j in path p , and 0 otherwise. This link-path indicator is provided by Yen[Yen 1971]; and analogously, $\delta_i^{r,p}$ is the node-path indicator.

Variables: $w_{r,p}$: 1 if demand r is provisioned by path p , and 0 otherwise; $x_{i,t,a}$: 1 if VNF instance a is of type t and is provisioned by node i , and 0 otherwise; $y_{i,t,r}$: 1 if request r is provisioned by a VNF instance of type t at node i , and 0 otherwise. z_i : processing load of physical node i . Fig. B.1 illustrates an example of a traffic request posed by clients and how it is represented in our model.

Objective Function: To minimize the number of VNF instances needed to serve all requests:

$$\min : \sum_i \sum_a \sum_t x_{i,t,a} \quad (\text{B.1})$$

This function is a good indicator of how efficiently one can map traffic requests into a given infrastructure. However, this can be easily changed to consider other costs, such

as CAPEX.

Constraints:

$$\sum_t \sum_a CPU[t] \times x_{i,t,a} \leq \overline{CPU}; \forall i \quad (B.2)$$

$$\sum_{r:t \in \mathbf{T}(r)} b(r) \times y_{i,t,r} \leq \sum_a Proc[t] \times x_{i,t,a}; \forall i, t \quad (B.3)$$

$$y_{i,t,r} \leq \sum_a x_{i,t,a}; \forall i, t, r \quad (B.4)$$

$$\sum_p w_{r,p} = 1; \forall r \quad (B.5)$$

$$\sum_r \sum_p \delta_{i,j}^{r,p} \times b(r) \times w_{r,p} \leq \overline{Link}; \forall i, j \quad (B.6)$$

$$\sum_i y_{i,t,r} = 1; \forall r, t : t \in \mathbf{T}(r) \quad (B.7)$$

$$w_{r,p} \leq \sum_i \sum_j \delta_{i,j}^{r,p} \times y_{i,t,r}; \forall p, r, t : t \in \mathbf{T}(r) \quad (B.8)$$

$$z_i = \sum_a \sum_t Proc[t] \times x_{i,t,a} + \sum_p \sum_{i \neq s(r) \wedge i \neq d(r)}^r \delta_i^{r,p} \times b(r) \times w_{r,p}; \forall i \quad (B.9)$$

$$z_i \leq \overline{Proc}; \forall i \quad (B.10)$$

Eq. (B.2) ensures that the number of CPUs used by all VNF instances of all types will not exceed the number of CPUs of the respective physical node. Eq. (B.3) ensures that VNF instances of type t provisioned by physical node i can process all the requests for network service t in node i . Here, we consider that the processing capacity required by request r is equal to its traffic bandwidth. Eq. (B.4) ensures that, if request r is provisioned by node i and requires a VNF instance of type t , there must be some instance of VNF type t at that node. Eq. (B.5) indicates that every request will be provisioned by a single route, because traffic partitioning is not allowed. Eq. (B.6) ensures that the link capacity is not exceeded. Eq. (B.7) ensures that all requests will be provisioned. Eq. (B.8) ensures that VNF requests can only be provisioned by nodes that are part of the route that provision this request. In the Fat-Tree configuration, the VNFs can only be allocated on source and destination nodes, and thus Eq. (B.8) must be replaced by $w_{r,p} \leq y_{s(r),t,r} + y_{d(r),t,r}; \forall p, r, t : t \in r$, where $s(r)$ and $d(r)$ are the source and destination nodes from request r . Eq. (B.9) calculates the physical node processing load, where the first term of the sum is related to the processing load of the VNFs instances and the second term is related to transit traffic, which is zero for the Fat-Tree. Eq. (B.10) defines the upper limit of processing load for the physical nodes.

B.2 Tradeoff analysis based on port cost

The Fat-Tree topology offers good scalability at the cost of investing in switches. On the other hand, the Hypercube offers a higher number of links at server nodes (e.g., a 3D hypercube offers 3 links per node versus 1 link per node in the Fat-Tree), but consumes part of its processing capacity for forwarding traffic. Fig. B.2.a shows how the number

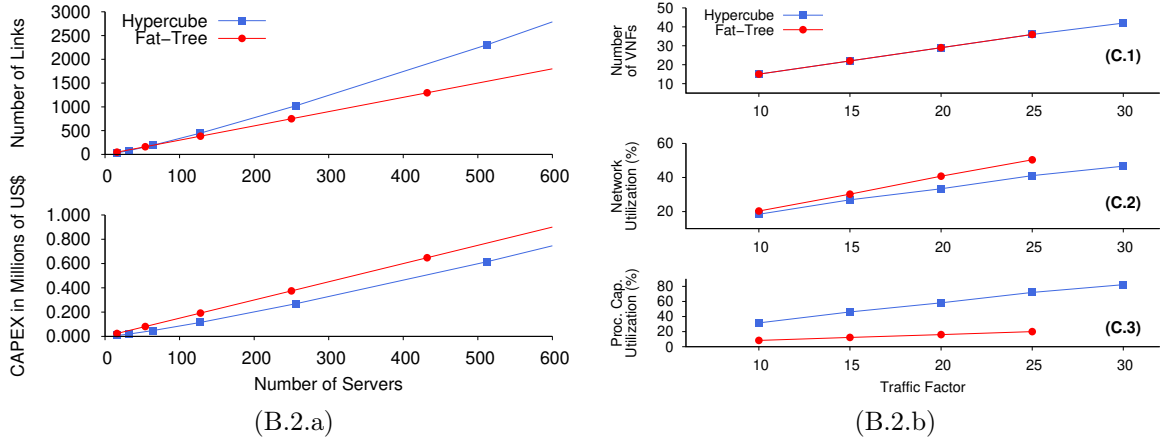


Figure B.2: (a) Comparison between the Fat-Tree and the Hypercube: No. of links vs. no. of servers, and CAPEX vs. no. of servers. (b) Results from ILP model: no. of provisioned VNF instances, network utilization, and processing capacity utilization vs. traffic factor.

of links and the CAPEX grow with the number of servers for both topologies, based on data from Herker et al.[Herker et al. 2015]. One should consider the fact that per-port cost in switches is roughly twice as much the cost of ports in network interface cards, for 10 GbE ports[Herker et al. 2015]. Hence, despite Hypercubes bearing more links, their overall costs can be considerably lower than Fat-Trees. In addition, the rich connectivity of Hypercubes may benefit complex NFV orchestration tasks.

B.3 Tradeoff evaluation for growing traffic demands

Our tests consider the Fat-Tree and the Hypercube topologies with 16 server nodes, which have the same resource profile (4Gbps processing capacity and 16 CPU cores) and are connected by 1Gbps links. The VNF profiles were based on Bari et al.[Bari et al. 2015] with the following types, number of CPUs, and processing capacities: Firewall (4 cores, 900Mbps), Proxy (4 cores, 900Mbps), NAT (2 cores, 900Mbps), and IDS (8 cores, 600Mbps). Each traffic request requires a service chaining of 3 VNF instances of different types (e.g., Firewall \rightarrow IDS \rightarrow Proxy).

To simulate the system response under increasing demand, we vary a traffic factor (tf) in six steps with the following values: 10, 15, 20, 25, 30, 35. The total traffic load of one step is represented by a set of 120 requests, which are individually calculated by randomly picking a baseline bandwidth (in Mbps) from 1, 2, 3, 4, 5 and multiplying it by the traffic factor of that step.

Fig. B.2.b summarizes the results obtained when minimizing the number of VNF instances. The first graph of Fig. B.2.b shows that VNF allocation presented similar results for both topologies, considering requests were provisioned by the same number of

VNFs instances for all levels of traffic.

Given that the connection between servers in the Fat-Tree is mediated by switches, the Fat-Tree link utilization is higher than in the Hypercube, as shown in the second graph of Fig. B.2.b. On the other hand, as Hypercube servers also process transit traffic, Hypercube processing capacity utilization is higher than in Fat-Tree, as shown in the third graph of Fig. B.2.b.

The link bottleneck in Fat-Tree was reached at the fifth step ($tf = 30$) when it was no longer able to provision the requests. Since the servers have only one connection in the Fat-Tree, the links saturate before the system reaches its maximum processing capacity. In contrast, since the servers in the Hypercube have more links, it did not reach the link bottleneck, but it was also only able to provision a solution up to $tf = 30$, due to its processing capacity limitation.

In conclusion, the Hypercube is able to deliver equivalent orchestration efficiency as the Fat-Tree at lower cost, provided its processing capacity bottleneck is respected. Alternatively, part of the switch cost elimination in the topology design could be re-invested in extra processing capacity in the Hypercube.

B.4 Analysis

This appendix presented some tradeoffs for the Fat-Tree and the Hypercube as representative topologies, which can help cloud operators on designing NFV solutions in edge DCs. Preliminary results indicated that server-centric architectures can support efficient NFV orchestration due to its high connectivity and lower CAPEX when compared to conventional network-centric topologies. In addition, there are some properties that can be explored, such as other objective functions, multiple shortest paths between nodes, native stateless routing schemes, and distributed optical switching [Dominicini et al. 2016, Cui et al. 2012]. These results can be further extended for other topologies with larger number of nodes.