Towards network softwarization : a modular approach for network control delegation

Hardik Soni

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Towards Network Softwarization: A Modular Approach for Network Control Delegation

Hardik SONI

INRIA

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE **d'**Université Côte d'Azur
Dirigée par : Thierry TURLETTI / Walid DABBOUS
Soutenue le : Avril 20, 2018

Devant le jury, composé de :
Hossam AFIFI, Professeur, Telecom Sud Paris
André-Luc Beylot, Professeur, IRIT/ENSEEIHT
Mathieu BOUET, Chercheur, Thales TCS
Giuseppe BIANCHI, Professeur, University of Roma
Guillaume URVOY-KELLER, Professeur, I3S/UNSA
Thierry TURLETTI, DR, Inria Sophia Antipolis
Walid DABBOUS, DR, Inria Sophia Antipolis

Inria
INVENTEURS DU MONDE NUMÉRIQUE

UNIVERSITÉ CÔTE D'AZUR

Inria INVENTEURS DU MONDE NUMÉRIQUE

École Doctorale STIC
Sciences et Technologies de l'Information et de la Communication
Centre de recherche: Inria (EPI DIANA)

Thèse de Doctorat

Présentée en vue de l'obtention du
grade de Docteur en Sciences

de
l'UNIVERSITÉ CÔTE D'AZUR
Mention : INFORMATIQUE

par

Hardik SONI

Une Approche Modulaire avec Délégation de contrôle pour les Réseaux Programmables

Dirigée par : Thierry TURLETTI
Walid DABBOUS

à soutenir le 20 avril 2018

devant le Jury Composé de:

Hossam AFIFI	-	Professeur, Telecom Sud Paris	*Rapporteur*
André-Luc BEYLOT	-	Professeur, IRIT/ENSEEIHT	*Rapporteur*
Mathieu BOUET	-	Chercheur, Thales TCS	*Rapporteur*
Giuseppe BIANCHI	-	Professor, University of Roma	*Examinateur*
Guillaume URVOY-KELLER	-	Professeur, I3S/UNSA	*Examinateur*
Thierry TURLETTI	-	DR Inria Sophia Antipolis	*Directeur*
Walid DABBOUS	-	DR Inria Sophia Antipolis	*Co-Directeur*

Acknowledgements

This thesis summarizes my research results during my Ph.D. study at the research team DIANA, Inria, France from 2014 to 2017. First, I would like to express my sincere gratitude to my supervisors Dr. Thierry Turletti and Dr. Walid Dabbous for trusting me with this thesis topic and their guidance, patience, constant encouragement, vision and immense knowledge. Their mentoring and support helped me to pass difficult moments of the research and raised confidence to address challenging research problems. It is my great pleasure to be their student.

I thank Dr. Hitoshi Asaeda from NICT, Japan for collaborating with me and providing valuable feedback during my thesis work. I convey my gratitude to Prof. Hossam AFIFI, Prof. André-Luc BEYLOT and Dr. Mathieu BOUET for their thorough review of my thesis and their valuable comments and suggestions, which have helped me to bring my thesis to its current form.

I am extremely grateful to the members of DIANA team for their support and interesting discussion on wide-range of topics, which helped me to broaden my perspectives. I specifically thank Christine Foggia, our team assistant, who helped me out with many bureaucratic tasks and contributed greatly to the smooth running of my PhD. Also, I thank Dr. Vikas Vikram Singh, Assistant Professor IIT Delhi for sharing his PhD student life experience and providing motivation.

Above all I thank my Mom, Dad, Kaka, Bhabhi, Dada, Dadi and my sisters Arpi and Rinku for their support, patience, understanding and tolerating my absence at back home in India on various occasions.

Une Approche Modulaire avec Délégation de contrôle pour les Réseaux Programmables

Résumé: Les opérateurs de réseau sont confrontés à de grands défis en termes de coût et de complexité pour intégrer les nouvelles technologies de communication (e.g., 4G, 5G, fibre optique) et pour répondre aux demandes croissantes des nouveaux services réseau adaptés aux nouveaux cas d'utilisation. La "softwarization" des opérations réseau à l'aide des paradigmes SDN (Software Defined Networking) et NFV (Network Function Virtualization) est en mesure de simplifier le contrôle et la gestion des réseaux et de fournir des services réseau de manière efficace. Les réseaux programmables SDN permettent de dissocier le plan de contrôle du plan de données et de centraliser le plan de contrôle pour simplifier la gestion du réseau et obtenir une vision globale. Cependant, ceci amène des problèmes de passage à l'échelle difficiles à résoudre. Par ailleurs, en dissociant la partie matérielle de la partie logicielle des routeurs, NFV permet d'implanter de manière flexible et à moindre coût toutes sortes de fonctions réseau. La contrepartie est une dégradation des performances due à l'implantation en logiciel des fonctions réseau qui sont déportées des routeurs.

Pour aborder les problèmes de passage à l'échelle et de performance des paradigmes SDN/NFV, nous proposons dans la première partie de la thèse, une architecture modulaire de gestion et de contrôle du réseau, dans laquelle le contrôleur SDN délègue une partie de ses responsabilités à des fonctions réseau spécifiques qui sont instanciées à des emplacements stratégiques de l'infrastructure réseau. Nous avons choisi un exemple d'application de streaming vidéo en direct (comme Facebook Live ou Periscope) utilisant un service de multicast IP car il illustre bien les problèmes de passage à l'échelle des réseaux programmables. Notre solution exploite les avantages du paradigme NFV pour résoudre le problème de scalabilité du plan de contrôle centralisé SDN en délégant le traitement du trafic de contrôle propre au service multicast à des fonctions réseau spécifiques (appelées MNF) implantées en logiciel et exécutées dans un environnement NFV localisé à la périphérie du réseau. Notre approche fournit une gestion flexible des groupes multicast qui passe à l'échelle. De plus, elle permet de bénéficier de la vision globale du contrôle centralisé apportée par SDN pour déployer de nouvelles politiques d'ingénierie du trafic comme L2BM (Lazy Load Balance Multicast) dans les réseaux de fournisseurs d'accès à Internet (FAI) programmables. L'évaluation de cette approche est délicate à mettre en œuvre car la communauté de recherche ne dispose pas facilement d'infrastructure SDN à grande échelle réaliste. Pour évaluer notre solution, nous avons élaboré l'outil DiG qui permet d'exploiter l'énorme quantité de ressources disponibles dans une grille de calcul, pour émuler facilement de tels environnements. DiG prend en compte les contraintes physiques (mémoire, CPU, capacité des liens) pour fournir un environnement d'évaluation réaliste et paramétrable avec des conditions contrôlées.

La solution que nous proposons délègue le contrôle et la gestion du réseau concernant le service de multicast aux fonctions spécifiques MNF exécutées dans un environnement NFV. Idéalement, pour davantage d'efficacité, toutes ces fonctions spécifiques devraient être implantées directement au sein des routeurs avec du hardware programmable mais cela nécessite que ces nouveaux routeurs puissent exécuter

de manière indépendante plusieurs fonctions réseau à la fois. Le langage de programmation P4 est une technologie prometteuse pour programmer le traitement des paquets de données dans les routeurs programmables (hardware et logiciels). Cependant, avec P4 il n'est pas possible d'exécuter sur le même routeur des programmes d'applications qui ont été développés et compilés de manière indépendante. Dans la deuxième partie de la thèse, nous proposons une approche originale pour résoudre ce problème. Cette solution, appelée P4Bricks, permet à des applications développées de manière indépendante de pouvoir contrôler et gérer leur trafic de données par le biais de routeurs hardware programmables.

Mots-clés: Réseaux Programmables, SDN, NFV, Multicast, Plan de données modulaire, P4, Composition du programme de réseau

Towards Network Softwarization: A Modular Approach for Network Control Delegation

Abstract: Network operators are facing great challenges in terms of cost and complexity in order to incorporate new communication technologies (e.g., 4G, 5G, fiber) and to keep up with increasing demands of new network services to address emerging use cases. Softwarizing the network operations using Software-Defined Networking (SDN) and Network Function Virtualization (NFV) paradigms can simplify control and management of networks and provide network services in a cost effective way. SDN decouples control and data traffic processing in the network and centralizes the control traffic processing to simplify the network management, but may face scalability issues due to the same reasons. NFV decouples hardware and software of network appliances for cost effective operations of network services, but faces performance degradation issues due to data traffic processing in software.

In order to address scalability and performance issues in SDN/NFV, we propose in the first part of the thesis, a *modular network control and management architecture*, in which the SDN controller delegates part of its responsibilities to specific network functions instantiated in network devices at strategic locations in the infrastructure. We have chosen to focus on a modern application using an IP multicast service for live video streaming applications (e.g., Facebook Live or Periscope) that illustrates well the SDN scalability problems. Our solution exploits benefits of the NFV paradigm to address the scalability issue of centralized SDN control plane by offloading processing of multicast service specific control traffic to Multicast Network Functions (MNFs) implemented in software and executed in NFV environment at the edge of the network. Our approach provides smart, flexible and scalable group management and leverages centralized control of SDN for Lazy Load Balance Multicast (L2BM) traffic engineering policy in software defined ISP networks. Evaluation of this approach is tricky, as real world SDN testbeds are costly and not easily available for the research community. So, we designed a tool that leverages the huge amount of resources available in the grid, to easily emulate such scenarios. Our tool, called DiG, takes into account the physical resources (memory, CPU, link capacity) constraints to provide a realistic evaluation environment with controlled conditions.

Our NFV-based approach requires multiple application specific functions (e.g., MNFs) to control and manage the network devices and process the related data traffic in an independent way. Ideally, these specific functions should be implemented directly on hardware programmable routers. In this case, new routers must be able to execute multiple independently developed programs. Packet-level programming language P4, one of the promising SDN-enabling technologies, allows applications to program their data traffic processing on P4 compatible network devices. In the second part of the thesis, we propose a novel approach to deploy and execute multiple independently developed and compiled applications programs on the same network device. This solution, called P4Bricks, allows multiple applications to control and manage their data traffic, independently. P4Bricks merges programmable blocks (parsers/deparsers and packet processing pipelines) of P4 programs according to processing semantics (parallel or sequential) provided at the time of deployment.

Keywords: Network Softwarization, SDN, NFV, Data plane programming, Multicast, Modular Data plane, P4, Network Program Composition

Contents

List of Figures

List of Abbreviations

SDN	**S**oftware-**D**efined **N**etworking
TSP	**T**elecommunication **S**ervices **P**rovider
ISP	**I**nternet **S**ervice **P**rovider
WAN	**W**ide-**A**rea **N**etwork
NFV	**N**etwork **F**unction **V**irtualization
COTS	**C**ommercial **O**ff-**T**he-**S**helf
UHD	**U**ltra **H**igh **D**efinition
IGMP	**I**nternet **G**roup **M**anagement **P**rotocol
PIM	**P**rotocol **I**ndependent **M**ulticast
MNF	**M**ulticast **N**etwork **F**unction
L2BM	**L**azy **L**oad **B**alance **M**ulticast
DiG	**D**ata-centers **i**n the **G**rid
NAT	**N**etwork **A**ddress **T**ranslation
SNMP	**S**imple **N**etwork **M**anagement **P**rotocol
ForCES	**W**hat **S**tands **F**or
POF	**P**rotocol **O**blivious **F**orwarding
P4	**P**rogramming **P**rotocol-Independent **P**acket **P**rocessors
RMT	**R**econfigurable **M**atch **T**ables
API	**A**pplication **P**rogramming **I**nterface
TLS	**T**ransport **L**ayer **S**ecurity
SSL	**S**ecure **S**ocket **L**ayer
HSDN	**W**hat **S**oftware-**D**efined **N**etworking
ECMP	**E**qual-**c**ost **M**ulti-**p**ath
TE	**T**raffic **E**ngineering
SDPA	**S**tateful **D**ata **P**lane **A**rchitecture
FSM	**F**inite **S**tate **M**achines
(XFSM)	e**X**tended **F**inite **S**tate **M**achines
NIB	**N**etwork **I**nformation **B**ase
CAPEX	**Cap**ital **Ex**penditure
OPEX	**Op**erational **Ex**penditure
IT	**I**nformation **T**echnology
IDS	**I**ntrusion **D**etection **S**ystem
IPS	**I**ntrusion **P**revention **S**ystem
OVS	**O**pen **vS**witch
DPDK	**D**ata **P**lane **D**evelopment **K**it
VPP	**V**ector **P**acket **P**rocessing
NIC	**N**etwork **I**nterface **C**ard
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
VM	**V**irtual **M**achine
NF	**N**etwork **F**unction

IP	**I**nternet **P**rotocol
QoS	**Q**uality-**o**f-**S**ervice
IPTV	**I**nternet **P**rotocol **T**elevision
NFVI-PoPs	**NFV I**nfrastructure **P**oint **o**f **P**resences
NFV-based COs	**NFV-based C**entral **O**ffices
PIM-SM	**P**rotocol **I**ndependent **M**ulticast - **S**parse-**M**ode
VNF	**V**irtual **N**etwork **F**unction
NC	**N**etwork **C**ontroller
SIP	**S**ession **I**nitiation **P**rotocol
SAP	**S**ession **A**nnouncement **P**rotocol
MSDP	**M**ulticast **S**ource **D**iscovery **P**rotocol
BFS	**B**readth **F**irst **S**earch
OF	**O**pen**F**low
OVSDB	**O**pen **vS**witch **D**atab**b**ase
DST	**D**ynamic **S**teiner **T**ree
DST-PL	**D**ynamic **S**teiner **T**ree **P**ath-**L**ength
DST-LU	**D**ynamic **S**teiner **T**ree **L**ink **U**tilization
NNFDAR	**N**earest **N**ode **F**irst **D**ijkstra **A**lgorithm considering **R**esidual capacities
SD	**S**tandard **D**efinition
HD	**H**igh **D**efinition
XMPP	E**x**tensible **M**essaging and **P**resence **P**rotocol
OTT	**O**ver-**t**he-**T**op
SDM	**S**oftware **D**efined **M**ulticast
DVMRP	**D**istance **V**ector **M**ulticast **R**outing **P**rotocol
MOSPF	**M**ulticast **O**pen **S**hortest **P**ath **F**irst
L2TPv3	**L**ayer2 **T**unnelling **P**rotocol **V**ersion **3**
VNE	**V**irtual **N**etwork **E**mbedding
MAT	**M**atch-**A**ction **T**able
DAG	**D**irected **A**cyclic **G**raph
IR	**I**ntermediate-**R**epresentation
CFG	**C**ontrol **F**low **G**raph
ID	**ID**entifier
UID	**U**nique **ID**entifier
UIDT	**U**nique **ID**entifier **T**able
OS	**O**peration **S**chedule
OSG	**O**peration **S**chedule **G**raph
OWO	**O**ut-of-order **W**rite **O**perations

1 Overview

Contents

1.1 SDN and NFV enabling Network Softwarization

Over the past decade, the Software Defined Networking (SDN) paradigm has shown a great potential in simplifying control and management of networks compared to traditional distributed networking. It has facilitated rapid transition of research innovations into production to address ever changing requirements of various types of networks such as data centers, Internet Service Providers (ISPs), Telecommunication Service Providers (TSPs), enterprise and wide-area networks. For instance, Google has deployed software defined Wide-area network (WAN) (B4 [Jain *et al.* 2013]) connecting its data center to achieve its traffic engineering requirement at low cost and reduced over-provisioning of network link capacities. However, while SDN provides simplified network control and management due to its architectural principles, it still faces several challenges. Separation of control and data traffic processing and centralization of the control are the core principles of SDN (See Figure 1.1). B4 has successfully leveraged one of the core principles of SDN, the centralized control over the forwarding devices in the network, to engineer its traffic requirements. However, the same centralized control poses scalability challenges for the network control applications and functions requiring fine-grained control over frequent network control events (e.g., flow arrival, statistics collection) in Mahout [Curtis *et al.* 2011a] and DevoFlow [Jain *et al.* 2013]. Numerous proposals studying pros and cons of SDN principles on different network control applications like routing, traffic engineering, statistics collector, in-network load-balancing and multicast have helped to evolve SDN as a promising approach for easy network control and management [Yeganeh *et al.* 2013, Bifulco *et al.* 2016].

In parallel to the SDN paradigm for simplified network control and management, Network Function Virtualization (NFV) [Chiosi *et al.* 2012] has emerged as a flexible and cost-effective way of accelerating the deployment of new network services to

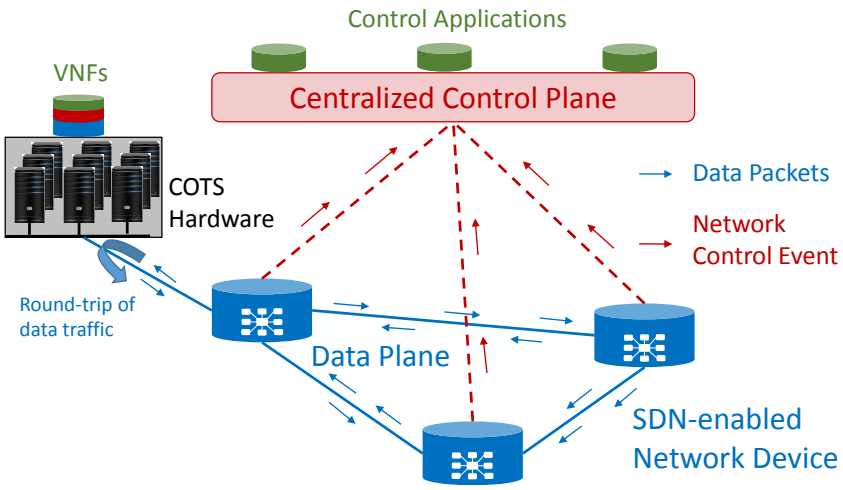


Figure 1.1: Network Softwarization using SDN/NFV

mitigate demanding and rapidly changing service requirements from network end-users and business use-cases. Deploying proprietary and specialized hardware with dedicated software for the required packet processing to enable new network services incurs a heavy cost, huge production delays and hinders innovations. The NFV paradigm separates packet processing hardware from its software and advocates the use of IT standard Consumer-Off-The-Shelf (COTS) hardware located at strategic places in the network. This helps in accelerating the deployment of new network services in a cost effective way and provides the needed agility to service providers for innovations. Indeed, services can be developed, deployed, test and integrated using software only without developing costly ASIC based hardware. Technological evolution in virtualization has enabled packet processing in software using pool of cheap COTS hardware resources. However, although performance metrics of packet processing in software are encouraging, they are still not comparable with native processing using proprietary hardware. NFV shifts in-network data traffic processing functions into virtualized environment. Consequently, line rate data traffic is required to make a round-trip to the COTS hardware resources that execute Virtual Network Functions(VNFs), see Figure 1.1. Basically, NFV trades packet processing performance of network services for their flexible and cost-effective deployment.

SDN provides softwarized control over the forwarding devices in the network by allowing to modify their behavior from outside the devices. The NFV paradigm advocates to rely only on network function specific software and use general purpose COTS compute hardware to deploy the software, either in virtualized or native environments. These together have enable software programs to transform network operations according to demands of network services, enabling *Network Softwarization*.

1.2 Issues with Network Softwarization

SDN and NFV both aim to accelerate the deployment of innovative ideas and new network services. Even though independently conceptualized, SDN/NFV together

provide cost-effective approach for simplified network management and network service deployment by softwarizing the network. Particularly, SDN is used for providing connectivity and controlling the forwarding devices, whereas NFV is used for more complex data traffic processing than merely forwarding, as shown in Figure 1.1.

However, the centralized SDN control plane suffers from scalability issues due to control applications processing frequent network events in the control plane. Section 2.2 describes various control and data plane enhancements proposed in the literature to address these issues. New control plane architectures (distributed, hierarchical and hybrid) are proposed to scale the processing of frequent network events while maintaining logically centralized abstractions. However, these approaches increase complexity of the control plane (e.g., consistency issue [Reitblatt *et al.* 2011,Kuzniar *et al.* 2014], restricted global network view, high response time to network events in data plane, controller placement issue [Heller *et al.* 2012], etc.) and do not provide the required level of flexibility to scale according to requirements of control application and function. Also, deploying a control application that processes frequent network events using these architectures still poses scalability threats to all other applications. Indeed, when an application processes frequent network events in the control plane, it congests the control plane and thereby, can affect performance of the other applications. Using a stateful data plane can allow forwarding devices to maintain the required local state for local processing of frequent network events. However, it is difficult to estimate the scale of required memory to store and manage the elaborate state information in the forwarding devices. Hence, a stateful data plane does not provide enough flexibility to scale according to the network service and traffic demands.

In the NFV paradigm, the data traffic is steered to the COTS compute hardware resources located in the network to be processed by VNFs. This degrades the overall performance of the network due to round trip of line rate data traffic outside the SDN data plane and its processing in software, without customized hardware.

In this thesis, we propose a modular network control and management architecture to address (1) the scalability issue of the centralized SDN control plane and (2) the performance degradation issue of the NFV paradigm.

1.3 Thesis Proposal: Modular Network Control and Management

In order to address the scalability issue of the centralized SDN control plane, we delegate the applications specific network control traffic processing to the functions running in the virtualized environment. We allow VNFs to control and manage traffic pertaining to the applications. So, VNFs executed on COTS computer hardware can push their line rate data traffic processing in the forwarding devices of SDN data plane. This approach requires ***multiple*** independently developed network control applications and functions to allow control and management of data traffic and forwarding devices in the network. Our proposal keeps maximum processing of data traffic in the SDN data plane for VNFs, thereby saving the round trip to COTS

compute hardware and uses hardware of forwarding devices to process data traffic at line rate.

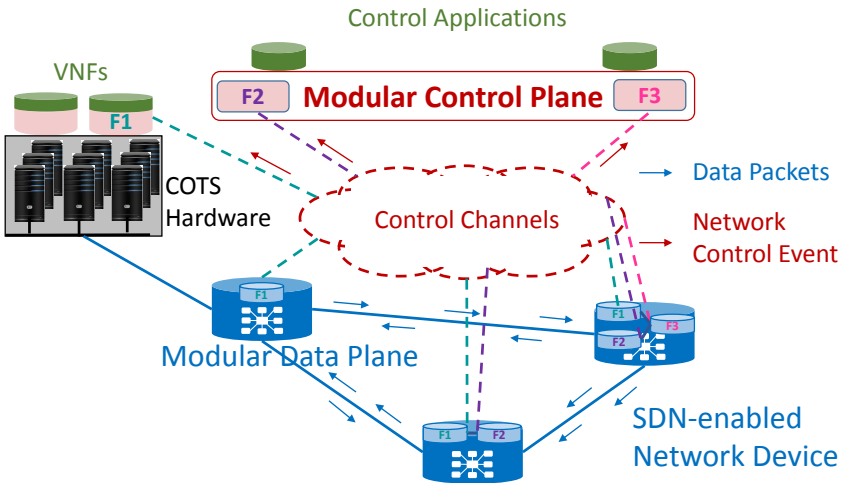


Figure 1.2: Modular Network Control and Management using SDN/NFV

Figure 1.2 depicts the architecture of modular network control and management. A modular control plane allows the deployment of multiple independently written control applications or functions (e.g., *F2* and *F3* shown in Figure 1.2) to process shared data traffic in the network without a centralized hypervisor (e.g., CoVisor [Jin *et al.* 2015]) composing the applications and functions. In addition, it facilitates control delegation, where control functions pertaining to specific applications and traffic can be executed in NFV environment. For example, control function *F1* processes control traffic using COTS compute hardware in the virtualized environment provided by NFV. Modular network control plane allows each application and function to define its own control data plane interface. Each application can receive only specified network events to process without interfering processing of other control applications. The computational power for control traffic processing can be flexibly scaled on demand by leveraging the NFV paradigm.

Current SDN data plane architectures provide exclusive control of forwarding devices to the monolithic centralized control plane. However, some SDN-enabling technologies, under special configuration, can allow control and management of the same forwarding device from multiple controlling agents.[1] In such a case, all the agents can modify data traffic processing behavior of the device configured by any agent. This may result in nondeterministic and inconsistent data traffic processing by the forwarding device. Hence, to deploy modular network control and management, it is imperative that each control application or function be able to add and remove its data traffic processing functionality in the shared forwarding devices with a processing semantic. Network operators should be able to define the resultant processing behavior of the device by composing combining packet processing functionalities of all the applications and functions using different composition semantics. SDN-enabled

[1]Multiple OpenFlow-enabled SDN controller can control and manage the same OpenFlow-enabled forwarding device using EQUAL mode.

network devices must be able to be programmed, controlled and managed by ***multiple*** control applications and functions. Hence, modularity in the SDN data plane becomes essential to perform modular network control delegation. We propose a modular SDN data plane, where each application can independently specify its data traffic processing behavior in the SDN data plane. For example, *F1*, *F2* and *F3* can specify their individual data traffic processing behavior to the required forwarding devices in the network. Those forwarding devices are controlled and managed in a simultaneous way by multiple control applications and functions. They allow to compose packet processing functionalities of all the applications and functions using composition semantics like sequential (*F1* followed by *F2*) or Parallel (simultaneous packet processing using *F1* and *F2*).

We consider that *modularized network control and management* is paramount to realize *softwarization of networks*. It may be perceived that modularized network control and management is indeed a distribution control plane approach, but we argue that a distributed control plane splits the network control from a centralized abstraction. On the other hand, our modularized network control and management approach splits the network control from forwarding devices in the SDN data plane and still maintains a centralized abstraction for control applications and functions.

Next, we outline the thesis structure along with our contributions.

1.4 Thesis Outline

In Chapter 2, we introduce network softwarization. First we describe basic tenets of the SDN paradigm and its evolution in the past decade in Section 2.1 including technologies enabling SDN paradigm in Section 2.1.1. Then in Section 2.2, we discuss about various control architectures to implement network services (e.g., IP Multicast) for SDN enabled networks. Next, we briefly introduce notions of NFV in Section 2.3 along with its enabler technologies. We discuss amalgamation of SDN, NFV and Network Programming in Section 2.4. The rest of the thesis comprises of two parts: (I) Modular Approach for Network Control Delegation and (II) Modular SDN Data Plane Architecture.

1.4.1 Part I: Modular Approach for Network Control Delegation

In the first part of the thesis, our primary proposal is to offload application specific control traffic to a function running in a virtualized environment. In Chapter 3, we employ this application specific control delegation to provide flexible scaling capability to IP multicast services.

We have selected live high quality video streaming as one of the network services requiring agile deployment and flexible scaling capability to satisfy its traffic demands. Indeed, the massive increase of live video traffic on the Internet and the advent of High Definition (HD), Ultra High Definition (UHD) videos have placed a great strain on ISP networks. IP Multicast is a well-known technology, which can be used to implement flexible, bandwidth-efficient and scalable multicast delivery

services. However, multicast has failed to achieve Internet-wide support so far [Diot *et al.* 2000]. Even for the limited deployment in managed networks for services such as IPTV, it requires complex integration of specialized multicast enabled routers and protocols, traffic engineering and QoS mechanisms in the networks. This increases the complexity of network management along with its associated cost due to the need of specialized routers in the infrastructure.

In Chapter 3, we use network softwarization while modularizing network control to deploy IP multicast services in ISP networks for live video streaming applications (e.g., Facebook Live and Periscope). Multicast routers need to maintain multicast group membership specific state in their memory, execute group membership management protocols (e.g., Internet Group Management Protocol (IGMP) and Protocol Independent Multicast (PIM)) and appropriately replicate IP packets in the network to deliver them from a single source to multiple recipients. We use the NFV approach and separate hardware and software of the specialized routers by (1) delegating the multicast group membership management functionality to the functions implemented in software, called Multicast Network Functions (MNFs), and (2) using SDN enabled network devices to achieve high performance packet forwarding for live video stream network services, Essentially, we delegate application specific network control and state management to dedicated function running in NFV environment. Use of specialized network functions to process group membership management messages addresses the scalability issue emerging from centralized control plane of the SDN architecture. This eliminates requirement of specialized multicast routers and provides scaling flexibility in deploying the service in a cost effective way by using SDN enabled network devices only. Use of NFV provides dynamic, flexible and cost effective way to scale IP multicast capability of the network based on traffic demands.

Also, we propose a traffic engineering mechanism, Lazy Load Balancing Multicast (L2BM) [Soni *et al.* 2017], to provide bandwidth guarantee to packet streams of live videos, which leverages simplified network control and management facilitated by the SDN paradigm. Providing bandwidth guarantee to network services requires continuous statistics collection of network's links utilization and network resource sharing between the network services and best-effort traffic in the network. Statistics collection is costly and may generate frequent network events depending on variations in traffic. Our L2BM proposal addresses this issue thanks to a global network view provided by the SDN paradigm. We evaluate L2BM by simulating complex real world traffic scenarios in ISP core networks for live video streaming network services and also by emulating an ISP core network with resource guarantee using an automation tool we designed, called Data centers in the Grid (DiG) [Soni *et al.* 2015].

In Chapter 4, we present the DiG tool, which emulates SDN-enabled networks with resource guarantee. Most SDN enabled network experiments are performed with traffic traces using emulators (e.g., Mininet [Lantz *et al.* 2010], Maxinet [Wette *et al.* 2014]) or simulators (e.g. ns-3 [Consortium 2008]) due to restricted access to real SDN enabled networks. Therefore, experiment results may be biased or noisy due to modeling techniques of simulators or unaccounted and excessive usage of physical resources in case of emulation(Figure 4 in [Tazaki *et al.* 2013]). DiG employs virtualization technologies used for NFV to emulate SDN enabled network topologies with guaranteed resource allocation on parallel and distributed computing

infrastructure like Grid'5000 [Balouek *et al.* 2013]. It emulates network links of given capacity, hosts with demanded CPU, memory and disk I/O bandwidth guarantee, while respecting constraints on the available physical resources present in the grid. We use DiG to emulate an ISP core network and evaluate L2BM with IP multicast traffic.

Our contributions related to the first part are following.

1. H. Soni, W. Dabbous, T. Turletti and H. Asaeda, "NFV-Based Scalable Guaran-teed-Bandwidth Multicast Service for Software Defined ISP Networks," in *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1157-1170, Dec. 2017. doi: 10.1109/TNSM.2017.2759167
2. H. Soni, W. Dabbous, T. Turletti and H. Asaeda, "Scalable guaranteed-bandwidth multicast service in software defined ISP networks," *2017 IEEE International Conference on Communications (ICC)*, Paris, 2017, pp. 1-7. doi: 10.1109/ICC-.2017.7996652
3. H. Soni, D. Saucez and T. Turletti, "DiG: Data-centers in the Grid," *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, San Francisco, CA, 2015, pp. 4-6. doi: 10.1109/NFV-SDN.2015.7387391

1.4.2 Part II: Modular SDN Data Plane Architecture

In our NFV-based proposal (Part I, Chapter 3), MNFs process multicast group membership management related control traffic in the ISP network and manage forwarding devices only for some specific traffic to deploy live video streaming network services. Meanwhile, the rest of the control traffic in the network is processed by a centralized SDN controller, which also manages all the forwarding devices in the network. This delegation of application specific network control requires multiple control applications to manage forwarding devices in the SDN enabled network. Hence, modularized control and management of networks is required to offload application specific control traffic by delegating network control to its NF. Also, it allows data packet processing VNFs to maximize the number of packets processed within the SDN data plane for performance benefits. The functionality based on modularized network control requires multiple control applications and network functions to manage forwarding devices in the SDN enabled network.

To achieve modularized deployment, control and management of control applications and functions in the network, it is mandatory to allow every control application and function independently program datapath of the devices and directly manage network devices according to its packet processing requirements. Also, every control application and function should have dedicated communication channel and control interface to control and manage its programmed datapath.

In the second part of the thesis, we describe a modular SDN data plane. Using the packet processing P4 [Bosshart *et al.* 2014] language, control applications and

functions can program the datapath and define the communication channel for forwarding devices in the data plane. P4 provides higher flexibility to program functionalities for programmable network devices. It allows to program extraction of protocol headers (using programmable parsers/deparsers), processing of headers and their reassembly to form a packet. It enables a network control application or function to control and manage the devices directly using the communication interface defined in the program. Particularly, P4 brings higher programmability and flexibility in the SDN paradigm along with simplified network control and management. However, it usually requires to describe all packet processing functionalities for a given programmable network device within a single program. This approach monopolizes the device by a single large program, which prevents possible addition of new functionalities by other independently written network control applications and services. However, if multiple independently written P4 programs are executed on the same forwarding devices, a modular deployment along with control and management of packet processing behavior can be done. In Chapter 5, we present the design of P4Bricks to enable modularized deployment, control and management of independently written control applications and functions. It allows to execute multiple independently developed P4 programs of network control applications and services on the same network device according to their packet processing requirements.

Finally, in Chapter 6, we conclude our proposal and discuss future research directions and avenues opened by modular delegation of network control.

Our contributions related to the second part is following.

1. Hardik Soni, Thierry Turletti, Walid Dabbous. P4Bricks: Enabling multi-processing using Linker-based network data plane architecture. HAL Report, 2018. (hal-01632431v2)

2 Introduction to Network Softwarization

Contents

2.1 Software Defined Networking

In the traditional distributed networking paradigm, independent and self-governing devices are connected together to share, forward and transport data among each other. These network devices are custom built to perform one or more functions in the network such as routing, firewall and Network Address Translation (NAT), as shown in Figure 2.1. The network devices are managed and configured using protocols like Simple Network Management Protocol (SNMP) [Schönwälder 2008] and NETCONF [Enns *et al.* 2011]. However, they autonomously control the function specific packet processing behavior according to control information exchanged using the distributed protocols and algorithms. Indeed, network devices process data packets based on autonomously and intelligently gathered control and management information from multiple sources with different mechanisms. The software and hardware devices residing in the paths of the control information exchange constitute the control plane of the network and those residing in the paths of the data packets form the data plane. In the distributed networking paradigm, the independent and self-governing devices pose various control and management challenges. They are costly and customized for intelligent protocols, packet processing and algorithms

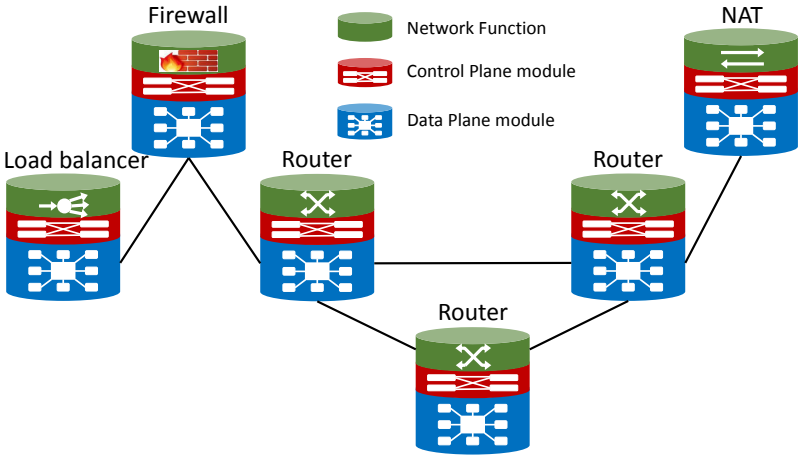


Figure 2.1: Example of distributed networking paradigm

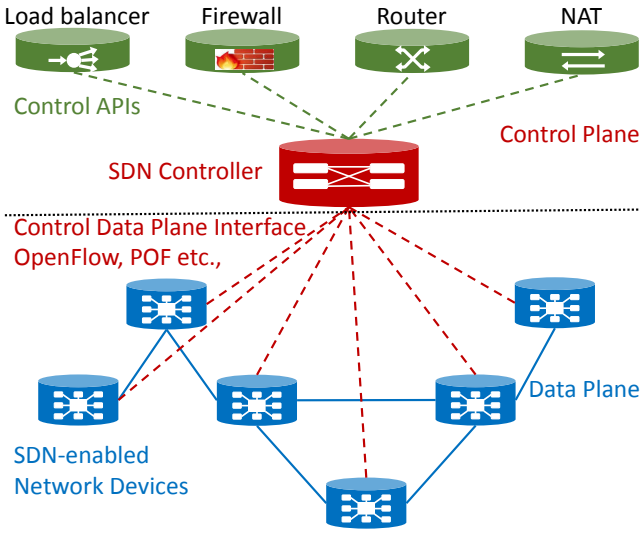


Figure 2.2: Example of SDN paradigm

with propriety hardware as well software. This makes maintenance, debugging and management of different types of independent devices complex for network operators. Once the devices are deployed, their packet processing behavior can not be changed to implement new solutions. Moreover, deploying innovative solutions for new requirements face long production delays and huge development cost of hardware and software of network devices, making them *closed*.

The SDN paradigm addresses the control and management challenges of network devices by separating the intelligent control plane from the data plane of software and hardware network devices (see Figure 2.2). It keeps only fundamental packet processing abstractions in network's data plane devices, which is used to program the required packet processing in the network by the control plane devices. Data plane devices are built based on fundamental abstractions of packet processing defined

by the SDN enabling technology for the data plane devices, like OpenFlow [McKeown *et al.* 2008], ForCES (Forwarding and Control Element Separation) [Halpern *et al.* 2010], POF (Protocol-oblivious Forwarding) [Song 2013] and P4 [Bosshart *et al.* 2014]. In the SDN terminology, data plane devices are often referred as *forwarding devices*. The packet processing behavior of data plane devices are controlled by a logically centralized control plane of the network. The centralized control plane controls all the devices in the network's data plane, essentially providing a global network view to simplify control and management of the network and its data traffic. The entity implementing the logically centralized control plane is generally termed as the *SDN controller*. A SDN controller provides control APIs to network functions such as NAT or firewall to control their packet processing behavior in the network. The SDN control plane uses configuration and management protocols and technologies (e.g., OpenFlow, ForCES, POF), which provides standardized open interfaces to manage devices in the data plane and control their packet processing behavior. Separation of control and data plane along with open interfaces to communicate between them makes the data plane network devices *open* to modify their packet processing behavior, programmatically even after their deployment. P4 is a programming language for protocol-independent packet processing, which can program reconfigurable forwarding devices (e.g., RMT [Bosshart *et al.* 2013b], Intel FlexPipe). It allows each program to define its own communication interface between control and data plane to manage and control the programmed packet processing behavior of the device. P4 extends the SDN principles by *opening* network devices in order not only to control their packet processing behavior, but also to program them. Specifically, it allows to write packet parsing in order to extract the required fields, process the parsed fields and reassemble the packet using the processed fields.

2.1.1 SDN Enabling Technologies

Even though principles of the SDN paradigm have gained more traction with the advent of OpenFlow [McKeown *et al.* 2008], ForCES has used the separation of control and data plane since 2003 to simplify the internal architecture of network devices. However, ForCES does not centralize the control plane of all the devices in the network. ForCES merely separates the control plane from the data plane to enable modular development of network devices, which allows forwarding hardware residing in the data plane to be combined with any control logic software with ease and flexibility. The ForCES approach keeps the control plane of the device in close vicinity of its forwarding element or in the same physical hardware. Meanwhile, SDN with OpenFlow not only separates the control plane of the devices but centralizes it at the network controller. It renders network devices with only their data plane element implementing OpenFlow's packet processing abstractions. OpenFlow based network controllers implement a centralized control plane and can be located anywhere in the network. POF enhances the OpenFlow's packet processing abstractions, which provides better flexibility in controlling and managing data plane elements. We briefly introduce OpenFlow and POF, as they have become widely known SDN enabling technology in recent years, attracting major attention from industry as well as academia.

2.1.1.1 OpenFlow

With the distributed networking paradigm, every proprietary network device built for specific network function implements some form of flow-tables and uses similar primitive functions to process packets at line rate. A flow is a set of packets with matching values for the given set of fields in each packet headers and a flow-table consists of field values for different flows and corresponding functions to process the packet. OpenFlow, in its simplest form, provides flow-table based packet processing abstraction for OpenFlow-enabled forwarding devices. It implements the identified common set of packet processing functions in the devices. Also, OpenFlow provides an open and standardized protocol to manage the flow-tables from control plane devices, specifically the SDN controllers. SDN controllers communicate with the forwarding devices with a secure channel using this open protocol.

OpenFlow-enabled forwarding devices consist of three components: Flow-Tables, the OpenFlow agent and a secure channel (see Figure 2.3). Flow-tables and a common set of packet processing functions reside in the datapath of the devices, which processes packets at line rate and is generally implemented in hardware. The OpenFlow protocol agent and the secure channel are located in the control path of the devices, implemented in software. A flow is defined using a set of fields along with values for each field to match against the ones presents in packets. An entry in flow-table com-

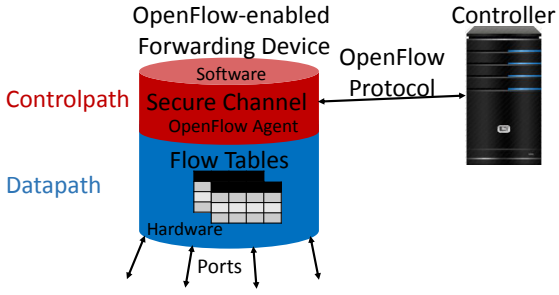


Figure 2.3: Architecture of OpenFlow-enabled forwarding device

prises of three fields: (1) Flow matching rule to match packets, (2) Action functions associated to the flow match rule and (3) Statistics to record packet and byte counts associated with the flow along with time elapsed since last match. A flow matching rule can have packet header fields, hardware specific fields (e.g., ingress port) and metadata fields as match fields. Action functions in the entry are a subset of the common set of packet processing functions used to process the matched packet. However, OpenFlow-enabled forwarding devices are not limited to these common set of functions and can process packets using extended action functions. On a packet arrival, datapath of the device parses the packet to extract values for header fields of various protocols such as Ethernet, IPv4, IPv6, TCP, UDP, VLAN or VXLAN. It performs look-up in a flow-table using the matching rule to find an entry corresponding to the extracted values of packet header fields along with metadata and hardware specific field values. If there is a table-hit as a result of the look-up, this means that a matching flow-entry is found and so, the datapath executes the action functions associated with the entry. Otherwise, it executes actions defined by the *table-miss*

flow entry for the table. Every table must have actions defined to handle a table-miss; the actions could be for instance sending the packet to controller, dropping the packet or continue the look-up with other flow tables. Packet processing abstractions of OpenFlow datapath have evolved considerably since their inception in order to incorporate the needs of requirement of various network functions. OpenFlow version 1.1 and higher provide an abstraction for flow-table pipeline processing of packets, in which multiple flow-tables can be programmed along with their execution order to process a packet.

OpenFlow-enabled forwarding devices and the controller communicate using the OpenFlow protocol on a secure transport channel (e.g., Secure Sockets Layer (SSL) or Transport Layer Security (TLS)). Hence, transport layer connectivity between the controller and forwarding devices are prerequisite to exchange OpenFlow messages. The SDN controller is responsible for providing connectivity among forwarding devices in the data plane. If the controller resides within the network of data plane elements, it is referred as in-band controller. In such a scenario, separate distributed protocol or static configuration is required to establish initial connectivity between the controller and the forwarding devices. If the controller resides in a separate network, referred as out-of-band controller, each forwarding device uses a static configuration to establish transport connectivity to the controller.

The OpenFlow protocol messages are classified in three major types, *Controller-to-switch*, *Asynchronous* and *Symmetric*. Controller-to-switch messages aim to manage flow-tables, entries within the flow-tables and interfaces, to retrieve statistics and to inspect other state information of the device. Asynchronous messages originated from the device are used to notify the controller for events like new packet or table-miss in datapath, flow-entry expiry and interface up-down. Symmetric messages are triggered by either the device or the controller and sent without specific request from the other side. They provide basic utilities for peer connection (e.g., message exchange on connection setup, check liveness, notify failure and notify experimental capability to peer). Also, the OpenFlow protocol allows to extend messages to manage and configure extended packet processing abstraction (e.g., action functions) supported by the device.

Using OpenFlow, a controller can connect to OpenFlow-enabled forwarding devices in three roles: *Master*, *Slave* and *Equal*. There can be only one master controller for each forwarding device in the network, which is allowed to control, manage and program the datapath of the device. However, a device can have multiple slave controllers, which receive all the network events and have read-only access to datapath of the device. Controllers with equal role can simultaneously control, manage and program the datapath of the same forwarding device. A forwarding device with a master controller can not have another controller with equal role.

Flow-tables in the datapath of OpenFlow follow match+action based packet processing. Datapath of OpenFlow-enabled forwarding devices are required to support parsing of header fields of every possible protocol with their header types to facilitate creation of any flow match rule. Indeed, this hinders programming of datapath from the controller and restricts packet processing to only supported header types by the device. OpenFlow does not allow to maintain application specific per packet state

information in the forwarding devices. P4 [Bosshart *et al.* 2014] and POF extend OpenFlow to mitigate these limitations and provide more flexible packet processing abstraction for enhanced programming, configuration of datapath.

2.1.1.2 Protocol-oblivious Forwarding (POF)

POF extends OpenFlow packet processing abstractions by introducing three new capabilities in the datapath of the forwarding devices. First, POF removes dependency on protocol-specific configuration of flow match rules of flow-tables in the datapath of forwarding devices. It uses the {offset, length} pair to locate data in the bitstream of the packet. Second, POF makes agnostics all packet processing action functions in the datapath protocol header fields, by defining a generic flow instruction set. Third, it allows to store and manage flow metadata, which has lifespan of the entire flow in the forwarding devices. It can store state information for a flow and use it to process the packet of the flow throughout its lifetime.

2.1.1.3 Programming Protocol-independent Packet Processors (P4) in brief

P4 is a high-level language, based on programmable blocks, which is used for programming protocol-independent packet processors. The programmable parser, deparser and logical match-action pipeline are the main programmable data plane blocks. Parser blocks are programmed by defining packet header types and fields, declaring instances of the types and defining Finite State Machine (FSM) to extract the header instances from the packets' bit streams. Match-action units are defined as tables (MATs) using match keys and actions. Match keys can be header fields or program's state variables and actions may modify and create header instances, their fields and hardware target specific fields. To create the control flow required for packet processing, programmers can use *if* and *switch* constructs to select next tables to process based on processing logic. Finally, packets are reassembled using the processed header instances by programming the deparser control block. Apart from programmable blocks, P4 provides APIs and a control interface to manage flow entries in MATs at runtime. P4 is described in more detail in Section 5.2.1

2.1.1.4 SDN enabling tools and software

The first specification for OpenFlow v1.0 was drafted in year 2008. Since then SDN has significantly changed thanks to open source as well industrial efforts and more importantly huge development efforts to build tools to support the SDN ecosystem. SDN controller and SDN-enabled forwarding devices (switches) supporting OpenFlow protocol have been major focus of development for industry and academic research communities. Here, we provide only a highlight of development efforts to build tools and software enabling SDN. NOX [Nicira Networks 2008], C++ based, and its python counterpart POX were early SDN controller developed to program OpenFlow 1.0 enabled switches. Beacon [Erickson 2013] written in Java and Ruby

based Trema [Takamiya *et al.* 2017] are result of some more development effort for SDN controller platforms. However, Ryu [Ryu SDN Framework Community 2014], Floodlight [Big Switch Networks 2013], ONOS [The ONOS Community 2014] and OpenDayLight [The OpenDaylight Foundation 2013] are widely adopted and have been able to continuously evolve with advances in SDN technologies. In Chapter 3, we use Floodlight for implementing our proposed L2BM traffic engineering policy.

Apart from controllers, OpenFlow-enabled forwarding devices are necessary to realize the SDN paradigm. Among many open source projects, Open vSwitch [Pfaff *et al.* 2015] has been widely used and adopted by the community. It is based on the Linux bridge legacy source code and has evolved with OpenFLow specification versions by including new features. Moreover, it supports many other protocols and features to support traditional distributed networking.

2.2 SDN Control and Data Plane Architectures

Network control applications and functions managing specific networks (e.g., data centers with new intensive traffic flow [Benson *et al.* 2010]) or requiring fine-grained control over frequent network events incur heavy load on the controller [Curtis *et al.* 2011b]. The reason is that the centralized SDN architecture concentrates the control traffic generated by such network events at the controller, which is implemented in software and has limited compute power. The control and management of concentrated control traffic depends on functional characteristics of applications implemented using the centralized controller. Applications like elephant flow detection (requiring statistics collection), stateful firewall, DNS-tunnel detection etc., generate frequent network events, which can be handled and processed without requiring global network information. However, network functions like routing requires a global network view as routing decisions are made considering network topology and links information. Next we discuss various control and data plane architectures proposed in the literature to scale network control applications and functions considering their control requirement over network events.

2.2.1 Hierarchical Control Plane

Kandoo [Yeganeh & Ganjali 2012] and Hierarchical SDN (HSDN) [Fang *et al.* 2015] structure control and data plane elements in hierarchy to address scalability issues emerging from fine-grained control over network.

The Kandoo framework employs two layers of controllers and distinguish between global and local network control applications. Local controllers at the bottom layer does not possess the global network view and processes frequent events that do not require network-wide state information but can be handled using forwarding device specific local states. Local controllers reside near to forwarding devices in data plane and they execute local control applications. The root controller at the top layer is logically centralized; it maintains the global network view and controls the local controllers. It solicits local controllers to relay network events pertaining to

global control applications, which require network-wide state information to process. Kandoo uses framework specific messages to communicate between global and local controllers. Kandoo's logically centralized root controller offloads processing of frequent network events to local controllers for local control applications requiring fine-grained control over network (e.g., elephant flow detection). The bottom layer controller maintains the state information required to process frequent network events for local control applications and controls one or more forwarding devices using the OpenFlow protocol. Essentially, Kandoo uses the local controllers to maintain the state information required to process frequent network events near to forwarding devices in the data plane.

HSDN specifically focuses on routing in data center networks to achieve hyper scalability and elasticity. It follows a proactive approach to handle high rate of new flow arrivals instead of reactively processing them. It pre-programs flow-tables in forwarding devices to establish all the paths in the network and uses labels to identify all the entire paths instead of only the destinations. HSDN uses a divide-and-conquer approach to partition the data plane network and arrange the network's forwarding devices in multi-layered hierarchy. The control plane of HSDN can make use of a MPLS based label stack or any labeling mechanism to implement ECMP and Traffic Engineering (TE). HSDN logically structures forwarding devices in the data plane to achieve hyper scalability for routing, ECMP and TE applications. Unlike Kandoo, HSDN does not use a multi-layer of control nodes. It structures them in a hierarchy and maintain application specific state information at the bottom layer controllers near the data plane forwarding devices. Next, we discuss an alternative approach allowing forwarding devices to maintain application specific state information, so that they can be programmed using the controller to process frequent events in their datapaths.

2.2.2 Stateful Data Plane and Packet Processing Abstractions

DevoFlow [Jain *et al.* 2013], SmartSouth [Schiff *et al.* 2014], OpenState [Bianchi *et al.* 2014] and Stateful Data Plane Architecture (SDPA) [Sun *et al.* 2017] enhance the OpenFlow's stateless match+action processing abstraction and allow to store application specific per packet state in forwarding devices. Forwarding devices can independently process network events incorporating stored local state information in the match+action processing abstraction.

DevoFlow introduces *rule cloning* and *local actions* to devolve control to forwarding devices. It allows forwarding devices to clone and derive new flow match rules from existing flow entries and state information stored in a boolean flag. Also, forwarding devices can take actions without invoking controller to handle port failure and provide multipath support for load balancing. DevoFlow enables forwarding devices with *Triggers*, *Reports* and *Approximate Counters* processes frequent events related to statistics collection locally. Essentially, DevoFlow adds some control intelligence in the forwarding devices using state information and local actions to delegate processing of network events, which do not require network-wide state.

SmartSouth [Schiff *et al.* 2014] implements a template function in the forwarding devices to offload event processing from the controller to forwarding devices. A template function can be used for different applications like Snapshot, Anycast, Blackhole and Critical node detection and it minimizes the involvement of the centralized controller.

OpenState provides *eXtended Finite State Machines* (XFSM) based packet processing abstractions along with stateless stages of flow-table for packet processing. OpenState allows packet processing using stateful match+action stages implemented using a *State table* and an *XFSM table* per stage. It uses a state table to match packet headers and find the corresponding state value. The XFSM table matches with packet header and state values to find the corresponding action and accordingly process the packet. OpenState allows programming of flow-tables of OpenFlow and per packet state for stateful packet processing in the forwarding devices. SDPA provides infinite state machine based stateful packet processing abstraction in addition of OpenState's XFSM. It uses two tables (State Transition and Action tables) to realize the functionality of the OpenState's XFSM table.

Usually, SDN controller handles message exchanges required to support specific protocols like ARP (Address Resolution Protocol) and ICMP (Internet Control Message Protocol). Such protocols require to maintain complex state for interaction and based on the state generate various types of packets. Handling complex protocols message exchange at SDN control provides great flexibility but impacts performance and scalability of SDN controller for large scale networks. InSPired Switches [Bifulco *et al.* 2016] allow in-switch generation of packets. It provides APIs to program packet generation within the switch by specifying trigger conditions, packet header formats, fields to set and forwarding action to takes on the generated packet. SDN controller can delegate message exchanges of protocols to forwarding devices.

SNAP [Arashloo *et al.* 2016] is a programming language that provides a stateful network-wide packet processing abstraction along with a centralized network programming model by considering the network as a *one big switch* having global and persistent arrays to read/write state information. The SNAP compiler generates forwarding match+action rules for each forwarding device and determines the placement of program state (arrays) across them. The forwarding rules make sure that packets travel through forwarding devices with the required states for processing.

DevoFlow and SmartSouth introduce specific packet processing mechanisms to process frequent network events using stored state information, whereas OpenState and SDPA provide a more general state-machine based packet processing abstraction for the same. SNAP enables programming of stateful data plane with a centralized control abstraction to simplify programming and management of distributed of stateful devices in the network data plane.

2.2.3 Distributed Control Plane

The centralized SDN control plane is vulnerable to single point of failure. Hence, this poses scalability as well as reliability issues for the controller in the network. Onix [Koponen *et al.* 2010], HyperFlow [Tootoonchian & Ganjali 2010], ONOS [Berde

et al. 2014], ElastiCon [Dixit *et al.* 2014] and Beehive [Yeganeh & Ganjali 2014] propose physically distributed implementations of the logically centralized control plane to address scalability and reliability issues. These early proposals of distributed SDN control planes consider stateless forwarding devices in the network data plane. As the distributed control plane is realized by running multiple instances of controller, it requires to distribute and provide a consistent global network view across all the instances. Onix stores the global network view as a Network Information Base (NIB) in form of a graph, where each node represents a network entity within the topology. Onix replicates and distributes the NIB data across multiple running instances of the SDN controller. It hides the complexity of maintaining the consistent global network view from control applications and provides general APIs to manage the network. Onix allows applications to use scalability mechanisms like partitioning, aggregating and structuring instances in a hierarchy or a federation. It provides applications the control to manage the trade-off between consistency and the durability of network state as required.

HyperFlow is a distributed event-based SDN control plane. Instead of maintaining the global network view in a data structure like NIB of Onix, HyperFlow controller instances use the publish/subscribe messaging mechanism to synchronize their network-wide views. Each controller instance publishes network events that update the global network view and other instances replay all the published events to reconstruct the global network view. Each controller processes the local network events without generating traffic in publish/subscribe channels, hence reducing the message exchange and increasing the scalability. Onix and HyperFlow require a number of running instances of controllers to be configured statically. ElastiCon allows dynamic increase and decrease of controller instances and balance the control traffic generated due to network events across them according to load conditions.

Beehive provides an asynchronous event based framework to program centralized control applications. It considers control applications as asynchronous message handlers capable of storing their state in dictionaries. Beehive automatically deducts the state to message mapping and guarantees that each key in dictionaries is accessed and managed by only one instance of the control applications. It converts a centralized application into distributed one by replicating it state, instrumenting it at runtime and dynamically changing the placement of its instances.

2.2.4 Hybrid Control Plane

Hierarchical control plane architectures can process frequent network events near data path and provide fine-grained control over network. However, hierarchical control plane architectures may not provide the shortest path between nodes, which poses the path stretch problem. As the number of levels in the hierarchy increases to scale large networks, path stretch between nodes increases. Distributed control plane architectures face a super-linear computation complexity problem for applications like routing in presence of large networks. In flat plane mode, they can communicate using distributed protocols to reduce the path stretch. Orion [Fu *et al.* 2015] provides an hybrid control plane architecture to address these problems. It allows controller instances to be structured in a hierarchy and in a flat plane.

2.3 Network Function Virtualization

The primary goal of networks is to provide data and communication services to share information in various forms across heterogeneous endpoint devices (e.g., mobile, television, data centers, desktop, etc.). Networks consist of a geographically scattered myriad of specialized network appliances, called *middleboxes*, to enable network services across endpoints. As the novel and demanding use cases for new network services emerge due to user growth and technological advancements (e.g., 4G, 5G, fiber, 4K video streaming), new and innovative revenue generating services have to be deployed in the network within critical time frame. Hence, network operators need to procure off-the-shelf middleboxes or design, develop and test new ones with customized hardware and software. Also, those new appliances have to be integrated into the existing network without interfering with already deployed services. This inhibits rapid and on-time deployment of new services, incurs heavy CAPEX to enable them and OPEX due to increase in diversity of middleboxes and network complexity.

To address all these problems, a large number of network operators came together in 2012 and led an effort by drafting an introductory white paper on Network Function Virtualization (NFV) [Chiosi *et al.* 2012]. This study focuses on transforming the network architecture and its operations for easy development and deployment of services at low cost. The fundamental principle of NFV is to separate the software from the customized proprietary hardware of network appliances. One of the easiest approach proposed by industry is to employ the virtualization technology widely used in compute and IT domain to host services in Clouds and Data Centers. NFV advocates to use industry standard servers, switching and storage devices capable of running packet processing softwares for network services or specific functions. Network operators can instantiate software on these devices to add new network services without deploying other service specific middleboxes. These software components are usually termed as Virtual Network Functions (VNFs). Standard hardware devices can be located in network operator's data centers, aggregation sites and customer premises and can be moved and integrated in various locations in the network as required. The hardware and software components required to deploy and manage VNFs are termed as the NFV Infrastructure (NFVI). NFVI provides the virtualized environment to execute network applications and service specific functions in software. Indeed, the SDN paradigm with open forwarding devices in the data plane can be the primary choice to deploy NFV's commodity hardware infrastructure.

2.3.1 NFV Use Cases and Enabling Factors

The NFV paradigm opens up many opportunities to transform network architecture and services. NFV can be employed to process data plane traffic and execute control plane functions of services in Telecommunication Service Providers (TSPs) and ISP networks. Hence, use cases for NFV are not limited to only existing network services, instead NFV can deploy new network services, which were not previously feasible due to prohibitively high cost, complex integration of technologies and incompatible deployment environments. Some of the elementary network devices performing

switching and security functions are obvious candidates to move to NFV, which include switching elements like routers, NAT, gateways and security devices firewalls, IDS/IPS, virus and span scanners, etc. As the physical layer communication technology evolves, TSP/ISPs require to roll out new mobile network technology by deploying functions to support related protocol stacks, communication transport and core network technologies. TSP's core and access network nodes enabling mobile services like 3G, 4G, VoLTE, 5G etc. can tremendously benefit from the NFV approach. Some other use cases are network monitoring, measurement and diagnosis functions; authentication, access and policy control; application and network optimization to deploy CDN (Content Distribution Network) nodes, caches, etc.

Cloud computing technologies are considered as primary enablers of NFV. Virtualization is the heart of cloud computing mechanisms. Hardware virtualization accelerated by native hardware for performance and software switches (e.g., Open vSwitch) to connect virtualized machines are primary enablers of NFV. Advances in virtualization technology like lightweight container based virtualization, though comes with dependency on native platform, have also propelled NFV. As NFV uses virtualization for processing packet at line rate, it requires high speed packet processing in software on industry standard compute servers. This can be achieved using high volume multi-core CPUs with high I/O bandwidth, smart NICs, TCP offloading and advanced technology like Data Plane Development Kit (DPDK) [Linux Foundation 2013] and Vector Packet Processing (VPP) [FD.io 2017] to process packets in software on these standard compute servers. Apart from virtualization, cloud technologies provide automation tools and mechanisms to control and manage network functions for resiliency and efficient utilization of hardware resources. For example, orchestration and management systems of cloud technology can provide means to instantiate virtual network functions, allocate physical resources to them, support failover, migrate, and snapshot the virtual functions.

The use of standard high volume compute servers built using general purpose compute architecture (x86, ARM etc.,) replaces customized and expensive ASIC hardware built for network services. A high volume availability of the servers and their components provide cost effective and a more economical way for packet processing compared to specialized middleboxes.

2.3.2 Benefits of NFV

The NFV paradigm reduces hardware equipment costs and other OPEX by aggregating scattered appliances to cloud, data center or aggregation sites, leveraging more economical industry standard high performance servers compared to function specific middlebox hardware built using ASICs. NFV allows rapid innovation and roll-out of new services. Network operators can target services based on geographic or customer sets. They do not have to rely on high scale and revenue generating demand to cover investment on the service specific hardware. NFV enables easy test, integration and development process, as it can run production and test software on the same standard high performance servers. NFV provides on demand scaling of network services, thereby providing a great flexibility in deployment network services according to traffic requirements and end-user demands. Its software-based approach opens network

service development market for smaller enterprises, academia, etc., bringing more innovation and creating healthy ecosystems. NFV can leverage power management features of standard servers, switches and storage devices and employ workload optimization to reduce energy consumptions of the hardware infrastructure. NFV can support in-service software upgrade by launching new VMs, redirecting traffic and, if required, synchronize the state of old and new VMs.

2.3.3 Challenges for NFV

The biggest challenge NFV face is performance degradation. Even though NFV employs high volume industry standard servers, virtualized network functions may not match packet processing performance of proprietary hardware appliances built using ASICs. The primary goal is to minimize the degradation in performance using appropriate software technology so that latency, throughput and processing overhead is decreased. A clear separation of software and hardware using virtualization must allow execution of any virtual appliance in different but standard datacenters or cloud platforms. In this aspect, the challenge is to define an unified interface to decouple software appliances and hardware infrastructure. The NFV architecture must co-exist with operators' legacy network equipments, creating a hybrid network in which virtual as well physical network appliance operate together. The NFV paradigm can succeed, only if a unified and standardized management and orchestration architecture exists to easily integrate new functions or virtual appliances into the operators' hardware infrastructure. Also, control and management automation of functions is mandatory to scale NFV. As NFV aggregates hardware and shares it with multiple network functions and services, it is necessary to provide guarantee on service performance and network stability, avoiding interferences from other services.

2.3.4 Tools, software and open-source initiatives for NFV

Virtualization technologies are at the core of the NFV paradigm. Indeed, the success of virtualization technologies in IT operations in enterprises and for compute services has motivated network operators to separate hardware and software for packet processing using network functions. To enable NFV, hardware to lightweight process-level virtualization technologies (e.g., VirtualBox, Qemu, Linux containers, Docker) are not sufficient. A more comprehensive approach is required to enable NFV, where orchestration and management tools are responsible for launching virtual instances on compute resources, diverting network traffic towards the instances, allocating compute and network resources, providing service guarantee, robustness and resiliency for the virtualized network functions. Hence, the European Telecommunications Standards Institute (ETSI) has developed specifications for Open Source NFV Management and Orchestration (MANO) software stack, called OSM [The ETSI OSM Community 2016]. Apart from ETSI effort, OpenStack and Apache CloudStack™ are widely used and represent popular platforms to enable NFV or virtualize cloud infrastructures.

2.4 SDN, NFV and Network Programming

In the SDN paradigm, network control applications like routing, topology discovery, statistics collection etc., process control traffic generated by network events by centralized (at least logically) control plane of the network. Early SDN proposals advocated to keep forwarding devices in the data plane as *dumb switches*, which are not capable of storing state information and locally process frequent network events as dictated by the controller. They used a distributed architecture for logically centralized control plane as outlined in Section 2.2.3. SDN controllers could not delegate specific control traffic processing to dumb forwarding devices in data plane. In order to offload control traffic processing and delegate network control, hierarchical control plane, stateful data plane and hybrid control plane approaches were proposed, as described in Section 2.2.1, 2.2.2 and 2.2.4.

In the NFV paradigm, various NFs like firewall (stateful or stateless), NAT, Intrusion Detection/Prevention Systems (IDS/IPS), DNS tunnel detection, load-balancers etc., process the data plane traffic at line rate and are executed in virtualized environment on low cost COTS. These VNFs are deployed in NFVI available at various locations in the network and traffic is steered through them. They are instantiated, migrated or deactivated according to change in traffic conditions in the network. This deployment approach using network softwarization restricts the network operator to implement custom, fine-grained packet processing functions and reuse common modules for packet processing across multiple VNFs, as many of these VNFs perform similar processing steps on the same packet. For example, packet processing modules for checksum computation, statistics computation of traffic flows or filtering can be shared across multiple VNFs. Moreover, migrating an entire middlebox in virtualized environment poses performance issues.

To address the above issues, OpenBox [Bremler-Barr *et al.* 2016] and Slick [Anwer *et al.* 2015] use the SDN and NFV principles together to provide programming abstractions in order to develop, deploy and manage NFs.

They provide common processing blocks (OBIs in OpenBox) or elements (in Slick) for reusability and to develop and efficiently deploy new NFs. Network controllers in OpenBox and Slick steer the traffic through OBIs and elements, respectively, running on in-network compute machines, specialized hardware or VMs. They use the same centralized network controller to manage instances of common processing blocks in the network (OBI instances or Slick elements), thereby extending the centralized SDN control plane for NF programming and management. Slick and OpenBox provide general message construct *configure* and a protocol, respectively, to develop, deploy and manage NFs from the centralized controller. Slick provides programming abstraction to specify a high-level packet processing policy, which indicates sequence of elements to process for a particular traffic flow.

Instead of deploying on in-network compute machines, specialized hardware or VMs, many of the common processing blocks (e.g., HeaderClassifier, RegexClassifier, Alert in OpenBox) or NFs can be implemented in the datapath of forwarding devices by programming or configuring them, directly. Such an architecture allows a high number of packets to be processed in multiple NFs within the datapaths of the forwarding

devices in the SDN data plane. It can enhance the overall network performance by maximizing the traffic processing within the SDN data plane. In particular, this approach eliminates the round-trip of packets from interfaces of NF to the forwarding devices and also saves computing power of COTS machines used to process packets at line rate in the fastpath of NFs. However, to maximize data traffic processing within the SDN data plane, it is imperative that the SDN data plane supports *modular* programming. Part II describes our proposal for a modular SDN data plane architecture using the P4 packet processing language.

Part I

Modular Approach for Network Control Delegation

3 NFV-based Scalable Guaranteed-Bandwidth Multicast Service for Software Defined ISP Networks

Contents

New applications where anyone can broadcast high quality video are becoming very popular. ISPs may take the opportunity to propose new high quality multicast services to their clients. Because of its centralized control plane, Software Defined Networking (SDN) enables the deployment of such a service in a flexible and bandwidth-efficient way. But deploying large-scale multicast services on SDN requires smart group membership management and a bandwidth reservation mechanism with QoS guarantees that should neither waste bandwidth nor impact too severely best effort traffic. In this chapter, we propose; (1) a scalable multicast group management mechanism based on a Network Function Virtualization (NFV) approach for Software Defined ISP networks to implement and deploy multicast services on the network edge, and (2) the Lazy Load Balancing Multicast (L2BM) routing algorithm for sharing the core network capacity in a friendly way between guaranteed-bandwidth multicast

traffic and best-effort traffic and that does not require costly real-time monitoring of link utilization. We have implemented the mechanism and algorithm, and evaluated them both in a simulator and a testbed. In the testbed, we experimented the group management at the edge and L2BM in the core with an Open vSwitch based QoS framework and evaluated the performance of L2BM with an exhaustive set of experiments on various realistic scenarios. The results show that L2BM outperforms other state-of-the art algorithms by being less aggressive with best-effort traffic and accepting about 5-15% more guaranteed-bandwidth multicast join requests.

3.1 Introduction

The massive increase of live video traffic on the Internet and the advent of Ultra High Definition (UHD) videos have placed a great strain on ISP networks. These networks follow a hierarchical structure for providing Internet access to millions of customers spread over large geographical areas and connected through heterogeneous access technologies and devices. Recently, Periscope[1] and Facebook Live Stream[2] over-the-top (OTT) applications, where anyone can broadcast their own channel, became very popular on both smartphones and computers. To support such new streaming applications and satisfy the users' demands, ISPs may decide to deploy high-quality multicast services in their networks. One solution is to use built-in multicast within their infrastructure to implement flexible, bandwidth-efficient and scalable multicast delivery services. Such an approach may enable the efficient deployment of many-to-many broadcast services such as Periscope, which could be extended to handle multicast group creation transparently on behalf of users. However, multicast has failed to achieve Internet-wide support so far [Diot *et al.* 2000], and even for the limited deployment in managed networks for services such as IPTV, it requires complex integration of specialized multicast enabled routers and protocols, traffic engineering and QoS mechanisms in the networks.

Software Defined Networking (SDN) appears to be an attractive approach to implement and deploy innovative multicast routing algorithms in ISP networks [Yap *et al.* 2010, Marcondes *et al.* 2012, Bondan *et al.* 2013, Tang *et al.* 2014, Craig *et al.* 2015, Zhang *et al.* 2015, Ruckert *et al.* 2016], thanks to its logically centralized control plane. More specifically, in Software Defined ISP networks, live video streaming applications can benefit from QoS guaranteed dynamic multicast tree construction algorithms [Kodialam *et al.* 2003, Chakraborty *et al.* 2003, Seok *et al.* 2002, Crichigno & Baran 2004, Youm *et al.* 2013] that exploit the global view of the network.

In addition, ISPs could exploit fine-grained control over QoS guaranteed multicast and best-effort traffic to implement traffic engineering policies that are friendly to low priority best-effort traffic. Several advanced multicast routing algorithms integrating load balancing techniques have been proposed in the literature to better utilize the network bandwidth, avoid traffic concentration and limit congestion in the network [Tang *et al.* 2014, Craig *et al.* 2015, Kodialam *et al.* 2003, Chakraborty

[1] Periscope URL: https://www.periscope.tv/.
[2] Facebook Live Stream URL: https://live.fb.com/

et al. 2003, Seok *et al.* 2002, Crichigno & Baran 2004]. However, these approaches require costly real-time monitoring of link utilization to allow network resources sharing between the QoS guaranteed and best effort traffic classes according to ISP traffic management policies.

Moreover, SDN-based centralized architectures suffer from well-known scalability issues. Different approaches based on either distributed [Yeganeh & Ganjali 2016, Phemius *et al.* 2014, Koponen *et al.* 2010, Dixit *et al.* 2014] and hierarchical [Yeganeh & Ganjali 2012, Fu *et al.* 2015] control planes or on stateful data planes [Bianchi *et al.* 2014, Song 2013] have been proposed to address SDN scalability issues in general. Distributed controllers usually need costly state synchronization mechanisms. Therefore, only the approaches that propose delegation [Yeganeh & Ganjali 2012, Santos *et al.* 2014] could be followed but they require to implement the whole functionalities of controllers at each router. Indeed, in the presence of large-scale multicast applications, extra processing is required at edge routers to handle locally all Internet Group Management Protocol (IGMP) membership messages [Cain *et al.* 2015] that would otherwise be flooded to the controller.

In this chapter, we address the two following problems: (1) how to avoid implosion of IGMP group membership messages at the SDN controller and (2) how to deploy guaranteed-bandwidth multicast services in Software Defined ISP networks with low cost and while being friendly with best effort traffic.

To address the first problem, we propose to exploit the hierarchical structure of ISP networks and to use Network Function Virtualization (NFV). In short, we delegate when needed the multicast group membership management through specific virtual network functions (VNFs) running at the edge of the network.

To address the second problem, we propose a novel threshold-based load balancing algorithm in which a certain amount of link capacity in the ISP's infrastructure is reserved in priority for guaranteed-bandwidth traffic. This means that in absence of guaranteed-bandwidth traffic, best-effort traffic can use the capacity reserved for guaranteed-bandwidth traffic. Hence, we dynamically increase the capacity share by gradually increasing the threshold. This approach is friendly to best-effort traffic and helps in indirectly load balancing the guaranteed-bandwidth traffic without the need of real-time link traffic monitoring mechanisms, as the controller is responsible for accepting or rejecting multicast subscription requests and is aware of bandwidth requirements and network link capacities.

Our contributions in this chapter are the following: (1) an original solution to handle multicast group management in a scalable way on Software Defined ISPs with multicast networking functions running locally on NFV Infrastructure Point of Presences (NFVI-PoPs) and NFV-based Central Offices (NFV-based COs); (2) a smart multicast routing algorithm called L2BM (Lazy Load Balancing Multicast) for large-scale live video streaming applications, which runs on the SDN controller to route the streams across the NFVI-PoPs or NFV-based COs and follows a threshold-based traffic engineering policy for capacity sharing; (3) an implementation of our framework including the scalable group management approach and the L2BM multicast routing algorithm in Open vSwitches (OVS) [Pfaff *et al.* 2015] based QoS Framework using OpenFlow and Floodlight controllers and the evaluation of the L2BM

algorithm, with comparisons with state-of-the-art solutions. We provide a web site[3] that includes the implementation of our framework, the simulation code, the scenarios scripts to reproduce all the experiments and extended results obtained with scenarios not shown in the chapter.

The rest of the chapter is organized as follows: Section 3.2 presents our architectural approach for deploying scalable, flexible and hierarchical control for multicast group membership management at the network edge, Section 3.3 presents L2BM, Section 3.4 describes the implementation of our framework, Section 3.5 presents the evaluation of our multicast routing algorithm L2BM, Section 3.6 discusses the related work and Section 3.7 concludes the work.

3.2 Scalable Multicast Group Management for Software Defined ISPs

In this section, we tackle the problem of deploying multicast functionalities in a scalable and flexible way on Software Defined ISP networks.

Traditional multicast routing and management protocols such as PIM-SM (revised RFC) [Fenner *et al.* 2016] and IGMPv3 (revised RFC) [Cain *et al.* 2015] effectively establish and maintain multicast communication paths between sources and receivers. Multicast routers run complex state machine for group membership management of interfaces. In brief, they handle IGMP membership report messages sent by the receivers to manage group membership state, and accordingly send PIM Join/Prune messages to the upstream routers to coordinate the multicast routing paths. Deploying multicast functionalities in SDN without taking precautions can lead to congestion issues at the controller as edge SDN routers need to forward all IGMP, because they can not run complex group membership state machines neither store state information to take decisions in an autonomous way.

Let us consider a hierarchical ISP network as shown in Figure 3.1. With the advent of NFV, network aggregation points at central offices (COs) are being transformed into mini-datacenters. These NFV-based COs gather commercial-off-the-shelf (COTS) hardware that can run any network functions such as NATs, firewalls or caches [Chiosi *et al.* 2012]. Access networks aggregate the customers' access lines at the COs at the frontier of the metro ISP network. The metro ISP network interconnects the COs using relatively high capacity links compared to access network links. Similarly, the core network interconnects gateway Central offices serving as Points of Presence and including NFV infrastructure that we call NFVI-PoPs. With SDN, a controller is responsible for programming packet forwarding in its own domain. We refer to it as the Network Controller (NC) of a given domain.

In our approach, NCs delegate multicast group management functionalities to virtual network functions (VNFs) running at NFV Infrastructure at the edge of the metro networks. We call these functions MNFs for Multicast Network Functions, and define MNFs-H and MNFs-N. MNFs-H running in NFV-based COs exchange IGMP query

[3] See https://team.inria.fr/diana/software/l2bm/

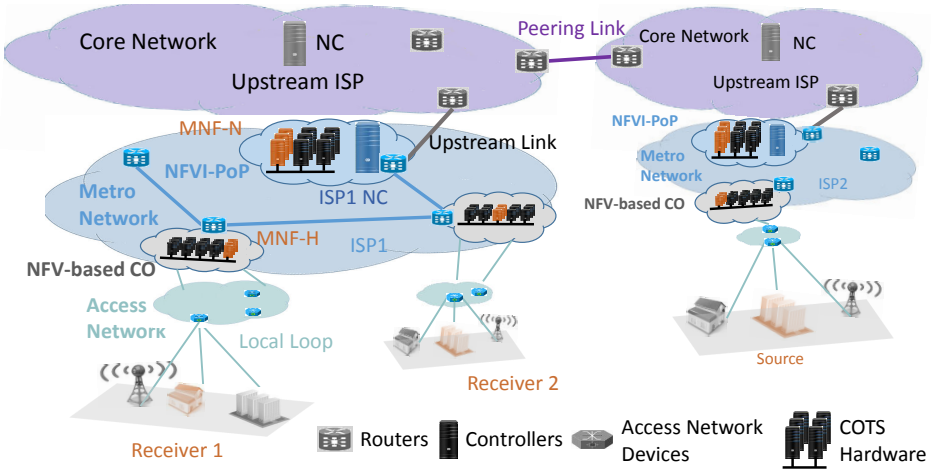


Figure 3.1: Example of Software Defined ISP Network with NFVI-PoPs

and report messages with the downstream receivers. MNFs-N running in NFVI-PoPs process PIM Join/Prune signals sent by MNFs-H based on the membership states of their NFV-based CO. Unlike traditional PIM Join/Prune messages, these signals do not update trees by themselves, but instead inform the corresponding MNFs-N to coordinate multicast tree update with the NC. Indeed, delegating group membership management processing at NFV-based COs can greatly reduce the concentration of control traffic emerging from multicast applications. Our solution aims to achieve scalability similarly to traditional distributed multicast protocols in SDN architecture in spite of its centralized control plane. It gives flexibility using NFV to enable multicast support on demand and does not put the burden of requiring multicast state management functionality on all the routers and especially core routers. NCs communicate with the NFV orchestrators that run on each NFV-based CO of the domain to instantiate MNFs when necessary. Note that NFV orchestrators are responsible for scaling in/out their VNFs according to the group membership traffic load, providing flexibility. We emphasize that implementing the MNFs functionalities requires several features that are not compatible with hardware SDN routers, which are usually dumb devices. In particular, it is necessary to run a state machine for implementing IGMP and for generating periodically membership queries to the multicast receivers. As described earlier, we argue that the presence of mini-datacenters in central offices (NFV-based COs) as shown in Figure 3.1 will enable running the MNFs functionalities as VNFs. If such datacenters are not deployed in central offices in the near future, MNFs could either be implemented as middleboxes running next to edge routers or integrated within software routers as switching at the edge is becoming virtual, handled on x86 cores as anticipated by SDNv2[4].

Let us now examine our proposed architecture with an example. At the start, in absence of MNFs running at the access NFV-based COs, the first group join request among the receiver hosts is forwarded as a packet-in to the metro NC. If the corresponding source or the multicast tree is already present in the metro network,

[4]See article "Time for an SDN Sequel? Scott Shenker Preaches SDN Version 2," www.sdxcentral.com/articles/news/scott-shenker-preaches-revised-sdn-sdnv2/2014/10/.

then the metro NC establishes[5] the bandwidth guaranteed path for the requested multicast flow between the edge router that receives the join request at the access NFV-based CO and the multicast tree. At the same time, the metro NC interacts with the NFV orchestrator of the access NFV-based CO to instantiate an MNF-H and with the NFV orchestrator co-located at the NFVI-PoP to instantiate an MNF-N. After that, the group specific IGMP traffic received by the edge router is redirected to the MNF-H and handled locally. In addition, the PIM Join/Prune signaling traffic is redirected to the MNF-N that manages group membership of all the NFVI-based COs and communicates with the metro NC to update the multicast tree for each group in the metro network. In case the access NFV-based CO is already receiving the requested multicast flow, the MNF-H is responsible for configuring the edge router to forward the multicast flow to the port where the IGMP membership report has been received. Once the processing of IGMP messages are delegated to MNF-H, both the metro NC and MNF-H can configure the SDN edge routers. This design makes all the flow tables in the edge router vulnerable to unauthorized modification from the corresponding MNF. Hence, careful programming of MNFs is required to avoid race conditions on flow tables and maintain consistency in routers tables.

Metro NCs inform upper level NCs in the hierarchy of the presence of all the multicast sources in their domain and also exchange this information with peering ISPs' NCs. We assume that the streaming application implements the required signaling protocols such as SIP, SAP and MSDP to announce and discover multicast groups.

On detecting a multicast source, an NC communicates with the orchestrator on its local NFVI-PoP to instantiate an MNF-N if the latter is not yet running, in order to store information on the new multicast source and process future Join/Prune signals. If neither the source nor the multicast tree corresponding to the PIM Join signal belongs to the domain, the MNF-N forwards the PIM Join signal to the upstream network through the upstream route set by the NC. If the source and the receivers are not in the same ISP, the Join signal will propagate through the peering link to reach the MNF-N corresponding to the source's ISP and a bandwidth guaranteed path will be established on both ISPs.

MNF-Hs are responsible for aggregating the group membership reports received from their NFV-based CO networks, and according to the state machine, they can send Join/Prune signals to the MNF-N for the different multicast groups. Hence, similar to multicast routers in traditional multicast protocols, MNFs can maintain the group membership state of their downstream receiver hosts. Figure 3.2 illustrates our approach.

Without the deployment of the MNF-H, the edge routers do not maintain multicast state and do not take decision to replicate the multicast stream to the required downstream interfaces in the centralized SDN based approach. Hence, all the multicast group membership messages are forwarded to the NC. In our proposed approach, once the MNF-H at the access NFVI-based CO receives a multicast stream, all the successive IGMP join requests received for the group at the edge router from the downstream access network are locally handled by the MNF-H. Hence, irrespective

[5]The algorithm used to dynamically construct multicast trees with bandwidth guarantee is described in Section 3.3.

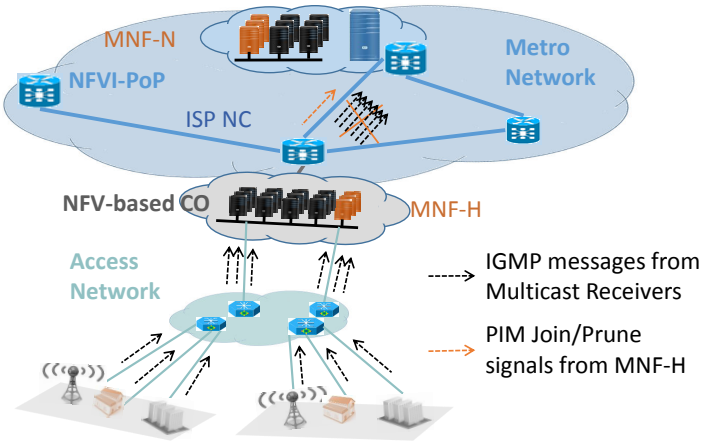


Figure 3.2: Multicast Group Membership Management Analysis

of the number of multicast group membership messages received for a group from end hosts in an access network, only the first IGMP join and the last IGMP leave requests result in sending a PIM Join/Prune signal from MNF-N to the metro NC in order to add/remove the NFVI-based CO from the multicast tree of the group. Therefore, with this mechanism NC is involved only for routing in core network and does not have to maintain IGMP state machines at any of the end hosts.

In Section 3.4, we describe the proof-of-concept implementation of MNFs with Open vSwitches that we use to validate our proposal of delegating group membership management to VNFs for scalability.

3.3 Lazy Load Balancing Multicast routing Algorithm (L2BM)

In this section, we describe L2BM, a threshold-based load balancing routing algorithm proposed to deploy a guaranteed-bandwidth multicast service in ISP networks. L2BM dynamically builds a multicast tree to deliver traffic to member NFVI-PoPs in Core networks or NFVI-based COs in ISP's Metro. It routes multicast streams on-demand to the member NFVI-PoPs or NFVI-based COs in the Core or Metro network, respectively, by programming SDN enabled forwarding devices in the network. L2BM attempts to be friendly with best-effort traffic and remove the need of real-time link measurement mechanisms, which are usually required when deploying load balancing mechanisms.

The main idea is to reserve in priority a certain fraction of link capacity, referred as the *threshold*, for guaranteed-bandwidth multicast services and to restrict the corresponding traffic to this threshold through traffic shaping. Then, to make sure that the best-effort traffic can use the reserved link capacity in the absence of guaranteed-bandwidth traffic, we use in forwarding devices Hierarchical Token Bucket [Devera 2002], which is a classful queuing discipline that allows sharing the link capacity

with flows of different priorities. More precisely, we associate a threshold parameter to each multicast group join request received at NFV-based COs. While connecting the NFV-based CO serving as a node to the multicast tree of the requested group, the L2BM algorithm avoids the links with utilization equal or greater than the current threshold value. L2BM attempts to reserve the required bandwidth on the minimum length reverse path from the receiver to any neighboring node in the multicast tree. If no reverse path to the tree can be found, L2BM increases the threshold value to consider previously avoided links and retries to attach the receiver to the target multicast tree. In presence of multiple shortest paths length with equal threshold value, L2BM selects the one with the least maximum utilization for guaranteed traffic among its links. This information is available at no cost at the NC as it keeps track of previous requests.

Algorithm 1 shows the pseudo-code of L2BM for adding a new node in the multicast tree, using notations defined in Table 3.1.

Table 3.1: Notations used in Algorithm 1

Symbols	Definition
$\mathcal{E}$	set of edges
$\mathcal{V}$	set of nodes
$\mathcal{V}_\mathcal{T}$	$\mathcal{V}$ for multicast tree $\mathcal{T}$ of group M
e_{vu}	edge from v to u
B_{vu}	link bandwidth consumption of e_{vu}
C_{vu}	link capacity of e_{vu}
U_{vu}	link utilization of e_{vu}; $U_{vu} = B_{vu}/C_{vu}$
θ	threshold parameter
θ_{init}	Initial value of θ
θ_{max}	Maximum value of θ
r	new receiver for group M
b	bandwidth requirement of group M
$\mathcal{P}$	FIFO queue, stores pruned nodes
$\mathcal{Q}$	FIFO queue, stores nodes to be explored
u	head node of the $\mathcal{P}$ queue
$Path[v]$	set of edges constructing path from v to r
$len(v)$	path length from node v to r
$U_{max}(Path[v])$	Max. link utilization of edges in $Path[v]$

Line 2 initializes the FIFO queue $\mathcal{Q}$ of graph nodes to be explored and path related variables. These graph nodes represent the SDN enabled forwarding devices in the Core or Metro network that compose the multicast tree. Then, the graph search starts with the initial value of the threshold (θ_{init}) to find a reverse-path from the new receiver, r, to the closest node in the multicast tree of the requested group. The threshold, θ_{init}, allows the links to be loaded to a specified value, before starting spreading the multicast traffic by taking longer paths. The algorithm performs Breadth First Search (BFS) to find a path to the tree considering only edges utilized below current threshold θ and pruning the rest. In line 5, the $ThresholdBFS$ function initializes the prune queue $\mathcal{P}$ and sets the local parameter θ_{next} to 1. The θ_{next} variable is used to record the minimum θ value required for the next recursive call of the $ThresholdBFS$ function. Queue $\mathcal{P}$ stores the nodes having any of their edges pruned due to an utilization higher than the threshold θ. In case none of the tree nodes is found, a recursive call to the function is done with queue $\mathcal{P}$ (the pruned

nodes) and θ set to θ_{next} to continue the search. The loop in line 6 starts the graph search from u, the head node of the queue $\mathcal{P}$. If node u is part of the tree for the requested multicast group, the algorithm terminates (lines 8-9). Line 10 adds node u in the *visited* set. The algorithm expands the search by considering each incoming edge e_{vu} to node u (line 11). It computes U^{new}, the new link utilization of the edge e_{vu} by adding the bandwidth demand b (line 12). If U^{new} is higher than the maximum capacity, θ_{max}, allocated for guaranteed traffic it discards the edge (lines 13-14). Otherwise, it further checks U^{new} against the current value of the threshold θ (line 15). If U^{new} is below θ, $Path^{new}$ and len^{new} are computed to reach v via u (lines 16,17). If another path of the same length to v already exists, the algorithm updates $Path[v]$ with the one having the lowest maximum edge utilization of the two (lines 18-20). Otherwise, node v is added in the queue $\mathcal{Q}$ and $Path$ and len are updated for v (lines 21-24). If U^{new} is above θ (lines 25-27), node u is added in prune queue $\mathcal{P}$ and θ_{next} is set to the minimum edge utilization value among all the pruned edges. If no tree node is found in the current search, the algorithm removes the nodes stored in $\mathcal{P}$ from *visited* set to consider pruned nodes in the next recursive call (line 29). Then, it makes the recursive call to the function with prune queue $\mathcal{P}$, round up to tenth of θ_{next} and *visited* set as input parameters (lines 28-30).

The algorithm is now explained through a simple example. Figure 3.3 shows a network with 4 nodes and a sequence of 3 events marked with numbers from T_0 to T_2. In this example, all links have the same capacity of 100 units. The current load and percentage link utilization are shown for each link e_{vu} using the notation (C_{vu}, U_{vu}). T_0 (in black) corresponds to the initial network state with existing multicast traffic. At T_1 (in red), receiver R_1 from node B joins a new multicast group with source S at A, which requires 20 units of bandwidth. As the link utilization of both e_{AB} and e_{DB} is equal to 0.1, the algorithm will explore both the edges, but will select tree node A increasing U_{AB} up to 0.3. At T_2 (in green), receiver R_2 from node D joins the same multicast group with source S. Again L2BM starts with $\theta = 0.1$, but ends up selecting the path A-C-D.

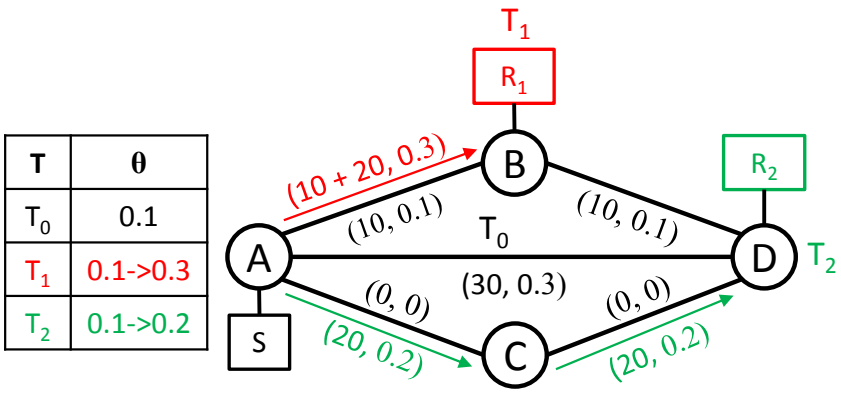


Figure 3.3: Example of L2BM functioning

When all the members from an access network leave a particular group, the MNF

Algorithm 1: Lazy Load Balancing Multicast

```
1  Function AddReceiverInTree(): Path
2      Q.Enqueue(r), len(r) = 0, visited ← ∅
3      return ThresholdBFS(Q, θ_init, visited)
4  Function ThresholdBFS(Q, θ, visited): Path
5      P : To store pruned nodes, θ_next = 1
6      while Q ≠ ∅ do
7          u ← Q.Dequeue()
8          if u ∈ V_T then
9              return Path[u]
10         visited ← visited ∪ u
11         foreach e_vu and v not in visited do
12             U^new ← U_vu + b/C_vu
13             if U^new ≥ θ_max then
14                 continue
15             if U^new ≤ θ then
16                 Path^new ← Path[u] ∪ e_vu
17                 len^new ← len(u) + 1
18                 if v ∈ Q then
19                     if len(v) = len^new and
                        U_max(Path[v]) > U_max(Path^new) then
20                         Path[v] ← Path^new
21                 else
22                     Q.Enqueue(v)
23                     len(v) ← len^new
24                     Path[v] ← Path^new
25             else
26                 P.Enqueue(u)
27                 θ_next = min(θ_next, U^new)
28     if P ≠ ∅ then
29         visited ← visited \ {v : ∀v ∈ P}
30         return ThresholdBFS(P, ⌈θ_next × 10⌉/10, visited)
31     return NULL
```

from the corresponding NFV-based CO has to notify the NC to remove its membership from the multicast tree. Node deletion from the tree is done by recursively removing the non-branch nodes in the reverse path of the stream till a branch node is encountered. This approach does not perturb the existing multicast tree, which prevents packet loss and reordering problems that could have emerged when restructuring the tree.

For each multicast join request, L2BM starts with an initial threshold value of θ_{init}. L2BM performs BFS using only edges with utilization below or equal the threshold. Nodes with higher link utilization are saved in prune queue $\mathcal{P}$ to continue, if needed, the search with increased threshold value through recursive calls. Let us consider the worst case scenario in which each call of $ThresholdBFS$ visits only one node and the rest of the nodes are enqueued in $\mathcal{P}$. This leads to at most 10 consecutive calls of $ThresholdBFS$ as the algorithm increases θ and rounds it up to tenth of the minimum of all link utilization operating above current θ, and each edge is visited exactly once. Hence, the order of run time cost of L2BM is the same as the one of BFS, $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$.

3.4 Testbed and Simulator Evaluation Frameworks

In this section, we first describe the proof-of-concept framework, based on an SDN controller and Open vSwitches, which implements the MNFs with the edge-based group management support and L2BM algorithm for Software Defined ISP networks. Then we present the simulator we implemented to evaluate the performance of L2BM on networks with high link capacity using different traffic scenarios, in a time efficient manner. The performance evaluation mechanisms and results are described in Section 3.5 using both setups.

3.4.1 Open vSwitch based QoS Framework

Providing guaranteed-bandwidth multicast services in Software Defined ISP networks requires allocating bandwidth resources on the multicast trees' links in a programmatic way. In particular, data plane network devices in the networks have to support QoS-based forwarding on the traffic flows and should be programmable. However, existing SDN protocols such as OpenFlow (OF) [McKeown *et al.* 2008] have limited support to program QoS on switches. To demonstrate the feasibility of guaranteed-bandwidth routing for multicast services in real networks, we implemented an SDN-controller module that provides the mechanisms to allocate bandwidth resources at the granularity of flow definition. In the following, we describe the implementation choices made for the SDN-controller module.

The OpenFlow protocol has gained widespread acceptance to implement the southbound interface of SDN, even though its specifications and features are still evolving. From the very first version of OpenFlow, programmable QoS support[6] was provided through simple queuing mechanisms, where one or multiple queues are attached to

[6]Note that queue creation and configuration are outside the scope of the OpenFlow version1.3.

the switch ports and flows are mapped to a specific queue to satisfy QoS requirements. In practice, the OF controller uses the *ofp_queue_get_config_request* message to retrieve queue information on a specific port. Then, OF switches reply back to the controller with *ofp_queue_get_config_reply* messages providing information on the queues mapped to the port. The OpenFlow version 1.3 and later versions specify the *Meter Table* feature to program QoS operations on a per-flow basis. However, traffic policing mechanisms such as rate-limiting throttle the bandwidth consumption of links by dropping packets, which may be problematic especially for time-sensitive flows. So, we rely instead on the Linux kernel QoS features and in particular, on the *tc* Linux command to configure Traffic Control (e.g., shaping, scheduling, policing and dropping) in a way it allows supporting bandwidth guarantee without loss in order to provide high QoE video [Evans *et al.* 2011].

LINC [linc dev 2015], CPqD [Lajos Kis *et al.* 2017] and Open vSwitch (OVS) are widely used software switches among existing implementations of OF-based software switches. We chose OVS because it has been shown that it can be ported to hardware [PICA8 2009] and achieve carrier-grade switching capability using specific acceleration software [6WIND 2017]. OVS offers a wide range of protocols like sFlow, NetFlow but it does not implement any QoS mechanism itself. However, it provides the support to configure its OF-enabled switch ports with a subset of Linux kernel QoS features that is sufficient to implement guaranteed-bandwidth multicast routing.

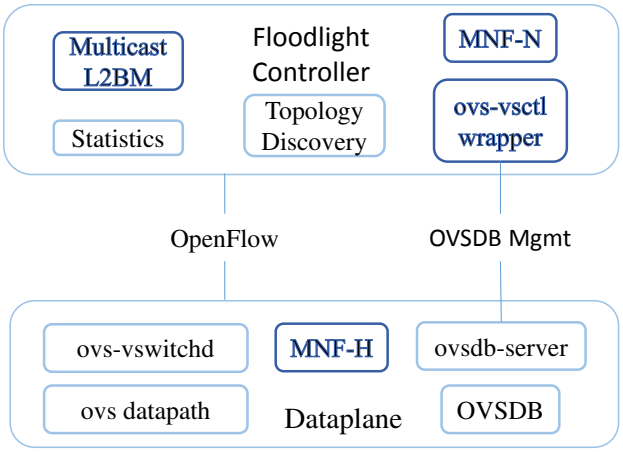


Figure 3.4: Queue-based QoS integration with OVS and Floodlight

To summarize, OVS is a multi-process system whose daemon called *ovs-vswitchd* is used to control all the software switch instances running on the hosting machine. *ovs-vswitchd* reads configuration from a database called Open vSwitch Database (OVSDB). While OF is used to program flow entries, the Open OVSDB management protocol is used to configure OVS itself through the *ovs-vsctl* command-line utility. The latter connects to the ovsdb-server process that maintains the OVS configuration database. *ovs-vsctl* is particularly important because it can create/delete/modify bridges, ports, interfaces and configure queues using the Linux kernel QoS features.

In order to provide an API for handling queues on remote software switches, we implemented a wrapper for *ovs-vsctl* and added it as a module to the Floodlight controller. We chose floodlight because of its simple and modular architecture along with ease of use. The controller module exposes APIs for queue management on data plane interfaces of OF forwarding devices, and through these APIs, other applications can perform traffic shaping, policing and scheduling features provided by *linux-tc*. This approach creates dependency on *ovs-vsctl* utility, but it avoids re-implementation of the OVSDB management protocol [Pfaff & Davie 2013] on the SDN controller. We evaluated our multicast service deployment solution in the Open vSwitch based QoS framework considering a two-level hierarchy ISP network, i.e., with metro and access networks.

3.4.2 Multicast Application Control Delegation to MNFs

MNFs implement IGMP and PIM message handlers. Those handlers require to maintain elaborate multicast state to aggregate membership messages received from downstream networks. MNFs also communicate with the controller to handle groups join/leave when needed and to receive the corresponding multicast streams based on membership status on downstream networks. We implemented MNF-H as a standalone control application to process IGMP messages to align with our proposal of delegating application specific network control to VNFs. For ease of experiments in absence of COTS and NFVI, we executed MNFs-H within the same operating system environment as Open vSwitches.

On the other hand, MNF-N is implemented as an independent server module in the Floodlight controller. The MNF-N server communicates with MNF-H to receive PIM Join/Prune signals but does not use the OpenFlow channel. We took this implementation decision to align it with the approach described in Figure 3.1.

The major functions of MNFs-H are to maintain multicast state, aggregate the IGMP messages received from the switch interfaces and update the flow tables in the switch accordingly. MNFs-H handle group membership management for IGMP messages received from end receivers. This requires updating the flow table in the switch programmatically. Hence, we use the lightweight OpenFlow driver *libfluid* [Vidal *et al.* 2014] to program the Open vSwitch and update the multicast group table as required. Such a use of specific network functions to process of group membership messages offloads group membership management traffic from SDN controller by preventing the repetitive forwarding of the messages from the same edge nodes to the controller. However, it requires to employ two control applications, MNF-H and the global SDN controller, to program the same OpenFlow-enabled edge switch. OpenFlow provides three roles, *Master*, *Slave* and *Equal*, to SDN controllers. We assign the *Equal* role to both the control applications so that both controllers can control, manage and program datapath of the same switch. Clearly, this approach has drawbacks and limitations, which we describe next.

3.4.2.1 Drawbacks and Limitations

OpenFlow does not restrict a controller to modify the datapath programmed by other controllers to process their traffic, when multiple controllers are configured with equal role for the same switch. In particular, it does not provide such a traffic isolation and security against malicious control applications. Considering the more general case where multiple applications are required to process the same packet, OpenFlow does not allow to specify packet processing operators to compose a policy comprising the applications to process the same packet. Also, all network events are broadcast to all the controllers. Hence, all the applications receive network events that they may not need to or must not process. In the part II of the thesis, we propose a modular SDN data plane architecture to address these drawbacks and limitations.

3.4.3 The Simulator

We have implemented a simulator that is able to generate different traffic scenarios, execute extensive experimental runs in time efficient manner and store a snapshot of network link utilization metrics after each multicast join and leave event in the network. Concerning link capacity allocation, the OVS-based L2BM testbed explicitly allocates link capacities to provide bandwidth guarantee to the multicast streams without the need of costly link monitoring system. To do that, it maintains the graph data structure of the topology, updates the links' consumption and allocates the bandwidth using the OVS-wrapper. Our simulator has exactly the same behavior without the need for emulating the network and making OVS-wrapper calls to allocate link capacity. More precisely, it simulates the Internet2-AL2S network topology by maintaining a graph data-structure and update the bandwidth consumption of each link based on routing decisions made by the multicast algorithms for each event. In this way, the most recent network view is provided to the running algorithm after each event.

However, in the testbed, multicast events may be processed in a different order than their arrival order. This mismatch is due to the controller implementation (e.g., single or multi-thread) and to delay variation in packet_in events sent to the controller by the switches. The simulator generates the multicast events trace with time stamped events according to traffic scenarios and sequentializes the events with nanosecond precision before feeding the event trace to the different algorithms. It does not wait between two events, thereby accelerating the execution of experiment run.

Unlike the testbed, the simulator does not run any SDN controller. Instead, it converts multicast join and leave events into function calls in the implementation to add or remove the receiver in the target multicast group. Hence, in the simulator, all the algorithms receive each event in the same order for a given experimental run even if the events are generated with very small time difference. However, in the case of the testbed, the events received in time duration of order of microseconds are simultaneously received by the controller without guaranteeing a specific processing order in the same experimental run across all the algorithms.

To compare the performance results between the testbed and the simulator, we executed 20 runs of the different workload experiments in the testbed environment, recorded these 20 event traces and fed them to the simulator. The results of Figure 3.6a with 95% confidence intervals show the same behavior of the routing algorithms in the simulator and in the testbed. For more details, see Section 3.5.5.

3.5 Evaluation of L2BM

In this section, we study the performance of L2BM for routing guaranteed-bandwidth multicast flows in a single domain ISP. The evaluation is done using the testbed and the simulator that are both described in Section 3.4. Without loss of generality, we consider that the guaranteed-bandwidth traffic is allowed to use the whole link capacity of the network (i.e., $\theta_{max} = 1$). We also assume that routers implement the Hierarchical Token Bucket queuing discipline so that best effort traffic can use the reserved bandwidth in the absence of guaranteed-bandwidth traffic.

3.5.1 Alternative Algorithms for L2BM

We compare L2BM with two multicast routing algorithms that implement greedy heuristics of the Dynamic Steiner Tree (DST) [Waxman 1988] algorithm with two different metrics: path-length (DST-PL) and link utilization (DST-LU). L2BM allocates the required bandwidth along the path to the new receiver. If it cannot find a path with enough available bandwidth for the multicast group requested by the new receiver, then it rejects the join request. We also implemented the guaranteed-bandwidth and admission control features in DST-PL and DST-LU algorithms. Note that DST-PL with guaranteed-bandwidth and admission control is an alternative implementation of NNFDAR presented in [Youm *et al.* 2013]. Both L2BM and DST-PL follow a *nearest node* approach with the path length metric proposed in [Waxman 1988], but in addition, L2BM attempts to limit the maximum link utilization below some threshold. With DST-LU, new receivers join the existing multicast tree using the path with the minimum total link utilization. As the initial threshold (θ_{init}) controls the multicast traffic load allowed on links before triggering load balancing, we use L2BM with low (0.1, 0.2), medium (0.4) and high (0.6) initial thresholds, θ_{init}, to study load balancing for different initial traffic loads on links.

3.5.2 Testbed and ISP Network Topology

Testing our bandwidth allocation implementation with Open vSwitches and Floodlight requires a testbed capable of emulating QoS-oriented SDN experiments. Existing emulation tools like Mininet [Lantz *et al.* 2010] do not consider physical resource constraints, hence the experiments can suffer from errors emerging from resource limitations of the host physical machines. We used the DiG [Soni *et al.* 2015] tool to automate the procedure of building target network topologies while respecting the physical resources constraints available on the testbed. Regarding the network topology, we chose *INTERNET2-AL2S* Figure 3.5 to represent an ISP network with 39

nodes and 51 bidirectional edges. Then we virtualized this topology using DiG on the Grid'5000 large-scale testbed. DiG implements routers using Open vSwitches and we configured it to allocate sufficient processing power (i.e., two computing cores) at each router for seamless packet switching at line rate. As the grid network uses 1Gbps links and *INTERNET2-AL2S* operates with 100Gbps links, we had to scale down the link capacity to 100Mbps in our testbed experiments. But, we use the simulator for the same network with 1, 10 and 100Gbps link capacities that are representative of links in different tier ISP networks. We present results with network of 1Gbps links and other results made are available[7].

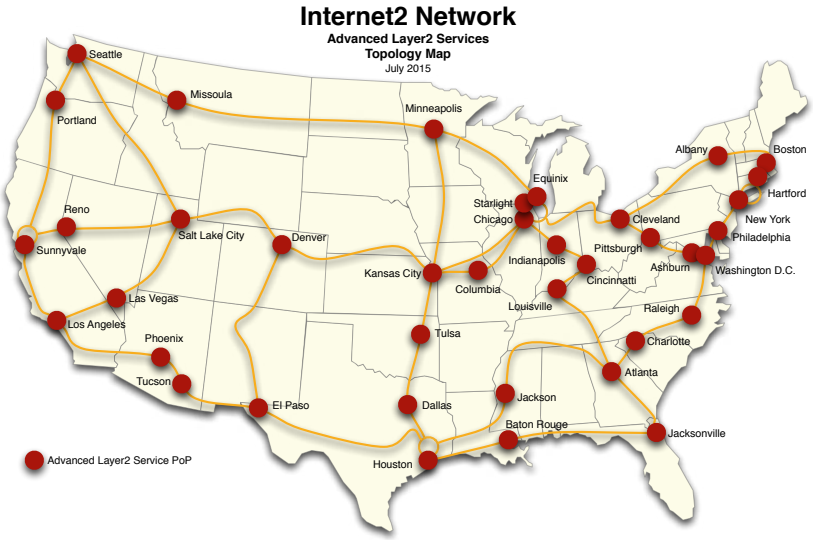


Figure 3.5: INTERNET2-AL2S Network Topology[8]

3.5.3 Multicast Traffic Scenarios

Properties of the multicast traffic vary according to the level we are in the hierarchy of ISP networks. The controller of a lower level ISP backbone network may be exposed to high end-viewers churn for a multicast session. Indeed, as each NFV-based CO covers a smaller number of end-receivers, their group memberships can be more volatile than for NFVI-PoPs of higher level ISP networks, which have a higher probability of maintaining their group memberships. To account for that in our evaluation, we generate workloads *without churn* and workloads *with churn* to simulate traffic conditions at a high and low level, respectively, in the hierarchy of ISPs networks. At a high level in the hierarchy, each NFVI-PoP aggregates a large number of multicast end receivers, resulting in static multicast sessions. In this case, NFVI-PoPs dynamically join multicast groups, but do not leave their groups, so such a traffic consumes the reserved network capacity θ_{max}, rapidly. Conversely, at a low level, end-receivers in the network join and leave multicast groups during the runs and NFV-based COs also change group membership, frequently.

[7] See URL https://team.inria.fr/diana/software/l2bm/

[8] From URL https://noc.net.internet2.edu/i2network/advanced-layer-2-service/maps-documentation/al2s-topology.html

In all the simulations, we generate different multicast traffic workloads by varying the number of multicast groups from 10 to 150 for a network with 1 Gbps link capacities to load the links sufficiently. We uniformly select multicast group session bandwidth demands for each multicast group in the workload from 1, 2, 5, 9, 12, 25, 50, 80 Mbps representative of different qualities of live video streams (SD, HD, UHD). Concerning the testbed, as we used 100 Mbps link capacity due to physical resource constraints, we generate workloads with lower bandwidth demands for multicast groups uniformly selected from 2, 5 and 8 Mbps.

To generate the workloads without churn, we launch multicast group senders with inter-arrival time exponentially distributed with a mean of $3s$ in order to avoid flash crowd and still have reasonably short simulation time. Once the sender of a multicast group appears in the system, join requests from receivers are sent with an inter-instantiation time exponentially distributed with a realistic mean $1/\lambda$ set to $5s$ [Veloso *et al.* 2002]. We generate the workloads with churn in a similar way but additionally enforce receiver nodes to leave the multicast group sessions at exponentially distributed time with a mean $1/\mu$ empirically set to $75s$. With this setup, each multicast group session in workload with churn has an average number of $(\lambda/\mu = 15)$ receivers at steady state. We chose to have a moderate number of users in order to avoid full broadcast scenarios, in which all routers in the metro network would belong to the multicast tree. We simulate the churn of receiver nodes in each multicast group session for a duration of $300s$. Note that we do consider source popularity in this study. This could be an interesting future work using our publicly available source code.

Furthermore, as proposed in [Hagiya *et al.* 1993], we consider two types of multicast traffic: *Homogeneous* and *Concentrated*. Workloads with Homogeneous multicast traffic (called *Homogeneous workloads*) generate multicast flows by utilizing the whole network in equal proportion. Such workloads are useful to compare the results obtained from the testbed and the ad-hoc simulator without limiting the experiments to a specific scenario. By contrast, workloads with Concentrated multicast traffic (called *Concentrated workloads*) aim to capture the congestion and traffic concentration on critical links for real-world scenarios, e.g., increase in video traffic during peak hours, live transmission of local events, etc. This usually results in higher load in some parts of the network.

We use Homogeneous workloads without churn to validate the results obtained from the ad-hoc simulator against the ones from the testbed. To this end, we distribute receiver and source nodes uniformly across the network and select traffic demands of multicast groups with a uniform probability from 2, 5 and 8 Mbps. To generate the Homogeneous workloads, we distribute the location of sender and receivers of each multicast group in a uniform way across the nodes in the network and we associate 13 receivers to each multicast group. Then we make 20 workload runs to compare the results obtained from the testbed and from the simulator as shown in Section 3.5.5.1.

Concentrated workloads are used to emulate realistic traffic concentration scenarios while analyzing the performance of the different load-balancing algorithms with the simulator. To generate them, we use a non-uniform probability distribution to select sources and receivers among network nodes. More precisely, we randomly select a

node, representing a hotspot, with a uniform distribution for each workload run. Then we compute the shortest path lengths from this node to all the other nodes in the network. We use the negative of these path lengths as exponents to compute the exponentially scaled distances from the node to all the other nodes. After that, we divide the exponentially scaled distance of each node with the sum of the distances of all the nodes to obtain the probability distribution of the nodes. To generate the Concentrated workloads without churn, we enforce congestion in hotspots with traffic generated by a subset of nodes in the vicinity. More precisely, we select 13 out of 39 nodes as receivers in the network and a source with the above probability distribution of the nodes for each multicast group. This set up has been chosen empirically to create congestion only on small parts of the network, not in the whole network. Finally, to generate Concentrated workloads with churn, we select the source nodes using only 33% of total network nodes. Then we instantiate 20 receivers for each multicast group, similar to multicast group sessions without churn and use an $M/M/\infty$ queue for each group to generate churn for a duration of $300s$. We execute 500 simulation runs of each workload for each algorithm to allow fair comparison among DST-PL, DST-LU and L2BM routing algorithms.

3.5.4 Evaluation Metrics

Measure of Link Utilization: We compute the three following measures of link utilization to evaluate the performance of the different multicast routing algorithms with the workloads described above: 1) Average (Avg) refers to the average utilization of all the links in the network, 2) Standard Deviation (StdDev) estimates the imbalance of traffic spread across the links and 3) Critical Links denotes the percentage of links with utilization higher than 90%. These three different measures of link utilization are used to analyze the network bandwidth consumption and qualitatively estimate the impact of guaranteed-bandwidth flows on best-effort traffic for the different algorithms. A high Avg value means that best-effort traffic will have less overall network bandwidth available. The StdDev measure illustrates uneven spread of available bandwidth across the network links. In particular, a high value of StdDev means a high probability of congestion for best-effort flows and unfair share of link capacities across the network between best-effort and guaranteed-bandwidth traffic. Finally, the Critical Links measure is used to estimate the concentration of guaranteed-bandwidth traffic in the network.

In the case of scenarios without churn, we use a snapshot of network links' utilization once all receivers have joined their multicast groups. Then we compute the average of the metrics over all the runs of the workload with the same number of multicast groups. We use the workloads by varying the number of multicast groups from 10 to 150 to study the behavior of the algorithms when increasing the traffic.

Regarding scenarios with churn, we take a snapshot of network links' utilization at every second, compute the Avg, StdDev and Critical Links metrics along with the Exponential Moving Average of the metrics to study the behavior of the different algorithms over the run duration. We validate the $M/M/\infty$ queue based simulation model as described in Section 3.5.3 by studying the link metrics over the entire period of a single workload run involving 130 multicast groups. For all the metrics,

we compute an exponential moving average with a smoothing factor of 0.6 for the whole run duration. Apart from validating the simulation model, we study the performance of all the algorithms over multiple runs similar to the scenarios without churn. Also, instead of taking a single snapshot of network links' utilization, we compute the average over the entire run of experiments.

Apart from link utilization metrics, we use the *Bandwidth Demands Acceptance Ratio* to compare the performance of the different algorithms. This ratio is obtained by computing the number of successful join group requests over the total number of join group requests sent. We compute this metric for scenarios with and without churn and for each run of the workload experiments. Then we average it over multiple runs and we plot the 95% confidence interval.

3.5.5 Results and Analysis

First, we study in Section 3.5.5.1 the link evaluation metrics obtained by running experiments on the testbed and on the ad-hoc simulator using the same workloads. After this analysis, all the results shown in the rest of the paper are obtained using the ad-hoc simulator and with the Concentrated workloads. Second, we study the performance of the different algorithms with the Concentrated workloads without churn in 3.5.5.2. Third, we validate the $M/M/\infty$ queue model for the Concentrated workloads with churn in 3.5.5.3 then we analyze the performance results obtained with the Concentrated workloads with churn in 3.5.5.4.

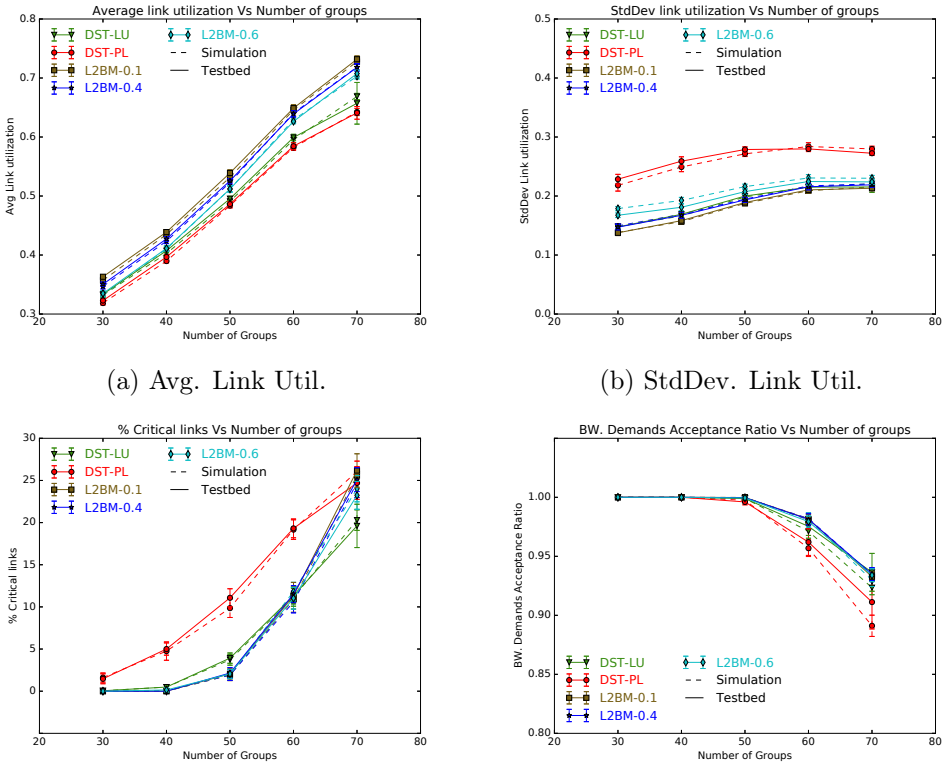
3.5.5.1 Validation of the Simulator with the Testbed

In order to compare performance results obtained with the testbed and the simulator, we perform 20 runs of each Homogeneous workloads without churn and plot average for each metric with 95% confidence interval in 3.6. Figures 3.6a-d show superimposed Avg and StdDev measures of link utilization along with the Critical Links metric obtained with the simulator and the testbed experiments for DST-PL, DST-LU and L2BM and for three different values of the initial threshold $\theta_{init} = \{0.1, 0.4, 0.6\}$.

As we can observe, the performance obtained with the simulator for each algorithm closely follows the results obtained in the testbed environment, which validates the implementation of the simulator. From now on, we use the simulator along with the Concentrated workloads to further study the performance of the different algorithms.

3.5.5.2 Simulation Results using Workloads without Churn

Here we evaluate the algorithms using multicast group session without churn of receiver nodes with the Concentrated workloads described in 3.5.3. Figures 3.7a-d show the Avg and StdDev measures of links' utilization, along with the Critical Links and bandwidth demands acceptance ratio metrics obtained without churn. DST-PL obtains 5-15% lower Avg, 5% higher StdDev and 5-12% lower bandwidth demands



(a) Avg. Link Util.

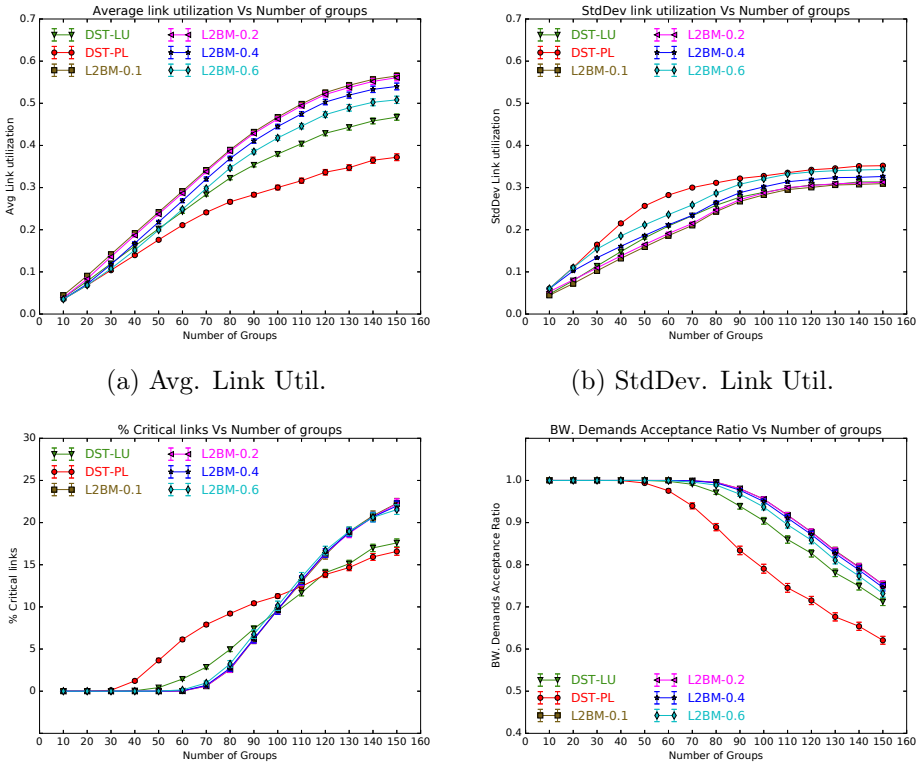
(b) StdDev. Link Util.

(c) Percentage of Critical Links

(d) Bw. Demands Acceptance Ratio

Figure 3.6: Comparison of Testbed and Simulation results

acceptance ratio compared to L2BM with $\theta_{init} = 0.1$. Indeed, DST-PL computes the shortest path to the multicast tree, which rapidly congest the critical links around the hot spots of network. For low traffic workloads (i.e., 10 to 60 multicast groups), all the algorithms accept 100% of the bandwidth demands. We recall that for all the algorithms, multicast join requests are satisfied only if sufficient bandwidth can be reserved on the links. The three variants of L2BM (particularly, $\theta_{init} = 0.10$) have 5% higher Avg link utilization than DST-LU and DST-PL. But unlike the rest of the algorithms, at low traffic the L2BM algorithms obtain zero Critical Links because it aggressively minimizes the maximum utilization of the links by not using the links operating above current threshold, see Figure 3.7c. We can note that the StdDev for DST-LU and L2BM does not vary significantly, so the two algorithms equally distribute the traffic across the network. However, we can observe for moderate traffic workloads (i.e., with 70 to 100 multicast groups) that DST-LU suffers from congestion on few Critical Links. At heavy traffic workloads (i.e., with more than 100 multicast groups) L2BM suffers from more congestion with 5% more critical links compared to DST-PL and DST-LU as shown in Figure 3.7c. In return, L2BM improves the bandwidth demands acceptance ratio by 5% and 15% compared to DST-LU and DST-PL, respectively. Overall, for a given reserved link capacity for guaranteed-bandwidth multicast traffic, L2BM is able to serve a higher percentage of bandwidth demand multicast requests compared to DST-LU and DST-PL.



(a) Avg. Link Util.

(b) StdDev. Link Util.

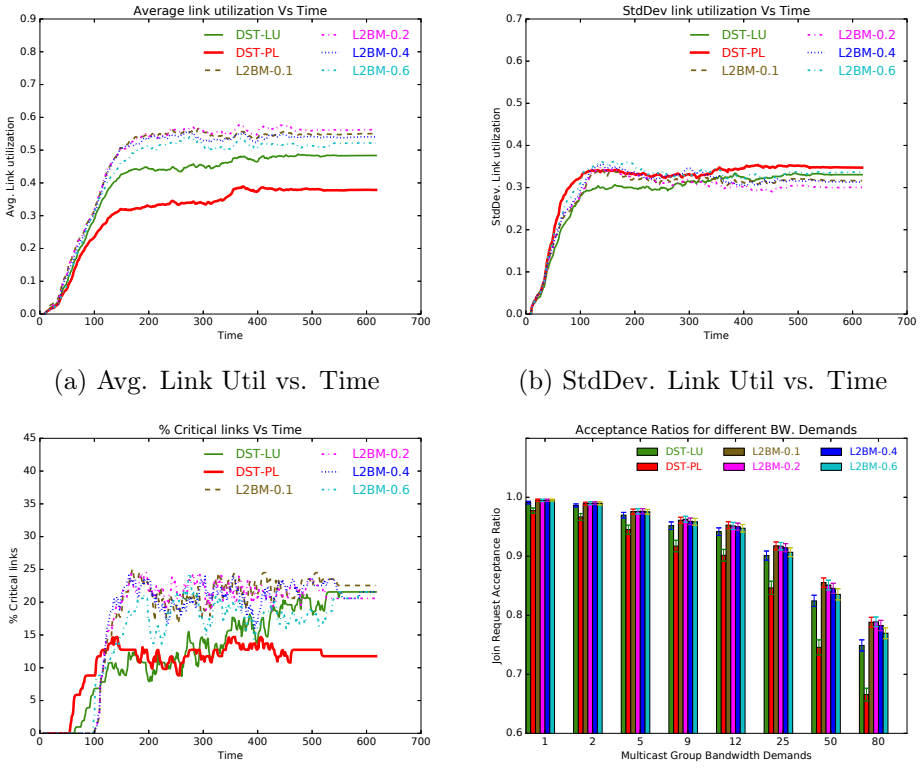
(c) Percentage of Critical Links

(d) Bw. Demands Acceptance Ratio

Figure 3.7: Results using Concentrated workloads without churn

3.5.5.3 Validation of the $M/M/\infty$ Model using Workloads with Churn

Next, we validate the $M/M/\infty$ queue based on the simulation model for the scenario with churn and the above mentioned link utilization metrics. We take an example run of moderate traffic workload of 130 multicast groups with Concentrated traffic scenario to assess the performance of the algorithms over the run duration to study the traffic pattern generated by the model over the run. Figures 3.8a, 3.8b and 3.8c show the Exponential Moving Average of the Avg and StdDev measures and the Critical Links metric at every second of the workload run. As we can observe, after continuous initial increase till 150 seconds in the value of all the metrics, the effect of churn begins and the links' utilization metrics maintain the value of all the metrics in a small range, e.g., the Avg link utilization value ranges from 0.3 to 0.4 for DST-PL, 0.4 to 0.5 for DST-LU and 0.6 to 0.6 for L2BMs. This can be explained by a first period in which multicast receivers join, followed by a period where receivers both join and leave based on $M/M/\infty$ queue model. Concerning the StdDev and the Critical Links metrics in Figures 3.8b and 3.8c, the performance of the algorithms are indistinguishable during the execution period. Therefore, we execute 500 runs of the workload and study the bandwidth demands acceptance ratio metric as shown in Figure 3.8d. The x-axis shows different values (1, 2, 5, 9, 12, 25, 50 and 80 Mbps) of bandwidth demands corresponding to multicast join requests whereas the y-axis



(a) Avg. Link Util vs. Time

(b) StdDev. Link Util vs. Time

(c) Percentage of Critical Links

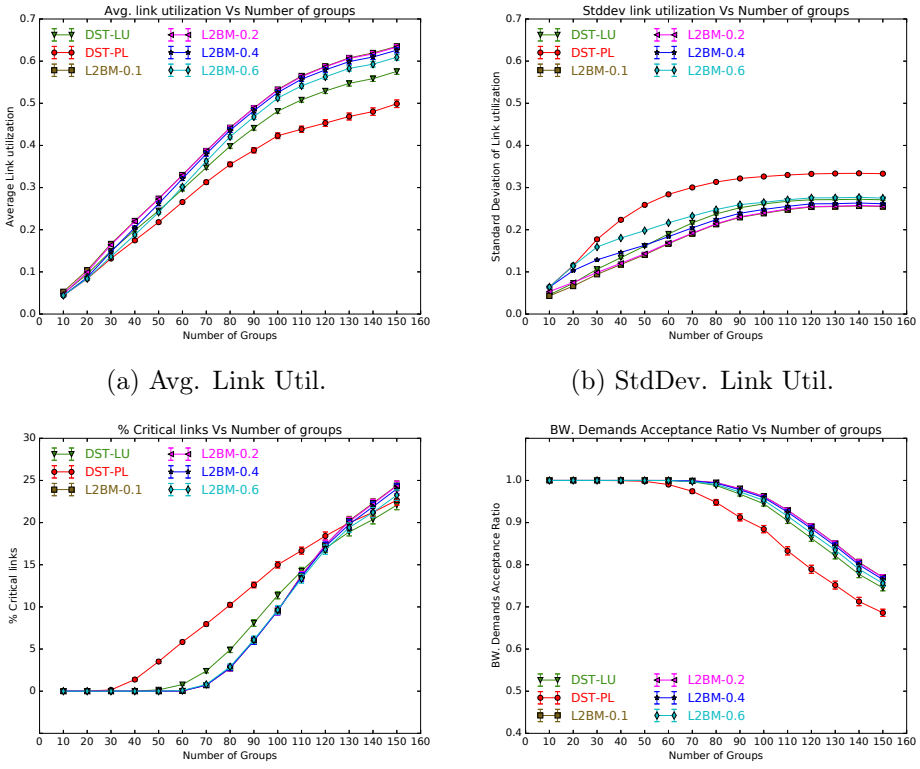
(d) Join Acceptance Ratio

Figure 3.8: Results using Concentrated workloads with churn and 130 multicast groups

shows the acceptance ratio obtained for each demand. For this workload, L2BM-0.1 accepts 2-3% more requests than DST-LU whatever bandwidth demands values of join requests. We note that none of the algorithms are heavily biased towards some bandwidth demands values of join requests. For all the algorithms, the join request acceptance ratio gradually drops when increasing the values of the bandwidth demands. The main reason is that low bandwidth demands values can be satisfied using small residual capacity available on heavily utilized links.

3.5.5.4 Simulation Results using Workloads with Churn

We further evaluate the algorithms using workloads with churn and different number of multicast groups varying from 10 to 150. Figures 3.9a and 3.9b show the Avg and StdDev measures of links' utilization. L2BM-0.1 uses 5% and 15% more Avg link utilization compared to DST-LU and DST-PL, respectively. see Figure 3.9a. Similar to the scenario without churn, all the algorithms accept 100% of guaranteed-bandwidth multicast requests for the low traffic workloads with 10 to 60 multicast groups, see Figure 3.9d. However, for the same workloads, L2BM does not use any link above 90% of the capacity as shown in Figure 3.9c. There is no significant difference between DST-LU and L2BM for bandwidth acceptance ratio in moderate traffic



(a) Avg. Link Util.

(b) StdDev. Link Util.

(c) Percentage of Critical Links

(d) Bw. Demands Acceptance Ratio

Figure 3.9: Results of L2BM, DST-PL and DST-LU using Concentrated workloads with churn

workloads(i.e., 70 to 100 multicast groups), but DST-LU suffers from congestion on 3-4% more links as shown in Figure 3.9c. As we increase the workload to heavy traffic, L2BM increases the percentage of Critical Links by 2-3%, but it increases the *Bandwidth Demand Acceptance Ratio* by 8-10% and 3-4% compared to DST-PL and DST-LU, respectively for $\theta_{init} = 0.1$.

Overall, in presence of low concentrated traffic, L2BM increases the Avg link utilization (due to the early load balancing that results in using paths longer than the shortest one) but minimizes the traffic concentration on a set of links near hot spots. Thereby, it provides more residual bandwidth on these links from θ_{max} link capacity allocated for guaranteed-bandwidth multicast traffic, being more accommodating and friendly to the best-effort traffic on these links. When the traffic is highly concentrated, L2BM is able to accept a higher number of guaranteed-bandwidth multicast requests than the other algorithms by using the threshold based technique. By doing so, L2BM allows to accept more join requests at the cost of slightly increasing the number of Critical Links as shown in 3.7c. Hence, it is able to more efficiently utilize the allocated network bandwidth on the links.

3.6 Related Work

Several approaches have been proposed to provide multicast solutions that leverage the logically centralized SDN control plane [Yap *et al.* 2010, Marcondes *et al.* 2012, Bondan *et al.* 2013, Craig *et al.* 2015, Tang *et al.* 2014, Zhang *et al.* 2015, Ruckert *et al.* 2016]. However, they do not address the specific scalability issue emerging from centralized processing of group membership messages at the controllers and do not provide low-cost best-effort friendly traffic engineering policy in presence of QoS guaranteed multicast traffic.

On the Scalability of the SDN Control Plane and of Multicast Group Management

First, we study the approaches addressing multicast group membership management regarding the SDN centralized control plane. In [Yap *et al.* 2010], authors have proposed an SDN-based multicasting scheme to implement XMPP chat sessions, referred as P2PChat. They propose to run a P2PChat Server to aggregate XMPP subscription messages, assign multicast IP addresses and submit IPs of the subscribers to a controller. However, this approach is not scalable because every join/leave request from the subscribers will result in an update message from the P2PChat Server to the controller. Also, the work does not address scenarios involving inter-domain multicast and group membership management in ISP networks, unlike us. CastFlow [Marcondes *et al.* 2012] proposes a clean-slate approach for multicast with membership churn, which does not use IGMP. A server component named Topology Server is responsible for multicast group configuration, and handles group join and leave events, but the replacement mechanism for IGMP messages is not detailed. Also, CastFlow does not address the scalability issue arising from centralized processing of group membership messages from receivers in real-world multicast scenarios. MultiFlow [Bondan *et al.* 2013] is another clean-slate approach that uses IGMP messages for group membership management. The work focuses on decreasing the delay in the configuration of the multicast groups. In [Craig *et al.* 2015], some optimization is proposed to prevent IGMP flooding in the network domain of the controller but still, all the IGMP membership messages have to reach the controller, which may overwhelm it for large multicast sessions.

In [Ruckert *et al.* 2016], authors have proposed the Software Defined Multicast (SDM) service and some APIs for efficient delivery of OTT multicast traffic in ISP networks. However, *SDM* relies on a centralized group management mechanism for its implementation in ISP networks and group management events are received from OTT content providers via some APIs, instead of using IGMP messages. Transparent to OTT clients, the *SDM* service converts OTT unicast traffic to multicast and vice versa. Multicast-to-unicast translation at the edge switches in the network may be an interesting approach for small groups but it is not able to scale with very large groups.

Regardless multicast, distributed [Yeganeh & Ganjali 2016, Phemius *et al.* 2014, Koponen *et al.* 2010] and hierarchical [Yeganeh & Ganjali 2012, Fu *et al.* 2015] approaches have been proposed in the literature to increase the scalability of the SDN control plane. However, sharing the network view and applications state across multiple controller instances is complex, costly and can lead to unacceptable latency to handle data plane events. Dynamically increasing the number of controllers (ElastiCon [Dixit *et al.* 2014]) and redistributing switches across them can help to handle surge in control workload (i.e., IGMP join/leave requests) [Dixit *et al.* 2014], but control elements still have to receive and handle all the IGMP messages and to coordinate to ensure a consistent view, which is problematic for large-scale multicast. Concerning hierarchical approaches, they require southbound protocol specific mechanisms to maintain consistency across the controllers within different hierarchies, which decreases the visibility of the global network. Our approach exploits the hierarchy of the network but does not use specific control messages or event type of southbound protocols as it is the case in Kandoo [Yeganeh & Ganjali 2012] to communicate between controllers at different levels in the hierarchy. Instead, messages exchanges among MNFs are done independently of the southbound protocol. This gives flexibility in programming network functions with the required degree of distributed processing and state management without altering the southbound control protocols neither the controllers.

Recently, adding stateful programmable control logic in the switches has been proposed to offload logically centralized controllers in OpenState [Bianchi *et al.* 2014], POF [Song 2013], P4 [Bosshart *et al.* 2014] and SNAP [Arashloo *et al.* 2016]. With the availability of Protocol Independent Switch Architecture [Bosshart *et al.* 2013a], it might be possible to provide the MNFs functionality by programming the switches. As switch technology evolves and provides stateful programmable control logic, it would be interesting to explore the possibility of implementing MNFs using such approaches.

In [Zhang *et al.* 2015], a network architecture similar than ours is proposed, where the SDN controller is responsible for setting up routing paths and NFV nodes to run specific NFs like packet filtering and video transcoding. However, they do not tackle the group membership scalability issue and their multicast routing algorithm targets a different objective than ours, minimizing the sum of network link and node utilization.

On Multicast Routing with Guaranteed-Bandwidth

Multicast routing with QoS guarantee and traffic engineering has a rich literature. However, due to limited deployment of multicast and distributed architecture of ISP networks, the bulk of the proposed algorithms e.g., [Kodialam *et al.* 2003, Chakraborty *et al.* 2003, Seok *et al.* 2002, Crichigno & Baran 2004, Youm *et al.* 2013] have never been used in practice by multicast routing protocols like DVMPR, PIM, and MOSPF. Note that none of these algorithms considers the impact of guaranteed-bandwidth multicast flows on best-effort or other lower priority guaranteed-QoS traffic.

In [Kodialam *et al.* 2003], authors have defined the Maximum Multicast Flow (MMF) problem. They propose to update the link weights and to use a greedy heuristic of nearest neighbor for the Dynamic Steiner Tree. In contrast, L2BM does not make such assumption neither requires prior knowledge of future demands. Also, the computational time complexity of our algorithm is equivalent to the one of breadth-first search, while to update the link weights, MMF has $\mathcal{O}(|\mathcal{E}|^2 * \log^2 Max(C_{uv}))$ time complexity.

Multiple QoS constrained Dynamic Multicast Routing (MQ-MDR) [Chakraborty *et al.* 2003] is a mechanism proposed to guarantee jitter, delay and loss metrics apart from bandwidth and to minimize the traffic load in the network. However, unlike L2BM, MQ-MDR requires that multicast join requests are tagged with some participation duration information, which are used to assign weights to the links.

Other objectives like minimizing the maximum link utilization can be used to efficiently serve future multicast join requests. For instance, the Hop-count Constrained Multicast Tree heuristic [Seok *et al.* 2002] aims to find a path connecting a receiver to the existing multicast tree that minimizes the maximum link utilization of the path and with a shorter length than with the Hop-count constraint. The Multi-objective Multicast routing Algorithm (MMA) [Crichigno & Baran 2004] considers the tree cost as another metric to minimize along with maximum link utilization. However, decreasing the Avg link utilization does not guarantee a higher acceptance ratio of guaranteed-bandwidth join requests as we show in Section 3.5.5. L2BM attempts both to reduce the Avg link utilization (to be friendly with best-effort traffic) and to accept more guaranteed-bandwidth join requests.

In [Youm *et al.* 2013], the Nearest Node First Dijkstra Algorithm (NNFDA) is proposed, corresponding to DST-PL described in Section 3.5.1. Our results show that L2BM is able to perform better than DST-PL both in terms of load-balancing and bandwidth demands acceptance ratio. An alternative load-balancing solution for multicast traffic consists in splitting each multicast flow in multiple thin-streams sent in different multicast trees, as proposed in DYNSDM [Ruckert *et al.* 2016]. However such an approach requires packets reordering at the receivers of the multicast stream, which increases jitter.

3.7 Conclusion

In this chapter, we propose an NFV-based approach to overcome the multicast scalability issue of centralized SDN architectures. Then we present a novel threshold-based load balancing algorithm to deploy at low cost a guaranteed-bandwidth multicast service that nicely cohabits with best effort traffic. Our solution uses a traffic engineering mechanism to split the network bandwidth between best-effort traffic and guaranteed-bandwidth multicast traffic. We show that it is able to accept 5% more bandwidth demands compared to state-of-the-art algorithms for traffic scenarios representing flash crowd and prime-time streaming of videos. The source code

and scripts used to evaluate our solution are made available to the community to ease reproduction of the experimental results[9].

Next, in Chapter 4, we describe our automation tool, DiG, developed to execute SDN experiments using high volume compute devices available in a grid. We used DiG to evaluate the L2BM traffic engineering mechanism for ISP core networks proposed in this chapter.

Then, in part II of the thesis, we use the P4 packet processing language to increase further programmability for in-network state management and computation at the edge of ISP networks. Particularly, we address drawbacks and limitations of realizing network control delegation with OpenFlow described in 3.4.2.1.

[9]See URL https://team.inria.fr/diana/software/l2bm/

4 SDN Experiments with Resource Guarantee

Contents

We are witnessing a considerable amount of research work related to SDN-enabled data center, cloud and ISP infrastructures but evaluations are often limited to small-scale scenarios as very few researchers have access to a real infrastructure to confront their ideas to reality. In this chapter, we present our experiment automation tool, DiG (*Data centers in the Grid*), which explicitly allocates physical resources in grids to emulate SDN-enabled data center, cloud or ISP networks. We used DiG, in Section 3.5, to evaluate the L2BM traffic engineering mechanism, proposed in Section 3.3 for ISP's SDN enabled core networks. DiG allows one to utilize grid infrastructures to evaluate research ideas pertaining to SDN-enabled network environments at massive scale and with real traffic workload. We have automated the procedure of building target network topologies while respecting available physical resources in the grid against the demand of links and hosts in the SDN-enabled experimental network. DiG can automatically build a large network topology composed of hundreds of servers and execute various traffic intensive workloads e.g., Hadoop Benchmarks to generate data center traffic. Also, we show that not only available physical resource capacity should be respected, but it is necessary to take into account resources' performance degradation due to tools and technologies used for emulation or virtualization. Hence, it is required to find *Operational Region* of resource considering performance degradation. Also, experimental networks should be emulated on physical infrastructure considering every physical resource's operational region as its available capacity for the emulation and virtualization technology.

4.1 Introduction

Most SDN experiments having data center, cloud and ISP network scenarios are performed with traffic traces using emulators (e.g., Mininet [Lantz *et al.* 2010], Maxinet [Wette *et al.* 2014]) or simulators (e.g. ns-3 [Consortium 2008]) due to restricted access to real production environments of companies like Amazon, Google, or Facebook. Therefore, experiment results may be biased or noisy due to modeling techniques of simulators or unaccounted and excessive usage of physical resources in case of emulation.

Prior to release of DiG [Soni *et al.* 2015], many tools were already available like Mininet [Lantz *et al.* 2010] and Maxinet [Wette *et al.* 2014] for running SDN experimentations. Among them, Maxinet is the closest to our work. However, it targets scalable emulation to create SDN enabled data center environments and relies on synthetic traffic generation models. Maxinet is built using Mininet, which has the capability to run real world applications to generate traffic. However, at the scale of hundreds of hosts, running such applications on emulated hosts consume computing resources and hinders emulation's scaling capability of network experiments. [Tazaki *et al.* 2013] Neither Maxinet nor Mininet provides guarantee on allocation of computing power (i.e., CPU cores) for emulated hosts to the scale of entire experiment with a minimum amount of physical resources.

In parallel of DiG, a Mininet-based Virtual Testbed for Distributed SDN Development (VT-D-SDN) [Lantz & O'Connor 2015] have been proposed. VT-D-SDN is more lightweight approach compared to emulation technology used by DiG. Also, it avoids configuration files and simplifies experimentations. However, VT-D-SDN do not provide resource guarantee. It does not take into account overhead of its cross-server tunneling technology while emulating experimental network links on the physical network links.

DISTributed systems EMulator (DISTEM) [Sarzyniec *et al.* 2013] can emulate complex network topologies, add heterogeneity (nodes and links with varying performance) and add faults to create special conditions in the experimental environment. However, early version of DISTEM [Sarzyniec *et al.* 2013] does not provide support to emulate SDN-enabled network environment with resource guarantee. Recently, authors of DISTEM[1] have added support for large SDN experiments with CPU core and link capacity allocation in their tool.

In this chapter we show importance using physical resources within their operational margin for resource guarantee to emulated network, which none of the emulation based testbed proposals have thrown light on. Using DiG, we aim (1) to create network topologies while respecting operational margin of available compute and network resource constraints and (2) to run real world applications and traffic on top of it. Since very few researchers have access to production SDN-enabled network environments and the majority of them has access to grid computing environments like Grid5000 [Balouek *et al.* 2013], the primary goal of our system is to build test environments for SDN-enabled networks in grid physical infrastructures. Using DiG,

[1]Distem Advanced Experiments http://distem.gforge.inria.fr/tuto_advanced.html#mapping-virtual-nodes-using-alevin September, 2016 http://cloud-days16.i3s.unice.fr/

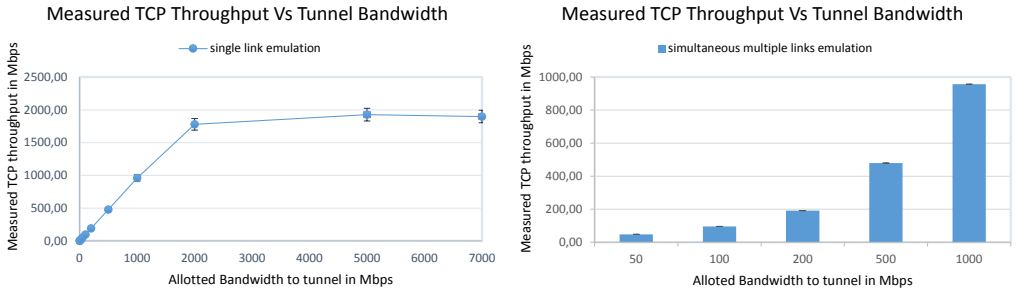
we can build overlay experimental networks by explicitly allocating available physical resources, like CPU and link capacity, up to their operational margins to the requirements of experimental network topologies, allowing to run real world data center applications on top of a grid with performance guarantees.

4.2 Operational Region and Resource Guarantee

DiG can instantiate network hosts by running virtual machines (VMs) or Docket containers. It creates forwarding devices using OpenFlow enabled switches on grid nodes. While emulating a experimental network, it is important to take into account operational regions of the available computing power of the grid nodes and the physical link capacity between each pair of grid nodes. In most of the cases, the physical network connectivity along with the computing power of the grid nodes are known by the experimenters. However, operational margins of the physical resources may not be known for the installed operating system and emulation technology available on grid infrastructure. In this section, we study an impact of emulation or virtualization technology used in DiG on available physical resources, with link capacity as a resource example.

We profile physical links for a layer 2 tunneling technology used in DiG to study performance degradation of its bandwidth capacity due to the technology along with native operating system environment. We create a pseudo-wire using L2TPv3 (Layer 2 Tunnelling Protocol Version 3) protocol to emulate a link between every node in the network. We use L2TPv3 tunneling protocol and regulate bandwidth of the tunnel using linux traffic-control to emulate a virtual link of a given capacity. We execute iperf to measure TCP throughput on both end of the tunnel to compute effective bandwidth on the emulated link. Figure 4.1a shows results of emulating a single link of varying capacities on a physical link with capacity 10Gbos. For each emulated link capacity, we perform 20 runs of the profiling experiment and plot the measured TCP throughput with 95% confidence interval. We can observe that beyond 2000 Mbps, TCP throughput does not linearly with increase in allotted bandwidth to experimental link. Therefore, physical link's operational region for L2TPv3 emulation technology can be considered up to 2000 Mbps instead of physical bandwidth of 10000 Mbps.

Next, we simultaneously emulated multiple links of different capacities on the same physical link with capacity 10000 Mbos. As we already know that operational margin of the physical link is 2000 Mbps, we restrict total capacity of the emulated links within 2000 Mbps. We perform 20 runs of this profiling experiment and plot the measured TCP throughput with 95% confidence interval for each emulated link capacity, as shown in Figure 4.1b. None of the emulated links shows performance degradation. because we allocated the physical link capacity within its operational margin. Every SDN experiments does not require to profile physical resources for their operational margin. Hence, DiG assumes that physical resources are already profiled for performance degradation due to emulation technology and their operational regions are known.



(a) Single Link Emulation (b) Multiple Links Emulation

Figure 4.1: Link emulation using L2TPv3 tunneling on a physical link with 10 Gbps

4.3 System Description

The DiG system is able to create experimental networks that carry real traffic between nodes running protocol stacks as in real world SDN-enabled networks. To achieve this, DiG implements a layer 2 overlay network on a grid infrastructure (see Figure 4.2) while respecting available physical resources and their operational margins. Having a layer 2 overlay network as an experimental network provides a bare-metal network environment to the SDN controllers and forwarding devices.

Figure 4.2 shows an example of embedding k=4,fat-tree SDN-enabled experimental network to physical hosts connected in a star topology in Grid5000 network. DiG system uses only hosts in the grid network to embed experimental network, because in most cases grid network devices like routers and switches are administrated by hosting organization. However, their physical network topologies are publicly available[2]. Similarly, Available physical resources can either publicly available [3] or trivially learned.

DiG creates out-of-band SDN-enabled experimental network by allowing experimenters to control and manage the network from outside the data plane. Experimenters can run a SDN controller of their preference and connect emulated forwarding devices in the data plane. DiG provides resource guarantees for data plane forwarding devices, links and hosts in the network. It allows to specify SDN controller's network location (IP address) to connect to forwarding devices using the same or different physical links, if available. DiG launches experiments network emulation from a centralized node, referred as *Deployment Node* in Figure 4.2. It allows to manage and monitor experiments from a centralized location.

[2]An example Grid5000 Network Topology at Rennes https://www.grid5000.fr/mediawiki/index.php/Rennes:Network

[3]An example Grid5000 Hardware Resource at Rennes https://www.grid5000.fr/mediawiki/index.php/Rennes:Hardware

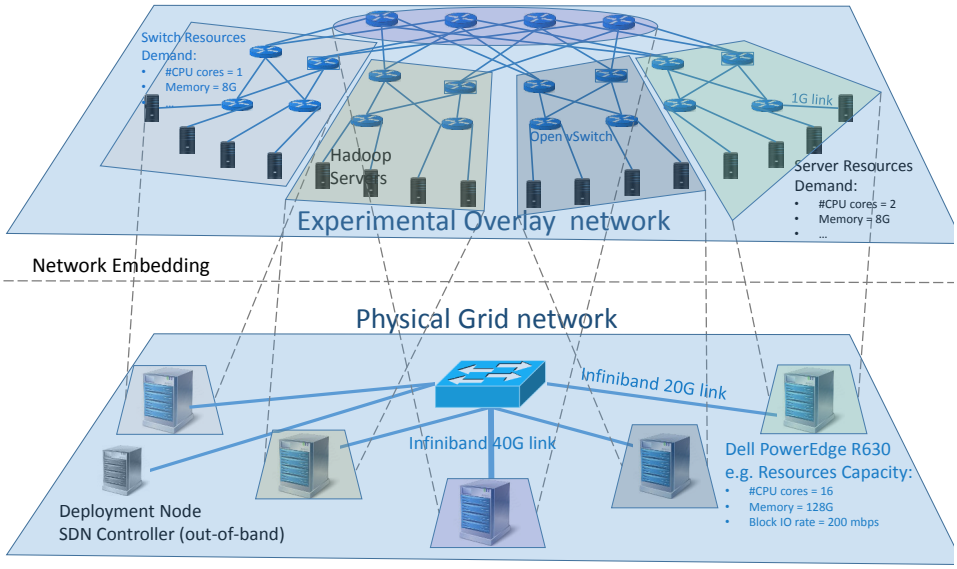


Figure 4.2: Embedding SDN-enabled experimental network with resource guaranteed

4.3.1 DiG Technical Description

The main challenge when designing DiG is to find the right trade-off between flexibility and ease of running SDN-enabled network experiments with resource guarantee. As DiG emulates networks with resource guarantee, it requires physical network information along with a SDN-enabled experimental network. Also, it can emulate a given network only if enough resources are available that satisfy the demand of the experimental network. Finally, using a single-click approach, it is able to create a layer-2 overlay network. We used a modular approach for DiG to provide flexibility and control to experimenters at various stages of the automation.

DiG maps the experimental network on the compute hosts of a physical grid network while satisfying the computing power requirements of all the nodes in the experimental network and not exceeding the computing capacity of grid hosts. Similarly, layer 2 overlay links are mapped by satisfying the demand of all the links in experimental topology while not utilizing physical links beyond their specified operational margin. So, the problem is reduced to the resolution of a Virtual Network Embedding (VNE) [Fischer *et al.* 2013] problem with constraints on nodes computing power and links capacity.

DiG comprises of three modules (see Figure 4.3) and uses three phases to set up experimental networks on grid infrastructures. Each phase generates an output in the form of text files and these files are used in the next phase as an input. This makes the system more flexible and facilitates modifications and integration of different phases implementations. The names of the three phases are *Experimental Network Embedding*, *Configuration Generation*, and *Deployment*. Note that each phase can be run in an independent way with appropriate input files, without the need of executing other phases. For example, Node Mapping can be generated by new VNE

algorithm, tool or even manually for required solution not generated or supported by algorithms in ALEVIN.

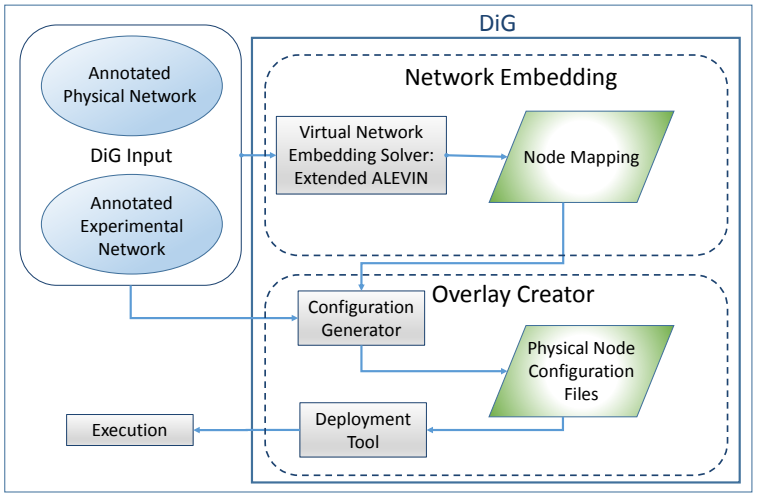


Figure 4.3: DiG Module interaction

4.3.1.1 Experimental Network Embedding

DiG solves a VNE problem using the ALEVIN [Beck *et al.* 2014] framework, which is used to generate the mapping between the experimental network and the grid physical infrastructure. ALEVIN is fed with the experimental and grid networks described in DOT language [4] in a text file. The experimental network is annotated with CPU cores requests for nodes, link capacity requests and other application-specific attributes like Hadoop node type for automation usage. The grid network is annotated with operational margin of CPU core capacity for grid nodes and link bandwidth capacity for physical links. The hosts are also annotated with IP addresses for automation purpose. ALEVIN uses CPU cores and link bandwidth attributes for both experimental and grid networks and it generates a node mapping file as shown in Fig. 4.3. The node mapping file is a text file that identifies the set of experimental network nodes mapped on each physical node.

4.3.1.2 Configuration Generator for Experimental Network

The Configuration Generator phase takes as input the mappings generated from the Experimental Network Embedding phase along with the descriptions of the experimental and grid networks in DOT format. However, the mapping file can be generated by any means, and not necessarily with the technique presented in Sec. 4.3.1.1. This allows (1) running different tools and algorithms for the network embedding step and (2) relaxing the strong dependency on the performance of embedding algorithms.

[4]DOT Language http://www.graphviz.org/doc/info/lang.html/

The Configuration Generator phase prepares the configuration files for each physical host based on the mapping. It contains the meta-data to instantiate the mapped part of the experimental network on physical hosts. The meta-data contains the appropriate commands and the parameters to instantiate the virtual machines and to map the virtual hosts to the physical nodes. It also contains the necessary information (e.g., source-destination UDP port numbers, IPs of grid nodes, tunnel unique IDs etc.) to create layer 2 tunneling protocol (i.e., L2TPv3) endpoints and links capacity information satisfying the experimental network bandwidth demand based on the mapping. Along with the experimental network configuration files, this phase generates files to bring up basic network utility (e.g., assigning IP to experimental network interface, routing etc.) in the hosts.

4.3.1.3 Deployment of Experimental Network

The last phase consists of the deployment of the experimental network using configuration files on the physical machines. DiG instantiates the virtual hosts in the experimental network on grid nodes, creates OpenFlow [McKeown *et al.* 2008] switches interconnected with L2TPv3 tunnels and controls the link bandwidth according to the requirements of the experimental network to emulate. It is also responsible to launch applications on virtual hosts of the experimental network. The Linux Traffic Control utility (*tc*) is used to control the bandwidth at the tunnel interfaces according to the links capacity requirements of the experimental network.

4.3.2 Management Network

Different versions of DiG tool emulate hosts with different virtualization technology. Experimenters can use DiG to virtualize complete host machines using *qemu-kvm* or lighter virtualzation technology *Docker* container. DiG creates dedicated management network to control and manage nodes of the experimental network emulated using complete machine virtualization. DiG uses physical network to manage containers running on remote physical hosts, if lightweight container based host virtualization is used.

4.3.2.1 Emulating hosts with complete machine virtualization

As mentioned above, the deployment phase launches applications in virtual hosts. DiG uses a designated node called *Manager node* in the grid infrastructure to launch the deployment phase in a centralized way. All the communications required for deployment purpose and management of experimental network are carried out on a dedicated management network isolated by experimental networks, as depicted in Fig. 4.4.

Each virtual host in the experimental network includes a management network interface. A management bridge is created on all the grid nodes including the manager node, as shown in Fig. 4.4. The virtual hosts of the experimental network running on a grid node are connected to the management bridge on the grid node through

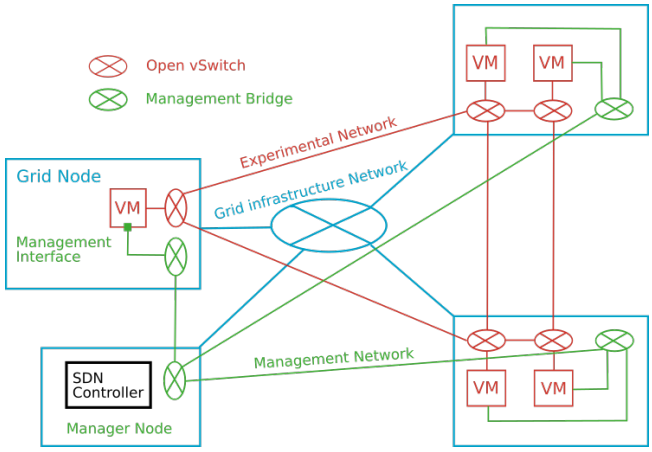


Figure 4.4: Experimental Overlay Network with Management Network

their management interface. The management bridge on each grid node is connected to the management bridge on the manager node. The Deployment tool is executed on the manager node; it uses the management network to dispatch the commands to launch applications on different VMs in the experimental network. This approach prevents any possible management traffic interfering in the experimental network that could distort experimental results.

4.4 Showcase and Usage

The primary goal of DiG is to create SDN-enabled experimental network with resource guarantee to imitate real world SDN-based data center and ISP topologies with high level of realism. Such network environments can be used for instance to test and evaluate performance of SDN controllers or routing algorithms with different real time traffic or topologies. We demonstrated the DiG in IEEE NFV-SDN, 2015 at San Francisco, USA. We showcased how to automatically emulate an OpenFlow data center k=4 Fat-tree topology in Grid5000 [5].

We use DiG to emulate a ISP core network topology to evaluate our traffic engineering proposal L2BM described in Chapter 3.3. Also, we use DiG to emulate data center topology k=8 Fat-tree to evaluate flow placement algorithm aOFFICER [Nguyen 2016]. We run Hadoop benchmark programs to generate data center traffic to evaluate aOFFICER. Hadoop is used in many real world data centers and many benchmark suites exist (e.g., HiBench [6]). Interestingly, Hadoop MapReduce applications generate a substantial amount of traffic during the data shuffling phase and particularly the TestDFSIO and TeraSort are MapReduce benchmark applications. Hence, they are primary choices for data-center workload generation to demonstrate the effectiveness of DiG.

[5] The video of the demo is available at https://youtu.be/wa1po2jmf7g
[6] https://github.com/intel-hadoop/HiBench

4.5 Conclusion

In this chapter, we presented the DiG tool to create SDN data center and ISP networks on a Grid infrastructure. We also showed the need of knowing performance cost of emulation and virtualization technology. DiG runs network embedding algorithms to emulate SDN-enabled network infrastructures in a Grid with performance guarantees. It automatically creates L2 overlay experimental networks and hosts based on the output of the embedding algorithms and can launch any off-the-shelf application on the experimental hosts to generate workload on the data center to evaluate. DiG is available to the community at URL http://team.inria.fr/diana/software/.

Part II

Modular SDN Data Plane Architecture

In this part, we describe a novel SDN data plane architecture required to realize modular network control and management. In part I of the thesis, we use a modular approach for application specific network control delegation and allow NFs running in NFVI to control and manage the pertaining traffic. Our modular control delegation approach requires to allow network control applications and functions to process the shared data traffic or isolate an application specific traffic from the rest of the applications.

Network hypervisors like CoVisor [Jin *et al.* 2015] also aim to modularize the deployment of network control applications or functions. CoVisor relies on a fixed control data plane interface, OpenFlow, and its properties. It decreases the programmatic flexibility of individually developed network control applications and functions and also constrain them through OpenFlow and its properties. This hypervisor approach adds an extra layer of OpenFlow communication. The CoVisor's centralized control plane remains vulnerable to scalability issues. Its composition of control applications can degrade performance of all the deployed applications and functions in the case only one of them is processing, inefficiently, frequent network events in its control plane. CoVisor enables modular deployment of applications but does not modularize the control and management of packet processing behavior of forwarding devices in the network.

Orthogonal to the CoVisor approach of composing packet processing functionalities of modularized control applications and functions in control plane, we compose the module's functionality in the fastpath of forwarding devices in the SDN data plane. We use the P4 packet processing language that provides more flexible packet processing abstraction than OpenFlow. In the next chapter, we describe the modular SDN data plane architecture, P4Bricks, which enables multiprocessing of control applications and functions developed and deployed as individual modules for packet processing functionalities in the network. P4Bricks also addresses the limitations of our OpenFlow based implementation mentioned in Section 3.4.2.

5 P4Bricks: Multiprocessing SDN Data Plane Using P4

Contents

Packet-level programming languages such as P4 usually require to describe all packet processing functionalities for a given programmable network device within a single program. However, this approach monopolizes the device by a single large network application program, which prevents possible addition of new functionalities by other independently written network applications. We propose P4Bricks, a system which aims to deploy and execute multiple independently developed and compiled P4 programs on the same reconfigurable hardware device. P4Bricks is based on a Linker component that merges the programmable parsers/deparsers and restructures the logical pipeline of P4 programs by refactoring, decomposing and scheduling the pipelines' tables. It merges P4 programs according to packet processing semantics (parallel or sequential) specified by the network operator and runs the programs on the stages of the same hardware pipeline, thereby enabling multiprocessing. In this work, we present the initial design of our system with an ongoing implementation and study P4 language's fundamental constructs facilitating merging of independently written programs.

5.1 Introduction

P4 [The P4 Language Consortium 2017] is a high-level language for programming protocol-independent packet processors. It allows reconfiguring packet processing behavior of already deployed data plane hardware devices, introducing new protocols and their processing to the devices, hence decoupling packet processing hardware from software. This provides high degree of flexibility for programming new network

applications and packet processing functionalities using reconfigurable hardware like RMT [Bosshart *et al.* 2013c] and Intel Flexpipe™. With P4, a network operator can execute one program at a time on the target reconfigurable network device. However, as the number of features or network applications to be supported by the device grows, P4 programs increase in complexity and size. The development and maintenance of such monolithic P4 programs containing all the possible features is error prone and needs huge time and effort as program complexity grows. On the other hand, this approach does not allow to easily compose independently written modules in a single P4 program. Network devices can have different packet processing requirements according to their role and location in the network. Deploying one large program for a small subset of applications results in inefficient utilization of the reconfigurable device resources.

P5 [Abhashkumar *et al.* 2017] optimizes resource utilization by leveraging policy intents specifying which features are required to remove excess applications and packet processing functionalities. However, with P5 there is still a large monolithic program to be configured based on the policy intents. ClickP4 [Zhou & Bi 2017] proposes a smart modular approach in which separate modules can be developed within the ClickP4 programming framework and then manually integrated into ClickP4 configuration files to create a large program. However, programmers are required to know the code of the ClickP4 library modules to integrate a new module into the ClickP4 framework as source code modifications may be required for modules on the already developed code base. Basically, P5 allows removing extra modules and features from already composed small P4 programs, whereas ClickP4 gives choice to select from a list of modules. Most importantly, with both P5 and ClickP4, packet processing functionalities on a device can not be easily composed using independently developed and compiled P4 programs.

Hyper4 [Hancock & van der Merwe 2016], HyperV [Zhang *et al.* 2017a] and MPVisor [Zhang *et al.* 2017b] propose virtualization of programmable data plane in order to deploy and run independently developed multiple P4 programs on the same network device at the same time. In these approaches, a general purpose P4 program working as a hypervisor for programmable data plane is developed, which can be configured to achieve functionally equivalent packet processing behavior of multiple P4 programs hosted by it. However, virtualization requires minimum 6-7× and 3-5× more match-action stages for every P4 program compared to its native execution for Hyper4 and HyperV, respectively. Also, such approaches show significant performance degradation for bandwidth and delay, thereby nullifying the benefit of high performance reconfigurable hardware.

Meanwhile, executing efficiently multiple P4 programs at a time on a same target device is highly desirable. We believe that a network operator should be able to easily add any features on its target device with programs potentially developed by different providers.

We present the design and architecture of P4Bricks, our under development system that aims to deploy and execute multiple independently developed and compiled P4 programs on the same reconfigurable device. P4Bricks comprises two components, called Linker and Runtime. The Linker component merges the programmable

parsers and deparsers and restructures the logical pipeline of P4 programs by refactoring, decomposing and scheduling their match-action tables (MATs). The Runtime component translates the dynamic table updates from the control planes of different applications into the tables of the merged pipeline. P4Bricks merges and executes MATs of multiple compiled P4 programs on the stages of the same hardware pipeline, thereby enabling multiprocessing. The idea is to provide a seamless execution environment for P4 programs in a multiprogram environment without any changes required in its control interface and MATs definitions. With P4Bricks network operators can specify the packet processing policy on the target device in terms of compiled P4 programs and composition operators using a simple command line interface.

This report presents the initial design of our system with an ongoing implementation and studies P4 language's fundamental constructs facilitating merging of independently written programs. It is organized as follows. Section 5.2 provides an overview of the P4Bricks system. Section 5.3 describes Linker, which composes compiled P4 programs using the only knowledge of MATs definitions. Then, Section 5.4 describes the Runtime module that interacts with the Linker component and the control plane of P4 programs to manage flow entries in the MATs of the programs.

5.2 System Overview

In this section, we provide a brief overview of the P4 language and we then introduce our system, describing merging of programmable blocks at link time and their management at runtime.

5.2.1 P4 background

P4 [The P4 Language Consortium 2017] is a high-level language, based on programmable blocks, used to define protocol-independent packet processing by programming data plane of reconfigurable target devices. A P4-compatible target manufacturer provides P4 architecture defining the programmable blocks and describing several hardware related information. Essentially, the P4 architecture of a target provides programmable components and declares interface to program them and exposes specific and already implemented constructs(e.g., checksum units and algorithms) that can be used and manipulated through APIs. The programmable parser, deparser and logical match-action pipeline are the main blocks used to program the data plane of packet processing targets. In addition of providing data plane programmability, P4 generates the APIs for control plane of the target device to communicate with the data plane. The APIs allow to manage the state of data plane objects from the control plane.

P4 provides the following fundamental abstractions to program reconfigurable target devices.

- *Header types* - to describe the format (ordered sequence of fields and their size) of headers within a packet.

- *Parser* - to describe all the possible sequences of headers within received packets, mechanism to identify the header sequences, headers and the values of the fields.
- *Actions* - are already implemented in the target, hence their behaviors are fixed. They are used manipulate header fields and metadata, also may take data as input from control plane at runtime.
- *Tables* - allows P4 programmer to define match keys and associate them with actions. A Match key can have multiple header and metadata fields. A Match key provides complex match and decision capabilities.
- *Match-action unit* - implements a table for execution at runtime. It performs table lookup using the match key to search associated actions and data, and executes the actions.
- *Control Flow* - to describe packet processing program, which includes invoking sequence of Match-action units. Packet reassembly can also be defined using control flow.
- *extern* - are architecture and hardware specific objects and implementations, which can be manipulated but not programmed. Because, their processing behaviors are implemented in the target.
- *Intrinsic metadata* - are architecture and target specific entities associated with each packet(e.g., interfaces)
- *User-defined metadata* - are data structures and variables defined in a program to maintain the program specific per packet state during packet processing.

The P4 language is constituted of four different sub-languages used for different purposes.

1. The core language - to describe types, variables, scoping, declarations, statements etc.
2. A sub-language for describing parsers - having specific constructs to describe packet parsing.
3. A sub-language for describing processing - using match-action units and to define traditional imperative control flow for packet processing
4. A sub-language for describing architecture - to define and declare types of programmable blocks for the target, architecture specific data and functions.

Next, we briefly discuss sub-languages for parsers and packet processing along with some of their fundamental constructs used to program Parsers, Deparser and logical match-action pipeline.

5.2.1.1 Programmable Parser and Packet Parsing in P4

All the packet processing targets must identify packets' protocol headers and their fields to determine how packets can be processed. Identifying and extracting protocol

headers and their fields within packets is called *packet parsing*. Packet parsing is a complex and challenging as the packet size, protocol header types and their payload vary across the networks and packets within the same network. Header type defines the format of a protocol header within a packet, programmers can describe it by specifying sequence of fields and their sizes. An example of header type definitions in P4 language for `Ethernet_h` and `IPv4_h` header types is given in 5.1. Every header type has a length field and an identifier field indicating length of the current header and encapsulated protocol type, respectively, to facilitate extraction of header fields and subsequent protocols headers. The first protocol header fields are extracted based on the network type from the start of packet bit-stream. Parser identifies the next protocol header to extract from the values of current protocol header's fields using a given *parse graph*, which captures all possible sequences of protocol headers using a DAG. Fixed parsers can parse packets according to the parser and protocol types defined at design time of the target. Where as programmable parsers allow to modify the parse graph and protocol headers types at any time according to new packet processing requirements.

P4 allows to define programmable parser blocks for P4-compatible reconfigurable target. Parser blocks are programmed by defining packet header types and fields, declaring instances of the types and defining a Finite State Machine (FSM) to extract the header instances from the packets' bit streams. The parse graph is encoded as FSM in P4. The FSM structure is described by defining *states* and *transitions* among the states. The programmer must define one start state for FSM of a P4 parser. P4 provides two logical final states named *accept* and *reject*, which are not part of the FSM defined by the programmer. The programmer can define next transition to accept state or reject state from any state of the parser to notify successful completion or failure packet in parsing, respectively (see code listing 5.1 for example definitions of states start and parse_ipv4 states). Each state has a name and a body comprising of a sequence of statements. The statements within a state's body describe the processing to perform when the parser transits to the state. As shown in code listing 5.1, they can be local variable declarations, assignments, function invocation(e.g., *verify* - to validate already parsed data) and method calls (to process parsed fields using *extract* and invoke other parser blocks defined within the same program). P4-compatible target implements the parser FSM in their programmable parser unit [Gibb *et al.* 2013], where FSM is converted into state transition table and loaded into the unit's memory blocks. Packets parsed into header using defined parser blocks instances are further processed using control blocks.

Listing 5.1: An example of Header Types and Parser in P4 [The P4 Language Consortium 2017]

```
typedef bit<48> EthernetAddress;
typedef bit<32> IPv4Address;
// Standard Ethernet header

header Ethernet_h {
  EthernetAddress dstAddr;
  EthernetAddress srcAddr;
  bit<16> etherType;
}
```

```
// IPv4 header (without options)
header IPv4_h {
  bit<4> version;
  bit<4> ihl;
  bit<8> diffserv;
  bit<16> totalLen;
  bit<16> identification;
  bit<3> flags;
  bit<13> fragOffset;
  bit<8> ttl;
  bit<8> protocol;
  bit<16> hdrChecksum;
  IPv4Address srcAddr;
  IPv4Address dstAddr;
}

// Structure of parsed headers
struct Parsed_packet {
  Ethernet_h ethernet;
  IPv4_h ip;
}

// Parser section
// User-defined errors that may be signaled during parsing
error {
  IPv4OptionsNotSupported,
  IPv4IncorrectVersion,
  IPv4ChecksumError
}

parser TopParser(packet_in b, out Parsed_packet p) {
  Checksum16() ck; // instantiate checksum unit

  state start {
    b.extract(p.ethernet);
    transition select(p.ethernet.etherType) {
      0x0800: parse_ipv4;
      // no default rule: all other packets rejected
    }
  }

  state parse_ipv4 {
  b.extract(p.ip);
  verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
  verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
  ck.clear();
  ck.update(p.ip);
  // Verify that packet checksum is zero
  verify(ck.get() == 16w0, error.IPv4ChecksumError);
  transition accept;
  }
}
```

5.2.1.2 Control Block

P4 provides a specific language constructs to program control block describing processing of the parsed headers and other metadata in the parser block. Actions and tables are among the fundamental packet processing abstractions declared within the body of control block. P4 uses the same control block but with specialized signature as the programming interface deparsing.

5.2.1.3 Programming match-action units

P4 language construct *tables* are used to describe packet processing in match-action units of targets. In this thesis, we use Match-Action Tables (MATs) to refer tables declared in P4 programs. Every definition of MAT must have a *key* (match key), *actions* and optionally, can have a *default action*, *entries* and *additional properties*, as shown in example code listing 5.2 for `ipv4_match` table. The key specifies data plane entities like header fields, program variables or metadata to be used to match values at runtime in hardware lookup table of a match-action unit. A key is described using a list of (field, *match_kind*) pairs, where field is a data plane entity and match_kind is a constant specifying an algorithm to match the values at runtime in the lookup table of the match-action unit. match_kind constants are useful to allocate specific memory type and resources to implement lookup table and generate control plane APIs used to manage entries in the MAT. Actions are code fragments processing data(e.g., header fields, metadata etc.,) in data plane and may additionally contain data that can be written by control plane and read by data plane. Actions used in every MAT must be defined in the P4 program. A MAT must declare all the possible actions which may appear in lookup table at runtime by assigning list of the actions to its *actions* property. For example, `Drop_action` and `Set_nhop` actions are declared in `ipv4_match` table definition in the example code listing 5.2. Actions allow control plane to influence packet processing behavior of data plane, dynamically. P4 allows to declare default action for MATs, which can be dynamically changed using control plane APIs. The default actions of a MAT is executed by the corresponding match-action unit, whenever key values do not match any entry in the lookup table. If the definition of a MAT does not have default action property declared and key values do not match any entry in the lookup table, then packet processing continues without any effect from the MAT. P4 allows to initialize look up tables by declaring a set of entries in definitions of MATs. These entries are constant and can not be changed by control plane at runtime, they can be only read. P4 allows to specify target architecture specific properties to pass additional information to compiler back-end of the target (e.g., table size, lookup table implementation hints etc.,).

Listing 5.2: An example of match-action table in P4 [The P4 Language Consortium 2017]

```
// Match-action pipeline section
control TopPipe(inout Parsed_packet headers,
          in error parseError, // parser error
          in InControl inCtrl, // input port
```

```
        out OutControl outCtrl) {
 IPv4Address nextHop; // local variable
   /**
  * Indicates that a packet is dropped by setting the
  * output port to the DROP_PORT
  */
   action Drop_action() {
   outCtrl.outputPort = DROP_PORT;
 }

 /**
  * Set the next hop and the output port.
  * Decrements ipv4 ttl field.
  * @param ivp4_dest ipv4 address of next hop
  * @param port output port
  */
 action Set_nhop(IPv4Address ipv4_dest, PortId port) {
   nextHop = ipv4_dest;
   headers.ip.ttl = headers.ip.ttl - 1;
   outCtrl.outputPort = port;
 }

 /**
  * Computes address of next IPv4 hop and output port
  * based on the IPv4 destination of the current packet.
  * Decrements packet IPv4 TTL.
  * @param nextHop IPv4 address of next hop
  */
 table ipv4_match {
   key = { headers.ip.dstAddr: lpm; } // longest-prefix match
   actions = {
     Drop_action;
     Set_nhop;
   }
   size = 1024;
   default_action = Drop_action;
 }

   ...
   // More Table definitions
   ...

   // Programming control flow
 apply {
   if (parseError != error.NoError) {
     Drop_action(); // invoke drop directly
     return;
   }
   ipv4_match.apply(); // Match result will go into nextHop
   if (outCtrl.outputPort == DROP_PORT) return;
       ...
       ...
 }
```

In control block, programmers can invoke a MAT using its *apply* method, which is a language construct provided by P4. Call to an apply method on a MAT instance execute the MAT using match-action unit of the target. The call returns a *struct* having a *boolean* and an *enum* as its fields. The return type struct and its member enum are automatically generated by the P4 compiler. The boolean member specifies if a matching entry is found in the lookup table or not. The enum member indicates the type of action executed as a result the execution. The boolean and enum fields can be used in *if* and *switch* construct to program packet processing control in the control block.

5.2.1.4 Deparser Control Block

Packets are reassembled using the processed header instances by programming the deparser control block. Deparsing is described using a control block having a mandatory specific parameter of type packet_out, as shown in code listing 5.3. P4 provides a language construct, *emit*, to reassemble the packet. Emit takes header instance as an argument and if the header instance is valid, it is appended to the packet. If the header is not valid, no operation is performed. An example deparser block in Section 5.3 appends Ethernet header, computes validity of IP header and if it is valid (modified), computes a new checksum before appending it to the packet.

Listing 5.3: An example of deparser block in P4 [The P4 Language Consortium 2017]

```
// deparser section
control TopDeparser(inout Parsed_packet p, packet_out b) {
  Checksum16() ck;
  apply {
    b.emit(p.ethernet);
    if (p.ip.isValid()) {
      ck.clear(); // prepare checksum unit
      p.ip.hdrChecksum = 16w0; // clear checksum
      ck.update(p.ip); // compute new checksum.
      p.ip.hdrChecksum = ck.get();
    }
    b.emit(p.ip);
  }
}
```

5.2.2 The P4Bricks System

P4Bricks enables network operators to deploy and execute multiple independently developed and compiled P4 programs on the same reconfigurable target device. P4Bricks allows network operator to define packet processing policy of the device using P4 programs and composition operators. It merges P4 programs according to packet processing semantics (parallel or sequential) specified by the network operator and runs the programs on the stages of the same hardware pipeline, thereby enabling multiprocessing. P4Bricks comprises two components, called Linker and Runtime. Linker takes as input the data plane configuration files generated by P4

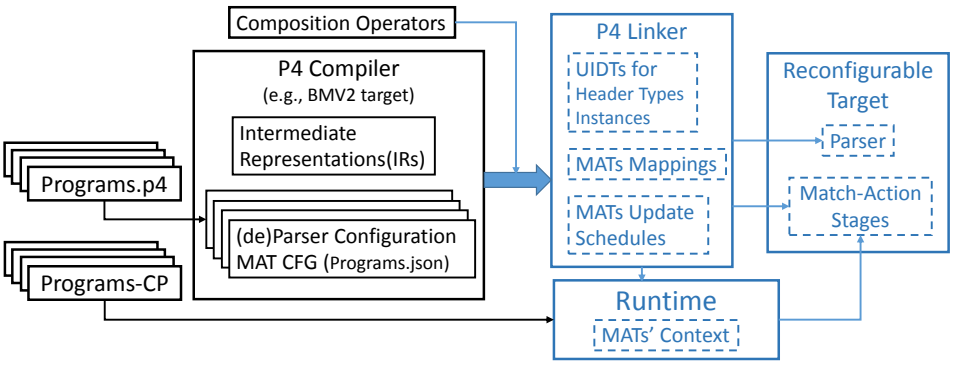


Figure 5.1: System Overview

compiler from source files of independently written P4 programs along with composition operators to apply on the programs, see Figure 5.1. It merges the programmable parser, pipeline and deparser blocks defined in the compiled configuration files of P4 programs according to packet processing policy described using parallel and sequential composition operators. It restructures the logical pipeline of P4 programs by refactoring, decomposing and scheduling their match-action tables (MATs). Linker does not assume knowledge of flow entries in MATs of P4 programs while merging the pipelines at link time. On the other hand, Runtime processes the MATs flow updates from the control plane of P4 programs at runtime, in accordance with composition operators and traffic isolation enforced by the network operator. The Runtime component translates the dynamic table updates from the control planes of different applications into the tables of the merged pipeline.

In order to merge the parsers, Linker has to identify equivalent header types and header instances present in the different P4 programs to enable sharing of common header types and instances. Two packet header types are said equivalent if they have the same format. Linker maps header types to Unique IDentifiers (UIDs) and stores the mappings between program specific IDs and UIDs in a table called header types UID Table (UIDT). Equivalent packet header instances are identified while merging the parse graphs defined in parser block of P4 programs. The parse graphs are merged by matching the sampling locations of parsers in the packet bit stream and the instances of header types extracted from the location. The mapping between program specific IDs and UIDs is also stored in a table called header instances UIDT. As merging parsers of two programs creates another parser, the composition operators can be recursively applied to merge parsers of any number of programs.

P4Bricks considers the packet header instances (extracted or emitted) along with *User-defined* and *Intrinsic* metadata associated with each packet as *data plane resources*. We consider the packet header instances and intrinsic metadata as shared data plane resources accessed for packet processing by the logical pipelines of the different P4 programs. However, we do not share user-defined metadata of one program with the other P4 programs because they are used by programmers to store the program specific state during packet processing. Linker replaces program specific IDs of header types and instances given by the compiler with the mapped ones in UIDTs before merging pipelines according to the composition operators. This unifies

different references (IDs and names) for equivalent header types and instances used in P4 programs, thereby identifying sharing of data plane resources. As *Intrinsic* metadata represents data structures and variable related to architecture and target, they are uniformly referenced across independently developed P4 programs. User-defined metadata of every program is always given a unique ID using program's name to restrict other programs from accessing it.

P4 allows to program packet deparsing using a control block having at least one parameter of a specialized type, `packet_out`. A deparser control block allows to generate the *emit* sequence of header instances to reassemble the packet, as defined by the programmer. If the instance is valid, emitting a header instance will append the instance to the outgoing packet. Similar to parser, two deparser blocks can be merged if the P4 compiler generates a DAG of *emit* instances, which encodes all possible *emits* functions after appending a header instance. However, as the P4 compiler generates a list to append header instances, the topological order (providing relative location of appending the header instance) can not be identified. The topological order is essential requirement to merge instances of different header types at the same level of the network stack. Let us consider that deparser of program P_A emits header instances in the order of Ethernet, IPv4, ICMP and TCP, whereas deparser of program P_B provides emit sequence of Ethernet, VLAN, IPv6 and UDP. Because of the semantics of *emit* calls that append the header instance only if it is valid, and the strict order imposed by the sequences of these calls, it is not possible to identify the correct merged emit order of header instances. In particular, it is not possible to deduct that the Ethernet header can be followed by either VLAN, IPv6 or IPv4. In our current P4Bricks implementation, we use the merged parse graph to identify the topological order among the header instances to be compatible with current P4 specifications. We note that if a future version of P4 makes use of DAGs to represent deparser control block, this could allow merging of deparsers without dependence to the parser block.

In P4, MATs are defined using 1) match keys composed of header fields and runtime state data of the program, and 2) actions to be taken based on matching on the keys. The packet processing control flow in the compiled configuration file of a P4 program is commonly represented as a DAG, called control flow graph (CFG), with each node representing packet processing using a MAT. The edges in CFG represent control dependency among MATs capturing packet processing order in the program. For each pipeline of a program, Linker decomposes the MATs CFG by adding resources as nodes, splitting each MAT node into match and action *control nodes*, and adding dependencies between control nodes and resource nodes according to resources accessed. Linker generates read-write *operation schedule graph* (OSG) for each resource from the decomposed CFG to capture all possible access orders and types (read or write) of operations executed on the resource due to packet processing control flow in the pipeline. Linker merges packet processing CFGs of all the P4 programs and the OSGs generated from them for each resource according to composition operators. Then, Linker refactors the MATs, regenerates the CFG and maps the refactored MATs to physical pipeline stages while respecting the merged read-write OSG for each resource, the MAT control flow of all the P4 programs and available physical match memory type and capacity in the stages. We introduce two

concepts to facilitate this restructuring : 1) Vertically decomposing the MATs into sub MATs and 2) Performing out-of-order write operations in OSG of any resource. These techniques allow mapping of a sub MAT on available physical match memory type (e.g., exact, ternary or Longest Prefix Match) in the physical pipeline stage, even if the complete MAT can not be scheduled to the stage due to control dependency. Apart from creating new MATs and CFGs, Linker produces the mappings between new MATs mapped to physical pipeline stages and the MATs of all the P4 programs. It also prepares the MATs mappings and update schedules of the decomposed MATs that will be used by the Runtime component, as shown in Figure 5.1.

Runtime executes in the control plane of the target device and acts as a proxy to the control plane of P4 programs in order to manage their MATs defined in configuration files. It uses UIDTs and MAT mappings generated during linking to translate MATs update from the control plane of programs to the tables mapped to physical pipeline stages by Linker. Runtime is responsible for maintaining referential integrity across the sub MATs of a decomposed MAT to provide consistent MAT update. For every decomposed MAT with its sub MATs mapped to different stages of physical pipeline, Runtime updates the entries of the sub MATs according to the schedule generated by Linker. Moreover, it regulates the flow updates from the control plane of all the P4 programs to enforce flow space isolation dictated by the network operator.

5.3 Linker

In this section, we describe the static linking process of compiled configuration files of multiple independently written P4 programs.

5.3.1 Merging Parsers and Deparsers

The parser block in P4 is modeled as a FSM, which encodes a directed acyclic parse graph. Each vertex represents a state and the edges describe the state transitions. A P4 parser FSM has one *start* state and two logical final states, *accept* and *reject*. We call programmers defined states having transition to *accept* and *reject* states as *accept-transition* and *reject-transition* states, as shown in Figure 5.2. Using *extract* construct of P4, each state can extract zero or more header instances by advancing the current index in the bit stream of the incoming packet according to the header type definition. The other fundamental *select* construct of P4 allows to specify lookup fields and value to program state transitions or identify next header types in the packet. Apart from *extract* and *select*, P4 provides other constructs namely *verify*, *set* and *lookahead* respectively for error handling, variable assignments and reading the bits in the stream beyond the current index without incrementing it. If the boolean condition in argument of verify statement evaluates to true, execution of successive statements in the state continues without any interference. Otherwise, it results in immediate transition to reject state. Hence, we consider the states consisting verify statements as reject-transition states.

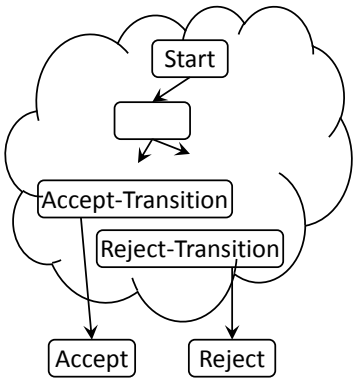


Figure 5.2: Parser FSM Structure

Essentially, merging of two parsers requires creating a union of their Directed Acyclic Graphs (DAGs) encoded as FSMs. However, as programs are independently implemented, they may not use the same identifiers and names. So, it is necessary to identify the equivalent header types, instances and parse states defined in the parser blocks. Also, explicit state transitions to accept and reject states and implicit transitions to these states resulting from error handling function call verify are required to merge for semantically correct merging of two parser FSMs. In the following we define the notion of equivalence of header types, parse states and header instances between two programs and explain how to find the equivalence relationship. Then, we describe our method to merge the states and create union of DAGs and thereby FSMs.

5.3.1.1 Equivalence of Header Types

A header type is defined as an ordered sequence of fields with bit-widths. Two fixed-length header types are equivalent if their ordered sequences of bit-widths are the same. Regarding variable length types, the length of the variable bit-width fields must depend on the same fixed width field in the header types and their maximum lengths must be identical. The length indicator field is uniquely identified by its bit-width and its offset from the start of the header. Using these definitions, we create a UID for each header type as an ordered sequence of bit-widths corresponding to its fields. In case of a variable length type, a UID is created considering its maximum possible length and the identifiers of the length indicator field. Header types UIDT maintains the mapping between UIDs and program specific identifiers for all the header types in all the programs.

5.3.1.2 Equivalence of Parse States and Header Instances

Parse states extract the bit streams into instances of header types defined in the program. So, the equivalence of parse states and header instances are correlated. The instances extracted from equivalent parse states are mapped to each other in

the instance UIDT. A parse state of a P4 program's parser is equivalent to a parse state of another program's parser if they satisfy the following conditions:

C1: both states extract bits from the same location in the bit stream of the packet. So, the current bit index points to the same location in the bit stream of the packet when the parsers visit the states.

C2: both states advance the current bit index by the same number of bits and extract the equivalent header types.

C3: if both states have *select* expression, then the lookup fields used in the expressions should be equivalent[1].

These conditions cover the scenario of *lookahead* construct also, where the states do not advance the current bit index but read the same set of bits to identify next transition states or store the value.

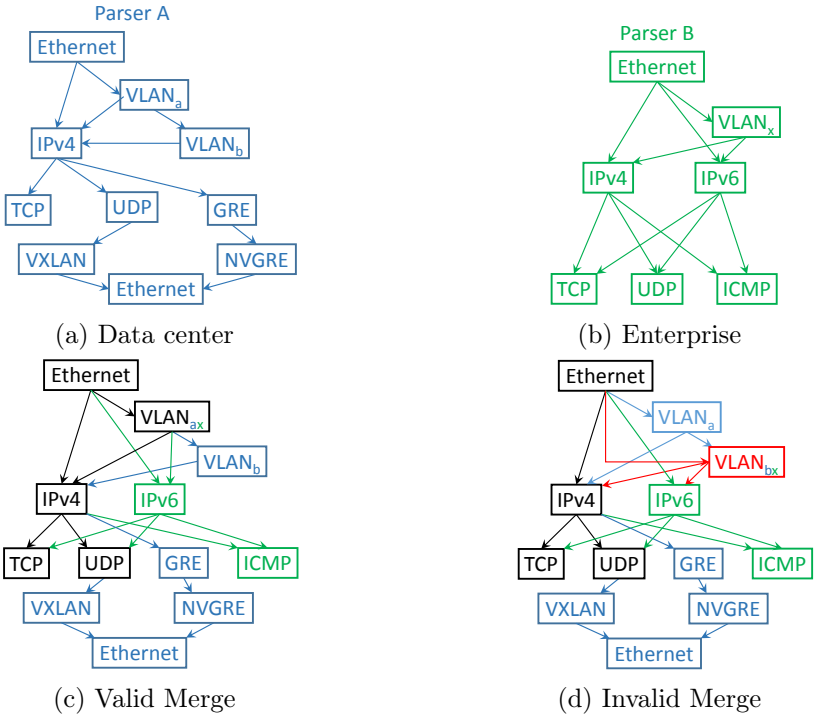
Algorithm 2: Identifying equivalent parse states and header instances

input : Parse graphs $PG_A = (\mathcal{S}_A, \mathcal{T}_A)$ & $PG_B = (\mathcal{S}_B, \mathcal{T}_B)$ of programs A & B
output: EquivalentStatesMap - Equivalent States mapping
$UIDT_{HeaderInstance}$ - Header Instances mapping

while $\mathcal{S}_A \neq \emptyset \vee \mathcal{S}_B \neq \emptyset$ **do** // Topological-order traversal of parse graphs
  $States_A \leftarrow$ `GetNodesWithoutIncomingEdge(`$\mathcal{S}_A$`)` // Nodes with 0-indegree
  $States_B \leftarrow$ `GetNodesWithoutIncomingEdge(`$\mathcal{S}_B$`)`
  foreach s_A *in* $States_A$ **do** // Mapping equivalent pairs of states
    foreach s_B *in* $States_B$ **do**
      if `ExtrHdrType(`s_A`)` $\equiv$ `ExtrHdrType(`s_B`)` **and** `LookupFields(`s_A`)` $\equiv$ `LookupFields(`s_B`)` **then** // Verifying conditions C2 and C3
        Add (s_A, s_B) in EquivalentStatesMap
        Map instances extracted from s_A, s_B in $UIDT_{HeaderInstance}$
        Remove s_A, s_B and their outgoing edges
        break
  for *remaining* s *in* $States_A \cup States_B$ **do** // Add mappings for unique states
    Add (s, s) in EquivalentStatesMap // Maps unique state to itself
    Map instance extracted from s to itself in $UIDT_{HeaderInstance}$
    Remove s and its outgoing edges

Let us take an example of merging two parse graphs of P4 programs to process Data center and Enterprise[2] network traffic, shown in Figures 5.3a and 5.3b, respectively. The parse graph of Data center has two VLAN states extracting double tagged headers in two VLAN instances a and b, whereas the parse graph of the Enterprise

[1]We assume all keyset expressions in *select* to be known values at compile time.
[2]The parse graphs are inspired from Figure 3 in [Gibb *et al.* 2013].



(a) Data center (b) Enterprise

(c) Valid Merge (d) Invalid Merge

Figure 5.3: Merging parse graphs of two parsers

network has a single VLAN state extracting header into the single instance named x. As shown in Figures 5.3c and 5.3d, when merging the two parse graphs, $VLAN_x$ should be equivalent to $VLAN_a$ but not to $VLAN_b$. Indeed, only $VLAN_x$ and $VLAN_a$ states extract the bits from the same current bit in the topological order to satisfy condition C1. Algorithm 2 describes the steps to identify equivalent parse states and header instances. First, we select the set of nodes (i.e., states) having 0 in-degree at each iteration for each parser graph in order to traverse the parse graphs in topological order (lines 2-3). Then, we find the equivalence states by verifying C2 and C3 conditions for all possible pairs of states from the two sets (line 6). After that, we remove the 0 in-degree nodes with their outgoing edges and continue iterating till all the nodes are visited. In the worst case scenario, each parse graph can have a start state connected to all the remaining nodes (i.e., star topology), *foreach* loops in lines 4-5 will execute the *if* condition in line 6 for every pair of states (s_A, s_B) from $\mathcal{S}_A \times \mathcal{S}_B$. Hence, the worst case time complexity of Algorithm 2 is $\mathcal{O}(|\mathcal{S}_A| \times |\mathcal{S}_B|)$.

5.3.1.3 Traffic Isolation and Error Handling

Parsers of two programs may transit to different final states (*accept* or *reject*) while parsing the same packet's bit stream. In case of error detected by the *verify* statement in a parse state of one parser, the parser transits to *reject* state. However, the equivalent state from another parser may not have *verify* statement with the same condition and can transit to different states instead of the final *reject* state. Moreover, we illustrate the need of traffic isolation emerging from different possible transitions

from equivalent states of two parsers with the example shown in Figure 5.3, where one parser transits to a final state and another to an intermediate state from the equivalent state. In case of double tagged VLAN packet, the Enterprise program transits to *reject* state after extracting first VLAN header ($VLAN_x$). However, the Data center program continues extracting next VLAN header. Hence, parsed packet headers of double tagged VLAN traffic should be exclusively processed by the MATs of the Data center program and MATs of the Enterprise program should process VLAN traffic only if the packets have a single tag. Another example, if the match keys of MATs defined in both the programs use only TCP header fields, the IPv6 based TCP traffic should not be processed by the MATs of the Data center program. In order to provide traffic isolation emerging from parsers resulting in different final states for the same packet and seamless error handling to all the parsers, we devise an error handling mechanism constituting an intrinsic metadata field and P4Bricks target specific implementation of the *verify* call.

We add an intrinsic metadata field, called *indicator*, with a bit width equal to the number of P4 programs to be merged. Each bit of this field indicates validity of the packet header instances for a program. In every state of the merged parse graph, we add an assignment statement masking the indicator field with a binary code. The binary code specifies per program validity of the packet if the parser transits to the state. Moreover, we appropriately set the bits of binary code, while merging parsers' *accept-transition* and *reject-transition* states to indicate the validity of packet header instances for the programs according to the final states in their parse graphs.

We prepare a binary code for each merged state while merging equivalence parse states of parsers. First, we assign the binary code according to the equivalence relationship among the parsers states. For instance, in the example of Figure 5.3c, $VLAN_b$ and $IPV6$ will mask the indicator field with $0b01$ and $0b10$ binary codes. Next, when an *accept-transition* state from one parser (UDP of Enterprise, Figure 5.3b) is required to merge with equivalent intermediate state (UDP of Data Center, Figure 5.3a) from another, we set the bit in the binary code corresponding to the former parser (Enterprise) in all successive states in the merged parser. Hence, $VXLAN$ will mask the indicator field with binary code $0b11$, even if it does not have any equivalent state in Enterprise parser. The merged parse graph transits to final *reject* state only when all the parsers transit to their logical *reject* state, otherwise merged parser transits to final *accept* state.

Linker replaces the *verify* statements in compiled P4 configuration file to P4Bricks data plane architecture specific implementation with the following signature.

```
extern void verify(in bool condition, in error err, in int
    program_id, inout bit<W> indicator);
```

For each P4 program, Linker allocates the parser error variable to hold error code generated by the program's parser, if the error base type is defined in configuration files of the program. Here, `program_id` is used to identify the parser error variable and set it to `err`, if `condition` evaluates to false. Also, the bit associated with the caller P4 program in `indicator` is set to 0 to indicate invalidity of the packet

header instances for the program. The semantics of the *verify* statement for P4Bricks enabled data plane architecture is given below.

```
P4BricksParserModel.verify(bool condition, error err, int
    program_id, bit<W> indicator) {
     if (condition == false) {
          P4BricksParserModel.parseError[program_id] = err;
          indicator = indicator & (~(1 << n));
     }
     if (indicator == 0)
          goto reject;
}
```

We emphasize here that P4 programmers use the *verify* statement with the signature provided by the P4 language. The Linker component in P4Bricks is responsible for translating the *verify* statements of each program by invoking the calls with appropriate values of the indicator field and `program_id` arguments.

5.3.1.4 Merging of Parse Graphs

We begin with merging the *select* transitions (i.e., edges in parse graphs) of two equivalent states by taking union of their keysets to create a single state. Regarding transition state of each *select* case, we find their equivalent state from the mappings and recursively merge them as described in Algorithm 3. If the two states have the same keyset, the corresponding transition states must be equivalent and we merge them too. We note that for a same keyset, two parsers can not transit to two different states. For instance, the value $0x0800$ of *EtherType* can not be used to transit to IPv4 state by one parser and to IPv6 state by the other. Allowing such ambiguous transitions creates non-deterministic FSM, resulting in a scenario where a packet can be parsed in different ways, which creates an ambiguity during packet processing. In the worst case scenario, there may not be any equivalent pair of parse states. Hence, Algorithm 3 will call `MergeStates` function $(|\mathcal{S}_A| + |\mathcal{S}_B|)$ times, accessing all the states the parse graphs once and merging their transitions based on keysets. Hence, the worst case time complexity of the algorithm is $\mathcal{O}(|\mathcal{S}_A| + |\mathcal{S}_B| + (|\mathcal{T}_A| \times |\mathcal{T}_B|))$.

In the case of sequential processing, merging the parsers of two programs is not sufficient to find the equivalent header instances between them. Let us consider an example of chaining of encapsulation-decapsulation network functions, where the first program pushes new header instances and the second parses the pushed header instances to process traffic in the network. Executing them on the same target with sequential composition requires identifying the equivalence relationship between header instances. For this purpose, we map the topological order of instances parsed from the merged parse graph to the sequence of emitted header instances in the deparser control block of the first program. Algorithm 4 describes detailed steps for it. In algorithm 4, we iteratively map 0 in-degree instances from the merged parser to the emitted instance from the deparser of the first program in sequence and removing instances from both. At each iteration, we search for an unmapped instance of merged parser having equivalent header type to the emitted unmapped

Algorithm 3: Merging of parse graphs

input : Parse graphs $PG_A = (\mathcal{S}_A, \mathcal{T}_A)$ & $PG_B = (\mathcal{S}_B, \mathcal{T}_B)$ of programs A & B
EquivalentStatesMap - States mapping

1 **Function** `MergeParseGraph`(PG_A, PG_B)
2 $InS_A \leftarrow$ InitState($\mathcal{S}_A$), $InS_B \leftarrow$ InitState($\mathcal{S}_B$)
3 `MergeStates` (InS_A, InS_B, $0b0$) // Merge init states of both parse graphs
4 **Function** `MergeStates`(S_A, S_B, $IndFlag$) // Recursively merge parse states
5 **if** S_A *and* S_B *are already merged* **then**
6 **return** merged state S_{Mab}
7 Set bits for program A and B in $IndMask_{Mab}$
8 $IndMask_{Mab} \leftarrow IndMask_{Mab} \parallel IndFlag$
// Merge state transitions using select case-lists of S_A and S_B
9 S_B.KeysetToNextStateMap $\leftarrow$ `CreateMap` (S_B.SelectCaseList)
10 **foreach** ($Keyset_A, NS_A$) *pair in* S_A.SelectCaseList **do**
11 $NS_B \leftarrow S_B$.KeysetToNextStateMap.Find($Keyset_A$)
12 **if** NS_B **then** // S_A & S_B have transitions for the same keyset
// Next States must be Equivalent
13 **Assert** (EquivalentStatesMap (NS_B, NS_A) == 1)
14 Remove $Keyset_A$ from S_B.KeysetToNextStateMap
15 **else** // Unique keyset across the case-lists
16 $NS_B \leftarrow$ EquivalentStatesMap.Find(NS_A))
17 **if** S_A *is accept-transition state* **then**
18 set bit for program A in $IndFlag$ // packet valid for program A
19 $NS_{Mab} \leftarrow$ `MergeStates` (NS_A, NS_B, $IndFlag$) // Recursive call
20 S_{Mab}.AddSelectCase($Keyset_A$, NS_{Mab}) // Add Merged State transitions
21 **for** *remaining* ($Keyset_B, NS_B$) *pairs in* S_B.KeysetToNextStateMap **do**
22 Repeat lines 16-20 using $Keyset_B, NS_B, S_B$
23 **return** S_{Mab}

instance. If there is any such instance pair, we create an equivalence mapping between them. Similar to Algorithm 2, the worst case time complexity of Algorithm 4 is $\mathcal{O}(|EmitHIList| \times |\mathcal{S}_M|)$.

Algorithm 4: Identifying equivalent header instances in sequential processing

input : $PG_{Merged} = (\mathcal{S}_M, \mathcal{T}_M)$ Merged Parse graph
EmitHIList - Emit order sequence of header instances

```
1  while EmitHIList not empty ∨ S_M ≠ ∅ do
2      States_B ← GetNodesWithoutIncomingEdge(S_M)
3      foreach hi_E in EmitHIList do        // Mapping instances using header type
4          foreach s_M in States_B do
5              hi_M ← GetHeaderInstanceExtractedFromState (s_M)
6              if HdrTypeof(hi_E) ≡ HdrTypeof(hi_M) and hi_M is unmapped
                then
7                  Map (hi_E, hi_M) in UIDT_HeaderInstance
8                  Remove s_M and their outgoing edges
9                  break
10             Remove hi_E from the list
```

Apart from *select*, *extract* and *verify*, parse states may have assignment, variable and constant declarations statements. Variables and constants declared in any program are user-defined metadata and are not shared with other programs. We concatenate the lists of assignment, variable and constant declaration statements used in equivalent states along with translated *verify* statements, even though they may perform redundant operations (e.g., verifying checksum or fields' value). Our current design for Linker fails to merge parsers of P4 programs and stops linking, if any of the equivalent parse states has assignment and method call statements modifying the shared data plane resources.

5.3.1.5 Deparser

P4 programmers can use *emit* function calls in a specialized control block defined for packet deparsing. If the header instance specified in a function argument is valid, it is appended to the packet else the function call does not perform any operation. To merge deparsers, we use merged parse graph to identify topological order of header instances. We note that the use of merged parse graph is not a semantically correct approach for finding the topological order among header instances in the deparsers of programs. Indeed, a parser may use a completely different network protocol stack to extract and process header instances than the one used in the deparser to reassembly the packet. More precisely, P4 programmer may (1) define parser to extract instances of some of the defined header types thereby decapsulating packet data from the header instances, (2) process the packet and finally (3) emit instances of disjoint header types to encapsulate the packet data before forwarding it to a interface. In such scenario, use of merged parse graph does not provide topological order among header instances to be emitted before sending out packets.

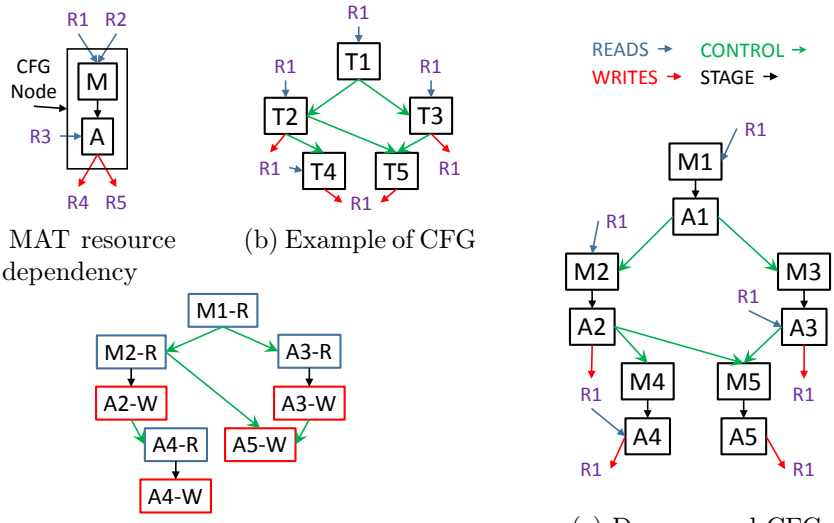
Hence, we restrict merging to P4 programs, which use the same network protocol stack for parsing and deparsing of packets and do not perform encapsulations. This allows the merging of P4 programs to be compatible with semantics of deparser control block described in current P4-16 language specification version 1.0.0 and still find topological order of header instances emitted by deparsers. In this work, we generate emit sequence of header instances by performing topological sort on the merged parse graph. We note that, we can perform sequential merging of parsers of an encapsulating P4 program followed by a decapsulating P4 program, because DAG in parser of decapsulating program is indirectly providing topological order of encapsulated header instance. However, in case of merging deparsers of multiple decapsulating programs, the topological order of header instances being emitted can not be identified with the current P4 specifications, thereby restricting the use of our proposal.

5.3.2 Logical Pipelines Restructuring

First, we replace all the occurrences of program specific identifiers of header types and instances with their UIDs using the mappings in the UIDTs generated while merging the parser and deparser blocks. Then, we decompose CFG stored in each P4 program's configuration file generated by P4 compiler. We create an OSG for each data plane resource from the decomposed CFG of each P4 program to capture all possible sequences of operations performed on the resource by the program. We describe decomposition of CFG and construction of OSG in Section 5.3.2.1. For each data plane resource, we apply composition operators on its OSGs generated from multiple P4 programs and create a single OSG for the resource, as explained in Section 5.3.2.2. This OSG captures all possible sequences of operations performed on the resource under given composition of P4 programs. Next, we merge decomposed CFGs of P4 programs according to composition operator by adding required control dependencies across the nodes of CFGs (Section 5.3.2.3). Also, we add dependencies from *indicator* metadata field to the control node without incoming edge in decomposed CFG of each P4 program. Such a control node specifies entry point of packet processing control flow of a given P4 program. Using the decomposed-and-merged CFGs and each data plane resource's OSG, we refactor original MATs of P4 programs and map them on physical pipeline stages. In Section 5.3.2.4, we describe the mechanism to map and refactor MATs on physical pipeline stages. To facilitate refactoring of MATs and mapping them on physical pipeline stages while respecting dependencies in merged OSGs and decomposed CFGs, we introduce 1) Vertically decomposing the MATs into sub MATs and 2) Performing out-of-order write operations in OSG of any resource in Sections 5.3.2.5 and 5.3.2.6, respectively.

5.3.2.1 Decomposing CFGs and Constructing Operation Schedules

First we decompose CFGs and capture schedules of the operations performed on each data plane resources. Each CFG node represents packet processing by a MAT involving match and action phases. We split each CFG node and reinterpret the



(a) MAT resource dependency

(b) Example of CFG

(c) Decomposed CFG

(d) OSG for resource R1

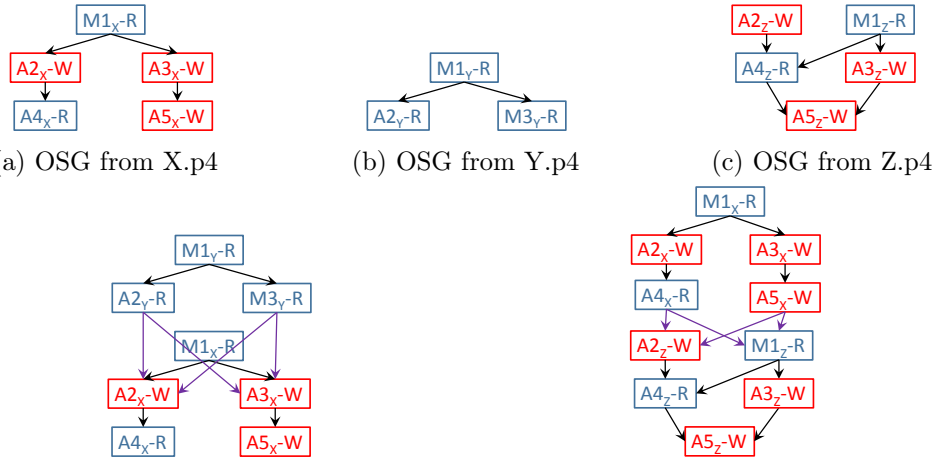
Figure 5.4: An example of CFG decomposition and OSG construction for a resource

packet processing phases as match and action *control nodes* operating on shared data plane resources, as shown in Figure 5.4a.

Match control nodes can only perform read operations on resources representing key fields. Action control nodes can perform read and write operations on resources, because all the actions declared within a table may take resources as input and modify them.

We add stage dependency from match to action control node of the same table. We decompose the CFG shown in 5.4b using this MAT representation. Figure 5.4c shows an example where several control nodes in the pipeline access a single resource R1. The decomposed CFG captures all control dependencies of the P4 program among control nodes and their operations on the data plane resources. CFGs generated by the P4 compiler are DAGs, but decomposed CFG may not be DAG. Because, we added every data plane resource as a graph node and a resource can be accessed by different control node for read and write operations, thereby creating cycles. Figure 5.4c does not explicitly show any cycle, because resource node R1 is repeated for legible pictorial view of decomposed CFG diagram. Otherwise, R1-M1-A1-M2-A2-R1 is one of the many possible cycles.

We create an *operation schedule graph* (called OSG) for each resource from the decomposed CFG to capture all possible sequences of operations that could be performed on the resource at runtime. As the CFG generated by P4 compiler for a P4 program is a DAG, so is the OSG of each resource derived from the CFG. Because, we merely split each CFG node into match and action control nodes, add stage dependency between them but do not add any packet processing control edge in CFG for decomposition. Within an OSG, nodes can be of two types (*node-R, node-W*), depending on the type of operation (i.e., read or write) performed on the resource by either a match or an action control node. Figure 5.4d shows the OSG for resource R1 created from the graph of Figure 5.4c. Even if the decomposed CFG of a program



(a) OSG from X.p4 (b) OSG from Y.p4 (c) OSG from Z.p4

(d) Parallel (X ‖ Y) (e) Sequential (X ≫ Z)

Figure 5.5: Two Different Merging of Resource's OSGs

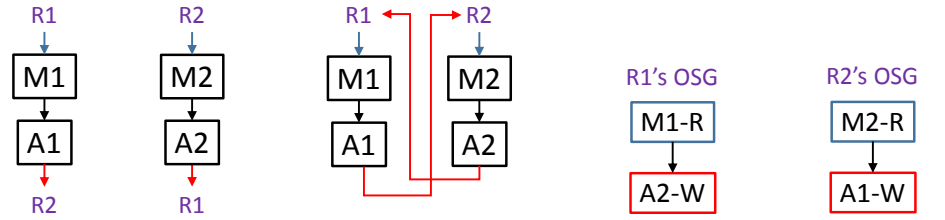
has cycles, all the access to resource nodes in a cycle follows a directed path in OSGs of the resources. For each data plane resource, we create one OSG per P4 program (derived from its CFG) and we merge all the OSGs by applying the composition operators specified by the network operator.

5.3.2.2 Applying Composition Operators on Resources

Multiple P4 programs can process traffic in parallel provided that they are processing disjoint data plane resources (or traffic flows). When processing disjoint data plane resources, there can only be one program with write operations on a given data plane resource and all the read operations on this resource from other programs must complete before any possible write operation.

Figure 5.5a and 5.5b show an example of OSGs for a resource derived from CFGs of two programs X and Y. All the read operations of the graph from Y must be scheduled before any write operations of the graph from X. To apply parallel composition for shared traffic flows, we add dependencies from all possible last reads of schedule graph from Y to all possible first writes of schedule graph from X, as shown in Figure 5.5d. In case of programs operating on disjoint traffic flows, we do not add any node dependency across the OSGs and consider merged graph as disconnected acyclic graph. In this case, Runtime enforces isolation of flows of programs in their MATs.

In case of sequential composition, once the first program completes all its operations on a data plane resource then and only then the next program is allowed to access the resource. We take an example of merging resource's OSGs from two programs X and Z, shown in Figures 5.5a and 5.5c, according to X after Z sequential composition. To apply the sequential composition's constraints, we add dependencies from all possible last operations of X to all possible first operations of Z as shown in Figure 5.5e. Similarly, we can merge resource's OSGs using any number of P4 programs by adding



(a) P1 (b) P2 (c) P1 ‖ P2 (d) OSG R1 (e) OSG R2

Figure 5.6: Parallel Composition of decomposed CFGs and OSGs

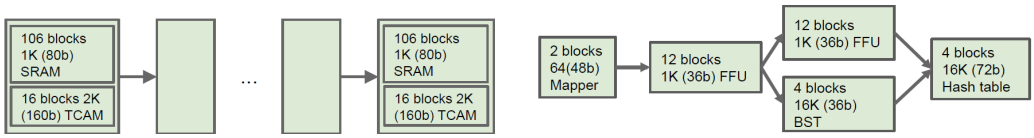
appropriate node dependencies among them, according to the specified operators and order. We use the merged OSGs of all the data plane resources, while restructuring the logical pipelines and mapping the MATs to physical pipeline stages.

5.3.2.3 Merging Decomposed CFGs

We merge decomposed CFGs by creating union of their nodes (i.e., data plane resource and control nodes) and edges. Every decomposed CFG has unique control nodes and control edges compared to other programs' decomposed CFGs. Because P4 programs may access shared data plane resources and perform operation on them, graph union connects decomposed CFGs of the programs through common resource nodes. If programs are accessing disjoint set of data plane resources, union of decomposed CFGs creates disconnected graph. The disconnected graph implies that merged program can have multiple control flows and MATs of each CFG can be scheduled and mapped on physical pipeline stages without any dependency from other CFGs' MATs. We add dependencies between *indicator* metadata field, created while merging parsers, and the match control node representing entry point of packet processing control flow in decomposed CFG of each P4 program. It allows to enforce required packet processing isolation emerging from packet parsing using the merged parser.

In case of parallel composition, we do not need to add any dependencies across the control nodes, as all the programs should process the assigned traffic flows. In case of sequential composition, some actions from the first program must be executed before the next program can start processing, even if there is no dependencies between operation schedules of any shared data plane resource from the composition constraint. For example, drop actions from a firewall program must be scheduled before any possible action for the next program. Therefore we add control node dependencies from such specific action control nodes of the first program to all the control nodes without incoming control edge of next program.

The merging of decomposed CFGs captures all possible control flow paths of each P4 program's CFG under specified composition. Similar to decomposed CFG, decomposed-and-merged CFGs can also have cycles. Every access sequence on every resource node resulting from a cycle should follow a corresponding directed path in the resource's merged OSG. However, under parallel composition, there may not exist a directed path for access sequences of all the resources in their respective merged



(a) RMT (b) Intel Flex Pipe

Figure 5.7: DAG representation of physical pipelines in reconfigurable devices [Jose *et al.* 2015]

OSGs. Such a scenario emerges when program $P1$ writes on a set of resources read by program $P2$ and $P1$ reads a disjoint set of resources written by $P2$. Figures 5.6a and 5.6b shows decomposed CFGs of single table programs P1 and P2 both accessing resource R1 and R2. The cycle, R1-M1-A1-R2-M2-A2-R1, results in a sequence of operations (M1-R, A2-W) on resource R1 and (A1-W, M2-R) on resource R2 in the decomposed-and-merged CFG(Figure 5.6c). In the merged OSG of resource R1, a corresponding path can be found for the sequence of operations on it. However, for resource R2 no corresponding path can be found in its merged OSG (Figure 5.6e). If we consider the other direction R2-M2-A2-R1-M1-A1-R2 in the cycle, no corresponding path can be found in merged OSG of resource R1 instead of R2. So, in any case either of the resources is operated out of its scheduled order represented in OSG of the resource. To perform required operation sequence resulting from a cycle on a resource, we perform out-of-order write operation with respect to the resource's merged OSG not having directed path according to the operation sequence. We describe the procedure to schedule out-of-order write operation in OSGs in Section 5.3.2.6. Next, we discuss usage of merged OSGs and decomposed CFGs of the P4 programs to refactor MATs defined in the programs and map them on physical pipeline stages.

5.3.2.4 Refactoring MATs and mapping to physical pipeline stages

Physical pipeline of a hardware target can be modeled as a DAG of stages, as shown in Figure 5.7, with each stage having different memory types with finite capacities and match capabilities, as described in [Jose *et al.* 2015]. The DAG provides the execution order of stages and possible parallelism among them. Moreover, decomposed-and-merged CFGs and merged resources' OSGs provide the scheduling order constraints on 1) match and action control nodes and 2) operations performed by these nodes on the resources. We refactor MATs and orderly map them to the physical pipeline stages by recomposing match and actions control nodes of the decomposed-and-merged CFG while respecting the scheduling order and the match stage constraints, as outlined in Algorithm 5.

In OSG of a resource, a node-W (specifying write operation on the resource) without any incoming edge is considered as *schedule-ready*. Schedule-ready nodes satisfy all scheduling constraints and are considered as available to map to pipeline stages. Also, a node-R (read operation) without any incoming edge along with its exclusive descendants with read operation type are considered schedule-ready; e.g., $A2_Z$-W,

$M1_Z$-R and $M1_Y$-R, $M1_X$-R, $A2_Y$-R, $M3_Y$-R in Figures 5.5c and 5.5d, respectively, are schedule-ready nodes. In a decomposed CFG, a control node without any incoming control edge is considered as *control-ready*. If a control node has all of its read operations on resources in ready state, it is considered as *complete read-ready*. A control node without any incoming edge is considered as *schedule-ready* having no control dependency as well as all of its operations on resources are schedule-ready. Essentially, a complete read-ready and control-ready node is schedule-ready. We select the physical pipeline stages according to their execution order (topological order) to map the refactored MATs while respecting the constraints of stages (line 2). To efficiently utilize the match stage's memory, we relax memory capacity constraints originating from the width of MATs by vertically decomposing them. This allows allocating the match stage memory at the granularity of individual key field instead of all the key fields required to match the control node.

Memory allocation may map either a subset or all of the key fields of a match control node, thereby creating sub matches of the logical MAT. We allocate memory of the current physical match stages by selecting schedule-ready node-R pertaining to match control nodes from OSGs for all resources while respecting memory match type and capacity constraints of physical stages (line 5). This approach may map the logical match of a MAT defined in a program to multiple nonparallel match stages. We focus on realizing match stage memory allocation at granularity of match operation on individual data plane resources (i.e., individual key field) rather than presenting any specific algorithm to select match operations to map on memory stages. Thereby, we omit the description of `SelectionOperations` function used in line 5 in Algorithm 5. Note that we skip the time complexity analysis of Algorithm 5, as it depends on the implementation the function `SelectionOperations`, which is not described in this thesis. Once read-ready operations of resources are mapped to the match stage, we remove the nodes and edges from the schedule graphs by making all their predecessors point to their respective successors (lines 9-11). For all the sub matches and match control node not having its action node in complete read-ready state, we create sub MATs as described in section 5.3.2.5 and allocate the actions to the action stage of the physical pipeline (lines 15-18). If any control-ready and complete read-ready match node is mapped to the stage, we schedule its dependent action node provided that it is at least complete read-ready. Also, if any write operation from the action node is not schedule-ready (line 20), we perform out-of-order write to map the MAT by recomposing match and action control nodes as detailed in 5.3.2.6.

5.3.2.5 Decomposing logical MATs into sub MATs

For every match control node, we recompose a MAT using schedule-ready read operations allocated by a selection algorithm to map on multiple nonparallel stages. If all the read operations from a match control node are not mapped to the same stage by used selection algorithm, recomposition creates a sub MAT and vertically decomposes the logical MAT corresponding to the match control node. We create a new data plane resource of user-defined metadata type for every decomposition of a logical MAT. We term this resource as the *foreign key field* associated with the recomposed sub MAT and its logical MAT. The bit width of the foreign key field

Algorithm 5: Mapping MATs to Physical Pipeline Stages

input : $PG_{physical} = (\mathcal{S}, \mathcal{E})$ - DAG of physical pipeline stages $\mathcal{S}$ & edges $\mathcal{E}$
$OSG_m(r)$ - merged OSG of all the resource r $\in \mathcal{R}$
$dCFG_m$ - merged all the program's decomposed CFGs
output: MATMappings - Maps every MAT to Ordered list of its Sub MATs

```
while S is not empty do                    // Topological-order traversal of PG_physical
    S_match ← GetNodesWithoutIncomingEdge(S)
    foreach r in R do
        ReadyMatchOps[r] ← GetSchedReadyReadMatchOpNodes (OSG_m(r))
    AllocatedOps[r] ← SelectionOperations (ReadyMatchOps[r], S_match)
    foreach r in R do
        foreach Op in AllocatedOps[r] do
            Remove edge representing Op in dCFG_m
            foreach PrevOp in Predecessor (Op) do
                foreach SuccOp in Successor (Op) do
                    Add PrevOp to SuccOp directed edge in the OSG_m(r)
    MappedMatchCN ← GetMatchControlNodesFromOps (AllocatedOps)
    foreach MatchCN in MappedMatchCN do
        ActionCN ← GetNextActionControlNode (MatchCN)
        if MatchCN or ActionCN is not complete read-ready or MatchCN is
         not control-ready then
            Create a sub MAT using schedule-ready Ops by MatchCN
            Add Foreign key field resource node in S
            Push sub MAT in mapped ordered list for the logical MAT in
             MATMappings
        else   // MatchCN is schedule-ready and ActionCN is complete read-ready
            if ActionCN has not schedule-ready write op on a resource r then
                Perform out-of-order write in OSG_m(r)
    Remove S_match from PG_physical with their incident edges
return MATMappings
```

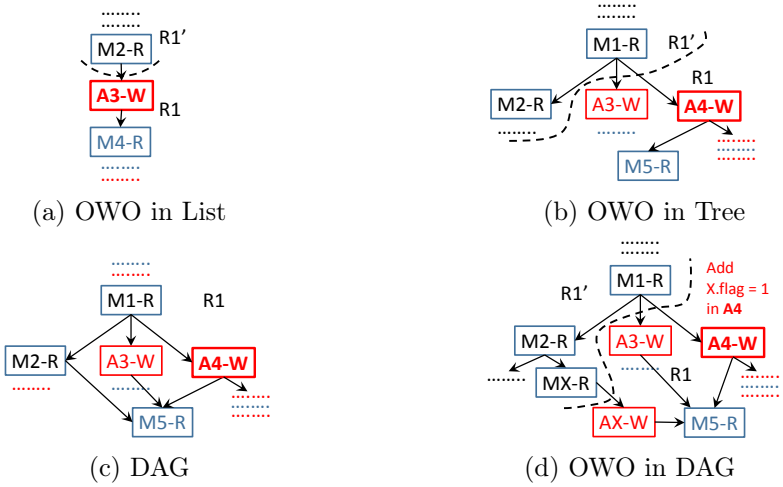
depends on the length of the logical MAT defined in its program. We vertically decompose logical MAT and add a new data plane resource field in order to efficiently use the memory of physical pipeline stages. We expect that the size of key fields of the sub MAT mapped on the physical stage is larger than or equal to the width of the foreign key field, thereby efficiently utilizing memory in the current match stage.

In this sub MAT, we add a new action setting the foreign key field to the given row ID on successful match at runtime. We consider sub MAT as *secondary table* having an action setting the foreign key field. Next, we remove dependency edges to the match control node from the selected key field resources. We add the resource dependency of the foreign key field to the match control node and add corresponding read-ready operations in the OSG of the field. In successive iterations of mapping physical stages, a sub MAT recomposed with the foreign key field resource in its match key fields is considered as the *primary table* being referenced by foreign key of the secondary table. A sub MAT can be primary table associated to multiple secondary tables referenced by their respected foreign keys. This creates multiple sub MATs using subsets of the match key set, hence vertically decomposing the MAT into multiple sub MATs. We store mappings between the decomposed MAT and its sub MATs (line 18). To realize flow updates in the decomposed MAT, all of its sub MATs are required to be updated with consistency at runtime. As the logical MAT is decomposed in multiple sub MATs with foreign key references, we create update schedule for sub MATs of every vertically decomposed logical MAT. The schedule provides a sequence to update sub MATs such that primary sub MAT is followed by its secondary, thereby maintaining referential integrity and consistency across the sub MATs for match and actions on foreign key fields. We arrange sub MATs of a vertically decomposed MAT in reverse topological order of their mapped physical pipeline stages. We push new sub MAT in front of the update sequence of sub MATs for the vertically decomposed MAT (line 18). The Runtime system uses this schedule to update the sub MATs mapped to the stages of the pipeline.

5.3.2.6 Out-of-order Write Operation

To explain scheduling of out-of-order write operations in an OSG, we start with the simplest case of an operation schedule (OS) list, followed by an OS tree and we finally address the directed acyclic graph case for OSG, as shown in Figure 5.8. Let us consider the first case shown in Figure 5.8a, where a resource's operation schedule is a list describing the sequence of operations on it. $A3$-W is scheduled out-of-order by performing all the ancestor operations of the node on the dummy resource R1', hence we split the schedule into two and move the one with all the ancestors to the dummy resource R1'. In case of a tree (shown in Figure 5.8b with $A4$-W as out-of-order write node), we additionally move all the read operation nodes (e.g., $M2$-R) before any write operation, exclusively reachable from the ancestors (e.g., $M1$-R) of the out-of-order write node. Finally, a tree can be transformed to a DAG by adding edges or paths from a node's siblings or ancestors to its descendants or itself, as shown in Figure 5.8c. In this case, the descendants of an out-of-order write node reachable using alternative (not involving the write node, e.g., $M5$-R) paths from its ancestors can be scheduled only once all of its other predecessor nodes (e.g., $M2$-R) are scheduled, including the predecessor of the out-of-order write node (i.e., $M1$-R).

Also, the value to be used for such nodes ($M5$-R) depends on two resources (R1 and R1') and during execution either of the resource will hold valid value depending on control flow of the execution at runtime. Hence, we add a flag as a data plane resource, a match (MX) and an action node (AX) nodes to copy the value of R1' to R1 by matching on the flag as shown in Figure 5.8d. This adds one MAT in the merged pipeline in order to recompose and map the MAT to the current pipeline stage using actions node not having all the write operations in ready state.



(a) OWO in List (b) OWO in Tree

(c) DAG (d) OWO in DAG

Figure 5.8: Out-of-order Write Operations (OWOs) and Types of Operation Schedules

5.4 Runtime

The Runtime system aims to manage the MATs refactored and mapped to the physical pipeline stages by Linker while restructuring the logical pipelines. It allows every P4 program's control plane to insert only the subset of flows assigned to the program by the network operator into the MATs of the pipelines. In order to apply parallel composition operator on disjoint traffic flows, Runtime only needs to enforce traffic flow assignment constraints on programs, because Linker does not merge MATs of the different programs while restructuring the pipelines. Runtime receives the MAT's flow update from every P4 program's control plane and uses the header instances UIDT to translate the flow updates from program's header IDs to UIDs given by the Linker. The other major functionality of Runtime is to provide consistency while updating the decomposed MAT, whose sub MATs are mapped to multiple nonparallel stages of the physical pipeline. Runtime provides consistency for updates logical MATs of a program, because MATs may be vertically split by Linker when logical pipelines are restructured. Runtime maintains referential integrity among the sub MATs of a decomposed MAT while performing Add, Delete and Modify flow updates. Runtime adds, updates and modifies flow entries in a decomposed MATs while respecting the update schedule of sub MATs of the decomposed MAT, generated by the Linker.

Algorithm 6: Add Flow at Runtime in a Decomposed MAT

```
input : Flow F = {M_kfv, A} with match M_kfv of key field-value & actions set A
        mat - UUID of the table to update flow F
        MATMappings - Maps every MAT to ordered list of its Sub MATs
1  Function AddFlow(F, mat, UpdateSchedList)
2      UpdateSchedList ← MATMappings.get(mat)
3      ActionSet ← F(A), MatchKFSet ← F(M_kfv)
4      foreach secondary SubMAT in ReverseOrder (UpdateSchedList) do
5          if not HasMatch (SubMAT, MatchKFSet) then
6              Create a new rowID for ForeignKeyField in Action of SubMAT
7          else
8              ForeignKeyField, rowID ← GetActionEntry (SubMAT, MatchKFSet)
9          ActionSet.Add(SetActionEntry (ForeignKeyField, rowID))
10         MatchKFSet.Add(MatchKeyValue (ForeignKeyField, rowID))
11     foreach SubMAT in UpdateSchedList do
12         if SubMAT has no flow entry for its subset of MatchKFSet then
13             Add Flow entry using subsets of MatchKFSet & ActionSet
14             Remove M_kfv & A of SubMAT from MatchKFSet & ActionSet
```

Linker may decompose a logical MAT by creating sub MATs using subsets of its match key fields. Flow entries of a decomposed MAT may have the same values for match key fields of some of its sub MATs. As sub MATs match flows using subsets of key fields of decomposed MATs, multiple flow entries of a decomposed MAT may share the same flow entry in sub MATs. At runtime, every secondary sub MAT assigns a row ID value, unique across the entries in the sub MAT, to its foreign key field in its action on successful match its key fields. Primary sub MATs match flows using the foreign key fields and other key fields of the decomposed MAT to further specialize the match on the flow entry and eventually execute the actions specified for the flow entry in the decomposed MAT.

Algorithm 6 realizes addition of new flow entry in a decomposed MAT. Before performing insertion, we match key fields of each sub MAT in reverse order of update schedule (packet processing order) to identify every sub MAT already having flow entry for its subset key fields of new flow of the decomposed MAT (line 4). If a sub MAT does not have matching flow with its subset key fields of new flow (line 5), we create new row IDs for the foreign key field of the sub MAT. We create a match and an action for the foreign key field (lines 9-10). Next, we insert flow entries in all the sub MATs in order of update schedule generated by the Linker. If a sub MAT does not have flow entry with its subset of key fields and foreign key fields, we add the corresponding new flow entry in the sub MAT (lines 12-14).

To delete a flow entry of a decomposed MAT (see Algorithm 7), we need to identify the flow entry stored across the sub MATs with intermediate foreign key fields. We match sub MATs in packet processing oder to find values of all the foreign key fields set by actions in sub MATs and store the sub MAT and foreign key field

Algorithm 7: Delete Flow at Runtime in a Decomposed MAT

input : Flow $F = \{M_{kfv}, A\}$ with match M_{kfv} of key field-value & actions set A
mat - UUID of the table to update flow F
MATMappings - Maps every MAT to ordered list of its Sub MATs

```
1  Function DeleteFlow(F, mat, UpdateSchedList)
2      UpdateSchedList ← MATMappings.get(mat)
3      MatchKFSet ← F(M_kfv),
4      foreach SubMAT in ReverseOrder (UpdateSchedList) do
5          ForeignKeyField, rowID ← GetActionEntry (SubMAT, MatchKFSet)
6          MatchKFSet.Add(MatchKeyValue (ForeignKeyField, rowID))
7          KeyMATMap.Add(ForeignKeyField, SubMAT)
8      foreach SubMAT in UpdateSchedList do
9          if SubMAT is in KeyMATMap then
10             Delete Flow entry f_sub having subset of MatchKFSet
11         foreach (ForeignKeyField, rowID) in f_sub do
12             if HasMatch (SubMAT, (ForeignKeyField, rowID)) then
13                 SecSubMAT ← KeyMATMap.GetSubMAT(ForeignKeyField)
14                 Remove SubMAT match keys from MatchKFSet
```

mapping.(lines 4-7). We match key field values of sub MATs in order of update schedule to delete flow entry of a decomposed MAT (line 8). We delete match entry from a secondary sub MAT (line 9) only if the foreign key fields values, set by the corresponding action, are not used for matching in primary sub MATs along with key fields of the decomposed MAT. If a primary sub MAT has multiple match entries having the same values for foreign key fields and different values for flow key fields, we do not further delete entries from secondary sub MATs of the decomposed MAT (lines 11- 14).

To update a flow entry of a decomposed MAT (Algorithm 8), we identify the flow entry stored across the sub MATs along with intermediate foreign key fields. We match key fields of flow pertaining to each sub MAT in reverse order of the update schedule (lines 4-8). The last primary sub MAT matching the remaining key fields of the flow and the foreign key fields uniquely identifies match entry of the decomposed MAT (line 9). Finally, we update action entry in the primary sub MAT (line 10).

A logical MAT can be decomposed in a number of sub MATs equals to the number of physical pipeline stages. Hence, the time complexity of Algorithms 6, 7 and 8 is $\mathcal{O}(|\mathcal{S}|)$, where $\mathcal{S}$ stands for the set of stages in the physical pipeline.

5.5 Conclusion

In this chapter, we outline our first design to compose independently written P4 programs. We describe methods used to merge parser blocks, use ID mappings

Algorithm 8: Modify Flow at Runtime in a Decomposed MAT

input : Flow $F = \{M_{kfv}, A\}$ with match M_{kfv} of key field-value & actions set A
mat - UUID of the table to update flow F
MATMappings - Maps every MAT to ordered list of its Sub MATs

```
1  Function ModifyFlow(F, mat, UpdateSchedList)
2      UpdateSchedList ← MATMappings.get(mat)
3      MatchKFSet ← F(M_kfv),
4      foreach SubMAT in ReverseOrder (UpdateSchedList) do
5          f_sub(M_kfv) ← GetKeyFieldValues (MatchKFSet, SubMAT (M_kf))
6          ForeignKeyField, rowID ← GetActionEntry (SubMAT, MatchKFSet)
7          Remove SubMAT match keys from MatchKFSet
8          MatchKFSet.Add(MatchKeyValue (ForeignKeyField, rowID))
9          if MatchKFSet is empty then
10             Update Flow entry with match f_sub(M_kfv) with actions set A
11             return
```

generated by them in merging of the pipelines. We show that it is possible to merge logical pipelines by considering packet headers and fields as shared resources and by applying composition operations to these shared resources. Deparser is represented as a control block in the P4 language, but we hope that our work provide a good motivation to rethink the design of the deparser to facilitate conceptually correct merging.

The current implementation status includes merging of parsers (5.3.1.1 to 5.3.1.4) and restructuring of pipelines (5.3.2.1 to 5.3.2.3 and 5.3.2.5) with out-of-order write operations.

6 Conclusions and Future Work

Contents

6.1 Summary and Conclusions

Advances in communication (e.g., 4G, 5G, fiber) and compute (smart devices) technologies with their pervasive usage have made the world communication-oriented, network-centric and more connected than ever. Leveraging the technological advances, a variety of use cases (e.g., live video streaming using mobile devices) and business models have emerged. Network operators are facing a great challenge in incorporating new communication technologies and to keep up with increasing and evolving demands of new network services addressing the emerging use cases. Necessity to innovate, develop, test, deploy and integrate new network devices, upgrade existing ones and configure them using vendor-specific tools at a rapid pace have made networks more complex and costly to operate.

SDN and NFV paradigms together promise to transform network architecture by *softwarizing* it, which can simplify network operations in a cost effective way and address the emerging use cases and the challenges. SDN decouples control and packet processing logic from forwarding devices and centralizes the control and intelligence of all the forwarding devices in the network, thereby allowing simplified control and management of network. On the other hand, NFV decouples software from hardware of network devices performing a fixed and specialized packet processing function in the network, thereby providing a cost effective softwarized approach for network service development, deployment, integration and management and upgradation. However, SDN and NFV face various challenges to realize the *softwarization of networks* vision. SDN applications processing frequent network events pose scalability challenges due to centralized control of the network. NFV faces performance issues due to data packet processing in without function-specific hardware. A lack of easy access to tools and testbed to perform SDN experiments with real world scenarios is also a major concern affecting evaluation of innovations in SDN research. With a paradigm

shift like distributed networking to centralize SDN, it is important to have a thriving ecosystem of tools and testbeds supporting new paradigm.

Live video streaming applications, which allows anyone to broadcast high quality live streams at anytime and from anywhere, are becoming widely-popular, as everyone having high access bandwidth and powerful smart devices at their disposal. IP multicast provides an efficient approach to enable network services to support such live video streaming applications. However, IP multicast is not widely adopted in the current distributed-network architecture. Even though SDN and NFV face critical challenges, they still hold great potential to enable next generation high bandwidth consuming applications like live video streaming in a cost effective way using IP multicast technology.

In Chapter 3, we propose to delegate multicast application specific network control to MNFs deployed in NFV environments. This allows to process massive group membership management traffic and offload the SDN controller to mitigate the scalability issue emerging from centralized SDN control. It provides a flexible to scale multicast capability of the network on demand by leveraging capability of NFV paradigm. OpenFlow does not provide composition features required to process the same traffic by multiple control applications and functions. Also, OpenFlow-enabled reconfigurable devices does not allow to enforce control isolation among different control applications and functions.

Also, in Chapter 3, we propose L2BM that leverages the global network view provided by centralized control to implement new traffic engineering policies with a great ease. In chapter 4, we described our automation tool DiG, which can emulate large scale SDN-enabled network providing required guarantee on performance of emulated resources like CPU cores, link capacity etc. We employed virtualization technologies used by NFV to emulate SDN-enabled networks. Also, we highlighted that performance degradation due to virtualization must be taken into account to provide performance guarantee for emulated resources.

From our experience with OpenFlow to control the same forwarding device with multiple SDN controllers, we learned that it is not sufficient to have modular SDN control plane, but it is also important to have modular SDN data plane to achieve *modularized network control and management*. We turned towards packet processing language P4, a more powerful and flexible packet processing abstraction to realize the SDN paradigm, to enable modularity in SDN data plane. P4 allows to describe parsing, deparsing and match-action pipeline to program packet processing in reconfigurable forwarding devices. However, all the packet processing functionalities of the device are required to be described in a single P4 program. This makes programs bloated, difficult to develop, test, debug and maintain. Also, it does not allow several network functions independently developed by different vendors to be deployed on the same device.

In Chapter 5, we proposed the design of the P4Bricks system to execute multiple independently developed P4 programs on the same reconfigurable hardware. Network operators can describe a composition of P4 programs using parallel and sequential operators. P4Bricks merges parsers, deparsers and match-action pipelines using semantics of composition operators and their order in composition. It considers header

fields and intrinsic metadata as a shared resources and applies composition operators to merge packet processing behavior of multiple independently written P4 programs. P4Bricks provides a modular SDN data plane architecture, hence it can enable modularized network control and management.

6.2 Future Works and Perspectives

In this thesis, we explored various avenues of the SDN and NFV paradigms in order to realize softwarized network control. We used NFV for flexible offloading of network control to specialized VNFs, which motivated us to enable modular network control and management. To achieve modular network control and management, we proposed a modular SDN data plane architecture using low level packet processing language P4. Also, we employed virtualization technologies to design a SDN experimentation tool with resource guarantee, which allows to conduct SDN research experiments using COTS compute devices available in a grid. We believe that for effective softwarization of networks, modularity in the SDN data plane is essential. Next, we describe some of our ongoing work and immediate research aspirations along with possible future directions of each avenue of SDN-NFV paradigm explored in this thesis.

6.2.1 Ongoing Work

We presented the design only for modular SDN data plane architecture in Chapter 5, our implementation of P4Bricks system is ongoing. We plan to evaluate P4Bricks and compare the hardware resource overhead with data plane virtualization approaches such as Hyper4 and HyperV, mentioned in 5.1. Also, we will implement MNFs using P4 to offload IGMP and group membership management control traffic processing.

6.2.2 Inter-domain IP Multicast Revival

Softwarization of networks with modular control and management will allow various new inter-domain multicast architecture to be deployed with ease. For example, deployment issues mentioned in ODMT [Basuki *et al.* 2012] can be addressed by deploying dedicated network functions for Controller Node (CoN), Forwarding Node (FoN) and Controller node Resolver (CR) described in the paper. Indeed, modularity in the SDN data plane can allow to push network control at the edge of ISP networks and allow functions running at the edges to program the core, that too on demand.

6.2.3 Performance Guaranteed Virtualization for Testbed and Cloud Platform

In Chapter 4, we demonstrated the need of finding operational margins of resources with a given virtualization technology. New tools and approaches should be developed to automate profiling of physical resources with various virtualization technologies and find resources' operational margins to guarantee performance of allocated resources. An automation tool to profile the hardware resources of infrastructure in different configurations of their allocation should be embedded in every emulation based testbed. Enabling OpenStack with such automated profiling tool can tremendously help to guarantee performance of resources with high confidence.

6.2.4 Modular SDN Data Plane

We believe modular SDN data plane will provide great flexibility for modular network control and management. This will unravel many more future research topics, which we enlist here.

6.2.4.1 Enhancements in Data Plane Programming Language

We believe our proposed P4Bricks system will motivate to reconsider the design for deparser control block. The deparser block, like parser block should have dedicate sub-language to capture the detailed programming logic and use it to compose the deparser blocks of P4 programs. P4Bricks should motivate to add more features and constructs (e.g., partial header types and definitions) in the P4 language for ease of programming and simplified composition of the programs. We consider P4 an apt candidate to extend and use to program OPP (Open Packet Processor) [Bianchi *et al.* 2016] devices. The OPP extension of P4 may require OpenState specific constructs to leverage all the packet processing abstractions provided by OpenState [Bianchi *et al.* 2014].

6.2.4.2 Dismantle and Reassemble SDN controller

It will be necessary to reconsider the design of network operating systems like ONOS [Berde *et al.* 2014]. However, the change would be surprisingly subtle and may not affect control APIs and management applications using them. Current implementations of SDN controllers only have a single channel to control and manage the datapath of forwarding devices. They distribute network events received from the channel to functionalities implemented as code-modules. They receive network events by either event based or service oriented architecture and expose APIs. A modular SDN data plane will allow all the code-modules to directly control the devices while still exposing the same APIs. For instance, a topology discovery module still exposes the same control APIs (east-west) to interact with other modules but it can have its own control data plane protocol and receive network events required

only to manage its datapath. This subtle change should improve the performance and increase flexibility in software development of SDN controllers.

6.2.4.3 Programmable Deep Packet Processing

P4, POF and OpenState allow stateful packet processing in the datapath, but capability of packet processing devices based reconfigurable hardware architectures (e.g., RMT [Bosshart *et al.* 2013b], d-RMT [Chole *et al.* 2017], OPP) to maintain complex state in memory is debatable. It would be interesting to explore the possibilities of processing complex stateful and layer 4 and above protocols in the TCP/IP stack using the available state memory of current reconfigurable devices and architectures.

6.2.4.4 An Operating System for Reconfigurable Devices

Our modular SDN data plane approach obviously leads us to develop an OS for reconfigurable devices. P4Bricks can deploy a packet processing graph described with P4 programs and composition operators. However, adding and removing P4 programs at runtime without device downtime is still a research problem to address. We are working on extending P4Bricks to address live updates and transform the system into an OS for reconfigurable devices in the SDN data plane.

Bibliography

[6WIND 2017] 6WIND. *6WIND VIRTUAL ACCELERATOR*. http://www.6wind.com/products/speedseries/6wind-virtual-accelerator/, 2017.

[Abhashkumar *et al.* 2017] Anubhavnidhi Abhashkumar, Jeongkeun Lee, Jean Tourrilhes, Sujata Banerjee, Wenfei Wu, Joon-Myung Kang and Aditya Akella. *P5: Policy-driven Optimization of P4 Pipeline*. In Proceedings of the Symposium on SDN Research, SOSR '17, pages 136–142, New York, NY, USA, 2017. ACM.

[Anwer *et al.* 2015] Bilal Anwer, Theophilus Benson, Nick Feamster and Dave Levin. *Programming Slick Network Functions*. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, pages 14:1–14:13, New York, NY, USA, 2015. ACM.

[Arashloo *et al.* 2016] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford and David Walker. *SNAP: Stateful Network-Wide Abstractions for Packet Processing*. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

[Balouek *et al.* 2013] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr and Luc Sarzyniec. *Adding Virtualization Capabilities to the Grid'5000 Testbed*. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann and Tony Shan, editors, Cloud Computing and Services Science, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[Basuki *et al.* 2012] Achmad Basuki, Achmad Husni Thamrin, Hitoshi Asaeda and Jun Murai. *ODMT: On-demand Inter-domain Multicast Tunneling*. Journal of Information Processing, vol. 20, no. 2, pages 347–357, 2012.

[Beck *et al.* 2014] M.T. Beck, C. Linnhoff-Popien, A. Fischer, F. Kokot and H. de Meer. *A simulation framework for Virtual Network Embedding algorithms*. In Telecommunications Network Strategy and Planning Symposium (Networks), 2014 16th International, pages 1–6, Sept 2014.

[Benson *et al.* 2010] Theophilus Benson, Aditya Akella and David A. Maltz. *Network Traffic Characteristics of Data Centers in the Wild*. In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[Berde *et al.* 2014] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin

Radoslavov, William Snow and Guru Parulkar. *ONOS: Towards an Open, Distributed SDN OS*. In Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[Bianchi *et al.* 2014] Giuseppe Bianchi, Marco Bonola, Antonio Capone and Carmelo Cascone. *OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch*. SIGCOMM Comput. Commun. Rev., vol. 44, no. 2, pages 44–51, April 2014.

[Bianchi *et al.* 2016] Giuseppe Bianchi, Marco Bonola, Salvatore Pontarelli, Davide Sanvito, Antonio Capone and Carmelo Cascone. *Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing*. CoRR, vol. abs/1605.01977, 2016.

[Bifulco *et al.* 2016] Roberto Bifulco, Julien Boite, Mathieu Bouet and Fabian Schneider. *Improving SDN with InSPired Switches*. In Proceedings of the Symposium on SDN Research, SOSR '16, pages 11:1–11:12, New York, NY, USA, 2016. ACM.

[Big Switch Networks 2013] Big Switch Networks. *Floodlight*, 2013.

[Bondan *et al.* 2013] Lucas Bondan, Lucas Fernando Müller and Maicon Kist. *Multiflow: Multicast clean-slate with anticipated route calculation on OpenFlow programmable networks*. Journal of Applied Computing Research, vol. 2, no. 2, pages 68–74, 2013.

[Bosshart *et al.* 2013a] Pat Bosshart*et al. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN*. In ACM SIGCOMM, New York, NY, USA, 2013.

[Bosshart *et al.* 2013b] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica and Mark Horowitz. *Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN*. SIGCOMM Comput. Commun. Rev., vol. 43, no. 4, pages 99–110, August 2013.

[Bosshart *et al.* 2013c] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica and Mark Horowitz. *Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN*. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[Bosshart *et al.* 2014] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese and David Walker. *P4: Programming Protocol-independent Packet Processors*. SIGCOMM Comput. Commun. Rev., vol. 44, no. 3, pages 87–95, July 2014.

[Bremler-Barr *et al.* 2016] Anat Bremler-Barr, Yotam Harchol and David Hay. *OpenBox: A Software-Defined Framework for Developing, Deploying, and*

Managing Network Functions. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 511–524, New York, NY, USA, 2016. ACM.

[Cain *et al.* 2015] Brad Cain, Steve Deering, Isidor Kouvelas, Bill Fenner and Ajit Thyagarajan. *Internet Group Management Protocol, Version 3.* RFC 3376, October 2015.

[Chakraborty *et al.* 2003] Debasish Chakraborty*et al. A dynamic multicast routing satisfying multiple QoS constraints.* Wiley IJNM, vol. 13, no. 5, 2003.

[Chiosi *et al.* 2012] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, Javier Benitez, Uwe Michel, Herbert Damker, Kenichi Ogaki, Tetsuro Matsuzaki, Masaki Fukui, Katsuhiro Shimano, Dominique Delisle, Quentin Loudier, Christos Kolias, Ivano Guardini, Elena Demaria, Roberto Minerva, Antonio Manzalini, Diego López, Francisco Javier, Ramón Salguero, Frank Ruhl and Prodip Sen. *Network Functions Virtualisation - Introductory White Paper.* ETSI, 2012.

[Chole *et al.* 2017] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda and Tom Edsall. *dRMT: Disaggregated Programmable Switching.* In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, pages 1–14, New York, NY, USA, 2017. ACM.

[Consortium 2008] NS-3 Consortium. *ns-3 Project page.* http://www.nsnam.org/, 2008. [Online; accessed 11-August-2015].

[Craig *et al.* 2015] Alexander Craig, Nandy*et al. Load balancing for multicast traffic in SDN using real-time link cost modification.* In IEEE ICC, June 2015.

[Crichigno & Baran 2004] J. Crichigno and B. Baran. *Multiobjective multicast routing algorithm for traffic engineering.* In IEEE ICCCN, pages 301–306, Oct 2004.

[Curtis *et al.* 2011a] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma and Sujata Banerjee. *DevoFlow: Scaling Flow Management for High-performance Networks.* SIGCOMM Comput. Commun. Rev., vol. 41, no. 4, pages 254–265, August 2011.

[Curtis *et al.* 2011b] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma and Sujata Banerjee. *DevoFlow: Scaling Flow Management for High-performance Networks.* In Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[Devera 2002] Martin Devera. *HTB - Hierarchy Token Bucket, .* https://linux.die.net/man/8/tc-htb, 2002.

[Diot *et al.* 2000] C. Diot, B.N. Levine*et al. Deployment issues for the IP multicast service and architecture.* IEEE Network, vol. 14, no. 1, pages 78–88, Jan 2000.

[Dixit *et al.* 2014] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman and Ramana Kompella. *ElastiCon: An Elastic Distributed Sdn Controller*. In Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '14, pages 17–28, New York, NY, USA, 2014. ACM.

[Enns *et al.* 2011] Rob Enns, Martin Bjorklund, Andy Bierman and Jürgen Schönwälder. *Network Configuration Protocol (NETCONF)*. RFC 6241, June 2011.

[Erickson 2013] David Erickson. *The Beacon Openflow Controller*. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[Evans *et al.* 2011] John Evans, Ali Begen, Jason Greengrass and Clarence Filsfils. *Toward Lossless Video Transport*. IEEE Internet Computing, vol. 15, pages 48–57, 2011.

[Fang *et al.* 2015] Luyuan Fang, Fabio Chiussi, Deepak Bansal, Vijay Gill, Tony Lin, Jeff Cox and Gary Ratterree. *Hierarchical SDN for the Hyper-scale, Hyper-elastic Data Center and Cloud*. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, pages 7:1–7:13, New York, NY, USA, 2015. ACM.

[FD.io 2017] FD.io. *Vector Packet Processing*. https://fd.io/technology/, 2017.

[Fenner *et al.* 2016] Bill Fenner, Mark Handley, Isidor Kouvelas and Hugh Holbrook. *PIM-SM: Protocol Specification (revised)*. RFC 7761, March 2016.

[Fischer *et al.* 2013] A. Fischer, J.F. Botero, M. Till Beck, H. de Meer and X. Hesselbach. *Virtual Network Embedding: A Survey*. Communications Surveys Tutorials, IEEE, vol. 15, no. 4, pages 1888–1906, Fourth 2013.

[Fu *et al.* 2015] Y. Fu, J. Bi, Z. Chen, K. Gao, B. Zhang, G. Chen and J. Wu. *A Hybrid Hierarchical Control Plane for Flow-Based Large-Scale Software-Defined Networks*. IEEE Transactions on Network and Service Management, vol. 12, no. 2, pages 117–131, June 2015.

[Gibb *et al.* 2013] G. Gibb, G. Varghese, M. Horowitz and N. McKeown. *Design principles for packet parsers*. In Architectures for Networking and Communications Systems, pages 13–24, Oct 2013.

[Hagiya *et al.* 1993] N Hagiya*et al. Concentrated traffic in non-hierarchical networks under alternate routing scheme: a behavior analysis*. In IEEE GLOBECOM, 1993.

[Halpern *et al.* 2010] Joel M. Halpern, Robert HAAS, Avri Doria, Ligang Dong, Weiming Wang, Hormuzd M. Khosravi, Jamal Hadi Salim and Ram Gopal. *Forwarding and Control Element Separation (ForCES) Protocol Specification*. RFC 5810, March 2010.

[Hancock & van der Merwe 2016] David Hancock and Jacobus van der Merwe. *HyPer4: Using P4 to Virtualize the Programmable Data Plane*. In Proceedings

of the 12th International on Conference on Emerging Networking EXperiments and Technologies, CoNEXT '16, pages 35–49, New York, NY, USA, 2016. ACM.

[Heller *et al.* 2012] Brandon Heller, Rob Sherwood and Nick McKeown. *The Controller Placement Problem.* In Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM.

[Jain *et al.* 2013] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart and Amin Vahdat. *B4: Experience with a Globally-deployed Software Defined Wan.* SIGCOMM Comput. Commun. Rev., vol. 43, no. 4, pages 3–14, August 2013.

[Jin *et al.* 2015] Xin Jin, Jennifer Gossels, Jennifer Rexford and David Walker. *CoVisor: A Compositional Hypervisor for Software-defined Networks.* In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15, pages 87–101, Berkeley, CA, USA, 2015. USENIX Association.

[Jose *et al.* 2015] Lavanya Jose, Lisa Yan, George Varghese and Nick McKeown. *Compiling Packet Programs to Reconfigurable Switches.* In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15, pages 103–115, Berkeley, CA, USA, 2015. USENIX Association.

[Kodialam *et al.* 2003] Murali Kodialam, TV Lakshman and Sudipta Sengupta. *Online multicast routing with bandwidth guarantees: a new approach using multicast network flow.* IEEE/ACM ToN, vol. 11, no. 4, pages 676–686, 2003.

[Koponen *et al.* 2010] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama and Scott Shenker. *Onix: A Distributed Control Platform for Large-scale Production Networks.* In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pages 351–364, Berkeley, CA, USA, 2010. USENIX Association.

[Kuzniar *et al.* 2014] Maciej Kuzniar, Peter Peresini and Dejan Kostić. *Providing Reliable FIB Update Acknowledgments in SDN.* In Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14, pages 415–422, New York, NY, USA, 2014. ACM.

[Lajos Kis *et al.* 2017] Zoltán Lajos Kis, Jean Tourrilhes, Khai Nguyen Dinh, Thanh Le Dinh, yu iwata, Yuval Adler and Hiroyasu OHYAMA. *CPqD Software Switch.* https://github.com/CPqD/ofsoftswitch13, 2017.

[Lantz & O'Connor 2015] Bob Lantz and Brian O'Connor. *A Mininet-based Virtual Testbed for Distributed SDN Development.* In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, pages 365–366, New York, NY, USA, 2015. ACM.

[Lantz *et al.* 2010] Bob Lantz, Brandon Heller and Nick McKeown. *A Network in a Laptop: Rapid Prototyping for Software-defined Networks*. In ACM HotSDN, 2010.

[linc dev 2015] linc dev. *LINC Switch*. https://github.com/FlowForwarding/LINC-Switch, 2015.

[Linux Foundation 2013] project Linux Foundation. *Intel Data Plane Development Kit*. http://dpdk.org/, 2013.

[Marcondes *et al.* 2012] Cesar AC Marcondes*et al.* *CastFlow: Clean-slate multicast approach using in-advance path processing in programmable networks*. In IEEE ISCC, 2012.

[McKeown *et al.* 2008] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker and Jonathan Turner. *OpenFlow: Enabling Innovation in Campus Networks*. SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pages 69–74, March 2008.

[Nguyen 2016] Xuan-Nam Nguyen. *The OpenFlow rules placement problem : a black box approach*. Theses, Université Nice Sophia Antipolis, April 2016.

[Nicira Networks 2008] Nicira Networks. *NOX Network Control Platform*, 2008.

[Pfaff & Davie 2013] Ben Pfaff and Bruce Davie. *The Open vSwitch Database Management Protocol*. RFC 7047, December 2013.

[Pfaff *et al.* 2015] Ben Pfaff, Justin Pettit, Teemu Koponen*et al.* *The Design and Implementation of Open vSwitch*. In 12th ACM NSDI, May 2015.

[Phemius *et al.* 2014] Kévin Phemius, Mathieu Bouet and Jérémie Leguay. *DISCO: Distributed multi-domain SDN controllers*. In IEEE NOMS, pages 1–4, May 2014.

[PICA8 2009] PICA8. *PICA8 White Box SDN,*. http://www.pica8.com/resources/technology, 2009.

[Reitblatt *et al.* 2011] Mark Reitblatt, Nate Foster, Jennifer Rexford and David Walker. *Consistent Updates for Software-defined Networks: Change You Can Believe in!* In Proceedings of the 10th ACM Workshop on Hot Topics in Networks, HotNets-X, pages 7:1–7:6, New York, NY, USA, 2011. ACM.

[Ruckert *et al.* 2016] Julius Ruckert*et al.* *Flexible, Efficient, and Scalable Software-Defined OTT Multicast for ISP Environments with DynSDM*. IEEE TNSM, 2016.

[Ryu SDN Framework Community 2014] Ryu SDN Framework Community. *Ryu*, 2014.

[Santos *et al.* 2014] M.A. Santos, B.A. Nunes, K. Obraczka, T. Turletti, B.T. de Oliveira and C.B. Margi. *Decentralizing SDN's Control Plane*. In IEEE LCN, 2014.

[Sarzyniec *et al.* 2013] L. Sarzyniec, T. Buchert, E. Jeanvoine and L. Nussbaum. *Design and Evaluation of a Virtual Experimental Environment for Distributed*

Systems. In 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pages 172–179, Feb 2013.

[Schiff *et al.* 2014] Liron Schiff, Michael Borokhovich and Stefan Schmid. *Reclaiming the Brain: Useful OpenFlow Functions in the Data Plane*. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII, pages 7:1–7:7, New York, NY, USA, 2014. ACM.

[Schönwälder 2008] Jürgen Schönwälder. *Simple Network Management Protocol (SNMP) Context EngineID Discovery*. RFC 5343, September 2008.

[Seok *et al.* 2002] Yongho Seok, Youngseok Lee, Yanghee Choi and Changhoon Kim. *Explicit multicast routing algorithms for constrained traffic engineering*. In IEEE ISCC, 2002.

[Song 2013] Haoyu Song. *Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane*. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[Soni *et al.* 2015] Hardik Soni, Damien Saucez and Thierry Turletti. *DiG: Datacenters in the Grid*. In IEEE NFV-SDN, pages 4–6, Nov 2015.

[Soni *et al.* 2017] H. Soni, W. Dabbous, T. Turletti and H. Asaeda. *NFV-Based Scalable Guaranteed-Bandwidth Multicast Service for Software Defined ISP Networks*. IEEE Transactions on Network and Service Management, vol. 14, no. 4, pages 1157–1170, Dec 2017.

[Sun *et al.* 2017] C. Sun, J. Bi, H. Chen, H. Hu, Z. Zheng, S. Zhu and C. Wu. *SDPA: Toward a Stateful Data Plane in Software-Defined Networking*. IEEE/ACM Transactions on Networking, vol. 25, no. 6, pages 3294–3308, Dec 2017.

[Takamiya *et al.* 2017] Yasuhito Takamiya, sugyo and Nick Karanatsios. *Trema*, 2017.

[Tang *et al.* 2014] Siyuan Tang, Bei Hua and Dongyang Wang. *Realizing video streaming multicast over SDN networks*. In EAI CHINACOM, pages 90–95, Aug 2014.

[Tazaki *et al.* 2013] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti and Walid Dabbous. *Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments*. In Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, pages 217–228, New York, NY, USA, 2013. ACM.

[The ETSI OSM Community 2016] The ETSI OSM Community. *Open Source MANO: An ETSI OSM Community White Paper*, October 2016.

[The ONOS Community 2014] The ONOS Community. *ONOS*, 2014.

[The OpenDaylight Foundation 2013] The OpenDaylight Foundation. *OpenDaylight*, 2013.

[The P4 Language Consortium 2017] The P4 Language Consortium. *$P4_{16}$ Language Specification version 1.0.0*, May 2017.

[Tootoonchian & Ganjali 2010] Amin Tootoonchian and Yashar Ganjali. *HyperFlow: A Distributed Control Plane for OpenFlow*. In Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[Veloso *et al.* 2002] Eveline Veloso, Virgílio Almeida, Wagner Meira, Azer Bestavros and Shudong Jin. *A Hierarchical Characterization of a Live Streaming Media Workload*. In ACM IMW, pages 117–130, Nov 2002.

[Vidal *et al.* 2014] Allan Vidal, Eder Leão Fernandes, Christian Esteve Rothenberg and Marcos Rogério Salvador. *libfluid. The ONF OpenFlow Driver*. http://opennetworkingfoundation.github.io/libfluid/, 2014.

[Waxman 1988] B. M. Waxman. *Routing of multipoint connections*. IEEE JSAC, vol. 6, no. 9, pages 1617–1622, Dec 1988.

[Wette *et al.* 2014] P. Wette, M. Dräxler and A. Schwabe. *MaxiNet: Distributed emulation of software-defined networks*. In 2014 IFIP Networking Conference, pages 1–9, June 2014.

[Yap *et al.* 2010] Kok-Kiong Yap, Te-Yuan Huang, Ben Dodson, Monica S Lam and Nick McKeown. *Towards Software-friendly Networks*. In ACM APSys, 2010.

[Yeganeh & Ganjali 2012] Soheil Hassas Yeganeh and Yashar Ganjali. *Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications*. In ACM HotSDN, 2012.

[Yeganeh & Ganjali 2014] Soheil Hassas Yeganeh and Yashar Ganjali. *Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking*. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII, pages 13:1–13:7, New York, NY, USA, 2014. ACM.

[Yeganeh & Ganjali 2016] Soheil Hassas Yeganeh and Yashar Ganjali. *Beehive: Simple Distributed Programming in Software-Defined Networks*. In ACM SOSR, 2016.

[Yeganeh *et al.* 2013] S. H. Yeganeh, A. Tootoonchian and Y. Ganjali. *On scalability of software-defined networking*. IEEE Communications Magazine, vol. 51, no. 2, pages 136–141, February 2013.

[Youm *et al.* 2013] Sungkwan Youm, Taeshik Shon and Eui-Jik Kim. *Integrated multicast routing algorithms considering traffic engineering for broadband IPTV services*. EURASIP JWCN, vol. 2013, no. 1, page 127, 2013.

[Zhang *et al.* 2015] S. Q. Zhang*et al. Routing Algorithms for NFV Enabled Multicast Topology on SDN*. IEEE TNSM, vol. 12, no. 4, Dec 2015.

[Zhang *et al.* 2017a] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar and J. Wu. *HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane*. In 2017 26th International Conference on Computer Communication and Networks (ICCCN), pages 1–9, July 2017.

[Zhang *et al.* 2017b] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar and Jianping Wu. *MPVisor: A Modular Programmable Data Plane Hypervisor*. In Proceedings of the Symposium on SDN Research, SOSR '17, pages 179–180, New York, NY, USA, 2017. ACM.

[Zhou & Bi 2017] Yu Zhou and Jun Bi. *ClickP4: Towards Modular Programming of P4*. In Proceedings of the SIGCOMM Posters and Demos, SIGCOMM Posters and Demos '17, pages 100–102, New York, NY, USA, 2017. ACM.