Towards High Quality and Flexible Future Internet Architectures

Flexibele toekomstige internetarchitecturen van hoge kwaliteit

Sachin Sharma

Promotoren: prof. dr. ir. D. Colle, prof. dr. ir. M. Pickavet
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. D. De Zutter
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2015 - 2016

UNIVERSITEIT
GENT

ISBN 978-90-8578-892-8
NUR 986, 988
Wettelijk depot: D/2016/10.500/24

Ghent University
Faculty of Engineering and Architecture
Department of Information Technology

Promotors: Prof. dr. ir. Didier Colle
Prof. dr. ir. Mario Pickavet

Jury Members: Prof. dr. ir. Luc Taerwe, Ghent University, Belgium (Chairman)
Prof. dr. ir. Didier Colle, Ghent University (Supervisor)
Prof. dr. ir. Mario Pickavet, Ghent University (Supervisor)
Prof. dr. ir. Balazs Sonkoly, Budapest University of Technology and Economics
Prof. dr. ir. Kris Steenhaut, Vrije Universiteit Brussel
Prof. dr. ir. Sofie Van Hoecke, Ghent University
Dr. Wouter Tavernier, Ghent University (Secretary)
Dr. Dimitri Staessens, Ghent University

Ghent University
Faculty of Engineering and Architecture

Department of Information technology
Technologiepark-Zwijnaarde 15, 9052 Ghent, Belgium

Tel.: +32-9-331.49.00
Fax.: +32-9-331.48.99

Dissertation to obtain the degree of
Doctor of Computer Science Engineering
Academic year 2015-2016

Acknowledgments

Finally, my journey to obtain a PhD is going to be over. Pursuing the PhD was not an easy task for me, but it was full of tough challenges that I could not have overcome alone. Fortunately, there were many people, who contributed to its successful completion.

First of all, I am deeply grateful to my promoters, Prof. Didier Colle and Prof. Mario Pickavet, for providing me an opportunity to carry out my PhD research at Ghent University-iMinds. I submit my respectable thanks to them for believing in my potential and performance. Without their excellent vision, direction, and support, it would not have been possible for me to have a successful research work. In particular, I am thankful for their constructive research ideas, feedback, suggestions and modesty.

I express my gratitude to Prof. Piet Demeester and all the members of the IBCN group for providing me the platform to successfully carry out my research work. I would also like to thank the secretariat – Martine and Davinia – and other staff members at IBCN who have made my stay comfortable in Ghent. All these efforts have created an excellent stimulating international environment for researchers like me in the IBCN group.

Prof. Luc Taerwe, Prof. Balazs Sonkoly, Prof. Kris Steenhaut, Prof. Sofie Van Hoecke, Dr. Wouter Tavernier, and Dr. Dimitri Staessens, I thank you all for being part of my jury and helping me to improve the quality of my PhD dissertation. I also thank to Dr. Dimitri Staessens and Dr. Wouter Tavernier for guiding me during my PhD.

I would also like to thank all the other co-authors of my articles – David, Joao, Luis, Adam, Donal, Ricardo, Sahel, Nick, Charaka, Rob, Juhoon, Catalin, Wolfgang, and Rebecca – whose critical discussions, comments, and support were indeed very helpful. Thank you all for your valuable feedback and efforts.

I also take this opportunity to thank all the members of projects – SPARC, OFELIA, CityFlow, UNIFY, and MECANO – in which I was actively involved during my PhD. Thank you for providing me research directions during the duration of projects.

I have aIso been surrounded by what I think to be the best colleagues one can wish for in an office: Sahel, Thijs, Sofie, Sander, Steven, Thomas, Maarten, Wouter, Dimitri, Abhishek, Ward, Marlies, Bram, Domonico, Rodrigo, Ludwig,

Bart, Willem, Farhan, Krishnan, Selva, Arun, Wei, Jetmir, and Ratul. I thank you all for your's friendly behavior, lunch discussions, and motivations.

This acknowledgment is incomplete without mentioning all the friends that I met during my stay in Ghent: Jignesh, Dinesh, Sarvagya, Vinod, Santosh, Abhishek, Richa, Nithya, Krishnan, Sandeep, Yogesh, Orance, Karthik, Chetan, Shailesh, Amit, Apoorv, Chanakya, and Rajesh. I want to say thanks to you all for all the chit-chat, eating, and drinking parties. I actually know that this list is incomplete, but you are in it and you know it.

A special thanks goes out to all my college/school friends especially: Manish, Navdeep, Vipan, Chandan, Vijay, Sanjay, Yogesh, Sangita, Venkat, Damoder, Sunit, Nikunj and many more.

Finally, I wish to thank my wonderful family – my parents (Shyam Lal and Kesari Devi), my brother (Vinod), my sisters (Ranjana, Anju, Manju, and Attu), my brother-in-laws (Kamal, Vidya Ratan, Sanjeev, Chanderhas), my sister-in-law (Deepika), my grandmother (Judya), my wife (Chetna), and my son (Kalkin) – for their immense love and support. I express my deepest appreciation and special gratitude to my parents for their selfless support throughout my life. I also thank my parents-in-law (Ramesh and Champa), my nephews and nieces - Gudiya, Raghaw, Shuku, Nikhil, Arpit, Sambhivi, Subarh, Vedakshi, and my little Tiya for their love. A special thanks goes to my wife Chetna and my son Kalkin (born on 3^{rd} August 2014 in Ghent, Belgium) for their patience, understanding, and love, and giving me time to work for my PhD and write this book. I dedicate this dissertation to my family.

Love you all!!

Germany, April 2016
Sachin Sharma

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

ACK	Acknowledgment
ADSL	Asymmetric Digital Subscriber Line
ANTS	Active Node Transfer System
ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency NETwork
ARPU	Average Revenue Per User
AS	Autonomous System
ASIC	Application-Specific Integrated Circuits
ATM	Asynchronous Transfer Mode
ATPG	Automatic Test Packet Generation

B

BCAM	Binary Content Addressable Memory
BE	Best Effort
BFD	Bidirectional Forwarding Detection
BGP	Border Gateway Protocol
BRAS	Broadband Remote Access Server
BSD	Berkeley Software Distribution
BT	Basic Reference Topology

C

CAM	Content Addressable Memory
CDN	Content Delivery Network
CIDR	Classless Inter Domain Routing
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CT	Core Topology

D

DARPA	Defense Advanced Research Projects Agency
DCOM	Distributed Component Object Model
DHCP	Dynamic Host Configuration Protocol
DPDK	Data Plane Development Kit
DSL	Digital Subscriber Line
DSLAM	Digital Subscriber Line Access Multiplexer

F

FIB	Forwarding Information Base
FIBRE	Future Internet testbeds/experimentation between Brazil and Europe
ForCes	Forwarding and Control Element Separation
FTTB	Fiber To The Building
FTTH	Fiber To The Home

G

GENI	Global Environment for Network Innovations

GPL	General Public License

H

HD	High Definition
HP	High Priority
HSA	Header-Space analysis
HTTP	Hypertext Transfer Protocol

I

I2RS	Interface to the Routing System
IBSP	Inter-Burst Segregation Protocol
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IP	Internet Protocol
ISDN	Integrated Services Digital Network
IS-IS	Intermediate System to Intermediate System

L

LAN	Local Area Network
LFB	Logical Functional Blocks
LLDP	Link Layer Discovery Protocol
LPM	Longest Prefix Match
LOS	Loss of Signal
LT	Large Topology
LTE	Long Term Evolution

M

MAC	Media Access Control
MAN	Metropolitan Area Network
MPLS	Multiprotocol Label Switching
MTU	Maximum Transmission Unit

N

NCP	Network Control Points
NETCONF	NETwork CONFiguration protocol
NFV	Network Function Virtualization
NS	Network Operating System
NSIS	Next Steps in Signalling
NTP	Network Time Protocol

O

OFELIA	OpenFlow in Europe: Linking Infrastructure and Applications
OLT	Optical Line Terminal
ONF	Open Networking Foundation
ONOS	Open Network Operating System
ONU	Optical Network Unit
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First
oTCL	Object oriented extension of Tool Command Language
OTT	Over-The-Top
OVS-DB	Open vSwitch Database Management protocol

P

PPP	Point-to-Point Protocol

Q

QoS	Quality of Service

R

RAM	Random Access Memory
REST	Representational State Transfer
RFC	Request For Comments
RIB	Routing Information Base
RPR	Resilient Packet Ring
RT	Ring Topology

S

SCTP	Stream Control Transmission Protocol
SDN	Software Defined Networking
SMTP	Simple Mail Transfer Protocol
SLA	Service Level Agreement
SPARC	Split Architecture for Carrier-Grade Networks
SPC	Store Program Control
SRAM	Static Random Access Memory
STP	Spanning Tree Protocol

T

TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TLS	Transport Layer Security
TT	Triangular Topology

U

UNIFY	UNIFYing cloud and carrier Networks
UDP	User Datagram Protocol

V

VDSL	Very high bit rate Digital Subscriber Line
VLAN	Virtual Local Area Network
VM	Virtual Machine
VPS	Virtual Path Slice
VS	Vendor-Specific Extension

W

WAN	Wide Area Network

Samenvatting – Summary in Dutch –

Toestellen zoals routers en switches zijn doorheen de voorbije decennia een steeds belangrijker rol gaan spelen in de ondersteuning van internet-gebaseerde services. Deze netwerktoestellen bestaan uit twee soorten functionaliteit: data plane-functionaliteit en control plane-functionaliteit. Het data plane stuurt het verkeer door naar de bestemming, terwijl het control plane de noodzakelijke taken uitvoert om het data plane te configureren. Momenteel ondersteunen netwerktoestellen een significant aantal technieken om diensten aan te bieden over het internet. Ondanks het indrukwekkende parcours dat deze technieken hebben afgelegd, is er nood aan snellere innovatie om te kunnen voldoen aan de immer stijgende noden van gebruikers en applicaties. Dit houdt in dat er sneller nieuwe toepassingen moeten ontworpen kunnen worden, en dat flexibelere, complexere toestellen beschikbaar worden.

De huidige netwerkinfrastructuur daartegenover is duur, complex, kan zich moeilijk aanpassen an de wijzigende vereisten, en werkt een sterke afhankelijkheid van de fabrikanten in de hand. Om het hoofd te bieden aan deze problemen, worden Software geDefinieerd Netwerken (SDN) de afgelopen jaren naar voren geschoven als oplossing. SDN biedt de mogelijkheid om een flexibel network te ontwerpen, de bijhorende complexiteit te reduceren en beoogt eenvoudigere netwerkinnovatie. SDN bereikt dit door een standaardprotocol te definiëren voor de communicatie tussen het control en het data plane. Dit laat toe om het control plane los te koppelen van de netwerktoestellen en ze uit te voeren op specifieke, andere toestellen, die we controllers noemen. Op dit ogenblik is OpenFlow het de-facto SDN protocol voor de communicatie tussen het control- en data plane van netwerktoestellen.

In deze verhandeling onderzoeken we OpenFlow-netwerkarchitecturen en hoe deze architecturen kunnen aangepast worden in functie van toekomstige communicatiediensten. Het onderzoek richt zich op het aanbieden van snelle herstelling van fouten, automatische verificatie en configuratie, hoge servicekwaliteit en verliesvrije pakketbezorging.

Voor snelle herstelling van fouten focust deze thesis op het aanbieden van ultrasnel foutherstel van diensten die worden aangeboden in OpenFlow netwerken. Om dergelijk snel herstel te bereiken, moet een netwerk in staat zijn zich te herstellen in 50 ms of minder. We implementeren hiervoor twee welbekende hersteltechnieken: restoratie en protectie. In restoratie worden herstelpaden gezocht nadat er zich een fout voordoet, terwijl in het geval van protectie deze herstelpaden gezocht worden voordat er een fout optreedt. Hierdoor kan bij protectie het verkeer worden omgeleid langs een van deze paden, zodra er zich een fout voordoet. Uit het onderzoek naar snel foutherstel blijkt dat door de gecentraliseerde aard van OpenFlow het moeilijk is om met restoratie ultrasnel herstel te bereiken in een netwerk met een hoog aantal flows. Aangezien restoratie een significante tijdspanne nodig heeft om de herstelactiviteit te voltooien, kan OpenFlow protectie implementeren om aan herstelvereisten te voldoen. Protectie, waar herstelacties genomen worden door de OpenFlow toestellen zelf, kan probleemloos in het kader van gecentraliseerde controle. Het onderzoek besluit dat protectie de voorkeur krijgt om in SDN/OpenFlow een herstel van fouten in minder dan 50 ms te bereiken, zelfs in grote netwerken met een hoog aantal flows.

Vervolgens focust ons onderzoek op de verificatie van de data plane functionaliteit van OpenFlow netwerktoestellen, om fouten bij het vergelijken van flows te vermijden. Er zijn twee oorzaken voor deze fouten in de data plane functionaliteit: (1) bugs (in software of hardware) in de OpenFlow data plane-implementatie en (2) fouten in de FlowTable configuraties. Het doel van de verificatiemechanismen is om die pakketkarakteristieken te vinden die ofwel foutief, ofwel niet kunnen doorgestuurd worden door het data plane. Zonder deze verificatie kan het moeilijk zijn om de pakketten op te sporen die niet of foutief kunnen afgeleverd worden door het toestel.

Automatische configuratie hebben we in twee aspecten onderzocht: (1) automatische bootstrapping van OpenFlow netwerken en (2) automatische configuratie van routingprotocollen in OpenFlow netwerken. Voor automatische bootstrapping moeten OpenFlow toestellen automatisch een sessie opzetten met een controller (zonder enige manuele configuratie). Zon bootstrappingtaak is complex voor een netwerk (of zijn toestellen) waar control- en dataverkeer verstuurd worden op hetzelfde kanaal (i.e., in-band netwerken). Hierdoor moeten OpenFlow toestellen (zonder control plane functionaliteit) een pad (of sessie) naar de controller zoeken en opzetten door gebruik te maken van andere toestellen in een in-band netwerk. Om automatische bootstrapping uit te voeren, introduceren en evalueren we een methode waarbij de controller zijn eigen controlenetwerk vaststelt via de switches waarmee hij verbonden is door het OpenFlow-protocol. Het onderzoek besluit dat deze methode het bootstrappingproces in een minimale tijdspanne voltooit, waardoor de methode geschikt is voor grootschalige netwerken. Voor het tweede aspect, de automatische configuratie van routingprotocollen, stellen we een raamwerk voor dat een extra controller vergt die de netwerkconfiguraties ontdekt (bv. de onderliggende topologie). Na het ontvangen van deze configuraties, sturen de OpenFlow toestellen automatisch de routing aan. Het raamwerk is geëvalueerd aan de hand van complexe netwerkarchitecturen. Het besluit van deze evaluatie, in vergelijking met manuele configuratie, is dat het voorgestelde automatische raamwerk de configuratietijd van de routingprotocollen significant verkleint.

Met het oog op het verhogen van de kwaliteit van de aangeboden service (QoS), stellen we voor om een dynamische voorrangsregel in OpenFlow netwerken te introduceren, om een hoge QoS te voorzien voor gebruikers met hoge prioriteit. Om QoS te implementeren, introduceren we een raamwerk met een dynamische voorrangsregel voor het internet. Het raamwerk laat toe een pad, vrij van interferentie met ander verkeer, te creëren tussen twee eindpunten op verschillende autonome toestellen voor een gegeven applicatieflow (bv. WebHD Video Streaming of HD Video to Video). Het raamwerk is geëvalueerd in verschillende referentienetwerkscenarios voor een stad met 1 miljoen inwoners, waarbij xDSL (Digital Subscriber Line), LTE (Long-Term Evolution) en Fiber networkingscenarios werden nagebootst. De evaluatie bevestigt dat het raamwerk geschikt is voor het internet en dat het een hoge QoS voor verkeer van hoge prioriteit voorziet.

Om pakketgeschakelde netwerken (inclusief OpenFlow netwerken) te garanderen dat er geen pakketten zullen verloren gaan, stelt deze verhandeling het Inter Burst Segregation Protocol (IBSP) voor. Het protocol is geëvalueerd op netwerksimulatoren (NS-3) en door emulatie op een platform met hoge performantie (d.w.z. data plane ontwikkelingskit). Uit de evaluaties kunnen we besluiten dat zonder het gebruik van IBSP een pakketgeschakeld netwerk geen behoud van pakketten kan garanderen, zelfs niet bij laag bandbreedtegebruik. Bij gebruik van IBSP kan behoud van pakketten wel gegarandeerd worden, ook in netwerken waarbij nagenoeg de volledige bandbreedte in het netwerk wordt gebruikt.

Tot slot kunnen we in de context van SDN netwerkarchitecturen een aantal richtingen voor toekomstig onderzoek verkennen. Deze toekomstige richtingen zijn: (1) transitie van legacy netwerken naar SDN-netwerken, (2) performantie, (3) probleemoplossen en (4) veiligheid.

Summary

In recent decades, network devices such as switches and routers have been successfully developed and deployed to deliver a plethora of services over the Internet. These network devices contain two elements: data plane and control plane. The data plane forwards traffic towards its destination, while the control plane performs the necessary tasks that allow the data plane to make forwarding decisions. Currently, network devices support a significant number of technologies to deliver services over the Internet. Despite the impressive track record of these technologies, the need to accelerate innovations has been increasing to meet growing demands of users and applications. The accelerated pace of innovations means more features need to be implemented in short timeframes, meaning more flexible (and thus complex) devices are needed. Currently, the network infrastructure has become expensive, complex, prone to vendor-locking, and inflexible to adapt to the needs of changing requirements. To overcome these problems, Software Defined Networking (SDN) has been emerging in recent years. In fact, SDN has a potential in designing a flexible network, fostering innovations, and reducing complexity. SDN achieves these by defining a standard protocol for communication between the control and data plane. Therefore, it allows decoupling of the control plane from network devices and embedding it into external devices called controllers.

Currently, OpenFlow is the de-facto SDN protocol for communication between the control and data plane of network devices. In this dissertation, we investigate OpenFlow network architectures, and perform research on how these architectures can be adapted to be suited for future communication services. The research aims at providing fast failure recovery, automatic verification, automatic configuration, high quality-of-service, and loss-free packet-switching solutions to OpenFlow.

For fast failure recovery, this dissertation focuses on providing carrier-grade quality to services provisioned in OpenFlow networks. For achieving carrier-grade quality, a network should be able to recover from a failure within 50 ms. We implement two well-known recovery techniques, restoration and protection, in OpenFlow networks. In restoration, recovery paths are established after a failure occurs and in protection, recovery paths are established before a failure occurs and hence, when the failure is detected, traffic is redirected to the recovery path. The research with fast-failure recovery techniques concludes that due to the centralized nature of OpenFlow, it is difficult for restoration to achieve carrier-grade quality in a network containing a large number of flows. As restoration may take significant time to complete recovery activities, OpenFlow can implement protection to

meet the carrier-grade recovery requirement. Protection, where recovery actions are taken by OpenFlow devices themselves, does not suffer from limitations of centralized control. The research concludes that protection is a way in SDN/OpenFlow to achieve failure recovery within 50 ms, even in a large-scale network serving many flows.

We considered node and link failures in the above fast failure recovery study. However, failures can also be caused by other errors in the data plane functionality (such as matching errors). Therefore, in the next study, we focus on verification of the data plane functionality of OpenFlow network devices for finding flow-matching errors. There can be two reasons for these errors in the data plane functionality: (1) bugs (software or hardware) in OpenFlow data plane implementation and (2) errors in FlowTable configurations. The objective of verification is to find the packet-headers that cannot be forwarded or can be forwarded incorrectly through the data plane. In the absence of this verification, it may be difficult to find which packets cannot be delivered or can be delivered incorrectly by a device.

In addition, we perform research on automatic configuration in SDN/OpenFlow. Automatic configuration is researched for two aspects: (1) automatic bootstrapping of OpenFlow networks and (2) automatic configuration of routing protocols in OpenFlow networks. For the former aspect, OpenFlow devices have to automatically establish OpenFlow sessions with the controller (in the absence of any manual configurations). Such a bootstrapping task is complex for a network (or its devices) where control and data traffic are transmitted on the same channel (i.e., in-band networks). To perform automatic bootstrapping in these networks, we propose and evaluate a method in which the controller establishes its own control network through the switches that are connected to it through the OpenFlow protocol. The research concludes that the proposed method allows bootstrapping in a minimal time, making it suitable for a large-scale network. For the latter aspect, i.e., automatic configuration of routing protocols, we propose a framework which runs an additional module (i.e., a controller application) to discover network configurations (e.g., underlying topology). After receiving these configurations, the OpenFlow controller automatically configures routing protocols. The framework is evaluated using complex network topologies. The evaluation concludes that compared to manual configurations, the proposed automatic configuration framework decreases the time to configure routing protocols significantly. Furthermore, the proposed framework is used to configure routing protocols in the quality of service (QoS) study (discussed in the next paragraph).

For the QoS study, we propose a framework to enable a dynamic right of way in OpenFlow networks to provide high QoS for high priority users. The proposed framework establishes high QoS in paths discovered by routing protocols used in the Internet. The framework allows an interference-free path, from other traffic, between any two endpoints, on multiple autonomous systems, for a given application flow (e.g., WebHD Video Streaming or HD Video to Video). The framework is evaluated in distinct reference network-scenarios for a city with a

population of 1 million inhabitants, emulating xDSL (Digital Subscriber Line), LTE (Long-Term Evolution) and Fiber networking scenarios. The evaluations confirm the suitability of the framework for the Internet, providing high quality of service for high-priority traffic.

In addition to the above QoS study, this dissertation proposes a protocol, called inter-burst segregation protocol (IBSP), which can guarantee zero packet-loss in packet-switched networks (including OpenFlow networks). The protocol is evaluated through simulations on a network simulator (i.e., NS-3) and through emulations on a high-performance platform (i.e., data plane development kit). The evaluations conclude that without using IBSP, a packet-switched network cannot guarantee zero packet-loss, although the bandwidth usage in the network is low. However, using IBSP, zero packet-loss can be guaranteed, even though nearly all the bandwidth is consumed in the network.

We also contextualize our work according to the recent trends in future SDN network architectures, enabling directions for future work. The future directions are for: (1) transition of a legacy network to a SDN network, (2) performance concerns, (3) troubleshooting concerns, and (4) security concerns.

1 Introduction

"The beginning is the most important part of the work."

–Plato, The Republic

The Advanced Research Projects Agency NETwork (ARPANET) was the first packet-switched network that became the basis of the Internet. When the ARPANET first time went into operation in 1969, it was a network of just four computers located at different sites. However, the size of the ARPANET grew and it became a network of networks, the Internet. Today, the Internet comprises of a huge interconnection of thousands of networks. Although the overall Internet architecture is an unquestionable success, the underlying infrastructure has not progressed very well in order to meet changing requirements of the increasing number of users and applications. Currently, the Internet infrastructure has become too expensive to build, too complex to manage, too prone to vendor-locking, and too inflexible to adapt to the needs of changing requirements [1]. This is because the current Internet infrastructure relies on devices which contain two elements: control plane and data plane, and a propitiatory interface (i.e., closed implementation) for communication between them. The control plane performs the necessary tasks that allow the data plane to make forwarding decisions, while the data plane forwards packets towards their destinations (using the forwarding decisions made by the control plane).

The Internet (i.e., its networks) needs a solution that can be implemented without changing its infrastructure too much and spending a lot of money, and therefore, opens up new business opportunities. In recent years, Software Defined

Networking (SDN) has been emerging to address above problems. The concept of SDN is applicable to the networks that build up the Internet. In fact, SDN defines a standard interface for communication between the control and data plane and therefore, allows decoupling of the control plane from network devices.

SDN has gained significant interest from many research communities and many of the research challenges behind it are or have been widely investigated in several projects. Some of these projects are: GENI (Global Environment for Network Innovations) [2], SPARC (SPlit ARchitecture for Carrier-grade networks), OFELIA (OpenFlow in Europe: Linking Infrastructure and Applications) [4], and UNIFY (UNIFYing cloud and carrier networks) [5]. Industrial players such as Deutsche Telekom, Google, Microsoft, Verizon, and Yahoo! have shown substantial interest towards SDN and have formed ONF (Open Networking Foundation) to promote and adopt SDN through standardization [6]. There are already several SDN based commercial solutions available in the market. These are from NEC, HP, Brocade, Juniper, etc.

OpenFlow is currently the de-facto SDN standard protocol for communication between the control and data plane of network devices. In this PhD research, we investigate OpenFlow network architectures, and perform research on how these architectures can be adapted to be suited for future communication services. The research aims at providing fast failure recovery, high quality-of-service, automatic configurations, troubleshooting, and loss-free packet-switching solutions to OpenFlow. This chapter introduces the work performed for this dissertation.

Section 1.1 and Section 1.2 present a background to the Internet and its problems. Section 1.3 provides SDN initiatives (i.e., road towards SDN). Section 1.4 introduces OpenFlow and its functionalities. Section 1.5 and Section 1.6 present research challenges and an overview of the work performed, respectively. Finally, we list all the publications obtained during the PhD research (Section 1.7).

1.1 Background

In this section, we present a background to the concepts, which are important for this dissertation. Currently, The layered model is first introduced and then different types of network elements and addressing schemes are presented. As concepts of software or hardware based switches/routers, control plane, data plane and flows are important to understand the basics of SDN, these are explained subsequently in this section. In addition, an overview of the current Internet infrastructure is important to understand current Internet problems. Therefore, this is described in further subsections.

1.1.1 Layered model

The operation of a network, and thus, the communication between its nodes (or devices), is often categorized in a layered model in which each layer uses the services offered by the lower layer. There are two popular models for representing the layered structure of the Internet: Open Systems Interconnection (OSI) [8] and Transmission Control Protocol/Internet Protocol (TCP/IP) [9]. OSI is a theoretical reference model for developing protocol standards in networking, while TCP/IP is a model that is a result of research and development conducted on the ARPANET.

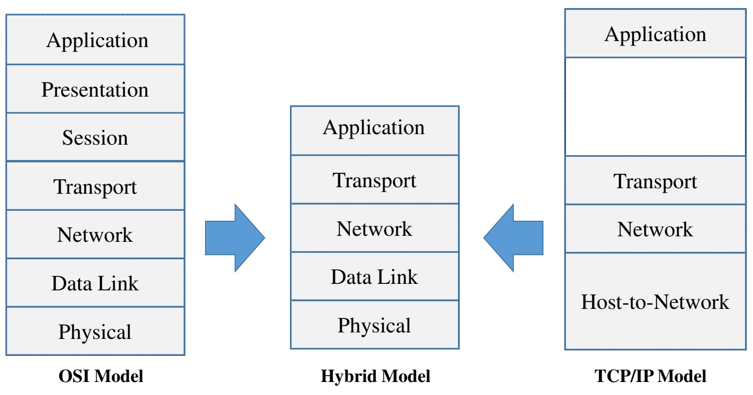


Figure 1.1: Layered models of the Internet (based on [9])

The TCP/IP model is not in conflict with the OSI model. Both models rely on the layered structure (See Fig. 1.1). However, the layers of TCP/IP are not in a one-to-one correspondence with the layers of OSI. The OSI model contains seven layers: application, presentation, session, transport, network, data link, and physical layer, while the TCP/IP model contains four layers: application, transport, network, and host-to-network layer. In the TCP/IP model, the presentation and session layers are not present. In fact, in the TCP/IP model, the application layer (or partially the transport layer) is responsible for functions that are performed by the presentation and session layers of the OSI model. In addition, the data link and physical layers of the OSI model are integrated in one layer (host-to-network layer) in the TCP/IP model. Fig. 1.1 also describes the hybrid model, which contains the main layers of both the models. Following paragraphs describe the layers of the hybrid model to present the OSI and TCP/IP model of the Internet:

1. **The application layer** allows users to run applications on network nodes. The layer does not define an application itself, but it defines services required to run the application. For example, application protocol HTTP (Hyper Text Transfer Protocol) defines how a web browser can pull the contents of a web page from a web server. The layer contains a variety

of protocols (such as HTTP, SMTP, and FTP) that are commonly needed by users and is responsible for exchanging information between applications, running on different end nodes.

2. **The transport layer** accepts data from the application layer, breaks it into small parts if needed, and provides communication between one application to another. Such communication is often end-to-end. The two most well known protocols for the transport layer are: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). If the transport layer protocol is TCP, the unit of data sent from the transport layer to the network layer is called Segment. However, if the transport layer protocol is UDP, the unit of data sent from the transport layer to the network layer is called datagram. In fact, TCP provides a reliable, connection oriented, flow control, and congestion control service, while UDP provides a connection-less service (without guaranteed delivery, flow control or congestion control) to the application layer. The meaning of terms such as connection-less, connection oriented, congestion control, reliability, and flow control is given below:

   Connection-less here means that data is sent from a source to a destination without establishing a session in advance (i.e., no acknowledgment is sent from the destination that it is ready and willing to accept data). In addition, the source does not attempt to monitor whether data is delivered to the destination. In contrast, connection oriented means that a connection between a source and a destination is first established through a handshaking process. Once the connection is established, the source sends data to the destination. The destination acknowledges then the receipt of any data sent by the source. If the acknowledgment of data is not received, the source re-sends the unacknowledged data. The TCP connection is called a reliable connection because it retransmits lost data and achieves guaranteed delivery. Using flow control mechanisms, TCP controls the data rate of a fast sender so that the receiver can handle all the incoming data. In addition, using congestion control mechanisms, TCP avoids congestion in the network.

3. **The network layer** includes the Internet protocol (IP). It adds the IP header to the segment/datagram received from the transport layer. The header of IP includes the source and destination IP address. IP provides a best effort service to deliver IP packets to their destination. It uses IP routing protocols (such as OSPF) to find a (shortest) route to the destination (see the next subsection). When sending an IP packet larger than the maximum transmission unit (MTU) allowed by the transmission link, the IP fragments the packet and sends each fragment separately to the data link layer. There are currently two versions of IP: IPv4 and IPv6. IPv4 utilizes a 32-bit address scheme, while IPv6 utilizes a 128 bit address scheme [9].

4. **The data link layer** is responsible for encapsulation of higher layer messages (i.e., IP packets) into frames that are sent through the physical layer. A frame includes the data link layer header on the top of an IP packet. The data link layer header includes the source and destination MAC (media access control) addresses. It also provides error control. Examples of data-link layer protocols are Ethernet IEEE 802.2 framing and Point to Point Protocol (PPP) framing.

5. **The physical layer** transmits raw bits through a transmission medium. Examples of the transmission mediums are: twisted pair cable, coaxial cable, optical fiber, and wireless medium. Typically, following factors are considered to choose a transmission medium: (1) transmission rate, (2) cost, (3) ability for installation, (4) capability to cope with environment conditions, and (5) distance.

We explained the hybrid model in the point of view of a source, i.e., the application layer sends data towards the physical layer (the top layer to the bottom layer direction). However, there are two additional cases: (1) the destination node receives data and (2) an intermediate node receives data. In the former case, nodes typically run all the layers and data is sent from the bottom layer to the top layer, while in the latter case nodes do not generally run all the layers. Intermediate nodes such as switches run the bottom two layers (the physical and data link layer), while nodes such as routers run the lower three layers (the physical, data link, and network layer). When data is received by an intermediate node, it is first sent from the bottom layer (physical layer) to the top layer to decide an output port through which data should be sent. Then, data is sent from the top layer to the bottom layer (physical layer) to transmit it over a transmission link. To decide the output port, nodes such as switches use MAC learning (discussed in the next subsection) and nodes such as routers use IP routing protocols.

1.1.2 Network elements

The Internet, a network of networks, is composed of a variety of network elements such as packet-switching devices (e.g., packet switches or routers) and circuit switching devices (e.g., SONET/SDH based). In circuit-switching devices, the forwarding decision takes place at the circuit level (i.e., based on the "position" of arriving bits, where the "position" is defined by space, time and wavelength). In contrast, in packet-switching devices, forwarding is performed on a per-packet basis. In context of IP routing, the forwarding decision in packet-switching devices is based on the IP destination address present in the IP packet header of the received packet. In this subsection, we describe some of the packet-switching devices, i.e., packet switches, routers, and middleboxes.

A packet switch uses data link layer technologies for forwarding packets. For example, an Ethernet switch uses Ethernet switching technologies for forwarding packets. In fact, an Ethernet switch contains the MAC address table and performs two operations: (1) MAC learning and (2) frame forwarding. MAC learning is responsible for building the MAC address table, while frame forwarding is responsible for sending out incoming frames to their destinations, based on the information available in the MAC address table. When an Ethernet switch receives a frame on a port, it performs MAC learning by searching the source MAC address (present in the frame header) of the frame in the MAC address table. If this MAC address is not present in the table, it creates a new entry in the table containing the source MAC address and the incoming port. Otherwise, the entry containing the source MAC address (in the table) is updated with the port information. After performing MAC learning, the switch forwards frames by first searching the destination MAC address of the frame in the table and then if the address is present, the frame is sent through the port information present in the corresponding entry in the table. However, if the address is not present (or it is a multicast or broadcast address) in the table, the frame is flooded in the network. To prevent loops, switches can run a spanning tree protocol (STP), which builds a spanning tree for a network and disables the links that are not part of the spanning tree.

A router is a network layer device that is responsible for forwarding IP data packets in a network. It runs routing protocols (such as Open Shortest Path First, OSPF) to make decisions for forwarding packets. These routing protocols typically send (or receive) link state information (messages) to (or from) neighboring routers and construct a topology of the network (from received link state information). It then adds path (shortest) information in the routing table to reach each destination. The network layer uses this information to forward incoming packets. Quagga [10] is an open software package based on the implementation of routers. It supports main standard routing protocols such as OSPF and BGP (Border Gateway Protocol), and can be installed in Unix-like platforms, particularly Linux, Solaris, and FreeBSD.

Middleboxes are intermediate devices that perform functions other than the standard functions of routers/switches (i.e., routing packets based on a path to reach a destination). The examples of middleboxes are: firewalls, NAT (Network Address Translation), load balancers, and web cache. Firewalls filter (i.e., accept or reject) traffic based on a set of predefined security rules. NAT translates private addresses (assigned in a local network) in packets into public addresses before these are forwarded to another network in the Internet and vice versa. A load balancer divides the load of a device into two or more devices and a web cache (or HTTP cache) is a server for the temporary storage (caching) of web content, such as HTML pages and images, to reduce bandwidth usage, server load etc.

1.1.3 Addressing schemes

In the layered model (shown in Section 1.1.1), the data link, network, and transport layers have their own addressing schemes. In this subsection, we describe these addressing schemes.

In the context of Ethernet switching, the link layer defines 48-bit MAC addresses for communication. In this case, the MAC addresses are specified by six groups of two hexadecimal digits separated by colons (e.g., 01:12:13:aa:bb:cc).

Table 1.1: Comparison among classful IP addresses. Here, N/D means "not defined".

class	Bits to start	Network ID size (in bits)	Host ID size (in bits)	start address	end address
A	0	7	24	0.0.0.0	127.255.255.255
B	10	14	16	128.0.0.0	191.255.255.255
C	110	21	8	192.0.0.0	223.255.255.255
D	1110	N/D	N/D	224.0.0.0	239.255.255.255
E	1111	N/D	N/D	240.0.0.0	255.255.255.255

For the network layer, IPv4 addressing schemes specify 32-bit addresses that are represented by four numbers separated by a dot (e.g., 192.168.11.12) (here, each number is 8 bit long). There are two schemes for IPv4 addressing: (1) classful addressing and (2) classless addressing. In classful addressing, IP addresses are divided into five classes: class A, class B, class C, class D, and class E. Class A, B, and C are for unicast addresses, while class D and E are for multicast and reserved addresses (mainly used for experimental and future use) respectively. The representation of all classful addresses is given in Table 1.1. Table 1.1 shows that class A, class B, class C, class D, and class E start with 0, 10, 110, 1110, and 1111 bits respectively. The network ID in Table 1.1 represents the network in which the device belongs and the host ID represents the device itself. In addition, it depicts the network size, the host size and the range of IP addresses in each class. The problem with classful addressing is that the gaps between sizes of different classes are enormous and therefore, a large number of IP addresses gets wasted [9]. For example, if a network has slightly more number of hosts than a particular class, it needs then either two networks of that class or the next class of a network. For example, for a network that has 400 hosts, it needs either a single class B network or two class C networks to assign each host an address. If a single class B network is allocated, a large number of host addresses gets wasted (because the number of hosts that can be specified in a class B network i.e., $(2^{16}-2)$, is significanly larger than the required host addresses, i.e., 400). However, if two class C networks are allocated, the number of available class C networks will exhaust quickly (because the number of networks that can be specified using class C addressing is only 2^{21}). Therefore, classful addressing is replaced by classless Inter-Domain Routing

(CIDR) in 1993.

In CIDR notation, an IP address is represented as A.B.C.D /p, where ”/p” is called the IP prefix or network prefix, and A, B, C, and D represent 8 bits of a 32 bit IPv4 address. The IP prefix determines the number of significant bits used to identify a network. For example, 192.10.15.10 /24 means that the first 24 bits are used to represent the network and the remaining 8 bits are used to identify hosts. Using CIDR notation, any number of contiguous bits can be assigned to identify networks. For example, if a network has 400 hosts. If CIDR is used, this network can be assigned an IP address with a network prefix of 23 (i.e. /23). This means, 9 bits are available for hosts, resulting into 512 available host IP addresses (i.e., very less wastage of IP addresses).

In contrast to the IP layer, the transport layer uses 16 bit’s unique port numbers to distinguish the segments and datagrams of separate applications.

1.1.4 Software switches/routers vs. hardware switches/routers

In this subsection, software and hardware based switches/routers are described. Software based switches/routers mean that packet forwarding is performed in software, while hardware based switches/routers mean that packet forwarding is performed in hardware.

1. **Software switches/routers**

   At the time when the ARPANET was used as an experimental network, most routers/switches were general-purpose Unix computers, running software to forward packets. A router/switch maintained a table (an efficient search data structure) in software for finding a route to a destination. In case of a router, the table is called routing table and in case of a switch, the table is called MAC address table. These tables contain information about the output interface where incoming packets (having a certain destination address) should be forwarded. When a packet arrives at a router/switch (i.e., Unix computer), software inside the router/switch extracts the destination address from the packet-header and looks up the address in the table. Based on the output interface information in the table, packets are forwarded to the destination.

2. **Hardware switches/routers**

   In the late 1980s, the ARPANET had grown from being an experimental network to a commercial network. At that time, new startups such as Cisco Systems and Wellfleet Communications started building a special-purpose, commercial version of routers/switches. The first major use of hardware acceleration in these networks was via the use of ASICs (application-specific integrated circuits) to perform high-speed hashing functions for table lookups. In the mid 1990s, advances in content addressable memory (CAM) made it possible to perform very high speed lookups (almost equal to the line rate speed). A CAM is a memory that performs the lookup task in a single clock cycle using a comparison circuitry. Unlike standard memory (random access memory) in which a memory address is used to return data stored at that address, a CAM is designed such that data stored on the CAM can be accessed by searching for the content itself and the memory retrieves one or more TCAM entries from where the content can be accessed.

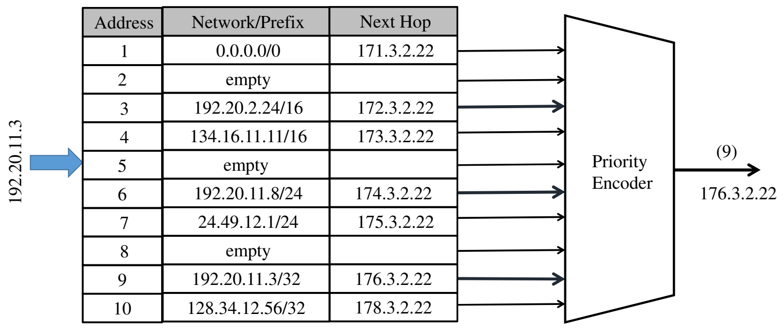
Currently, there are two types of content addressable memory (CAM): binary (BCAM) and ternary (TCAM). BCAM supports storing of binary bits (i.e., zero or one: 0,1), while TCAM supports storing of binary as well as don't care bits (0,1,X). BCAM is usually used to perform lookups in switches, while TCAM is usually used to perform lookups in routers. There are two major disadvantages of content addressable memory: high cost and high power consumption.

In switches, the lookup task (i.e., searching a destination address in the MAC address table) in hardware is somewhat a straightforward task, as lookups have to be performed on an exact match. However, the lookup task in routers (i.e., searching a destination IP address in the routing table) is complicated because devices have to perform lookups on the closest match on a network address, where the match may only be on the most significant bits of a network address [9]. As it is possible that multiple entries of the routing table can find a match for a destination address, the router has to select one of the matched entries for forwarding a packet. If more bits of the destination address are matched with an entry, the network address becomes more specific. Therefore, an entry with a longest prefix match (LPM) is selected for forwarding packets. LPM is the technique (leveraging the CIDR) used in routers to reduce routing table sizes.

Fig. 1.2 depicts an example of longest prefix matching in TCAM. The network address/prefix and the next hop information are present in TCAM. Fig. 1.2 shows that a search for IP address 192.20.11.3 in TCAM gives three matching results, i.e., (3),(6), and (9). Since the entries are sorted by the prefix length, the priority encoder (see Fig. 1.2) gives the next hop information of the lowest matching entry, i.e., (9), for longest prefix matching.

1.1.5 Basic Router Design - control plane and data plane

The basic design of routers can be presented by two elements: control and data plane (see Fig. 1.3). Fig. 1.3 also shows the interface between the control and data



Address	Network/Prefix	Next Hop
1	0.0.0.0/0	171.3.2.22
2	empty	
3	192.20.2.24/16	172.3.2.22
4	134.16.11.11/16	173.3.2.22
5	empty	
6	192.20.11.8/24	174.3.2.22
7	24.49.12.1/24	175.3.2.22
8	empty	
9	192.20.11.3/32	176.3.2.22
10	128.34.12.56/32	178.3.2.22

Figure 1.2: Longest Prefix Matching (LPM) example in TCAM (source [11])

plane. However, this interface between the control and data plane has always been a proprietary and closed implementation.

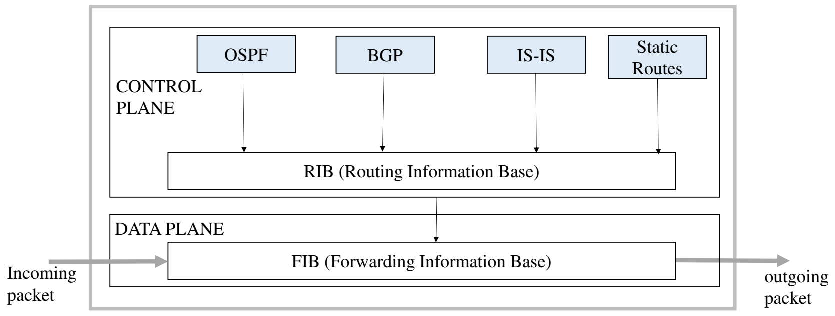


Figure 1.3: Basic router design

The control plane is the brain of routers. It consists of dynamic IP routing protocols, such as OSPF (open shortest path First), BGP (border gateway protocol), and IS-IS (intermediate System to intermediate System), and many other protocols, such as IGMP (Internet group management protocol), ICMP (Internet control message protocol), ARP (address resolution protocol)[1], BFD (bidirectional forwarding detection protocol) and so on. Fig. 1.3 shows that the control plane also contains the routing information base (RIB).

The RIB is the routing table where all IP routing information is stored. When a routing protocol learns a new route, it adds the route into the RIB. When a destination becomes unreachable, the respective route is removed from the RIB. In addition, a route can be added by an administrator (see static routes in Fig. 1.3).

[1]ARP is a layer 2 control plane protocol for switches

As the control plane is run on a low-end CPU (central processing unit), processing of packets is slower in the control plane than the data plane. Therefore, routes from the RIB are inserted into the data plane (i.e., in the FIB) for fast forwarding of packets. In addition, there can be multiple routes to the same destination in the RIB. Among these multiple routes, a single best route is installed in the FIB from the RIB. The FIB is a part of the data plane.

The data plane is responsible for packet buffering, packet scheduling, header modification and forwarding. It typically consists of ports that are used for the reception and transmission of packets, and the FIB. The FIB contains entries that are installed from the RIB. It uses a high speed lookup memory (such as TCAM) to store entries. The description of entries in a high speed memory (such as TCAM) is given in the previous subsection.

1.1.6 Packet-flow (or simply flow)

A flow is a sequence of packets traveling from a source to a certain destination (the destination can be a unicast, multicast, or broadcast destination) at a certain point in time. A flow can be uniquely identified by parameters such as: (1) source IP address, (2) destination IP address, (3) source port, (4) destination port, and (5) layer 4 protocols (TCP/UDP). Any combinations of these parameters can form a flow. For instance, when a web browser is opened and www.google.com is typed, this creates a new flow with the following parameters: (1) transport protocol: 6 (i.e., TCP), (2) source port, e.g., 1234, (3) destination port, 80, (4) source IP, e.g., 1.2.3.4, and (5) destination IP: the IP address of www.google.com. The concept of a flow is important, since it may be that packets from one flow are needed to be handled differently from others, by means of separate queues/actions. Therefore, using flow parameters, packets of different flows can be distinguished to apply different actions (such as traffic shaping).

1.1.7 Overview of the Internet Infrastructure

Networks can be classified according to their size, such as: (1) local area network (LAN), (2) metropolitan area network (MAN), and (3) wide area network (WAN). A LAN consists of a network that is restricted to a small area, typically a local office, house, or building. A MAN is larger than a LAN and can cover an area from several miles to tens of miles. A WAN occupies a very large area, such as a state, an entire country or the whole world. A WAN can contain multiple small networks such as LANs or MANs.

The Internet infrastructure can typically be divided into five network segments: the home network, access network, (metro-) aggregation network, core (backbone) network, and campus network (or data center) (see Fig. 1.4). A description of all these segments is given below:

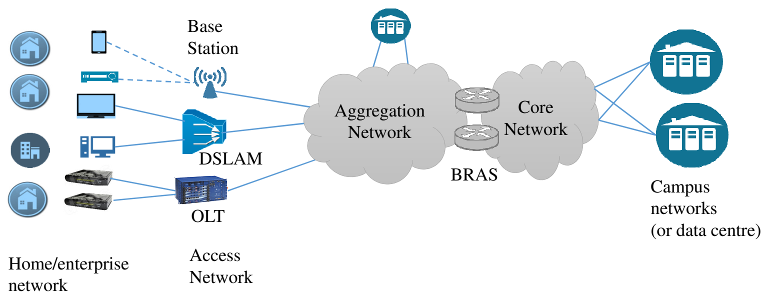


Figure 1.4: High level overview of Internet infrastructure segments

1. **Home Networks**

   A home network (a small company network) is a network that typically consists of a few number of end-nodes (e.g., personal computers) which are connected by a wired or wireless LAN. This normally spans a limited area (such as the size of a home or a building). Home networks (or LAN) has scaled up in speed from 10 Mb/s (in 1980s [12]) to 100 Gb/s (today) [13]. In future, the achieved speed is expected to be higher than today's achieved speed.

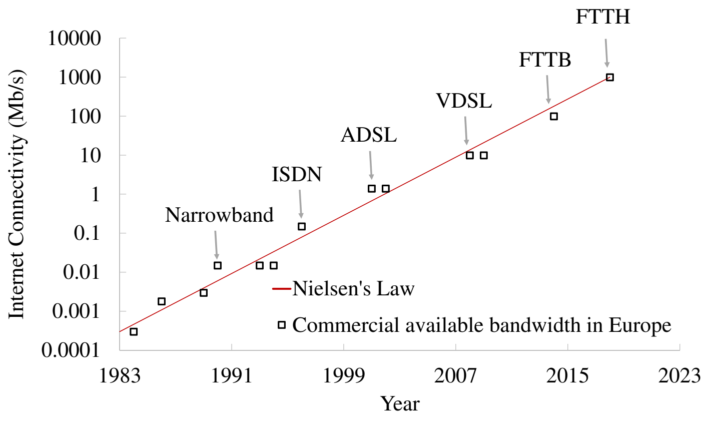


Figure 1.5: Bandwidth evolution in access networks [source: IEEE spectrum]

2. **Access networks**

   Access networks connect home and/or small business networks to the Internet. It is typically built up as a tree structure, where redundancy is limited to the connections of business users (mostly in a ring structure), and spans in a couple of kilometers. Typical data rates that can be currently achieved in these networks range from 100 Mb/s to 1 Gb/s per end user. Devices in these networks include: DSLAM (Digital Subscriber Line Access Multiplexer) in case of DSL (digital subscriber line), OLT (optical line terminal) in case of Fiber, and base stations in case of cellular mobile connections. Fig. 1.5 shows the different technologies that are used in access networks over several years in Europe to meet increasing bandwidth demands of users. The technologies used are: ISDN (integrated services digital network), ADSL (analog digital subscriber loop), VDSL (very high bit rate DSL), FTTB (fiber to the building), and FTTH (fiber to the home). Fig. 1.5 shows that the adoption of technologies matched with the Nielsen's law of bandwidth evolution [14].

3. **Aggregation or metro networks**

   Aggregation networks interconnect access networks in a ring or slightly meshed network topology of tens of network nodes. These networks aggregate traffic (from several access networks) and feed it into the core network. These networks typically span areas with the diameters of up to 50 km. Currently, the typical channel speed that is achieved in these networks is 10 Gb/s.

4. **Core or backbone networks**

   The core network connects aggregation networks to the core of the Internet (i.e., it connects countries and continents over large distances). BRAS (broadband remote access server) is typically used to connect aggregation networks to the core network. The core network topology is strongly meshed, and fastest technologies are used in this network to deliver data. This is the part of the Internet where fiber-optic cables (high speed) were used for the first time. Core networks have been changed dramatically in recent decades. Today, high bandwidth capacity such as 100 Gb/s is reality in these networks [15].

5. **Campus networks**

   Campus networks are generally run by some universities or corporate-sized companies (containing up to 1000 of nodes). Data centers can be considered to belong to this segment [16]. These networks are normally connected to the core (or to a metro network).

1.2 Problems of the Internet

The Internet has been changing radically communications to the extent that it is now our favorite medium of daily communication. A wide range of applications for news, entertainment, business, commerce, and social networking has been launched over the Internet. As the Internet is not able to efficiently support the increasing demands (such as performance, reliability, and scalability) of these applications, the Internet infrastructure has been upgraded constantly by enhancing the software or hardware part of its network devices. Every time the Internet faces a new challenge, new standards are proposed to overcome the challenge. The number of standards published per year by IETF (The Internet Engineering Task Force) is illustrated in Fig. 1.6.

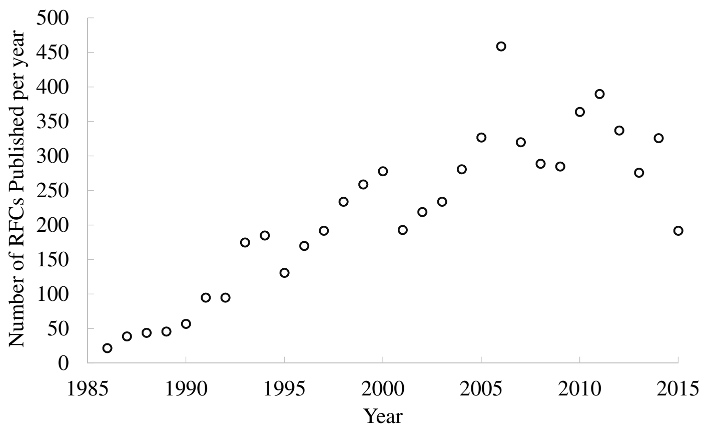


Figure 1.6: The number of standards published per year by IETF

There are following problems that occur frequently on the Internet:

1. **Inflexibility**

   Currently, it is not possible to quickly offer new services, which require changes in available protocols of network devices. These services must wait for vendors (and standards bodies such as IETF) to approve and incorporate new solutions in operating networks. Currently, the standardization process for a new solution (or protocol) is a long-lasting process. Even if operators find promising solutions, they need to wait for years to implement them in their networks. This might bury lots of interesting opportunities due to the lack of support from vendors. In addition, there is long release cycles for implementing a new solution in network devices, as the bug finding and testing cycle should be extensive and regress to prevent a network from failing. The result of long waiting time is that network operators have to deal years and years with old legacy equipment not yet capable of running the latest protocols.

2. **High complexity**

   Network operators deal with a variety of heterogeneous devices (e.g., routers, firewall, and switches), which have a progressively reduced life cycle (due to fast hardware and software additions). A reduced lifetime facilitates introduction of new protocols (to meet growing demands of users) and also increases the complexity of the network infrastructure.

3. **Manual configurations**

   Due to a reduced life time of network devices, network operators may have to manually configure devices containing new or old protocols many times. The problem is that manual configuration may result into frequent misconfiguration, increasing the deployment time of new protocols (functions) on the current Internet infrastructure.

4. **High Cost**

   It is already stated that lots of standard protocols (see Fig. 1.6) are proposed for the Internet to meet growing demands. Currently, all network devices (proprietary devices) implement almost all the standard protocols in their control plane. These protocols are implemented in a closed environment in networking (i.e., each vendor implements, develops, and tests a large amount of redundant code, which is is not available in an open software environment). This clearly increases the costs to software development.

5. **Vendor interoperability**

   Over the years standards have been developed for most relevant protocols that are used by network devices. Vendors implement these standards in a manner that allows heterogeneous devices from multiple vendors to function with one another. However, in addition to the implementation of these standards, vendors always add enhancements, which allow vendors to outperform their competition. As many vendors add such enhancements, the result is that each vendor device has difficulty to operate smoothly with products from another vendor.

Software Defined Networking (SDN) has been emerged to address above problems by making networks more programmable. One of the major drivers of SDN is its simplification. It simplifies the network infrastructure by allowing it to decouple the control plane (complex software) from network devices.

1.3 The road towards SDN

SDN has been receiving a considerable amount of attention in recent years. However, the idea of programmable networks is there from many years. We divide the SDN work into two categories: (1) early SDN initiatives (i.e., from 1980s to 2000s), and (2) recent SDN initiatives (from 2000s to until now).

1.3.1 Early SDN initiatives

The most important early initiatives in the direction of SDN can be categorized into three approaches: (1) Store Program Control (SPC) approach, (2) Active Networking approach, and (3) OpenSig Approach. Table 1.2 gives an overview of projects involved in these three approaches.

Table 1.2: Early SDN initiatives

	Early SDN initiative	Projects or Companies involved
1.	SPC Approach	AT&T Company
2.	Active Networking Approach	ANTs [21], Smart Packets [22], Netscript [23], and Switchware [24]
3.	OpenSig Approach	Tempest (switchlets) [25], Xbind [26],

A short description of these early initiatives is given below:

1. **SPC approach**

   The SPC approach [18] is the first approach to separate the control and data plane. This was introduced by AT&T in 1980s to improve the management and control of telephone networks. Prior to SPC, all telephonic calls were managed and controlled through the circuit switches involved in a call. However, due to a number of issues (such as limited processing power of circuit switches, limited visibility of network resources, and limited amount of programming that could be safely accomplished on these devices), SPC was introduced. Using SPC, all the administrative functions of setting up calls were offloaded to external entities called Network Control Points (NCP). NCP became the basis on which many telephonic network features (call centers, 800-numbers i.e., toll-free numbers, calling cards etc.), still in use today, have been built [19].

2. **Active Networking Approach**

   The Active Networking Approach (appeared in 1997) was mainly supported by the Defense Advanced Research Projects Agency (DARPA). The idea was to integrate programmability (i.e., the ability to access network devices) into Internet devices (such as routers/switches). The innovation here was that packets were no longer treated as passive. Rather, they were treated active in the sense that they carry programs for how to process data packets. The goal was to allow applications to specify the desired requirements on a per packet, per flow, or per application basis.

Two different programming models were proposed: (1) integrated model (also called capsule model) and (2) discrete model (also called programmable router/switch model). In the integrated model, programs (containing specific instructions for how to process packets) are integrated in data packets (in-band), and then executed at each router/switch along the path. In the discrete model, programs are injected into routers/switches separately from actual data packets (i.e., through out-of-band mechanisms). In this model, users or network operators first inject programs into routers/switches along the path, and when a data packet arrives, its header is examined and an appropriate pre-injected program is loaded to process the data packet. Projects such as ANTS (Active Network Transfer System) [21] and smart packets [22] were based on the notion of the integrated model, while projects such as Netscripts [23] and SwitchWare [24] were based on the notion of the discrete model.

3. **OpenSig Approach**

   The OpenSig approach (appeared in 1999) addresses network programmability by providing a set of open interfaces and programming environments (e.g., CORBA, DCOM, java) in network devices (such as ATM switches, IP, MPLS routers). The original motivation behind OpenSig was that complex control architectures of network devices could be restructured according to a minimum set of layers where the services available in each layer are accessible through open interfaces. The objective was to give access to end-users or third party service providers to program or customize network devices to obtain a required service. Projects such as Tempest [25], Xbind [26] were based on the OpenSig Approach.

   Xbind [26] develops a platform to create, deploy, and manage multimedia services i.e., it develops mechanisms for network resource allocations, multiple vendor switch control, and broadband signaling. On the other hand, the tempest partitions ATM (Asynchronous Transfer Mode) switch's resources (such as certain range of ports, virtual path identifier range or virtual circuit identifier) into switchlets [27], and then each switchlet is controlled independently by different virtual network managers [25]. The advantage of the tempest framework was the ability to execute diverse control architectures (using virtual network managers) over the same physical ATM network.

The SPC approach was proposed for telephone networks, while the Active Networking and OpenSig approaches were proposed for the Internet. In comparison to the OpenSig approach, the active networking approach adds more flexibility to service creation, but increases more complexity to programmable networks. Both the approaches (active networking and OpenSig approaches) neither gathered critical mass nor transferred to widespread use in the Internet. Therefore, these approaches were not successful to make the Internet programmable. There were three main reasons of this failure: (1) no standard interface for communication between the control and data plane, (2) no attention to practical issues like performance (such as overhead on the network), complexity, scalability, and security, and (3) no real interest from service providers and operators to use them on their infrastructures (may be due to the lack of an immediate compelling problem) [28].

1.3.2 Recent SDN initiatives

It is already stated that there was not much interest from service providers and operators for integration of OpenSig and Active Networking approaches over the Internet. However, in the early 2000s, the Internet experienced major changes in networking because new technologies (such as ADSL) emerged, providing high speed Internet access to users. At that time it was easier than before for a user to afford an Internet connection which could be used for all kinds of daily activities such as e-mail, exchange of large files, and multimedia activities. This mass adoption of the high-speed Internet resulted into launch of a significant number of applications/services over the Internet. Service providers and network operators then started showing lots of interests for network innovation, performance, quality of service, and management functions (such as automatic configuration). This shifted the attention of service providers, operators, and research communities towards programmable networks once more. This was strengthened by the improvement (i.e., performance wise) of servers which could now run control plane software more efficiently [29].

Table 1.3: Recent SDN initiatives

	SDN initiatives	Projects or activities involved
1.	IETF Initiatives	ForCES [30], NETCONF [31], I2RS [32]
2.	Clean Slate Initiatives	4D [34], ETHANE [35], SANE [36], and OpenFlow [37]

The movement (i.e., from the early SDN to the current SDN) did not occur at once, but it went through a series of intermediate steps. In this section, we discuss these intermediate steps. We divide the recent work into two initiatives: (1) IETF initiatives and (2) Clean Slate Programs initiatives (Table 1.3).

1. **IETF Initiatives**

   One of the limitations of early SDN initiatives was that there was no standard interface between the control and data plane of network devices. The IETF ForCES (Forwarding and Control Element Separation) [30] addresses this limitation by clearly defining the interface. It defines a standard framework and mechanism for the exchange of information between the control plane functionality (called the Control Element) and implementation of the data plane (called the Forwarding Element). It describes several basic building blocks and their control and also allows easy extension. ForCES works on a Master/Slave basis, where the forwarding element is a slave and the control element is a master. ForCES has been undergoing standardization since 2003.

   NETCONF (network configuration protocol) provides methods to install, manipulate, and delete configurations of network devices. The functioning of NETCONF is realized as remote procedure calls (RPC). It uses XML-based methods for configuring a network. Additionally, the YANG data modeling language [33] has been developed for specifying NETCONF operations.

   I2RS (interface to the routing system) provides a standard interface to the routing system (or process) for real time or event driven interaction (read/write access) through a collection of control or management interfaces. One of the main goals of I2RS is to make the RIB of routers programmable.

2. **Clean Slate Initiatives**

   The clean slate initiatives are: 4D [34], Ethane [35], SANE [36], and OpenFlow [37]. A short description of each initiative is given below:

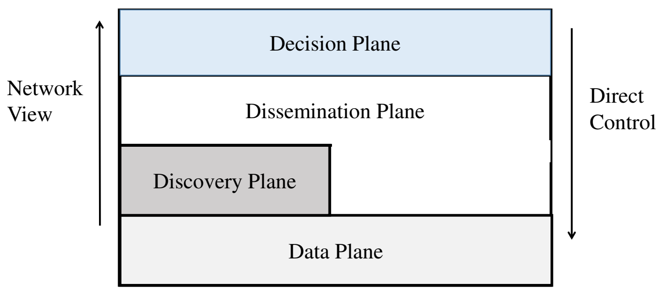


Figure 1.7: 4D clean slate architecture [source: [34]]

The 4D project envisioned the Internet architecture as four planes: (1) decision plane, (2) dissemination plane, (3) discovery plane, and (4) data plane (see Fig. 1.7). The decision plane is responsible for installing network configurations; the dissemination plane is responsible for providing information related to the view of the underlying network to the decision plane; the discovery plane allows network devices to discover the underlying topology; and the data plane is responsible for forwarding traffic.

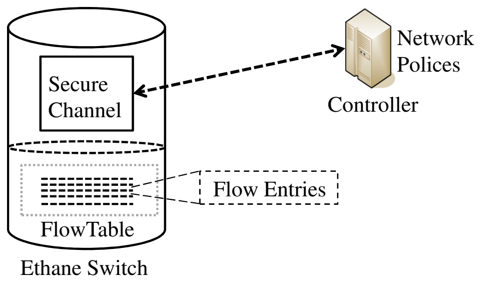


Figure 1.8: Ethane proposal

The Ethane project [35] (and its predecessor, SANE [36]) pursued the main ideas proposed in the 4D project for a centralized architecture and expanded it to incorporate security. In particular, the Ethane project proposed an architecture that contains two components: (1) An Ethane switch, which contains the FlowTable and a secure channel (see Fig. 1.8) and (2) the controller, which contains a set of policies to add in the Ethane switches through the secure channel. When a packet arrives at an Ethane switch, its packet-header is compared against the entries in the FlowTable. If a match is found, the action of that entry is performed. If no match is found, the packet is sent to the controller and the controller, thereafter, can add a new entry in the switch to forward packets.

The interesting fact about the Ethane project was that its switches could be deployed together with conventional Ethernet switches and without any modification to end hosts, allowing the widespread adoption of its architecture. The Ethane architecture was deployed at the campus of Stanford University in a period of a few months [35]. The Ethane project was very important, as the experiences gained by its design, implementation and deployment laid the foundation for what had thereafter become SDN (i.e., OpenFlow). In particular, Ethane is considered the immediate predecessor of OpenFlow, since the simple Ethane switches later became the basis of the OpenFlow concept. In fact, an OpenFlow switch is a generalized form of an Ethane's datapath switch. In addition, Ethane used a specific implementation of a controller (e.g., by routing flows securely). However, OpenFlow presents a more general implementation of a controller, which programs OpenFlow switches through the OpenFlow protocol.

OpenFlow was proposed because there was no practical way to experiment with new protocols in sufficiently realistic settings (e.g., at scale carrying real traffic) to gain the confidence needed for their widespread deployment. One important requirement for testing new protocols in production networks was a need for network programmability, which would simplify network management and service deployment, and would allow experimental and production networks to run simultaneously at the same infrastructure, each using a different set of forwarding rules. In this context, the OpenFlow protocol was proposed as a way for researchers to run experimental protocols in the network infrastructure they use everyday. The protocol provides mechanisms to program infrastructure devices using a set of primitives.

Although ForCES and OpenFlow follow the same principle i.e., defines a standard protocol for communication between the control and data plane), they are conceptually different. Some of these differences are described below:

1. In ForCES, the data-path element is represented by a set of logical functional blocks (LFB), each of which has a single specific function of processing packets. LFBs include a classifier and scheduler. Multiple LFBs in the data-path element are interconnected to form an LFB topology, which forms conceptual paths taken by packet flows within the data-path element. In contrast, the Openflow protocol does not expect any LFBs, but it expects the data-path element to have several tables (FlowTables).

2. ForCES does not define a new generic data plane function. However, OpenFlow defines a new generic data plane function by providing an extensive Flow-Match Header part.

3. A single LFB entry in ForCES can have only one action [30]. In order to perform multiple actions on packets, multiple LFBs are needed to be connected. However, in OpenFlow version 1.0, a single Flow Entry of the FlowTable can perform multiple actions on packets. From OpenFlow version 1.1, multiple actions can be performed using multiple tables.

ForCES fails to attract commercial applications, e.g., no mainstream router vendors have any motivation to adopt the concept. However, currently many OpenFlow commercial solutions (such as from NEC, HP) are available in the market. Due to strong support from industry, research, and academia, OpenFlow has been able to gather a widespread adoption. It is currently believed that

OpenFlow is the SDN de-facto standard [30]. Therefore, many researchers, vendors, and operators have formed ONF (Open Networking Foundation) to standardize the OpenFlow protocol.

1.4 Software Defined Networking using OpenFlow

OpenFlow is an SDN protocol that enables one or more entities (called controllers) in a network to interact with the data plane of network devices and to make adjustments, so that it can be adapted to meet the changing requirements. OpenFlow has been released in the form of specifications. The first two versions of the specifications (i.e., v1.0 and v1.1) were released by Stanford University in 2009 and 2011 respectively. However, since the third version (v1.2), ONF [6] has been releasing the next versions of OpenFlow. The current major OpenFlow version is 1.5. In this section, the Stanford and ONF design for OpenFlow are first presented, followed by an introduction to OpenFlow and its functionalities. We then provide a brief overview of extensions available in different OpenFlow versions. Furthermore, a brief description about OpenFlow switches (hardware and software) and controllers is presented.

1.4.1 Network design for OpenFlow

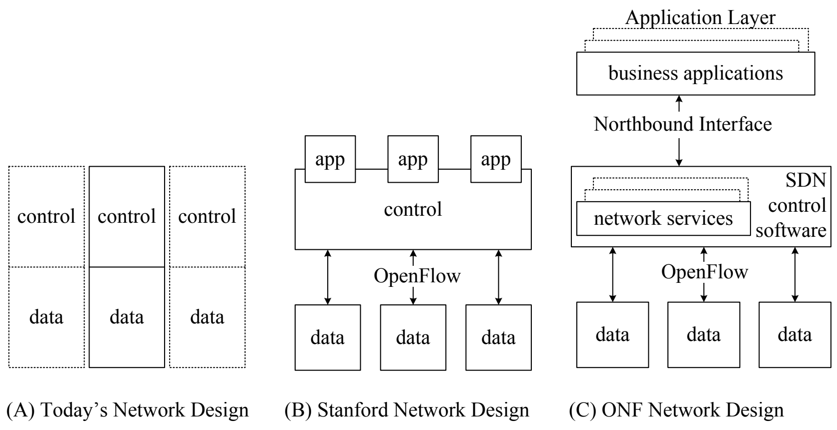


Figure 1.9: Design of today's and future networks

Fig. 1.9A shows the design of the today's network in which control and data plane layers are integrated in each device. Fig. 1.9B depicts the design proposed by Stanford University for future networks in which the control plane layer is decoupled from the data plane layer. In this design, the control plane layer is located in an external entity (the controller) and then the external entity communicates with the data plane layer through the OpenFlow protocol. Applications (which may be network or service related) are the pieces of software that are coupled in the controller. These applications can introduce new features in networks such as security, quality of service, forwarding schemes, and configurations.

Fig. 1.9C presents the design proposed by ONF. It extends the Stanford design by placing applications in a separate entity (i.e., the application layer) and the application layer then communicates with the controller through the northbound interface (see Fig. 1.9C). The application layer can receive a global and an abstracted view of the network from the controller and can use this information to provide appropriate instructions to the control plane layer to perform specific actions (such as security and quality-of-service) in the data plane layer.

1.4.2 Introduction to OpenFlow and its functionalities

OpenFlow networks consist of four components (Fig. 1.10): (1) data plane, (2) secure channel, (3) OpenFlow protocol, and (4) control plane. A description of all these components is given below:

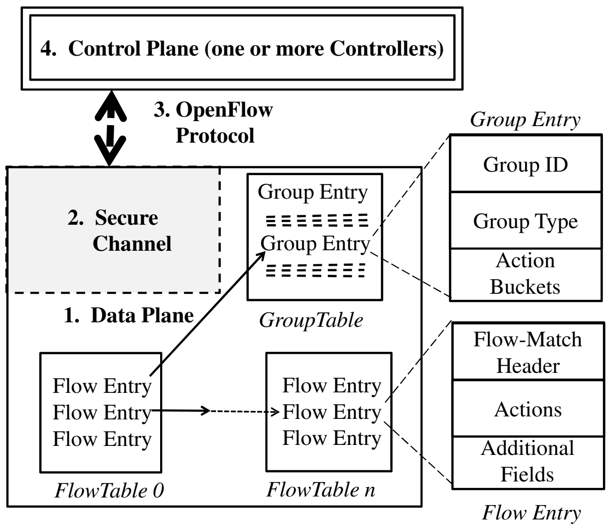


Figure 1.10: OpenFlow overview [source: OpenFlow specification 1.1]

1. **Data Plane**

   The data plane consists of FlowTables and the GroupTable (see Fig. 1.10). In the first version of OpenFlow (v1.0), only a single FlowTable can be present in a switch/router. However, in later versions, multiple FlowTables and a GroupTable can also be present in the switch/router. Using the OpenFlow protocol, both FlowTables and the GroupTable can be programmed via the controller.

The idea of the FlowTable is based on the fact that most modern routers/switches (network devices) contain a proprietary FIB (Forwarding Information Base) which is implemented in the forwarding hardware using TCAMs (Ternary Content Addressable Memory). OpenFlow provides an abstraction of the FIB by proposing FlowTables. A FlowTable is an extended version of the router FIB, which introduces extensible flow matching (i.e., matching on MAC, IP, transport layer, and many other fields) and actions for flows in networks.

*=wildcards

Flow-Match Header part							Actions	Additional fields		
Ingress port	Dst MAC	Src MAC	Src IP	Dst IP	Src port	Dst port	Output Port	Priority	statistics	
1	*	*	*	1.2.3.4	*	*	Port:2	100	325	
1	*	*	*	1.2.3.*	*	*	Port:3	101	123	

Figure 1.11: An example of a FlowTable

An entry in a FlowTable contains: (1) Flow-Match Header, which defines a flow, (2) actions, which define how a matched packet should be forwarded (i.e., forward to an output port or drop it) and (3) some additional fields such as priority, and statistics (see Fig. 1.10 and Fig. 1.11). The entry can be an exact match entry or a wildcarded entry. In the exact match entries, all the matching fields are defined for a Flow Entry, and in the wildcarded entries, some of fields can be wildcarded and therefore, these fields will not be matched against the incoming packets. Fig. 1.11 illustrates two wildcarded Flow Entries. In the first entry, the source MAC address, destination MAC address, source IP address, and transport layer ports are wildcarded fields. In the second entry, the last eight bits of the destination IP address are also wildcarded. The actions as Port:2 and Port:3 in Fig. 1.11 mean that packets matching the Flow-Match header will be forwarded through the output port whose number is 2 and 3, respectively. The other fields, shown in the figure, are priority and statistics.

When a packet arrives at an OpenFlow switch, it is matched against the Flow-Match Header (wildcarded or exact match) of the entries in the FlowTable. If a match is found, the statistics of that entry are updated and the actions are performed. If two or more matches are found, the actions of the highest priority number entry are performed. If no match is found, the packet (a part thereof) is forwarded to the controller. Thereafter, the controller determines how the packet can be handled. It may return the packet to the switch indicating the forwarding port, or it may add a Flow Entry in the switch to forward the packet.

Starting from OpenFlow v1.1, switches can have multiple FlowTables (see Fig. 1.10) in their data plane. The multiple FlowTables in a switch/router are sequentially numbered, starting at 0 (see Fig. 1.10). When a packet arrives at an OpenFlow switch, it is first matched with the first FlowTable (i.e., Table 0). If the action of the matched Flow Entry is Table i ($i > 0$), matching is continued at Table i. Table i can then forward the matched packet to another FlowTable. Note that the action can only be a FlowTable number which is greater than its own FlowTable number.

In addition to the multiple tables, the GroupTable concept is proposed in OpenFlow v1.1. A switch can have at most one GroupTable. The GroupTable supports more complex forwarding actions such as multicast routing, and fast-failover. In our research work, we used the fast-failover forwarding actions to implement a fast failure recovery scheme in OpenFlow. In fact, the GroupTable consists of Group Entries, which contain: (1) a unique identifier (GroupID), (2) GroupType and (3) action buckets (see Fig. 1.10). Typically, a Flow Entry redirects a packet to the GroupTable. In this case, the action of the Flow Entry is GroupID. The packet is then forwarded according to the respective Group Entry. Depending on the GroupType in the Group Entry, complex actions specified in the action buckets are performed.

Using the OpenFlow version earlier than 1.4, only packet-switching devices (i.e., where matching takes place on the packet-header) can be controlled by the OpenFlow protocol. However, since version 1.4, the OpenFlow protocol can also control the optical switches in which matching can take place on signal attributes such as channel spacing, frequency etc.

2. **Secure Channel**

   The secure channel of an OpenFlow switch (See Fig. 1.10) connects the switch with the controller. It is responsible for establishing and terminating an OpenFlow session with the controller. Generally, it is secured by TLS (Transport Layer Security) based encryption, although an unencrypted transport layer connection is also allowed. If the transport layer connection is used for communication, OpenFlow messages are encapsulated on top of the transport layer header. The transport layer protocol for establishing an OpenFlow session can be: TCP, SCTP (Stream Control Transmission Protocol), or UDP. As switches and the controller need a reliable connection between each other, TCP or SCTP is preferred over UDP. In addition, as not all the platforms support SCTP, TCP is mostly used for establishing sessions over the channel.

3. **OpenFlow Protocol**

   We present the OpenFlow protocol by describing the message exchange between an OpenFlow switch and the controller. Each OpenFlow message starts with an OpenFlow header which contains the OpenFlow version number, type, and the length of the message.

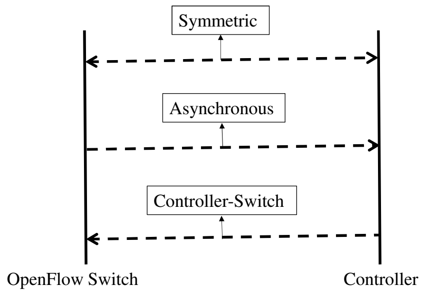


Figure 1.12: Categories of OpenFlow messages

   OpenFlow specifications divide messages into three categories: (1) Symmetric, (2) Asynchronous, and (3) Controller-Switch (Fig. 1.12). Symmetric means that the message can be sent in both directions (i.e., the "controller to switch" and the "switch to controller" direction), Asynchronous means that the message can be sent only in the "switch to controller" direction. Controller-switch means that the message can be sent only in the "controller to switch" direction. An overview of some of important OpenFlow messages which fall into these three categories is given in Table 1.4.

   The HELLO messages are exchanged after the secure channel (e.g., TCP channel) is established between the controller and the switch. These are exchanged to determine the version of OpenFlow supported by both sides. The ECHO messages are transmitted by either side to find that an OpenFlow session is still alive or not. The VENDOR messages are available for notifying vendor-specific extensions to the peer.

   The FEATURE_REQUEST message is sent by the controller to know the features (such as the number of tables and buffers) supported by the switch. In response to the FEATURE_REQUEST message, the switch transmits the FEATURE_REPLY message to the controller. Similarly, the controller

Table 1.4: Overview of some important OpenFlow Messages

Message Types	Category	Description
HELLO	Symmetric	To setup an OpenFlow session
ECHO_REQUEST	Symmetric	To check aliveness of the session
ECHO_REPLY	Symmetric	To reply ECHO_REQUEST
VENDOR	Symmetric	To notify vendor extensions
FEATURE_REQUEST	Controller-switch	To request features of switches
FEATURE_REPLY	Asynchronous	To reply features of switches
CONFIG_REQUEST	Controller-switch	To get configuration of a switch
CONFIG_REPLY	Asynchronous	To reply a CONFIG_REQUEST
PACKET_IN	Asynchronous	To send a packet to the controller
FLOW_REMOVED	Asynchronous	To notify flow removal
PORT_STATUS	Asynchronous	To notify the port status
ERROR	Asynchronous	To notify an error
PACKET_OUT	Controller-switch	To send a packet from a switch
FLOW_MOD	Controller-switch	To add a Flow Entry in a switch
PORT_MOD	Controller-switch	To modify a port
STATS_REQUEST	Controller-switch	To request statistics
STATS_REPLY	Controller-switch	To reply a STATS_REQUEST
BARRIER_REQUEST	Controller-switch	To send a barrier request
BARRIER_REPLY	Controller-switch	To reply a BARRIER_REQUEST

sends the CONFIG_REQEST message to the switch to know configuration parameters (such as the datapath id and the port numbers) and the switch replies then with the CONFIG_REPLY message.

Using a PACKET_IN message, the switch sends a data packet to the controller, when it does not have a matched Flow Entry for the packet. Control traffic is also sent to the controller via this message. With the FLOW_REMOVED message, the switch informs the controller that a Flow Entry is removed from the FlowTable, while using the PORT_STATUS message, the switch notifies the changes in the port status. Finally, the switch uses ERROR messages to notify the controller about errors occurred in processing OpenFlow messages.

The controller uses PACKET_OUT messages to send packets (data or control packets) to the switch for forwarding out through the data plane. Using the FLOW_MOD message, the controller adds/modifies/deletes Flow Entries in the switch. The PORT_MOD message is used to modify the status of an OpenFlow port.

With the STATS message pair, the controller obtains statistics (counters) from the switch while the BARRIER messages ensure that the particular OpenFlow commands from the controller have finished executing on the switch. When the switch receives the BARRIER_REQUEST message, it first completes execution of all the commands received prior to the BARRIER_REQUEST before executing any commands received after it. The switch then notifies the controller via the BARRIER_REPLY message.

4. **Control Plane**

   In an OpenFlow network, the controller (or a cluster of redundant controllers) implements the control-plane i.e, discovering a network topology and external end hosts (or adjacent network devices), computing forwarding entries, and installing them into network devices using the OpenFlow protocol.

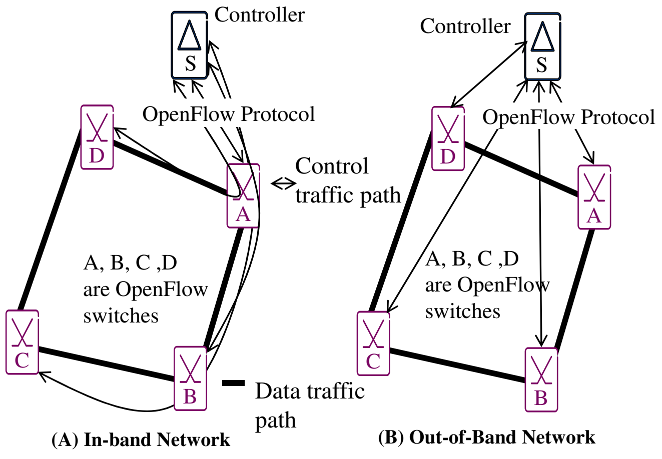


Figure 1.13: OpenFlow networks: In-band and Out-of-band. In in-band networks, switches use the same network for transmitting data traffic and control traffic. In out-of-band networks, switches use a separate network for transmitting control traffic

The controller connection with the switches can be out-of-band or in-band (See Fig. 1.13). In the case of an in-band connection, control messages are sent on the same channel used for transporting data traffic, whereas in the case of an out-of-band connection, control messages are sent on a different channel. As shown in Fig. 1.13, in the in-band connection, switches B, C and D share the same channel for transporting both control and data traffic, and in the out-of-band connection, switches use a different channel (a separate network) for transporting control and data traffic. Implementing an out-of-band connection is simple because the controller has a separate network to communicate with each switch in a data network. However, implementing an in-band control connection is complex, since switches have to search and establish a path to the controller (bootstrapping) through other switches in the data network. In our research, we propose a bootstrapping algorithm with which switches in an in-band network establish an OpenFlow session with the controller without having any manual configurations.

1.4.3 Extensions in different OpenFlow versions

For enabling widespread deployment in production and carrier environments, new OpenFlow versions provide additional functionalities (compared to described in the previous subsections). In this section, we describe some of these functionalities (such as queuing support, OpenFlow meters support, multiple controller support, and auxiliary connections support), which are used in (or related to) this PhD research.

From version 1.0, OpenFlow switches have the queuing support. OpenFlow v1.0 and v1.1 support queues with guaranteed minimum rates, while OpenFlow v1.2 and higher versions support both minimum and maximum rates for a given queue. However, creation of queues is out-of-scope for the OpenFlow protocol. With the OpenFlow protocol, a packet can be redirected through an already created queue. Although the OpenFlow protocol does not support creating or modifying queues, an OpenFlow capable switch can be queried to report queues attached to a specific port, and to report the guaranteed rate. In fact, separate protocols such as the OpenFlow management and configuration protocol (OF-CONFIG) [38] and the Open vSwitch Database Management Protocol (OVS-DB) [39] have been proposed to create queues in OpenFlow switches. In this PhD research work, for the quality-of-service work, we used the OVS-DB protocol to create queues in OpenFlow switches. Additionally, we proposed vendor-specific extensions to create queues through the OpenFlow protocol.

From version 1.3, OpenFlow supports additional mechanisms (i.e., using meters) to implement quality of service techniques. Unlike queues which cannot be created by the OpenFlow protocol, meters can be added in switches (in a Flow Entry) through the OpenFlow protocol. To add meters, an OpenFlow switch maintains the meter table. The meter table contains meter entries and each entry contains: (1) meter identifier, which is a 32 bit unique number to identify a meter, (2) meter band, which specifies the rate of band and the way in which incoming packets should be processed, and (3) counters, which are incremented when a packet is processed by the meter. Meters can be attached directly to a Flow Entry. An action of a Flow Entry can a meter identifier and thereby, it enables the Flow Entry to send a matched packet to the meter table. As an entry can contain multiple actions, the meter action is applied first, i.e., before any other actions [40].

From version 1.2, OpenFlow has multiple controllers support. Using this support, an OpenFlow switch can establish sessions with multiple controllers at a time. Currently, three roles are specified for controllers: (1) Equal, (2) Master, and (3) Slave. The default role of a controller is Equal. In this role, a controller has full access to OpenFlow switches, i.e., it can receive all the switch asynchronous messages, send controller-to-switch commands, and also send/receive symmetric messages. A controller can request its role to be changed to Slave. In this role, the controller has read-only access to switches. The default for this role is not to receive asynchronous messages, apart from port status messages. In addition, in this role, the controller is not permitted to execute controller-to-switch commands. Furthermore, a controller can request its role to be changed to Master. This role is similar to Equal and therefore, in this role, the controller has full access to OpenFlow switches. The difference between the Master and Equal role is that there can have multiple controllers with the Equal role, but there can be only one master controller in a network. In addition, only the Master controller can allow an Equal role controller to be changed to the Slave role or vice versa.

Traditionally, an OpenFlow channel between a switch and a controller consists of a single connection. However, from OpenFlow v1.3, to boost the performance of the OpenFlow channel, additional auxiliary connections between the controller and the switch can be established. Each auxiliary connection must use the same controller IP address as the main connection. However, transport protocols other than the main connection can be used for an auxiliary connection. Prior to establishing any auxiliary connections, the switch must first establish its main connection. If the main connection fails, all of the auxiliary connections, which have been established for the main connection, need to be taken down immediately.

1.4.4 OpenFlow capable hardware switches

Hardware implementations of OpenFlow devices hold the promise of operating the devices much faster than their software counterparts (soft devices). To understand how OpenFlow data plane components such as FlowTables and Flow Entries can be implemented into hardware switches, we briefly discuss the functional block diagram of some of today's hardware OpenFlow devices.

Fig. 1.14 depicts the functional block diagram. It shows matching in both software and hardware. In fact, many switches such as HP and NEC switches contain FlowTables in software as well as in hardware. Usually, the software table contains the full set of Flow Entries, while the hardware table contains a subset of the Flow Entries. When a packet arrives at a switch (Input Arbiter in Fig. 1.14), the packet is stored in the input queue and the packet-header is first extracted by the header extractor and then the packet header is matched against all the entries

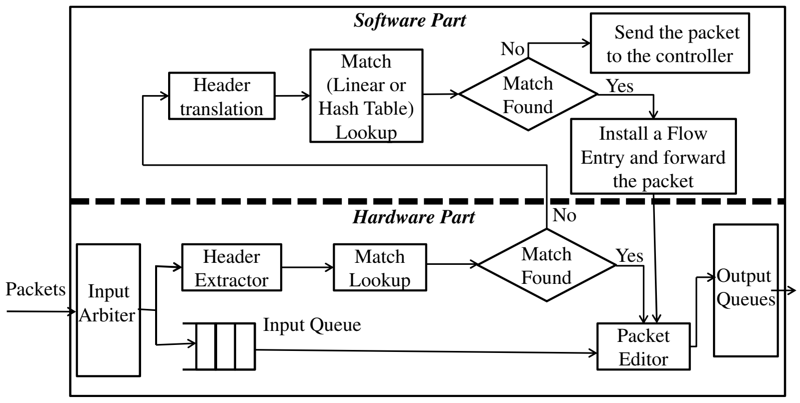


Figure 1.14: Functional block diagram of OpenFlow hardware switches

(TCAM or SRAM) present in hardware. If a match is found, the Packet-Editor forwards the packet to the output port/queue from the input queue. If no match is found, the packet is forwarded to software which translates again the packet into headers. If a matching entry is found in software, the entry is installed in the hardware FlowTable and the packet is forwarded through the Packet-Editor (Fig. 1.14). Otherwise, it is forwarded to the controller to define its action.

1.4.5 OpenFlow capable soft switches

Soft switches are software packages that can emulate a switch on a PC. The following soft switches are capable of emulating OpenFlow:

1. **Stanford reference implementation**

   The reference implementation is the first software release of an Openflow switch. This was implemented by Stanford University. The first version of the implementation could be run in kernel-space as well as in user-space. However, later versions can only be run in user-space.

2. **Open vSwitch**

   Open vSwitch is a production quality implementation of an SDN switch. It is designed to enable massive network emulation, while still supporting standard management interfaces and protocols (such as OpenFlow and Open vSwitch Database management protocol). Open vSwitch is targeted at multiple server virtualization deployments. Open vSwitch can be run in kernel mode as well as in user-space mode. The kernel space implementation contains FlowTables in both user-space and kernel space. The user-space table contains a full set of Flow Entries, while the kernel-space table contains a subset of the Flow Entries. When a packet arrives in the switch, it is first matched against all the entries present in the kernel. If no match is found, the packet is forwarded to user-space. If a matching entry is found in user-space, the matched entry is installed in kernel-space and the packet is forwarded. Otherwise, it is forwarded to the controller to define its action. The kernel space implementation of Open vSwitch emulates the implementation of many OpenFlow hardware switches such as NEC and HP switches. Currently, Open vSwitch supports many OpenFlow versions: v1.0, v1.1, v1.2, v1.3, and v1.4.

3. **Ericsson and CPqD OpenFlow switch**

   These switches contain different versions of OpenFlow, v1.1, v1.2, v1.3, and v1.4. The version 1.1 was implemented by Ericsson, while later versions were implemented by CPqD. These are user-space switch implementations of OpenFlow switches. The code is based on the original Stanford reference switch with the forwarding functionality being completely rewritten to support different OpenFlow versions.

1.4.6 OpenFlow controllers

Over the years, many controllers were released for OpenFlow. Table 1.5 shows the NOX/POX, Beacon, Maestro, Trema, Ryu, Floodlight, OpenDayLight, and ONOS controllers. Among all the mentioned controllers, NOX and POX were the first controllers that contained the OpenFlow support. The latest controllers are OpenDayLight and ONOS, which have the capability to support carrier-grade services in OpenFlow.

Table 1.5: Different OpenFlow controllers. OFv is the version of OpenFlow supported

Controller	Language	OFv	Platform
NOX/POX [41]	Python/c++	1.0/1.3	Linux
Beacon [42]	Java	1.3	Window/Linux/Mac
Maestro [43]	Java	1.0	Window/Linux/Mac
Trema [44]	Ruby,c	1.0	Linux
Ryu [45]	Python	1.0 - 1.4	Linux
Floodlight [46]	Java	1.0/1.3	Linux
OpenDayLight [47]	Java	1.0/1.3	Linux
ONOS [48]	Java	1.0/1.3	Linux

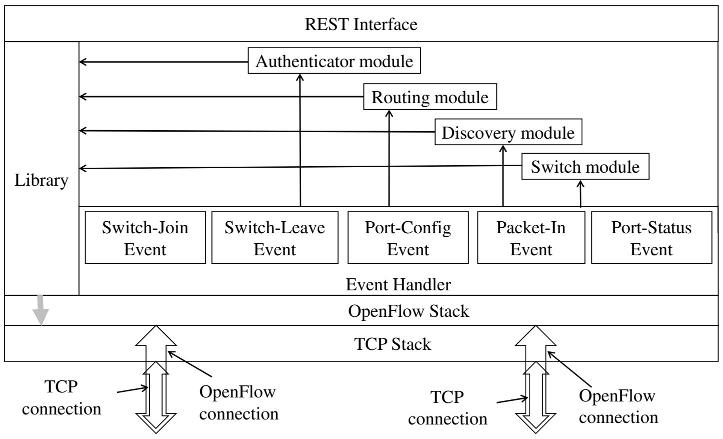


Figure 1.15: Basic controller architecture

The basic architecture of the controllers is given in Fig. 1.15. Fig. 1.15 illustrates that the controller may contain many different modules, such as authentication, discovery, routing, and switch modules. Using the authentication module, end hosts can be authenticated. Using the discovery module, the underlying topology of OpenFlow switches can be discovered. For discovering a topology, the controller sends LLDP (Link Layer Discovery Protocol) packets to be transmitted from an OpenFlow switch to its neighboring switches. When a neighboring switch receives an LLDP packet, it sends the packet back to the controller. From the received LLDP messages, the controller detects a link between switches and hence, detects a topology. Using the routing module, routing paths (shortest) can be calculated for a certain source and destination pair. This module uses the topology gathered using the discovery module to calculate a path between a source and destination.

Using the switch module, MAC learning can be performed to forward traffic. MAC learning operates by maintaining a mapping between the MAC (Media Access Control) addresses and the physical ports of OpenFlow switches by which the destinations can be reached. These mappings are learned by monitoring the source addresses of incoming packets. When a packet is received, the module updates or adds the source MAC address and incoming port in its MAC table. Besides this, it matches the destination address of packet with the addresses available in the MAC table. If the address matches, it adds a Flow Entry in a FlowTable of the corresponding OpenFlow switch so that the packet can be forwarded via the port, defined in the mapping of the MAC table. Otherwise, if the destination is a broadcast, multi-cast, or unknown uni-cast, the controller sends an OpenFlow packet to the OpenFlow switch to flood the corresponding data packet out of all ports. All these different modules are listed in [41]. Some controllers also contain a REST interface to communicate with an external application. Currently, controllers such as Floodlight, Ryu, OpenDayLight, and ONOS implement the REST interface.

To implement a module, the controllers have a library and event handler. With the library, modules can create and send OpenFlow messages to switches. With the event handler, different modules (see Fig. 1.15) receive events generated by the OpenFlow stack. The events, which are mostly used by modules, are: (1) Switch-join, which is generated when a switch establishes an OpenFlow session with the controller, (2) Switch-leave, which is generated when a switch disconnects from the controller, (3) Port-config, which is generated when the controller receives all port information, (4) Packet-in, which is generated when a packet is received to decide its forwarding action, (5) Port-status, which is generated when an LOS is detected or repaired in one of the ports in a switch.

1.5 Research challenges and possible solutions

In this section, we discuss the research challenges that are considered in this PhD research.

1.5.1 Fast failure recovery

When a failure occurs, network devices need to re-route traffic from an affected path to an alternative path. The problem with OpenFlow is that network devices depend on the centralized controller to establish a path. Hence, until the controller identifies a failed link (or node) and updates forwarding entries (Flow Entries) in all the relevant switches, packets traveling on the affected path will be dropped. Moreover, if the controller itself fails, it cannot establish routes (flows) in the network. Hence, one of the challenges of OpenFlow is to provide fast failure recovery in its networks.

Carrier-grade networks have a strict requirement that the network should recover from a failure within 50 ms. In fact, there is a service level agreement between the business customer and a service provider to deliver a reliable service. If requirements are not met, these are compensated for the loss of service. Therefore, carrier-grade networks typically implement two well-known fast failure recovery mechanisms – restoration and protection – in their networks. In the case of restoration, recovery paths can be either pre-planned or dynamically allocated, but the resources required by the recovery paths are not allocated until a failure occurs. Hence, when the failure occurs, additional signaling is needed to establish the restoration path. In the case of protection, recovery paths are always pre-planned and reserved before the failure occurs. Hence, when the failure occurs, no additional signaling is needed to establish the protection path and affected traffic can be immediately redirected. Protection is therefore a proactive mechanism and restoration is a reactive mechanism. There are different protection mechanisms applied in carrier grade networks. These are: $1 + 1$, $1 : 1$, $1 : N$ ($N > 1$) and $M : N$ [51].

In $1+1$ protection, one protection path is established exactly for protecting one working path, and traffic is permanently duplicated at the ingress node on both the paths. In $1 : 1$ protection, like 1+1 protection, one protection path is established for one working path, however, traffic is transmitted over only one path (working or protection) at a time. This leaves the opportunity to transport extra traffic along the protection path in failure-free conditions. In $1 : N$ protection, one protection path is dedicated for protecting N working paths. However, $M : N$ protection is the extension to $1 : N$ protection, where a set of M recovery paths protects a set of up to N working paths.

In fact, there can be three different domains in OpenFlow in which failures can happen:

1. Data plane domain, where a network device or a link between network devices fails.

2. Control-plane domain, where a connection between a network device and the controller fails.

3. Controller domain, where the controller itself fails.

Failure recovery is important in all the domains because a failure can cause a disruption of a service or prevent new service establishment in the network. For the data and control plane domain, OpenFlow may rely on the restoration and protection mechanisms (discussed above) to recover from the failure. However, for the controller domain, OpenFlow may rely on other mechanisms such as backup controller's mechanisms [52]. Hence, when a connection between one controller and a network device is lost, the network device may rely on the backup controller to take actions.

The problem with restoration in OpenFlow is that it puts a considerable load on the controller momentarily after the failure. This is because the controller has to reconfigure all affected flows in the network and therefore, has to send lots of messages to network devices to update Forwarding Entries. As restoration may take significant time to complete recovery activities, OpenFlow can implement protection to meet the carrier-grade requirement. Protection removes the need of network devices to contact the controller for reconfiguring affected flows. This is accomplished by pre-establishing the protection paths. The challenge here is that protection needs a method to redirect traffic to an alternative path without contacting the controller, when the failure occurs. In addition, OpenFlow may need to run additional protocols (such as Bi-directional Forwarding Detection) in network devices to detect failures.

1.5.2 Verification of data plane functionality

Although OpenFlow decouples the control plane functionality from the data plane functionality of network devices, verification of the data plane functionality for errors (such as configuration errors, software or hardware bugs) is one of the time consuming and challenging tasks in OpenFlow. This is because the data plane functionality contains two complex parts: (1) Flow-Match Header part and (2) action part. The Flow-Match header part can match different packet-headers, while the action part can implement very complex forwarding actions such as multipath routing, fast-failover, and flooding.

An action fault occurs when packets matching a Flow-Match header are processed incorrectly via the action part of the Flow Entries. Action faults can be due to software or hardware failures, congestion, mis-wiring, etc. Currently, automatic test packet generation (ATPG) mechanism [53] is proposed to verify a Flow Entry for action faults.

A Flow-Match Header fault occurs when matching of a packet with the Flow-Match Header gives an incorrect result (i.e., a packet gets matched with the Flow-Match Header which it should not, or a packet does not get a match although a matching Flow-Entry is present in the FlowTable). The problem with the verification of the Flow-Match Header is that the header space of the Flow-Match header is very large. As the header-space is increasing in each new OpenFlow version, probability of having matching errors is also increasing. In OpenFlow v1.0, flow matching can take place on the ingress port, Ethernet, IPv4, and transport layer headers. However, in the later versions, matching can take place on many additional fields (such as MPLS headers and IPv6 headers). For verification, we may need to verify matching of all these fields with different packet-headers. Additionally, if a Flow Entry contains wildcards in any of the matching-header fields, all the flows, which can match with the wildcarded Flow-Match Header, are required to be verified for flow-matching issues. Moreover, in OpenFlow v1.0 there is just one FlowTable, and from version 1.1, there can be a maximum of 255 FlowTables in a switch. Therefore, verification of the Flow-Match Header faults may require the verification of flow matching in each of these tables.

According to ONF, currently there are OpenFlow switches available from 26 different companies, and the switches from different companies differ substantially in both the data plane and the controller-switch interactions [49] [50]. The switches such as Quanta, HP, NEC, Ocedo, and Pica8 Pronto switches contain

FlowTables in hardware as well as in software. The Quanta switch [49] uses the FlowTable of software for packet forwarding only when the FlowTable of hardware is full. However, HP, NEC, and Ocedo switches use both software and hardware FlowTables for packet forwarding. The software table contains a full set of Flow Entries, and the hardware table contains a subset of the Flow Entries. When a packet arrives at a switch, it is first matched against all the entries present in hardware. If no match is found, the packet is forwarded for matching in software. Therefore, we may need to verify matching of packets in software as well as in hardware.

Currently, many tools are already proposed to verify OpenFlow switches. Some of these tools are FLOVER [54], FlowChecker [55], HSA (Header-Space analysis) [56], VeriFlow [57] and Anteater [58]. The challenge with all these tools is that these find errors by just analyzing the configuration of networks. However, finding all software or hardware errors is difficult without sending a packet in the network. Therefore, network operators have to debug manually by sending test packets (e.g., using ping) in the network. Manual debugging takes significant time in finding issues.

It is already stated that ATPG is proposed to find action faults. This tool automatically transmits test packets in the network to find errors. However, the problem is that ATPG is not able to find all the errors that can be present in the large header space of the Flow-Match Header. Therefore, in this dissertation, we propose a method which can verify the Flow-Match header of the Flow-Entries for flow-matching issues (see the next section).

1.5.3 Bootstrapping

Bootstrapping involves the mechanisms that bring a system from an initial state to an operational state. In this phase, a network device behaves like an end-host rather than as a forwarding device, i.e., the device cannot forward any packets yet. The challenge is that OpenFlow specifications do not yet describe that how OpenFlow devices can be automatically bootstrapped. Without performing these tasks automatically, an operator (or an engineer) may face a lot of manual configurations such as going to the field to perform initial configuration tasks (i.e., configuration of the network device's IP address and the controller IP address).

Bootstrapping by manual configurations is a time consuming task. In addition, manual configurations can result into bugs. Therefore, in traditional networks, operators typically use automatic ways to configure their devices. One of the most commonly used automatic configuration protocols in these networks is DHCP (Dynamic Host Configuration Protocol) [59]). For running the DHCP, the DHCP server is placed at a location (e.g., at a central location), which is accessible from network devices. Network devices then run DHCP clients and exchange messages with the DHCP server. The DHCP server then configures bootstrapping parameters (such as IP address) in devices.

Automatic configuration protocols (such as DHCP), which are used in traditional networks, can also be used in OpenFlow networks to configure bootstrapping parameters. For an out-of-band network, using the DHCP can be similar to a traditional network, as the DHCP server can be located at the controller (central location) and each OpenFlow device can run the DHCP client. As a separate network is present in the out-of-band network for the communication between OpenFlow devices and the controller, the DHCP server is able to exchange messages with all devices, and therefore, able to configure bootstrapping parameters.

The challenge is that the automatic configuration protocols, which are used in traditional networks, cannot be applied straightforward in an in-band OpenFlow network, since network devices cannot directly communicate with the controller. They need to search and establish a path to the controller (or the DHCP server) through the other switches in the network (See Fig. 1.13). Therefore, in this dissertation, we proposed a method with which the controller establishes its own control network to communicate with network devices (see the next section).

1.5.4 Quality of Service

The commercial challenge of the growth of the Internet is represented in Fig. 1.16. Web companies (webcos) such as Skype, Google, Netflix, Akamai, and Facebook use the internet infrastructure to transmit traffic to users, although there are limits to service level commitments they receive. The telecommunication (telco) companies, who invest massively in the infrastructure, continue to see a decline in the average revenue per user (ARPU) as ever decreasing revenue goes to the telco from users and the webcos/CDNs (content delivery network). Yet revenue and margins increase for the webcos. Currently, traffic from webcos is expected to keep increasing, as webcos launch more and more applications which are sent over-the-top (OTT) of telco networks. The problem here is that telcos are not currently interested in investing on additional infrastructure capacity to provide the required bandwidth for these applications without an adequate return on network capital employed.

Currently, the Internet works on a best effort basis (i.e., all bits on the Internet are treated equally today). The lack of differentiation on the Internet has caused significant business challenges for telcos. As differentiation is against net neutrality, it is difficult to introduce differentiation over the Internet. Net neutrality means that no bit of information should be prioritized over another on the Internet. This is a complicated regulatory issue, and a full discussion is out of scope of this research work. In [60], reasonable network management practices following net

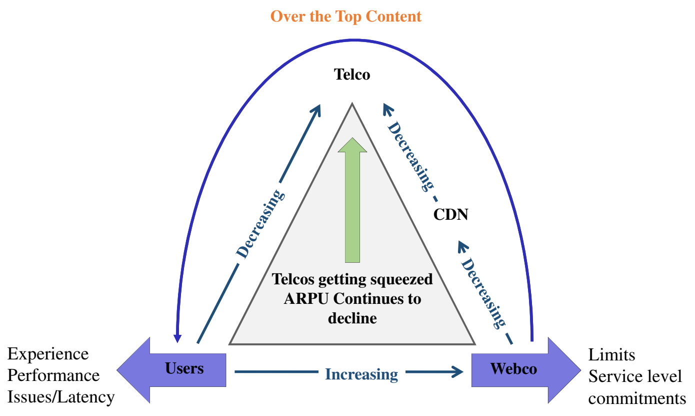

Over the Top Content
Telco
Decreasing
Decreasing
CDN
Decreasing
Telcos getting squeezed ARPU Continues to decline
Experience Performance Issues/Latency
Users
Increasing
Webco
Limits Service level commitments

Figure 1.16: The commercial challenge of the growth of the Internet [1]

neutrality have been introduced (or extended) for the Internet. These reasonable network management practices include:

1. Transparency: Broadband providers must disclose the network management practices, performance characteristics, and terms and conditions of their broadband services.

2. No blocking: Broadband providers are not allowed to block lawful content and applications that compete with their voice or video telephony services.

3. No unreasonable discrimination: Broadband providers must not unreasonably discriminate in transmitting lawful network traffic.

Currently, many quality of service (QOS) mechanisms, such as Diffserv (Differentiated Services) and IntServ (Integrated Services), are available for the Internet. IntServ has a scalability problem, as it is based on individual flows and DiffServ alleviates this problem by providing QoS based on aggregated flows. Currently, these mechanisms need a model which is based on the requirements of the reasonable network management practices.

As SDN/OpenFlow is considered to be one of the Future Internet Technologies, it would be beneficial to propose such a QoS model for the Internet that supports SDN/OpenFlow in its networks.

1.5.5 Loss-free packet switching

For decades network engineers have been trained to think in terms of macroscopic bandwidth, when scaling their networks. Macroscopic here means that the bandwidth is provisioned by simply adding a capacity that is equal to the average of the incoming rate of different data transfers. However, traffic such as media traffic is bursty in nature. When such a type of traffic is sent over a network, packets are sent in micro-bursts at the maximum speed, which may lead to congestion, as many bursts may be transmitted at the same time. Hence, provisioning of bandwidth by averaging the incoming rate of different data transfers is no longer valid or even meaningful for a network that is loaded with bursty traffic.

Over-provisioning is not a solution for this problem. This leads to continuously increasing investment costs in the network, and to even more inefficient usage of resources. Over-provisioning only helps to reduce or even mask a part of the risk and it does not guarantee zero packet-loss in the network. Currently, large-size buffers are also provided in devices of the network to decrease the packet-loss. The problem is that this can cause unnecessary delay and may give poor performance (buffer-bloat problem [61]).

To increase the reliability of data transport, many protocols are used in packet switched networks. One of such protocols is TCP. TCP provides reliable data transfer by having the destination, sending back acknowledgments to the source to signal proper in-order delivery of data packets. The transport of these acknowledgments back to the source takes a certain amount of time caused by the latency of the physical transport in networks. This latency is a consequence of the limitations in the transport speed, (mostly) related/limited to the speed of light. If the source has to wait for these acknowledgments to send the next data packets of media flows, links will be idle for a part of the time. Several mechanisms have also been developed to overcome these situations, such as increasing the TCP window-sizing. However, stretching these parameters to compensate fully for the long length of a link could compromise the reliability of the mechanism or create additional overhead in resending large portions of data in case of data loss.

Other protocols or standards to decrease the packet-loss are: (1) IEEE 802.17 (Resilient Packet Ring), and (2) IEEE 802.1Qbb (Priority-based Flow Control). In these standards a control packet (feedback packets in IEEE 802.17 and PAUSE frames in IEEE 802.1Qbb) is sent back to the sender to notify about congestion. The issue here is that if there is a large distance between the sender and congested node, the latter may suffer buffer overflow and packet-loss can happen. Currently, QoS mechanisms such as DiffServ is also proposed to deliver quality of service to high priority traffic. The problem is that DiffServ can only work for (small) fractions of high priority traffic. Additionally, it does not guarantee zero packet-loss in the network.

Currently, no solution is available, which can guarantee zero packet-loss in packet-switched networks in all network scenarios. In this thesis, we propose a method which can guarantee zero packet loss, although nearly all the bandwidth is consumed in a network (see the next section).

1.6 Research contributions

In this dissertation, we focus on fast failure recovery, verification, bootstrapping, quality-of-service, and loss-free packet-switching mechanisms for SDN/OpenFlow (see Fig. 1.17). All the mechanisms except the loss-free packet-switching mechanism are tested using SDN/OpenFlow. The loss-free packet-switching mechanism is tested using traditional network technologies. However, it is equally applicable to SDN/OpenFlow network architectures. In this dissertation, each of the following chapters corresponds to a journal article (as-is) which is already in a published or submitted state.

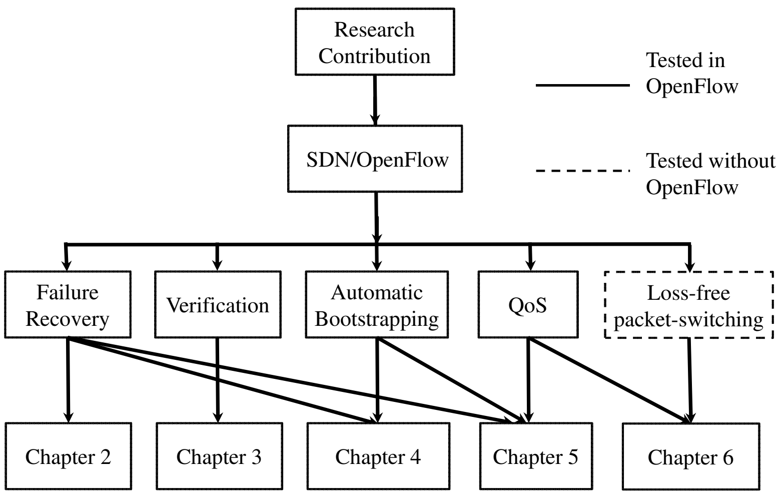


Figure 1.17: Overview of work performed

In Chapter 2, fast-failure recovery mechanisms, restoration and 1:1 path protection, are studied for an out-of-band OpenFlow network to recover from data plane failures. Data plane failures considered are: link and node failures. In the out-of-band network, a failure in the data plane does not affect the communication between network devices and the controller. Therefore, the controller is used to restore data traffic in restoration. In the case of 1:1 path protection, the controller proactively establishes two disjoint paths - working and protection - for data traffic and when the ingress switch detects the failure, it redirects traffic to the protection path without contacting the controller. A wide range of experiments are performed to verify the suitability of these mechanisms in carrier-grade networks. The experiments are performed by using topologies which differ with the number of network devices and the degree of meshedness. Additionally, a scalability experiment is performed to measure the recovery time with an increase in the number of data flows in a network. With the restoration experiments, it has been concluded that Openflow can restore traffic, but its dependency on the centralized controller means that it will be hard to achieve 50 ms restoration in a large-scale network serving many flows. With the protection experiments, it has been concluded that protection is a way in OpenFlow to achieve carrier-grade recovery requirements, even in a large-scale network serving many flows.

In Chapter 3, verification mechanisms are proposed to find failures in the matching of incoming packets with the Flow-Entries in FlowTables. These failures can occur due to configuration issues and software or hardware failures in network devices. The objective of verification is to find the packet-headers which cannot be matched or can be matched incorrectly with the Flow-Match header of a Flow Entry. To verify matching, the mechanism transmits test packets in the network and therefore, it requires additional resources - (1) computational resources of the controller (to generate and transmit test packets) and (2) bandwidth resources between the controller and network devices (to transmit/receive test packets) - to be reserved in the network. To decrease these requirements, we consider a network in which servers (custom machines) can be attached to a network device to transmit or receive test packets. The experiments are performed with a wide range topologies and networks (such as in-band and out-of-band networks). With the experiments, it has been concluded that the verification time depends on the bandwidth available in the network for verification. If bandwidth is unlimited, verification can be achieved in a very short time interval. However, if bandwidth limitations exist, the verification time might increase significantly.

In Chapter 4, bootstrapping, queuing, and fast failure recovery mechanisms are proposed for in-band OpenFlow networks. With the bootstrapping mechanisms, an OpenFlow device can be bootstrapped automatically without having any manual configurations. This chapter gives a brief overview of the bootstrapping mechanism, while Appendix A gives a detail description. With the queuing mechanisms, the queuing support of OpenFlow is extended to include priority numbers in queues. Using these queues, different traffic (control and data traffic) can be served with different priorities on the same channel of an in-band network. For fast-failure recovery, restoration and protection techniques are proposed for control traffic, while utilizing the previously proposed restoration and protection mechanisms of the out-of-band network for data traffic (see Chapter 2 discussion). In addition, practical challenges of implementing these mechanisms in existing open-source OpenFlow packages are discussed in this chapter. Furthermore, we implement these mechanisms in one of the OpenFlow software packages and perform extensive experiments. The experiments with the bootstrapping mechanism conclude that the proposed mechanism allows bootstrapping in a minimal time, which makes it suitable even for a large network. The experiments with the queuing mechanism conclude that data traffic does not affect the communication between the controller and switches, although data traffic and control traffic are sent on the same channel. The experiments with failure recovery mechanism conclude that carrier-grade quality can be achieved in OpenFlow.

In Chapter 5, we provide a proposal for the introduction of a dynamic OpenFlow capability in the Internet by looking at the commercial challenge to the growth of the internet (see the previous section) and describing a new operational model for the internet to address this challenge. The proposed model is evaluated using multi-autonomous system experiments in distinct reference network-scenarios for a city with a population of 1 million inhabitants, emulating xDSL (Digital Subscriber Line), LTE (Long-Term Evolution) and Fiber networking scenarios. Quality of service is established in this model by inserting QoS queues in paths discovered by routing protocols. For running routing protocols in OpenFlow, we also propose an automatic configuration framework. This framework is explained in detail in Appendix B. The proposed operational model running QoS mechanisms is tested in extensive emulation environments. The obtained results considering both control and data traffic scalability confirm the suitability of the proposed model for multiple autonomous systems scenarios of the Internet. In one of the experiments, failure recovery is also considered for QoS flows in the network. Regarding failure recovery, we did not focus on fast-failure recovery (Chapter 2 and Chapter 4) but instead, we focus on scenarios in which high-priority traffic should always get a higher precedence over best-effort traffic, even after a failure. Appendix C provides a detailed description of our proposed QoS framework for failure recovery.

In Chapter 6, we propose a protocol, called inter-burst segregation protocol (IBSP), which can guarantee zero packet-loss in packet-switched networks. IBSP guarantees zero packet-loss by controlling burstiness in each node of a network. For controlling burstiness, IBSP enforces separation between different bursts of sources at each node. We perform simulations to verify the proof-of-concept of IBSP and perform the emulations on a high performance platform (i.e., DPDK) to check the suitability of the protocol in a real environment. The experimental results confirm that IBSP guarantees zero packet-loss, although nearly all the bandwidth is consumed in the network. In addition, the jitter using IBSP is low.

Chapter 7 presents the main conclusions of this dissertation. It summarizes important messages and presents the future research directions.

1.7 Publications

The research results obtained during this PhD research have been published in scientific journals and presented at a series of international conferences. The following list provides an overview of the publications during my PhD research.

1.7.1 Publications in international journals (listed in the Science Citation Index [2])

1. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *OpenFlow: Meeting Carrier-Grade Recovery Requirements*, Computer Communications, Vol. 36(6), March 2013, pp. 656-665.

2. Marc Sune, Leonardo Bergesio, Hagen Woesner, Tom Rothe, Andreas Kopsel, Didier Colle, Bart Puype, Dimitra Simeonidou, Reza Nejabati, Mayur Channegowda, Mario Kind, Thomas Dietz, Achim Autenrieth, Vasileios Kotronis, Elio Salvadori, Stefano Salsano, Marc Korner, **Sachin Sharma**, *Design and Implementation of the OFELIA FP7 Facility*, Computer Networks, Vol. 61, March 2013, pp. 132-150.

3. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *Automatic configuration of routing control platforms in OpenFlow networks*, ACM SIGCOMM Computer Communication Review, Vol. 43(6), October 2013, pp. 491-492.

4. Mark Berman, Piet Demeester, Jae Woo Lee, Kiran Nagaraja, Michael Zink, Didier Colle, Dilip Kumar Krishnappa, Dipankar Raychaudhuri, Henning Schulzrinne, Ivan Seskar, **Sachin Sharma**, *Future Internets Escape the Simulator*, ACM CACM Magazine, Vol. 58(6), June 2015, pp. 78-89.

5. **Sachin Sharma**, Wouter Tavernier, Sahel Sahhaf, Didier Colle, Mario Pickavet, Piet Demeester, *Verification of Flow Matching Functionality in the Forwarding Plane of OpenFlow Networks*, IEICE Transactions on Communications, Vol. E98B (11), November 2015, pp. 2190-2201.

6. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *In-band control, queuing, and failure recovery functionalities for OpenFlow*, IEEE Network, Vol. 30(1), January 2016, pp. 106-112.

7. **Sachin Sharma**, David Palma, Joao Goncalves, Dimitri Staessens, Nick Johnson, Charaka Palansuriya, Ricardo Figueiredo, Luis Cordeiro, Donal Morris, Adam Carter, Rob Baxter, Didier Colle, *CityFlow, Enabling Quality of Service in the Internet: Opportunities, Challenges, and Experimentation*, Computer Networks, November 2015 (under revision).

8. **Sachin Sharma**, Didier Colle, Wouter Tavernier, Mario Pickavet, Piet Demeester, *Inter-burst Segregation Protocol guaranteeing loss-free packet*

[2]The publications listed are recognized as 'A1 publications', according to the following definition used by Ghent University: A1 publications are articles listed in the Science Citation Index Expanded, the Social Science Citation Index or the Arts and Humanities Citation Index of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper.

switched networks, IEEE Communications Letter, February 2016 (under revision).

1.7.2 Publications in international conferences (listed in the Science Citation Index [3])

9. Dimitri Staessens, **Sachin Sharma**, Didier Colle, Mario Pickavet, Piet Demeester, *Software Defined Networking: Meeting Carrier Grade Requirements*, 18th IEEE International Workshop on Local and Metropolitan networks (IEEE LANMAN), October 2011, pp. 1-6, North Carolina, USA.

10. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *Fast failure recovery for in-band OpenFlow networks*, 9th International Conference on Design of Reliable Communication Networks (DRCN), March 2013, pp. 44-51, Budapest, Hungry.

11. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *Automatic bootstrapping of OpenFlow networks*, 19th IEEE International Workshop on Local and Metropolitan networks (IEEE LANMAN), April 2013, pp. 1-6, Brussels, Belgium.

12. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *A demonstration of automatic bootstrapping of resilient OpenFlow networks*, IFIP/IEEE International Symposium on Integrated Network Management (IM), May 2013, pp. 1066-1067, Ghent, Belgium.

13. **Sachin Sharma**, Dimitri Staessens, Didier Colle, David Palma, Joao Goncalves, Mario Pickavet, Luis Cordeiro, Piet Demeester, *Demonstrating Resilient Quality of Service in Software Defined Networking*, IEEE Conference on Computer Communications Workshops (IEEE INFOCOM), April 2014, pp. 133-134, Toronto, Canada.

1.7.3 Publications in other international conferences

14. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *Enabling Fast Failure Recovery in OpenFlow Networks*, 8th International Workshop on the Design of Reliable Communication Networks (DRCN), October 2011, pp. 164-171, Krakow, Poland.

[3]The publications listed are recognized as ‘P1 publications’, according to the following definition used by Ghent University: P1 publications are proceedings listed in the Conference Proceedings Citation Index - Science or Conference Proceedings Citation Index - Social Science and Humanities of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper, except for publications that are classified as A1.

15. **Sachin Sharma**, Dimitri Staessens, Didier Colle, Mario Pickavet, Piet Demeester, *A Demonstration of Fast Failure Recovery in Software Defined Networking*, 8th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom), June 2012, pp. 411-414, Thessaloniki, Greece.

16. Sander Vrijders, Dimitri Staessens, **Sachin Sharma**, Pieter Simoens, Didier Colle and Mario Pickavet, *Optimization of data deployment in data centers using BitTorrent and OpenFlow*, Fire Engineering conference, November 2012, Ghent, Belgium.

17. **Sachin Sharma**, Dimitri Staessens, Didier Colle, David Palma, Joao Goncalves, Riccardo Figueiredo, Donal Morris, Mario Pickavet, and Piet Demeester, *Implementing quality of service for the software defined networking enabled future internet*, The European Workshop on Software Defined Networking (EWSDN), September 2014, pp. 49-54, Budapest, Hungry.

18. David Palma, Joao Goncalves, Bruno Sousa, Luis Cordeiro, Paulo Simoes, **Sachin Sharma**, and Dimitri Staessens, *The QueuePusher: enabling queue management in OpenFlow*, The European Workshop on Software Defined Networking (EWSDN), September 2014, pp. 125-126, Budapest, Hungry.

19. Adam Carter, Donal Morris, **Sachin Sharma**, Luis Cordeiro, Riccardo Figueiredo, Joao Gonalves, David Palma, Nick Johnson and Dimitri Staessens, *Cityflow: openflow city experiment Linking infrastructure and applications*, The European Workshop on Software Defined Networking (EWSDN), September 2014, pp. 129-130, Budapest, Hungry.

20. Joao Gonalves, David Palma, Luis Cordeiro, **Sachin Sharma**, Didier Colle, Adam Carter, Paulo Simoes, *Software-defined networking: guidelines for experimentation and validation in large-scale real world scenarios*, The conference on Artificial Intelligence Applications and Innovations (AIAI), September 2014, pp. 38-47, Rhodes, Greece.

21. Juhoon Kim, Catalin Meirosu, Ioanna Papafili, Rebecca Steinert, **Sachin Sharma**, Fritz-Joachim Westphal, Mario Kind, Apoorv Shukla, Felician Nemeth, Antonio Manzalini, *Service Provider DevOps for Large Scale Modern Network Services*, BDIM conference collocated with IM, pp.1391-1397, April 2015, Ottawa, Canada.

22. **Sachin Sharma**, Wouter Tavernier, Didier Colle, Mario Pickavet, Piet Demeester, *Verification of aggregated flows in OpenFlow networks*, IEEE Conference on Computer Communications Workshops (IEEE INFOCOM), April 2015, pp. 7-8, Hong Kong, China.

1.7.4 Publications in IETF Drafts

23. Catalin Meirosu, Antonio Manzalini, Juhoon Kim, Rebecca Steinert, **Sachin Sharma**, Guido Marchetto, Ioanna Papafili,, K. Pentikousis, S. Wright, *Service Provider DevOps for Software-Defined Telecom Infrastructures*, Work in Progress, IETF draft, 2015.

1.7.5 Other publications

24. Wolfgang John, Alisa Devlic, Zhemin Ding, David Jocha, Andras Kern, Mario Kind, Andreas Kpsel, Viktor Nordell, **Sachin Sharma**, Pontus Skldstrm, Dimitri Staessens, Attila Takacs, Steffen Topp, F. Westphal, Hagen Woesner, Andreas Gladisch, *Split Architecture for Large Scale Wide Area Networks*, arXiv preprint arXiv:1402.2228, February 2014.

25. Rebecca Steinert, Wolfgang John, Pontus Skldstrm, Bertrand Pechenot, Andrs Gulys, Istvn Pelle, Tams Lvai, Felicin Nmeth, Juhoon Kim, Catalin Meirosu, Xuejun Cai, Chunyan Fu, Kostas Pentikousis, **Sachin Sharma**, Ioanna Papafili, G. Marchetto, R. Sisto, F. Risso, P. Kreuger, J. Ekman, S. Liu, A. Manzalini, A. Shukla, S. Schmid, *Service Provider DevOps network capabilities and tools*, arXiv preprint arXiv: 1510.02818, October 2015.

26. Wolfgang John, Catalin Meirosu, Pontus Skldstrm, Felician Nemeth, Andras Gulyas, Mario Kind, **Sachin Sharma**, Ioanna Papafili, G. Agapiou, G. Marchetto, R. Sisto, R. Steinert, P. Kreuger, H. Abrahamsson, A. Manzalini, N. Sarrar, *Initial Service Provider DevOps concept, capabilities and proposed tools*, arXiv preprint arXiv:1510.02 220, October 2015.

1.7.6 Publications in national conferences

27. **Sachin Sharma**, Didier Colle, Mario Pickavet, *Failure recovery for OpenFlow networks*, 12th FEA PhD Symposium, December 2011, Ghent Belgium.

28. **Sachin Sharma**, Didier Colle, Mario Pickavet, *Enabling prioritization over the Internet*, 15th FEA PhD Symposium, December 2014, Ghent Belgium.

References

[1] Martn Casado, Teemu Koponen, Scott Shenker, Amin Tootoonchian, *Fabric: A Retrospective on Evolving SDN*, The first workshop on Hot Topics in Software Defined Networks (HotSDN), pp. 85-90, 2012.

[2] GENI Project [Online]. Available: https://www.geni.net/.

[3] SPARC Project [Online]. Available: http://www.fp7-sparc.eu/.

[4] OFELIA Project [Online]. Available: http://www.fp7-ofelia.eu/.

[5] UNIFY Project [Online]. Available: http://www.fp7-unify.eu/.

[6] ONF [Online]. Available: https://www.opennetworking.org/, 2012.

[7] Jennifer Rexford, Future Internet Architecture: Clean-Slate Versus Evolutionary Research, Communications of the ACM, Vol. 53(9), pp. 36-40, 2010.

[8] *ITU-T Recommendation X.200: Data Networks and Open Systems Communications: Open Systems Interconnection-model and notation*, 1994.

[9] Kurose, J. F. and Ross, K. W., *Computer Networking: A top-down approach featuring the Internet*, Reading: Addison-Wesley, 2001.

[10] Quagga [Online]. Available: http://www.nongnu.org/quagga/.

[11] D. Shah, P. Gupta, *Fast Updating Algorithms for TCAMs*, IEEE Macro, Vol. 21(1), pp. 36–47, 2001.

[12] H. T. Kung, *Gigabit Local Area Networks: A systems perspective*, IEEE Communications Magazine, pp. 79-89, 1992.

[13] P. Winzer, *Beyond 100G Ethernet*, IEEE Communications Magazine, Vol. 48(7), pp. 26-30, 2010.

[14] Nielsen's Law of Internet Bandwidth [Online]. Available: https://www.nngroup.com/articles/law-of-bandwidth/.

[15] Mathieu Tahon, Marlies Van der Wee, Sofie Verbrugge, Didier Colle, Mario Pickavet, *The Impact of Inter-platform. Competition on the Economic Viability of Municipal Fiber Networks*, OFC, 2014.

[16] Al-Fares, A. Loukissas, and A. Vahdat, *A scalable, commodity data center network architecture*, In ACM SIGCOMM Computer Communication Review, volume 38, pp. 6374, 2008.

[17] NEC SDN Fabric [Online]. Available: http://www.necam.com/SDN/.

[18] Sheinbein, D. and Weber, R.P., *Stored Program Controlled Network: 800 Service using SPC network capability*, The Bell System Technical Journal, Vol. 61(7), pp. 1737-1744, 1982.

[19] Van der Merwe, J., Cepleanu, A., D'Souza, K., Freeman, B., Greenberg, A., Knight, D., McMillan, R., Moloney, D., Mulligan, J., Nguyen, H., Nguyen, M., Ramarajan, A., Saad, S., Satterlee, M., Spencer, T., Toll, D., Zelingher, S., *Dynamic Connectivity Management with an Intelligent Route Service Control Point*, Proceedings of the SIGCOMM Workshop on Internet Network Management, pp. 29-34, 2006.

[20] J. T. Moore and Scott M. Nettles, *Towards practical programmable packets*, 20th IEEE Conf. on Computer Commun. (INFOCOM), pp. 41-50, 2001.

[21] David Wetherall, John Guttag and David Tennenhouse, *ANTS: Network Services without the Red Tape*, IEEE Computer, Vol. 32(4), 42-49, 1999.

[22] Schwartz, B., Jackson, A.W., Strayer, W.T., Wenyi Zhou, Rockwell, R.D., Partridge, C, *Smart Packets for active networks*, IEEE Second Conference on Open Architectures and Network Programming, 90-97, 1999.

[23] Silva, S., Yemini, Y., Florissi, D., *The NetScript active network system*, IEEE Journal on Selected Areas in Communications, Vol. 19(3), pp. 538-551, 2001.

[24] Alexander, D.S., Arbaugh, W.A., Keromytis, A.D. and Smith, J.M., *A secure active network environment architecture: realization in SwitchWare*, IEEE Network, Vol. 12(3), pp. 37-45, 1998.

[25] Van der Merwe, J.E., Rooney, S., Leslie, I., Crosby, S., *The Tempest-a practical framework for network programmability*, IEEE Network, Vol. 12(3), pp. 20-28, 1998.

[26] A. A. Lazar, K.-S. Lim, and F. Marconcini, *xbind: The system programmers manual*, tech. rep., Technical Report, 1996.

[27] Jaco E. van der Merwe and Ian M. Leslie, *Switchlets and dynamic virtual ATM networks*, the IFIP/IEEE International Symposium on Integrated Network Management (IM97), pp. 355368, 1997.

[28] Feamster, Nick, Rexford, Jennifer and Zegura, Ellen, *The Road to SDN*, ACM Queue, Vol. 11(12), 2013.

[29] Bo Han; Gopalakrishnan, V.; Lusheng Ji; Seungjoon Lee, *Network function virtualization: Challenges and opportunities for innovations*, in Communications Magazine, IEEE , vol.53, no.2, pp.90-97, Feb. 2015.

[30] A. Doria, J. Hadi Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, *Forwarding and Control Element Separation (ForCES) Protocol Specification*, Internet Engineering Task Force (IETF), RFC 5810, 2010.

[31] R. Enns, M. Bjorklund,J. Schoenwaelder, A. Bierman, *Network Configuration Protocol (NETCONF)*, IETF RFC 4741, 2011.

[32] A. Atlas, J. Halpern, S. Hares, D. Ward, T. Nadeau, *An Architecture for the Interface to the Routing System*, IETF Work in progress, 2015.

[33] M. Bjorklund, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, IETF RFC 6020, 2010.

[34] Greenberg Albert, Hjalmtysson Gisli, Maltz David A., Myers Andy, Rexford Jennifer, Xie, Geoffrey, Yan, Hong, Zhan, Jibin, Zhang Hui, *A Clean Slate 4D Approach to Network Control and Management*, ACM SIGCOMM Comput. Commun. Rev., Vol. 35(5), pp.41-54, 2005.

[35] Martin Casado , Michael J. Freedman , Justin Pettit , Jianying Luo , Nick McKeown , Scott Shenker, Ethane: taking control of the enterprise, ACM SIGCOMM Computer Communication Review, vol. 37(4), 2007.

[36] Martin Casado , Tal Garfinkel , Aditya Akella , Michael J. Freedman , Dan Boneh , Nick McKeown , Scott Shenker, *SANE: a protection architecture for enterprise networks*, Proceedings of the 15th conference on USENIX Security Symposium, 2006.

[37] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *OpenFlow: Enabling Innovation in Campus Networks*, SIGCOMM Comput. Commun. Rev., Vol. 38(2), pp. 69–74, 2008.

[38] Bansal et al., *Openflow management and configuration protocol*, 2014. [Online]. Available: https://www.opennetworking.org/sdnresources/onf -specifications/openflow-config.

[39] B Pfaff et al., *The open vswitch database management protocol*, IETF RFC 7047, Dec 2013.

[40] Open vSwitch Configurations [Online]. Available: http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt.

[41] NOX and POX repository [Online]. Available: https://github.com/noxrepo/.

[42] David Erickson, *The Beacon OpenFlow Controller*, HotSDN, pp. 2013.

[43] Maestro code Repository [Online]. Available: https://code.google.com/p/maestro-platform/.

[44] Trema OpenFlow controller [Online]. Available: https://trema.github.io/trema/.

[45] Ryu OpenFlow controller [Online]. Available: https://osrg.github.io/ryu/.

[46] Floodlight code Repository [Online]. Available: http://www.projectfloodlight.org/floodlight/.

[47] OpenDayLight code Repository [Online]. Available: https://www.opendaylight.org/.

[48] ONOS code Repository [Online]. Available: http://onosproject.org/.

[49] D. Y. Huang et. al., High-Fidelity Switch Models for Software-Defined Network Emulation, HotSDN, pp. 43-48, 2013.

[50] H. Pan et al., The FlowAdapter: enable flexible multi-table processing on legacy hardware, HotSDN, pp. 85-90, 2013.

[51] J. Vasseure, M. Pickavet, Piet Demeester, *Network Recovery - Protection and Restoration of Optical, SONET-SDH, IP, and MPLS*, Morgan Kaufmann Publishers, 2004.

[52] Benjamin J. van Asten, Niels L. M. van Adrichem, Fernando A. Kuipers, *Scalability and resilience of software-defined networking: an Overview*, arXiv:1408.6760, 2014.

[53] H. Zeng, P. Kazemian, G. Varghese and N. McKeown, *Automatic Test Packet Generation*, CoNEXT, pp. 241–252, 2012.

[54] S. Son, S. Shin, V. Yegneswaran, G. Gu, *Model checking invariant security properties in OpenFlow*, ICC, pp. 1974-1979, 2013.

[55] E. Al-Shaer and S. Al-Haj, *FlowChecker: configuration analysis and verification of federated openflow infrastructures*, SafeConfig, pp. 37–44, 2010.

[56] P. Kazemian, G. Varghese, N. McKeown, *Header Space Analysis: Static Checking for Networks*, NSDI, pp. 113–126, 2012.

[57] A. Khurshid, and W. Zhou, M. Caesar, Matthew, Godfrey, P. Brighten, *VeriFlow: Verifying Network-wide Invariants in Real Time*, HotSDN, pp. 49–54, 2012.

[58] M. Haohui, A. Khurshid, R. Agarwal, M. Caesar, P. Brighten Godfrey, S. T. King, *Debugging the Data Plane with Anteater*, ACM SIGCOMM, pp. 290–301, 2011.

[59] R. Droms, *Dynamic Host Configuration Protocol*, https://www.ietf.org/rfc/rfc2131.txt, RFC 2131, 1999.

[60] Federal Communications Commission FCC 10-201. [Online]. Available: https://apps.fcc.gov/edocs public/attachmatch/FCC-10-201A1.pdf.

[61] J. Gettys et. al., Bufferbloat: dark buffers in the internet, Communications of the ACM, Vol. 55(1), pp. 57–65, 2012.

[62] iLabt.iMinds [Online]. Available: http://ilabt.iminds.be/.

[63] Mininet Virtualization Platform [Online]. Available: http://mininet.org/.

[64] DPDK Platform [Online]. Available: http://dpdk.org/.

[65] NS3 [Online]. Available: https://www.nsnam.org/.

2 OpenFlow: Meeting carrier-grade recovery requirements

This chapter investigates fast-failure recovery techniques for meeting carrier-grade quality in an out-of-band OpenFlow network. Out-of-band means that there is a separate network for control traffic and for data traffic. To achieve carrier-grade quality, the network should be able to recover from a failure within 50 ms. Therefore, we apply two well-known failure recovery techniques, restoration and protection, in OpenFlow. In the case of restoration, recovery paths are established after a failure occurs and in the case of protection, recovery paths are established before a failure occurs and hence, when the failure is detected, traffic is redirected to the recovery path. The chapter concludes that OpenFlow can restore traffic, but its dependency on the centralized controller means that it is hard to achieve 50 ms restoration in a large network serving many flows, so we need to remove this dependency and switch to protection to meet the carrier-grade recovery requirement.

⋆ ⋆ ⋆

Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester

Published in Computer Communications, March 2013 [1]

Abstract OpenFlow, initially launched as a technology-enabling network and application experimentation in a campus network, has a disruptive potential in designing a flexible network, fostering innovation, reducing complexity and delivering the right economics. This paper focuses on fault tolerance of OpenFlow to deploy it in carrier-grade networks. The carrier-grade network has a strict requirement that the network should recover from a failure within a 50 ms interval. We apply two well-known recovery mechanisms to OpenFlow networks: restoration and protection, and run extensive emulation experiments. In OpenFlow, the controlling software is moved to one or more servers (controllers) which can control many switches. For fast failure recovery, the controller must notify all the affected switches about the recovery action within a ms interval. This leads to a significant load on the controller. We show that OpenFlow may not be able to achieve failure recovery within a 50 ms interval in this situation. We add the recovery action in the switches themselves so that the switches can do recovery without contacting the controller. We show that this approach can achieve recovery within 50 ms in a large-scale network serving many flows.

Keywords

Carrier-grade; OpenFlow; Protection; Restoration

2.1 Introduction

There is almost no practical way to experiment with new protocols in sufficiently realistic settings (e.g., at the scale carrying real traffic) to gain the confidence needed for their widespread deployment. In this context, OpenFlow [1] has caught attention of many researchers and router vendors. It is developed in a clean-slate future internet program by Stanford University, which aims to offer a programmable network to test new protocols in current Internet platforms. If operators want to program the behavior of networking devices such as routers or switches, they require direct programming of the forwarding hardware. The core idea of OpenFlow is to provide direct programming of a router or switch to monitor and modify the way in which the individual packets are handled by the device. It is based on the fact that most modern routers/switches contain a proprietary FIB

[1]Compared to the publication in Computer Communications, additional section "Related work" (Section 2.6), footnote 2, and references [22], [23], [24], [25] and [26] are added to the chapter.

(Forwarding Information Base) which is implemented in the forwarding hardware using TCAMs (Ternary Content Addressable Memory). OpenFlow provides the concept of a FlowTable that is an abstraction of the FIB. In addition to this, it provides a protocol to program the FIB via “adding/deleting/modifying” entries in the FlowTable. This is achieved by one or more separate devices (so-called controllers) that communicate with the OpenFlow switches via the OpenFlow protocol. The switch/router that exposes its FlowTable through the OpenFlow protocol is called an OpenFlow switch/router.

The standard switch/router consists of interlinked elements that handle forwarding of packets (data plane) as well as controlling of forwarding (control plane). The control plane of these switches/routers implements almost all the standard protocols. However, only few of the protocols are required and effectively used. This makes switches/routers complex, expensive, and difficult to extend with new functions. OpenFlow addresses these issues by separating the control and forwarding plane. An OpenFlow network has a centralized programming model, where one or more controllers manage the underlying switches. This design is highly flexible, since it is the controller that can decide what actions (forward or drop) have to be taken for the different packets and at the same time forwarding can be done in hardware. Furthermore, the new functions can also be deployed very easily by just changing the software of one or more controllers.

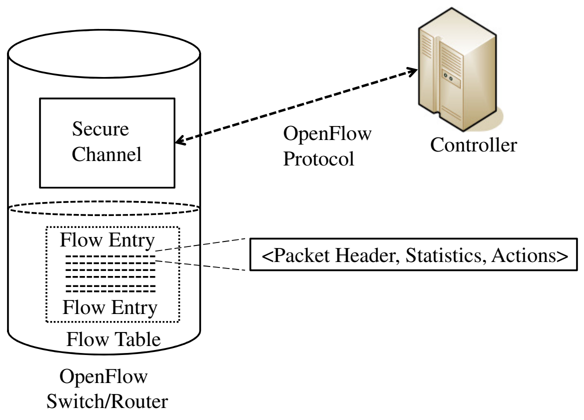


Figure 2.1: OpenFlow principle

An entry in the FlowTable consists of (1) a “packet header” that defines the flow, (2) “statistics” which keep track of matching packets per flow, and (3) “actions” which define how incoming packet should be processed. When a packet arrives at the OpenFlow switch, it is compared against the Flow Entries in the FlowTable. If a match is found, the actions of that entry are performed. If no match is found, the packet (a part thereof) is forwarded to the controller over the secure channel (shown in Figure 2.1). Thereafter, the controller can determine how the packet can be handled. It may return the packet to the switch indicating the forwarding port, or it may add the valid Flow Entries in the switches.

Large-scale experimental testbeds are available for researchers in the US through the GENI (Global Environment for Network Innovations) [2], in Japan through the JGN2plus research network [3], and in Europe through the OFELIA (OpenFlow in Europe: Linking Infrastructure and Applications) [4] projects. Several algorithms resulting from experiments on those testbeds have been deployed in many networks supporting various applications. Currently, industrial players such as Deutsche Telekom, Google, Microsoft, Verizon, and Yahoo have shown substantial interest towards OpenFlow and have formed ONF (Open Networking Foundation) [5] to standardize the OpenFlow protocol.

At present, one of the European projects named SPARC (SPlit ARchitecture Carrier-Grade Networks) [6] examines how carrier-grade networks can benefit from OpenFlow. Carrier-grade networks have a high capacity to support hundreds of thousands of customers and assume extremely high availability. Carrier-grade supports services such as cellular-telephone conversations, credit-card transactions that assume the availability of a reliable network. The requirement for the availability in commercial telecom networks is typically 99.999% [7]. The disruption of communication can suspend critical operations. In fact, there is a service level agreement between the business customer and a service provider to deliver a high-quality service. If a network operator is unable to meet the agreement, the service providers have to compensate for the loss of service. A failure recovery requirement in a carrier-grade network is sub-50 ms [8], which implies that the failure should be detected and the traffic should be recovered within a 50 ms interval.

Failure recovery in OpenFlow requires modification and addition of the Flow Entries in the OpenFlow switches. This paper studies fault tolerance of OpenFlow networks in large-scale carrier-grade networks. We implement two well-known mechanisms of failure recovery i.e., restoration and protection, in OpenFlow networks. In the case of restoration, alternative paths are not established until a failure occurs. In the case of protection, alternative paths are reserved before the failure occurs in the network. The controller in restoration must notify all the affected switches about a recovery action immediately. This leads to a significant load on the controller that delays the recovery action within the switches. We show that OpenFlow because of its centralized architecture may not be able to achieve failure recovery within a 50 ms interval in this situation. For protection, we add the recovery action in the switches themselves so that the switches can do recovery without contacting the controller. We perform an extensive emulation of our recovery mechanisms in different network scenarios and test OpenFlow with increased number of flows. We show that protection can achieve recovery within

50 ms in a large-scale network serving many flows.

The rest of the paper is organized as follows: Section 2.2 presents network resiliency, Section 2.3 describes the emulation environment, Section 2.4 presents results, Section 2.5 describes additional considerations, Section 2.6 describes the related work and finally Section 2.7 concludes.

2.2 Network resiliency

Network resilience is the ability to provide and maintain an acceptable level of service in presence of failures. We first describe general mechanisms that are used in carrier-grade networks to recover from failures. Then, we describe the integration of those mechanisms in OpenFlow in order to fulfill the network availability requirements of carrier-grade networks.

The recovery mechanisms [10, 11] that are used in carrier-grade networks are divided into two categories: restoration and protection. In the case of restoration, recovery paths can be either pre-planned or dynamically allocated, but resources required by the recovery paths are not allocated until a failure occurs. Thus, when the failure occurs, additional signaling is required to establish the restoration path. However, in the case of protection, recovery paths are always pre-planned and reserved before a failure occurs. Hence, when the failure occurs, no additional signaling is needed to establish the protection path and traffic can immediately be redirected. In segment protection, an end-to-end working path is divided into segments, each of which is protected by a unique backup segment. However, in path protection, a complete end-to-end working path is protected by a unique backup path. There are different types of protection schemes available for carrier grade networks. These are: 1+1, 1:1, 1:N and M:N protection [11].

Resilience is achieved in carrier-grade networks by first designing a network topology with failures in mind in order to provide alternate paths. The next step is adding the ability to detect the failures and react to them using a proper recovery mechanism. Loss of Signal (LOS) and Bidirectional Forwarding Detection (BFD) [9] are widely used to detect failures in carrier-grade networks. LOS can detect a failure in one particular port of the forwarding device, whereas BFD can detect failures in the path between any two forwarding devices. BFD is a simple Hello protocol that in many aspects is similar to the detection components of many routing protocols like OSPF (Open Shortest Path First). A pair of systems (end-to-end devices) transmits BFD packets periodically between each other, and if a system stops receiving BFD packets, the path between neighboring systems is assumed to have failed.

2.2.1 Resilience for an OpenFlow Network

A failure can be detected in OpenFlow by a Loss of Signal. It causes an OpenFlow port to change to the "down" state from the "up" state. An OpenFlow port is the port bounded to an OpenFlow switch to transmit and receive packets. This mechanism (i.e., LOS) detects only link-local failures and may be used in restoration. However, for path protection, end-to-end failure detection in any path in forwarding switches is required.

Recovery in OpenFlow can be done in essentially two different ways. One approach is to support the recovery mechanisms of a specific implemented protocol like MPLS (Multi Protocol Label Switching Protocol) [12, 13] into OpenFlow. The other approach is to build resilience, supporting recovery of arbitrary flows, regardless of the type of traffic they carry. We explore this option for OpenFlow networks.

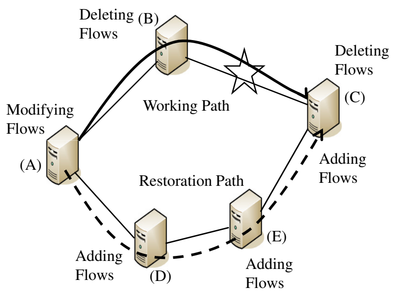


Figure 2.2: Restoration mechanism

Fast restoration in OpenFlow can be implemented in the controller. It requires an immediate action of the controller after a notification of a change in a link status (i.e., LOS). Failure recovery can be performed by removing affected Flow Entries and installing new entries in the affected switches as fast as possible following the failure notification [14, 15]. The restoration mechanism can be seen in Figure 2.2 which consists of OpenFlow switches A, B, C, D and E. Assuming the controller knows the network topology, we can calculate a path from a source node to a destination node. In Figure 2.2, the controller first installs path $< ABC >$ by adding the Flow Entries in the switches A, B and C. Once the controller receives the failure-notification message of link BC, it calculates a new path, i.e., $< ADEC >$. For OpenFlow switch A, as the flow in the Flow Entry for the working path ($< ABC >$) and the restoration path ($< ADEC >$) is identical but the action is different (i.e. to forward to switch B or D), the controller modifies the Flow Entry at A. In addition, for the restoration path ($< ADEC >$), there are no Flow Entries installed in the switches D, E, and C related to this flow, the controller must add these entries in the respective switches. The Flow Entries in

C for the working path ($< ABC >$) and the restoration path ($< ADEC >$) are different, since the incoming port is assumed to be a part of the matching header in Flow Entries. Once all the affected Flow Entries are updated/installed in all the switches, the flow is recovered from a failure. After the immediate action of restoration, the controller can clean up the other switches by deleting the Flow Entries at B and C related to the older path ($< ABC >$).

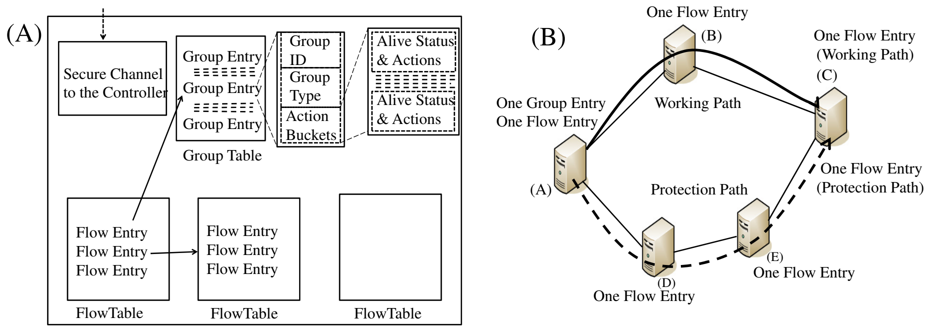


Figure 2.3: (A) Group Table Concept (B) Protection Mechanism

In the time between failure detection and completion of restoration, data packets may be lost. In order to further reduce the packet loss resulting from delay in the restoration action, we can turn to protection. Protection removes the need of OpenFlow switches to contact the controller for modification and addition of the Flow Entries to establish the alternative path. This can be accomplished by pre-computing the protection path and establishing it together with the working path. In 1+1 protection, traffic is duplicated at both the protected and working path, and in 1:1 protection, traffic is transmitted to the protection path upon a failure at the working path. Protection allows fast recovery, but requires a large FlowTable.

To implement a protection scheme, we applied the Group Table concept specified for OpenFlow in its version 1.1 [16]. Unlike the FlowTable, the GroupTable consists of Group Entries that in turn contain a number of actions. To execute any specific entry in the GroupTable, a Flow Entry forwards packets to a Group Entry having a specific group ID. Each Group Entry consists of the group ID (must be unique), a group type and a number of action buckets (shown in Figure 2.3A). An action bucket consists of an alive status (e.g., watch port and watch group in OpenFlow version 1.1 [16]) and a set of actions that are to be executed based on the value of the associated alive status. OpenFlow introduces the fast-failover group type [16] in order to perform fast failover without needing to involve the controller. This group type is important for our protection mechanism. Any group entry of this type consists of two or more action buckets with a well-defined order. A bucket is considered alive if its associated alive status is within a specific range (i.e., watch port or watch group is not equal to 0xffffffffL). The first action bucket describes what to do with a packet under normal conditions. If this action bucket has been declared as unavailable that is due to change in status of a bucket (i.e, 0xffffffffL), the packet is treated according to the next available bucket. The status of the bucket can be changed by the monitored port going into the "down" state or through other mechanisms such as BFD.

In our 1:1 path protection mechanism, the above Group Table concept of OpenFlow is used without any modification. We used BFD to detect failures. Once BFD declares a failure in the working path, the action bucket associated with this path in the GroupTable is made unavailable by changing the value of the alive status.

1:1 path protection can be seen in Figure 2.3B. When a packet arrives at the ingress OpenFlow switch (A), the controller installs two disjoint paths: one in $< ABC >$ and the other in $< ADEC >$. The ingress OpenFlow switch (A) is the switch that actually needs to take a switching action on a failure condition, i.e., to send a packet to B during the normal condition and to send a packet to D during the failure condition. For this particular flow of the packet, the Group Table concept can be applied at the ingress OpenFlow switch (A). The Group Entry in this switch may contain two action buckets: one for output port B and the other for output port D. In addition, one entry is added in the FlowTable of switch A, which points matched packets to the above Group Entry in the GroupTable. For the other switches, B and C for the working path, and D, E and C for the protection path, only one Flow Entry is added. Thus in our case, switch C contains two Flow Entries: one for the working path $< ABC >$ and the other for the protection path $< ADEC >$. Once a failure is detected by BFD in the working path, the ingress OpenFlow switch (A) changes the alive status of the first bucket in the Group Entry. Thus, the action related to the next bucket, whose output port is D, can be taken. As the Flow Entries in D, E and C related to the $< ADEC >$ path are already present, there is no need to install these in the respective switches upon the failure.

2.3 Emulation environment

In [14], we have shown that restoration meets the 50 ms recovery requirement when there are two flows in a small network. However, as carrier-grade networks are typically large, serving many flows, we emulated restoration and protection on increasing number of flows in a large-scale European network. The emulated European topologies, testbed and methodology are described in this section.

2.3.1 Emulation testbed and topologies

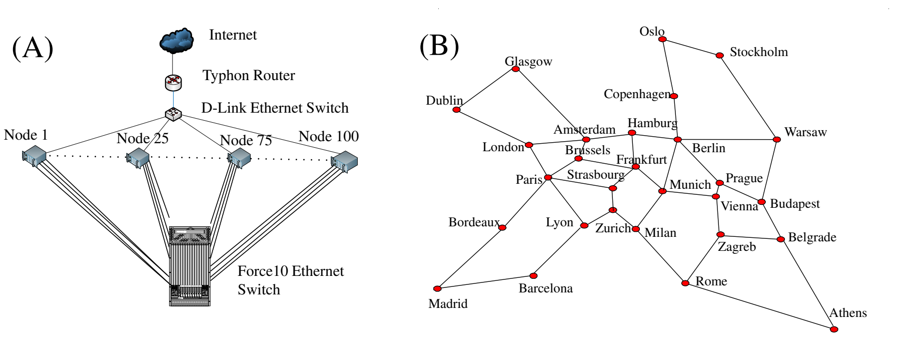


Figure 2.4: (A) Virtual wall testbed (B) BT topology

Our virtual-wall emulation testbed is a generic test environment (based on emulab [17]) for advanced network, distributive software and service evaluation. It consists of 100 physical nodes (dual processor, dual core servers and up to six 1 Gb/s interfaces per node) interconnected by a non-blocking 1.5 Tb/s Force10 Ethernet switch (shown in Figure 2.4A). It uses the concept of Virtual LAN (VLAN) to build arbitrary topologies. The nodes in our testbed may be used for network emulation (bandwidth, delay, packet loss) of a real network environment.

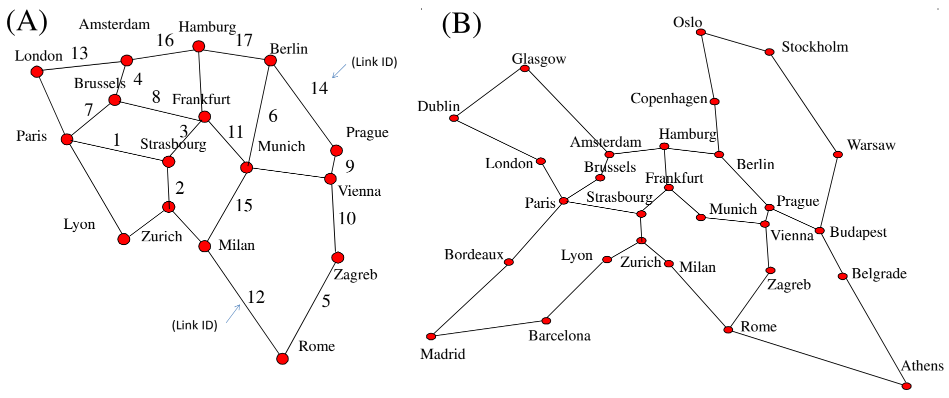


Figure 2.5: (A) CT topology (B) RT topology

We emulated an extensive failure recovery experiment using the topologies that were developed within the COST 266 action project. In this project, a basic reference topology (BT topology in Figure 2.4B) and variations of the BT topology (e.g., Figure 2.5), suited for a pan-European network, were designed. The variations of the BT topology, Core Topology (CT), Large Topology (LT), Ring Topology (RT) and Triangular Topology (TT), were obtained by varying the total number of nodes and the degree of meshedness. The CT, and LT differ with respect to the number of nodes. The BT consists of 28 nodes, the CT consists of 16 nodes, and the LT consists of 37 nodes. The other derived topologies contained the same number of nodes as the BT, but the difference lies in the degree of meshedness. The topology called Ring topology (RT), Figure 2.5B, is much sparser than the BT. The details of these topologies can also be found in [19]. We used the BT, CT and RT topologies for our experiments.

In our experiments, each node of the considered COST 266 topologies acted as an OpenFlow switch. In our emulation, we connected a server to each OpenFlow switch (not shown in Figure 2.4B and 2.5). We built these topologies in our virtual-wall testbed. A virtual-wall node (shown in Figure 2.4A) acted as an OpenFlow switch as well a server in our emulation. In our emulation, one CPU core of a virtual-wall node has been assigned to an OpenFlow switch and another CPU core has been assigned to a server. Each of the OpenFlow switches has also been provided a dedicated interface to a switched Ethernet LAN, which establishes a connection to the single controller, i.e., out-of-band connection. Out-of-band means that there is a separate channel for the control and data plane i.e. a failure in the data plane does not affect communication between the switches and the controller.

2.3.2 Emulation methodology

There are many extensions for the OpenFlow protocol. These extensions have been released publicly in the form of OpenFlow versions. In April 2012, OpenFlow version 1.3 has been released by ONF. In addition, many OpenFlow controllers are also available for controlling OpenFlow networks. These are NOX, Beacon, Onix, Helios and Maestro. We implemented restoration and protection in the NOX controller and in the OpenFlow version 1.1 (developed by Ericsson [18]) and used these for our emulations.

To evaluate the recovery time, each server generated packets to all other servers using the Linux kernel module PKTGEN. Each server sent packets to all the other servers at the constant interval of 6 ms. The size of the PKTGEN packets was 64 bytes. We manually configured the routing table in each server to transmit the packets to the OpenFlow network.

For failure detection in protection, each working path was monitored by adding an additional BFD flow in the OpenFlow switches. BFD transmits a failure notification message when its session breaks. To receive the failure notification message, a virtual link (the link between veth1 and veth2 in Figure 2.6) has been

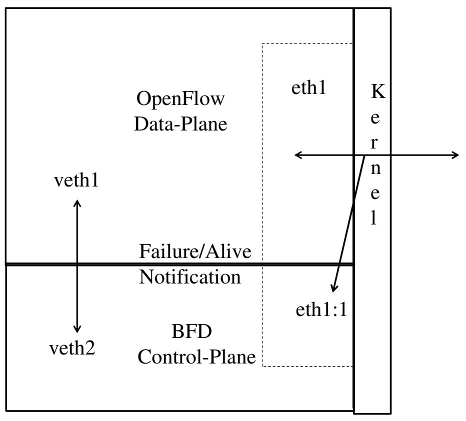


Figure 2.6: Integration of BFD in OpenFlow

created between the OpenFlow switch and BFD (two different processes in each OpenFlow switch). Furthermore, an alias of the OpenFlow port (eth1:1) has been created for BFD to receive packets from the OpenFlow port (eth1). The BFD failure detection time in our emulation was 40 to 44 ms. [2] For restoration, we did not establish a BFD session. The OpenFlow switches detected the failure when an LOS was declared by a port as a "port down" event.

We derive an analytical model for the recovery time in the next section. The first experiment was performed to measure parameters of the analytical model. We used these parameters to verify our model. The second experiment was carried out on the CT topology. In this experiment, each server sent two different flows to all other servers. This experiment was performed with 480 flows in the network. The aim of this experiment was to find the recovery time experimentally and compare with the mathematical results that are calculated via our analytical model. The third experiment was similar to the second experiment, but was performed with all the different topologies. The aim of this experiment was to compare the recovery time in different network topology scenarios. The fourth experiment was performed using a node failure instead of a link failure. The aim of this experiment was to test OpenFlow with a node failure condition. The fifth experiment was performed by increasing the number of flows in the CT topology. We increased the number of flows up to 24000 in this experiment. The aim of this experiment was to do a scalability experiment (i.e., how the number of flows affects the recovery time) in an OpenFlow network.

We now describe the second experiment in detail. All the further experiments are based on the second experiment. In the second experiment, we break a link

[2]The packet send interval is kept as 10 ms and if continuously 4 BFD packets are missed, BFD detects a failure.

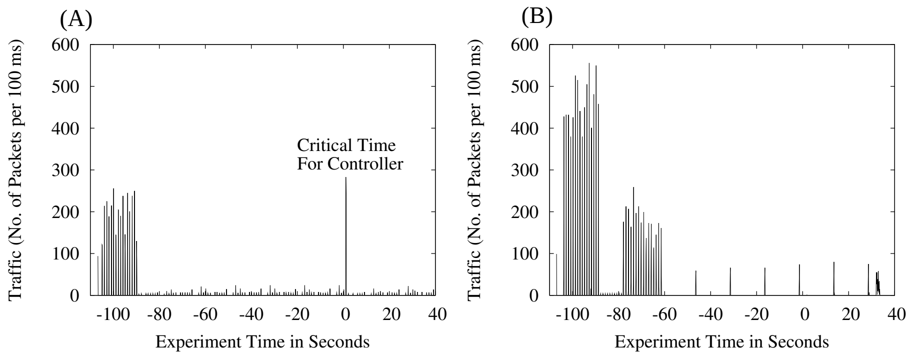


Figure 2.7: (A) Restoration (NOX intensity) (B) protection (NOX intensity)

at the 0 s and find the recovery time. We describe the link break by failing the London-Amsterdam link in the CT topology (Figure 2.5A). Figure 2.7 shows the traffic that was captured using the tcpdump utility in the NOX controller. At the beginning of the experiment, there were 16 spikes (from -106 to -92 s) after each one-second interval. These spikes were due to the traffic from the OpenFlow switches to establish paths between servers. The one-second between the spikes occured because we have started sending the pktgen traffic after waiting one-second between each server. The one-second interval was used to avoid overloading the NOX controller as the switches can try to establish too many flows in a short time span. The spikes in Figure 2.7B are higher than the spikes in Figure 2.7A because protection establishes an alternative path together with the working path. In protection, each OpenFlow switch also established the BFD sessions (the spikes from -78 to -64 s in Figure 2.7B) for each different working path. There were small spikes periodically in the controller traffic. These were the echo messages that were sent to check aliveness of the controller links. The height and number of these spikes are different in Figure 2.7A and 2.7B. This was due to the minor time difference between the start of both experiments. At the 0 s, we have failed the link London-Amsterdam by disabling the Ethernet interface at London. When the OpenFlow switch (London) detected this failure, the notification message was sent to the controller. Since the controller in restoration starts recovery upon a failure notification, there is a large spike at around 0 s in Figure 2.7A. These were the flow-mod messages and the acknowledgments of these messages sent to reestablish the new path. In protection, as backup flows were installed before the failure occurs, the controller does not take any action upon the failure. Therefore, we do not see any spike at 0 s in Figure 2.7B.

Figure 2.7 shows that in normal operation, the control network load is generally low. Implementing a high speed control network only for restoration (shown by critical time in Figure 2.7A) will probably not make sense. Implementing protection mechanisms in the switches will be more cost-efficient, it may slightly increase the bandwidth requirement at flow setup time due to extra protection information to be sent to the switch, but highly decrease the bandwidth requirements during the failures by allowing the switch to perform the protection switching without the controller interference.

2.4 Results

This section is structured according to the number of different experiments performed for our emulation.

2.4.1 Analytical model and measurement of its parameters

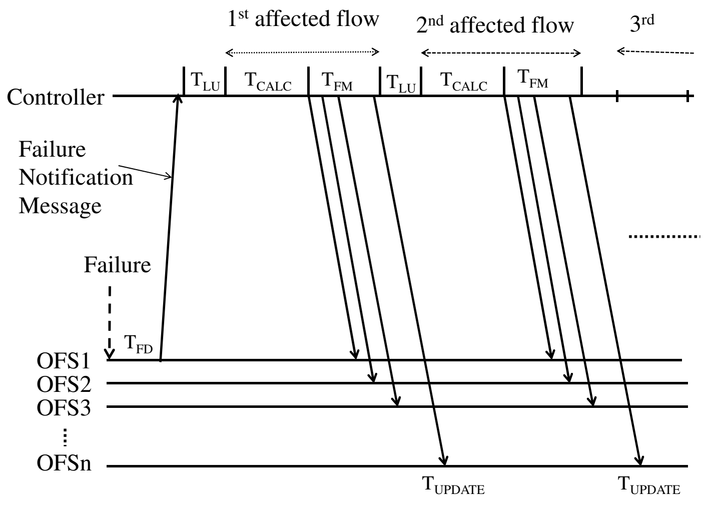


Figure 2.8: Analytical model for restoration

In this section, we derive an analytical model to calculate the failure recovery time in an OpenFlow network. Our model is based on the failure recovery model described in [10] and [20]. We extended this model for our implemented restoration scheme.

Figure 2.8 shows the recovery time in restoration, i.e, when the failure is detected (T_{FD}), the failure notification message is sent, the affected flow is searched (T_{LU}), a new path is calculated (T_{CALC}) and then the flow-mod messages (T_{FM}) are sent, then again the next affected flow is searched (T_{LU}) and so on. When switches receive the flow-mod message, the FlowTable is also updated with a new Flow Entry (T_{UPDATE}). We calculate the mathematical expression for the restoration time (T_R) for N affected flows (f) and it is shown in Eq. 2.1. T_{PROP} in Eq. 2.1 is the propagation time. $T_{LU,f}$, $T_{CALC,f}$ and $T_{FM,f}$ are the T_{LU}, T_{CALC} and T_{FM} for the affected flow f.

$$T_R = T_{FD} + T_{PROP} + \sum_{f=1}^{N} (T_{LU,f} + T_{CALC,f} + T_{FM,f}) + T_{PROP} + T_{UPDATE} \tag{2.1}$$

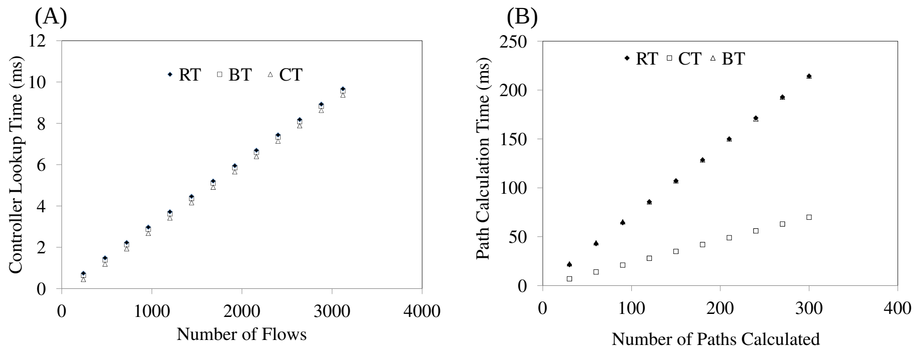


Figure 2.9: (A) Flow lookup time (affected/unaffected) (B) Controller path calculation time

In our implemented mechanism, the affected port is searched in the flow path linearly for the link failure. If it is found, the flow is declared to be affected. The lookup time for the number of flows is shown in Figure 2.9A.

If a flow is declared to be affected, a new shortest path is calculated in our algorithm. Figure 2.9B shows the path calculation time in the RT, BT and CT topology. The paths calculated in our experiment are the shortest paths obtained via the Dijkstra algorithm. The time complexity of the implemented Dijkstra algorithm is $O(n^2)$ where n is the number of nodes. This time can be decreased to $E + n \times log(n)$ (E is the number of edges) by using a Fibonacci heap for storing the topology graph. The path calculation times are measured for all topologies used in experiments and are shown in Figure 2.9B.

We measured the flow-mod transmission capacity of the controller over a 1000 Mbps link. Figure 2.10A shows the transmission capacity of the controller link when the flow-mod messages were transmitted over 1, 3, 4 and 12 TCP connections. We observed 38%, 100%, 100%, and 100% of the CPU utilization for 1, 3, 4 and 12 TCP connections respectively. The flow-mod transmission capacity decreased when the controller started transmitting flow-mod messages on all the 12 TCP connections. This is because of context switching between the different TCP connections. Figure 2.10A shows the maximum flow-mod transmission capacity as 15000 to 40000 packets per second.

A FlowTable in the OpenFlow 1.1 software, implemented by Ericsson, is a linear table. If a match is found, the entry is modified in the table. Otherwise, a

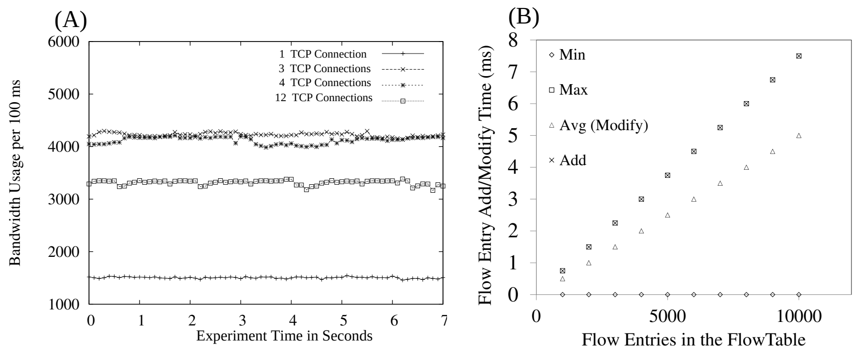


Figure 2.10: (A) Controller flow-mod transmission capacity (B) Flow entry modification/addition time

new entry is added at the end. The time for modification (minimum, average and maximum) and addition of an entry with respect to the number of Flow Entries in the table is shown in Figure 2.10B.

We calculated each time (flow lookup, path calculation, flow-mod transmission and flow-entry addition) separately to determine the recovery time (Eq. 2.1) which we compare with the experimental results in the next section.

Now we describe the time complexity of our protection mechanism. The recovery time in the protection mechanism depends on the time the switch takes to modify the alive-status of the Group Entries. The switch takes O (1) time (approximately 5.8 microsecond) to modify the alive-status of one Group Entry. For n Group Entries, it takes $O(n)$ time. For our experiment, the number of Group Entries in a switch is equal to the number of different paths that are established by it. Thus, the number of Group Entries (per switch) for the CT, BT, and RT topology is 15, 27, and 27 respectively.

2.4.2 Emulation results

We now show the results of the experiment in which the link London-Amsterdam was failed in the emulated CT topology (the second experiment). Figure 2.11 shows the traffic at the London-Amsterdam (from -0.2 to 0.3 s) link, which was captured at Amsterdam. As the port at London was disabled at 0 s, London stopped transmitting packets at this link. However, as the port at Amsterdam was not disabled, Amsterdam continued transmitting traffic on the same link until the controller establishes new fault-free paths. There is no traffic in Figure 2.11A after 0.240 s because all the traffic has been switched to some other fault-free path. However, the total traffic becomes equal to the BFD traffic after about 0.05 s in Figure 2.11B. This is because only BFD traffic remained on this link. It was not switched to some other path.

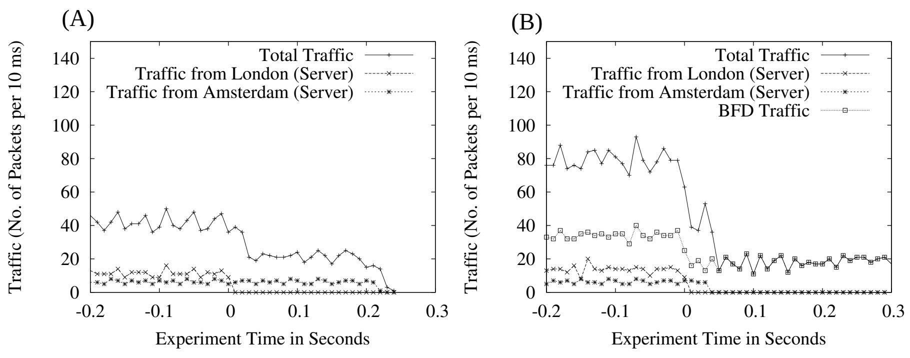


Figure 2.11: (A) Restoration (Traffic on the affected link) (B) Protection (Traffic on the affected link)

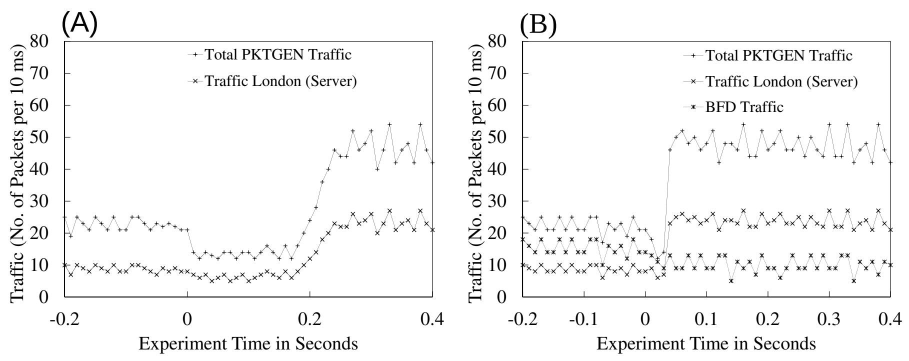


Figure 2.12: (A) Restoration (traffic on the restored link) (B) Protection (traffic on the protected path)

Figure 2.12 shows the traffic on the link London-Paris. Traffic London (Server) in Figure 2.12 is the traffic from London (server) to all the other servers. The traffic from -0.2 to 0.4 s is shown in Figure 2.12. After the failure recovery action at 0 s, this was only the link connecting London, so all the traffic from and towards London (server) has to pass through this link. At the time of the link failure (at 0 s), Figure 2.12 shows a drop in the total PKTGEN traffic on this link. The dropped traffic was the traffic that was coming from the link London-Amsterdam to the link London-Paris. Restoration/protection reroutes the affected traffic. Figure 2.12A shows that there is a drop in the total traffic for approximately 0.190 s, followed by a step-wise increase in traffic until 0.240 s when all flows in our experiment were restored. Figure 2.12B shows this time as approximately 0.040 s, followed by a stepwise increase until 0.050 s for protection. Figure 2.12B shows a small decrease in the BFD traffic after the failure because the BFD traffic from the link

London-Amsterdam through London was completely lost after the link failure.

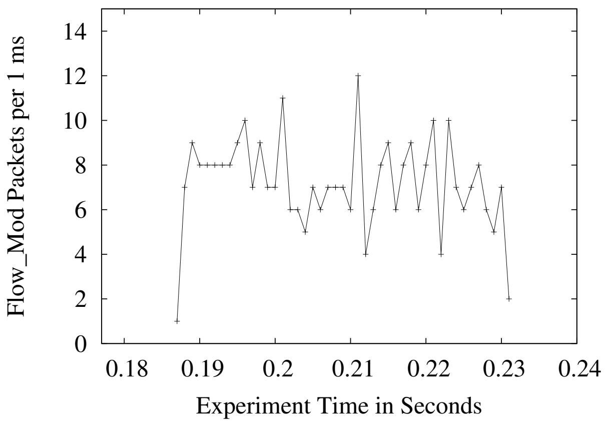


Figure 2.13: NOX Flow-mod traffic in restoration

In our restoration mechanism, London detected the failure at about 0.187 s. However, restoration took approximately 0.053 s to restore all the flows (0.187 to 0.240 s). Figure 2.13 shows a detail of 0.045 s interval (0.187 to 0.232 s) in which the NOX controller searched the affected flows, calculated new paths and sent flow-mod traffic to the switches.

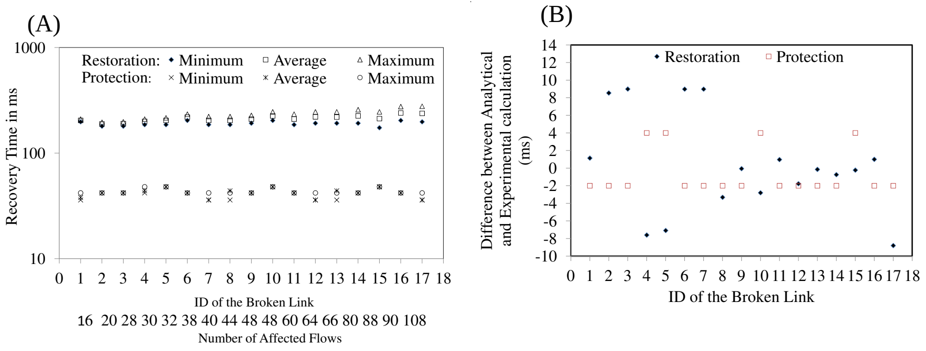


Figure 2.14: (A) Recovery time (experimental) (B) Recovery time difference

We did the link failure experiment for all the indicated links of Figure 2.5A. The results are depicted in Figure 2.14A. The x-axis shows the ID of the broken links and the number of affected flows. The y-axis shows the minimum restoration/protection time (the time it took to recover the first flow), the maximum restoration/protection time (the time it took to recover all the flows) and the average restoration/protection time (the expected time for any flow to be recovered after a failure). The recovery time is calculated as the number of packets dropped multiplied by the packet send interval. The links are ordered from left to right according to the number of flows that were affected by the link failure (Figure 2.14A) .

The Figure 2.14A shows the difference in the time when the first flow was restored after each failure. The reason behind this is that switches detected the failure at different time points. In our Linux node, it is difficult to measure this failure detection time in ms, as there is some random time between the moment we break the link and the moment the Linux system has actually disabled the link (both receive and transmit disabled). The first flow was restored in 2 to 3 ms. So the failure detection time in our experiment was approximately equal to the time when the first flow was restored. In Figure 2.14A, the maximum restoration time after failure detection is shown by the link ID 17, which is equal to 60 ms. Figure 2.14A illustrates the dependence on the number of flows that have been restored. Figure 2.14A also shows that all the flows were protected between 42 to 48 ms in protection. This includes the failure detection time of BFD which was approximately 40 to 44 ms. Figure 2.14B shows the difference between the analytical and experimental calculation of the restoration and protection time. In the restoration calculation, the failure detection time is assumed the time when the first flow restored in our experiment. In the protection calculation, it is assumed 44 ms. As the average path length in the CT topology is 3, we assumed that the controller has transmitted 4 flow-mod message per affected flow in restoration. Figure 2.14B shows the difference of ± 9 ms for restoration and the difference of -2 to +4 for protection. This difference includes the error in the recovery time calculation that is generated by the packet send interval (6 ms) in our experimental results.

Now we show the results of the link failure and the node failure experiment performed on all different topologies (the third and fourth experiment). In the link failure experiment, a link was broken, and the number of affected flows and the recovery time were calculated. In the node failure experiment, the failure was given by running “poweroff” command on one the CT OpenFlow switches that are present in all the topologies, and the number of affected flows and the recovery time were calculated. In this experiment, the controller receives a FIN message because an OpenFlow switch disconnects from the controller. The controller then starts the same restoration activity as performed in the link failure experiment. The results of both experiments are depicted in Figure 2.15. The x-axis shows the number of affected flows. The y-axis shows the recovery time after the first flow was restored in the experiment. (Link) and (Node) in Figure 2.15 are referred to the link and node failure experiment respectively. The number of servers in the CT, BT and RT are 16, 28 and 28 respectively. As each server transmitted packets to all the other servers present in the topology, we observed different number of flows

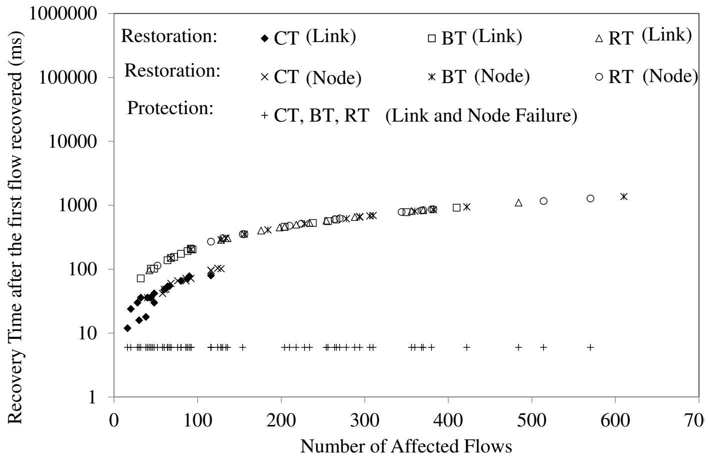


Figure 2.15: Link and node failure experiment on the CT, BT and RT topologies

affected by failing the same link or the same node in different topologies. Figure 2.15 shows that the restoration time depends on the number of affected flows in all the topologies. The results also show that the CT topology has less restoration time than the BT or RT topology because the path calculation time in the CT is less than the BT or RT topology.

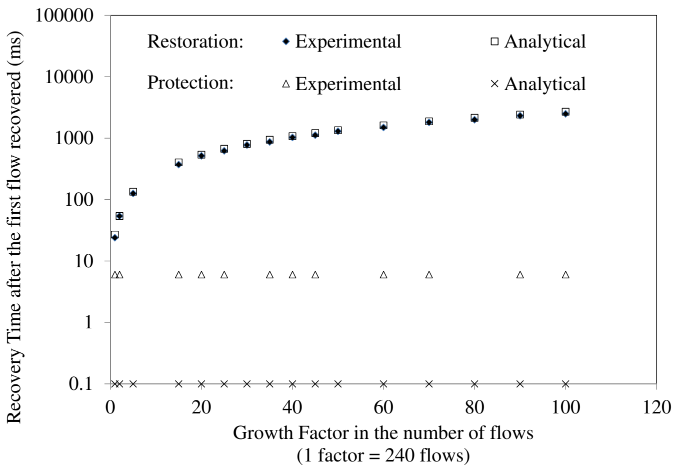


Figure 2.16: Scalability experiment

To test scalability, we did the experiment on the CT topology, where the number of flows from each server was increased by a factor n (1 to 100) and the link London-Amsterdam was failed during the experiment (the fifth experiment). The factor n means each server transmitted n different flows to all other servers. In the factor 1, there were 240 flows in the network and 33 were affected by the failure. In the factor 100, there were 24000 flows in the network and 3300 were affected by the failure. The experimental and analytical results are shown in Figure 2.16. We found a linear increment in the restoration time. We observed approximately 2.5 s restoration time when we increased the number of flows by the factor 100. However, in protection, we did not observe dependence on the increased number of flows. This is because in protection, we established 15 Group Entries (per switch) for all the flows in the CT topology, and modification of the affected entries has taken less than a 1 ms time ($O(n)$).

We evaluated restoration and protection in mesh topologies. Carrier-grade networks often feature a ring topology for the aggregation segments. For resiliency on rings, typically protection is used, as each switch has only two directions. If a connection is broken, traffic is sent along the other direction on the ring. A standardized protection solution for packet networks is Resilient Packet Ring (RPR, IEEE 802.17). RPR has two modes of protection, wrapping and steering. In wrapping, when a failure is detected, traffic going towards and from the failure direction is wrapped (looped) back to go in the opposite direction on the other ring (subject to the protection hierarchy). In steering, traffic is redirected in the source node to the opposite ring. Wrapping and steering can be easily implemented in OpenFlow by the GroupTable concept where the first action bucket in a Group Entry contains the action for the working path and the second action bucket contains the action for the protection path. Performance-wise, this will perform similarly (or even better, since only 2 action buckets are needed) to the protection on a mesh, so 50 ms protection in Openflow can be met on a ring.

2.5 Additional considerations

2.5.1 Memory size requirement in protection

Our results show that protection is better than restoration because the former reduces the time required for fault recovery and avoids the sudden increase of traffic load in the controller at the time of failure detection. However, protection needs to maintain additional information of alternative paths together with the working path. Thus, the memory requirement in protection is more than restoration.

In restoration, the controller replaces the Flow Entry of the working path in the ingress OpenFlow switch (e.g. switch A in Figure 2.2), which implies that one Flow Entry per flow is required at any time in this switch. However, the ingress OpenFlow switch for protection (e.g. switch A in Figure 2.3B) installs an additional Group Entry for failure recovery. Thus, the additional memory requirement of the ingress OpenFlow switch for protection is the size of GroupTable i.e. the number of Group Entries in the GroupTable.

As the TCAM memory is expensive, it is the size requirement of this memory which is important for OpenFlow switches. The size requirement of this memory depends on implementation of OpenFlow switches. The OpenFlow switch implemented in HP procurve 5400 zl series [21] manages FlowTables in hardware and in software. The FlowTable in software has the full set of Flow-Entries, and the FlowTable in hardware has the subset of Flow Entries. The FlowTable in hardware is managed using a TCAM that translates a Flow Entry into a TCAM entry. When a packet does not match to any TCAM entry, the packet is forwarded to software. If the matching entry is found in software, it is installed in the TCAM and the packet is forwarded.

Thus, the Flow Entries related to the protection path can be installed in software, and the Flow Entries related to the working path can be installed in hardware. Once a failure occurs in the working path, the Flow entries related to the protection path can be moved to the TCAM. Furthermore, as the GroupTable requires a 32 bit match on the Group ID, the GroupTable can be present in the static or dynamic RAM. Thus, for this type of OpenFlow switch implementation, the protection path does not require the additional Flow Entries to be installed in the TCAM. Therefore, the TCAM memory requirement of protection can be equal to restoration.

2.5.2 Reliability of the control plane

In this paper, we considered failures in the data-plane side i.e. recovery from a failure when a data traffic path fails. However, because Openflow is a split architecture (relying on the controller to take actions when a new flow is introduced in the network), reliability of the control plane is also an important issue. The controller should also be resilient against targeted attacks. There are multiple options for control plane resiliency. One can provide two controllers, each on a separate control network and when a connection to one controller is lost, the switch can switch over to the backup network. This is a very expensive solution. Another option is to try to restore the connection to the controller by routing the control traffic over the data network. When a switch loses a connection to the Openflow controller, it can send its control traffic to a neighboring switch, which will require the controller to detect such messages and establish Flow Entries for routing the control traffic through this neighbor switch. This through-the-data-plane solution is an intermediate step towards full in-band control. An effective scheme for carrier grade networks may be to implement out-of-band control in the failure-free scenario, switching to in-band control for switches which lose the controller connection after a failure. In-band control is supported in the Openflow specification. There could be a situation where the controller itself crashes. In this situation, we can have two controllers so that when the one controller crashes then OpenFlow switches can rely on a backup controller. In future work, we will consider those situations in OpenFlow networks.

2.6 Related work

To the best of our knowledge, the research presented in this chapter (or paper) is the first work performed for achieving carrier-grade quality in OpenFlow. However, currently, many other research groups and projects have been showing significant interest in exploring many other mechanisms to achieve fast failure recovery requirements (such as carrier-grade). Some of these works are listed below:

In [22], segment protection is implemented in an OpenFlow based Ethernet network. In this protection scheme, the working and backup paths (with different priorities) are pre-configured for a segment of the network and when a failure is detected in the working path, an auto reject mechanism (proposed for protection) removes the working path and therefore, enables traffic to be forwarded through the backup path Flow Entries.

In [23], protection schemes are implemented for each link in a network (instead of for each path). In these schemes, a protection path (and a BFD session) is established for each link and when a failure occurs in a link, traffic is redirected to the corresponding protection path. In addition, control traffic protection schemes are researched in [24]. In these schemes, multiple controllers are used to recover from controller failures scenario. In [25], a mechanism is proposed to recover from a failure when the fast-failover group type in an OpenFlow switch is not available. For this case, another group type (such as SELECT [16]) is used to implement protection.

Currently, the EU-FP7 BEBA project [26] has been showing a lot of interest in implementing fast failure recovery in SDN with zero packet loss regardless of controller reachability and even when OpenFlow's fast-failover feature cannot be used. The proposed mechanism is based on OpenState, an OpenFlow extension that allows a programmer to specify how forwarding rules should autonomously adapt in a stateful fashion, reducing the need to rely on remote controllers.

2.7 Conclusions

In this paper, we have presented restoration and path protection for OpenFlow to deploy it in a carrier grade network. We ran extensive experiments on emulated pan-European network topologies and tested OpenFlow in a real environment via our virtual-wall testbed facility. We showed that OpenFlow can restore traffic, but its dependency on the centralized controller means that it will be hard to achieve 50 ms restoration in a large-scale carrier grade network. We used the group table concept (recently proposed for OpenFlow) to implement protection. In this paper, we proposed the first implementation of protection based on the group table concept. Finally, we showed that OpenFlow can achieve the carrier-grade requirement of a 50 ms interval if protection is implemented in these networks to recover from failures.

Acknowledgment

The research leading to these results has received funding from the European Communitys Seventh Framework Programme (FP7) under Grant agreement no. 258457 (SPARC) and no. 258365 (OFELIA).

References

[1] N. McKeown, T. Andershnan, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *Openflow: Enabling innovation in campus networks*, ACM Computer Communication Review, Vol. 38, Issue 2, pp. 69-74, New York, USA, 2008.

[2] GENI [Online]. Available: http://www.geni.net.

[3] JGN2plus research and development test-bed network [Online]. Available: http://www.jgn.nict.go.jp/.

[4] OFELIA [Online]. Available: http://www.fp7-ofelia.eu/ .

[5] ONF [Online]. Available: https://www.opennetworking.org/.

[6] SPARC [Online]. Available: http://www.fp7-sparc.eu/ .

[7] D. Collins, *Carrier-grade Voice over IP*, McGrawa Hill, 2000.

[8] B. Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, *MPLS-TP requirements*, RFC 5654, IETF, 2009.

[9] D. Katz, and D. Ward, *Bidirectional Forwarding Detection*, RFC-5880, IETF, 2010.

[10] J. P. Vasseur, M. Pickavet, P. Demeester, *Network recovery: protection and restoration of optical, SONET-SDH, IP and MPLS*, Morgan Kaufmann, 2004.

[11] E. Mannie and D. Papadimitriou, *Recovery (Protection and Restoration) Terminology for Generalized Multi-Protocol Label Switching (GMPLS)*, RFC 4427, IETF, 2006.

[12] A. R. Sharafat, S. Das, G. Parulkar, and N. McKeown, *MPLS-TE and MPLS VPNs with Openflow*, ACM Computer Communication Review, Vol. 41, Issue 4, pp. 452-453, New York, USA, 2011.

[13] D. Jocha, A. Kern, A. Takacs, P. Skoldstrom, *MPLS-Openflow based access/aggregation network*, GENI Engineering Conference, Puerto Rico, US, 2011.

[14] S. Sharma, D. Staessens, D. Colle, M. Pickavet, P. Demeester, *Enabling Fast Failure Recovery in OpenFlow Networks*, Design of Reliable Communication Networks, pp. 164 - 171, Krakow, Poland, 2011.

[15] D. Staessens, S. Sharma, D. Colle, M. Pickavet, P. Demeester, *Software Defined Networking: Meeting Carrier Grade Requirements*, Local & Metropolitan Area Networks, pp. 1-6, North Carolina, USA 2011.

[16] OpenFlow Switch Specification: Version 1.1.0 (Wire Protocol 0x02) [Online]. Available: http://www.openflow.org/, 2011.

[17] Emulab Network Emulation [Online]. Available: http://www.emulab.net011.

[18] Ericsson OpenFlow and NOX Controller Software [Online]. Available:https://github.com/TrafficLab.

[19] S. D. Maesschalck, D. Colle, I. Lievens, M. Pickavet, P. Demeester, C. Mauz, M. Jaeger, R. Inkret, B. Mikac and J. Derkacz, *Pan-European Optical Transport Networks: An Availability-based Comparison, Photonic Network Communications*, Vol. 5, Issue 3, pp. 203-225, 2003.

[20] V. Sharma, F. Hellstrand, *Framework for Multi-Protocol Label Switching (MPLS)-based Recovery*, RFC 3469, IETF, 2003.

[21] OpenFlow switch HP procurve 5400 zl series [Online]. Available: http://www.openflow.org/wp/switch-hp/.

[22] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, *OpenFlow-Based Segment Protection in Ethernet Networks*, Journal of Optical Networks, Vol. 5(9), 2013

[23] L. M. Niels, J. Benjamin and A. Fernando, *Fast Recovery in Software-Defined Networks*, EWSDN, 2014

[24] Y. Hu, Wang Wendong, G. Xiangyang, C. H. Liu, X. Que and S. Cheng, *Control Traffic Protection in Software-Defined Networks*, IEEE Globecom, 2014

[25] K. Nguyen, Q. T. Minh, S. Yamada, *Novel Fast Switchover on OpenFlow Switch*, IEEE CCNC, 2014

[26] BEBA Project [Online]. Available: http://www.beba-project.eu/

3 Verification of Flow Matching Functionality in the Forwarding Plane of OpenFlow Networks

In the previous chapter, we considered node and link failures, and proposed failure detection and recovery mechanisms for OpenFlow. However, failures can also be caused by other errors in forwarding functionality (such as matching errors). This chapter proposes a mechanism which can be used to detect matching errors in the forwarding functionality of OpenFlow switches/routers. Finding all the matching errors is difficult by just analyzing the configuration of a network. Therefore, the mechanism transmits test packets in the network to find matching errors. The mechanism can be executed from the controller, or on an additional device or server (or virtual machines) attached to the network. The experimental results evaluate the trade-off between the verification time and the required resources, enabling the user of the mechanism to choose which bandwidth to reserve for given verification time.

⋆ ⋆ ⋆

Sachin Sharma, Wouter Tavernier, Sahel Sahhaf, Didier Colle, Mario Pickavet, Piet Demeester

Published in IEICE Transactions on Communications, November 2015.[1]

Abstract In OpenFlow, data and control plane are decoupled from switches or routers. While the data plane resides in the switches or routers, the control plane might be moved into one or more external servers (controllers). In this article, we propose verification mechanisms for the data plane functionality of switches. The latter consists of two parts: (1) Flow-Match Header part (to match a flow of incoming packets) and (2) action part (e.g., to forward incoming packets to an outgoing port). We propose a mechanism to verify the Flow-Match Header part of the data plane. The mechanism can be executed at the controller, or on an additional device or server (or virtual machines) attached to the network. Deploying a virtual machine (VM) or server for verification may decrease the load of the controller and/or consumed bandwidth between the controller and a switch. We propose a heuristic to place external verification devices or VMs in a network such that the verification time can be minimized. Verification time with respect to consumed resources are evaluated through emulation experiments. Results confirm that the verification time using the proposed heuristic is indeed shortened significantly, while requiring low bandwidth resources.

Keywords

OpenFlow; In-band; Out-of-band; Verification

3.1 Introduction

OpenFlow [1] decouples the control plane from the data plane of switches or routers and embeds it into one or more external servers (controllers). The core idea of OpenFlow is to provide programmability of the data plane from the controllers using the OpenFlow protocol. In fact, the controllers program the data plane by "adding/ modifying/deleting" entries in the FlowTables (forwarding table) of switches.

An entry in a FlowTable contains: (1) Flow-Match Header, which defines a flow, (2) actions, which define how packets should be forwarded (i.e., forward to an output port or to a different FlowTable) and (3) some additional fields such as priority, statistics and cookie identifier (Fig. 3.1). When a packet arrives at an OpenFlow switch, it is matched against the Flow-Match Header (wildcarded or

[1]Compared to the publication in IEICE Transactions on Communications, footnote 2, 3, and 4 are added to the chapter.

Flow-Match Header part							actions	additional fields		
Ingress port	Dst MAC	Src MAC	Src IP	Dst IP	Src port	Dst port	Port or Table	Priority	statistics	cookie identifier
1	*	*	*	1.2.3.*	1	1	Port:2	3245	325	2

*=wildcards

Figure 3.1: Example of a Flow Entry

exact match) of the entries of the FlowTable. If a match is found, the statistics of that entry is updated and the actions are performed. If two or more matches are found, the actions of the highest priority number entry are performed. If no match is found, the packet (a part thereof) is forwarded to the controller. Thereafter, the controller determines how the packet can be handled. It may return the packet to the switch indicating the forwarding port, or it may add a Flow Entry in the switch to forward the packet.

The research challenge that we consider in this article is the verification of matching of incoming packets with the Flow-Match Header of a Flow Entry. There can be two causes of incorrect or no matching of packets: (1) bugs in OpenFlow switch implementation and (2) errors in FlowTable configuration. Bugs in OpenFlow switch implementation [2] may be caused by bugs in the hardware or software part of switches. The errors in FlowTable configuration may be caused by: (1) bugs in controller software for the addition of a Flow Entry and/or (2) presence of a high priority error-prone Flow Entry (added manually or by a controller) that gives a match with the incoming packets [3]. The objective of verification is to find this incorrect or no matching and hence, to find the packet-headers that cannot be matched or can be matched incorrectly with the Flow-Match Header of a Flow Entry. In the absence of this verification, it may be difficult to find which packets cannot be delivered or can be delivered incorrectly by a switch.

Most of the existing verification tools such as HSA (Header Space Analysis) [4], Anteater [5], VeriFlow [6] detect matching issues by analyzing configuration of switches. However, without sending real packets, it is difficult for these tools to find software or hardware bugs in flow matching. Recently, the automatic test packet generation (ATPG) tool [7] is proposed to verify the network by transmitting test packets. ATPG verifies only one (or more than one) of the packet headers that gives a match with a wildcarded Flow Entry, whereas matching of all the other packet headers remains untested. In the (short) demo paper [8], we demonstrated a mechanism to verify this matching functionality of a switch, where the controller is used for verification. However, this mechanism requires modification of the Flow Entries, which may not be acceptable in operational networks. In this article, we propose a mechanism to verify the Flow Entries without modifying them.

Our mechanism transmits test packets to verify that all the packet-headers match correctly with the Flow-Match Header or not. However, if the test packets have to match with the Flow Entries of data packets, these would need additional bandwidth to be reserved for test packets on the links corresponding to the outgoing actions of the matched Flow Entries. To overcome the challenge (additional bandwidth requirement), we forward the test packets through duplicated Flow Entries, which either drop or forward the test packets to the controller (instead of forwarding these on the outgoing links).

Our mechanism copies all the Flow Entries of a FlowTable to a new table (i.e, any other FlowTable of a switch), and the new table is verified then by sending test packets and performing a verification activity. As the Flow-Match Headers of the Flow Entries in the new table is the exact copy of the Flow-Match headers in the original table, the assumption is that if a flow matching failure is reported by the new table, the failure is also present in the original table. Moreover, as the action part (verification of the action part is already covered by ATPG) of the Flow Entries in the new table is different from the corresponding Flow Entries in the original table, all the test packets can be forwarded through the new table without being sent on the outgoing action of the original Flow Entries. Therefore, the objective of copying Flow Entries is to: (1) maintain the same set of Flow-Match Headers in the new table (as the original table) for finding FlowTable specific matching issues [3], and (2) prevent forwarding of test packets through outgoing links (actions of Flow Entries in the original table) of a switch and therefore, decreasing the bandwidth requirements of those links for verification. Furthermore, as the mechanism does not verify the original Flow Entries, there may be some cases when our mechanism is not able to verify matching functionality for some software or hardware bugs. These cases are described in Section 3.2.1.

Our mechanism can be executed by the controller. However, it requires computational resources (to generate test packets) of the controller and the bandwidth resources between the controller and a switch (which are also utilized for the other controller activities) to perform verification. Therefore, to decrease the controller requirements for these resources, we consider a network in which servers (custom machines) can be attached to a switch or router to transmit or receive packets [9]. For verification, either these servers can be used directly for verification or these can host virtual machines (VM) to perform verification.

Deploying a VM or server over the network infrastructure may increase the cost, as it requires additional resources (memory, CPU etc.,) to be reserved for verification. Nevertheless, the VM may need to transmit test packets to a switch through the data path links of other switches, as all the switches may not have a direct link with the VM. Therefore, if the VM is placed at a location from where test packets need to travel a long path or same links to reach all or some switches, the bandwidth requirements of all these links for verification will increase. Hence, we also propose a heuristic to efficiently place the VMs or servers and efficiently select a path between switches and the VMs so that the verification time is reduced and resource requirements are distributed over many links in the network.

We perform extensive emulation experiments on the Fed4Fire testbed facility provided by iMinds [10]. In our experiments, verification using the controller and a VM (or server) is emulated. Software matching errors are emulated by translating some packets of a flow to incorrect headers. This leads to matching of these packets with an incorrect Flow Entry. In our results we show the verification time (the time required to find these matching errors) with respect to resources available in the network. The experiments validate the proposed approach and evaluate the trade-off between the verification time and the required resources, enabling the user of the mechanism to choose which bandwidth to reserve for given verification time. Additionally, our proposed heuristic is compared for different types of topologies which vary with the number of switches and degree of meshedness. The results show that the network can be verified in limited time, when a VM is placed using our proposed heuristics.

The mechanism is implemented as part of the verification functionality of the integrated prototype of the UNIFY project [9]. Section 3.2 provides errors in Flow Matching Functionality and Section 3.3 presents our proposed mechanism, Section 3.4 presents in-band or out-of-band verification, Section 3.5 and Section 3.6 present emulations and results, and then finally Section 3.7 concludes.

3.2 Errors in flow-matching functionality

In OpenFlow switches, there can be two types of Flow Entries: (1) exact match and (2) wildcarded Entries [11]. In case of the exact match entries, all the matching-header fields are specified for a Flow Entry. However, in case of the wildcarded entries, some of fields can be wildcarded for a Flow Entry. Finding matching issues in the exact match entries is simple, as these entries can match only one type of a flow. However, as wildcarded flows can match many different flows, it is complex to verify matching of these flows in OpenFlow networks. Our mechanism can verify the matching of these wildcarded flows. However, the similar mechanism can be used to verify the exact match flows. In this section, we describe the Flow-Matching bugs that can be present in an OpenFlow switch and the errors that can be identified by our proposed verification mechanism.

3.2.1 Software or hardware bugs in flow matching

The proposed mechanism can be used to find errors due to configuration issues and software or hardware bugs of flow matching. These bugs can be present at any block of software or hardware programming [2]. Fig. 3.2 depicts the functional blocks of OpenFlow switches such as HP, NEC switches [13] [14], and indicates where flow matching occurs. It also shows in which blocks matching-related bugs can be present, and which errors can be detected through our mechanism. Here, errors mean which packet-headers are not matched correctly or can be matched incorrectly by the Flow-Match Header due to a software or hardware bug.

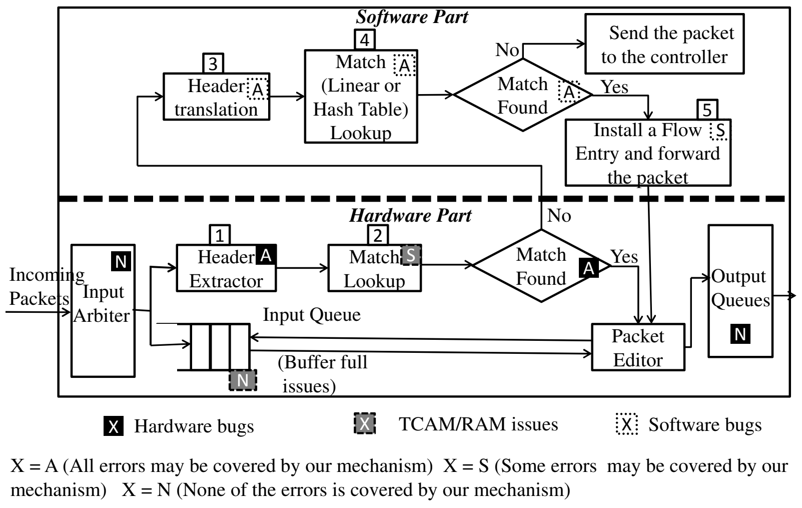


Figure 3.2: Hardware or software bugs in OpenFlow switches

Figure 3.2 shows matching in both software and hardware. In fact, many switches such as HP, NEC switches contain FlowTables in software as well as in hardware. Usually, the software table contains the full set of Flow Entries, while the hardware table contains a subset of the Flow Entries. When a packet arrives at a switch (Input Arbiter in Fig. 3.2), the packet is stored in the input queue and the packet-header is first extracted by the header extractor and then the packet header is matched against all the entries (TCAM or SRAM) present in hardware. If a match is found, the Packet-Editor forwards the packet to the output port/queue from the input queue. If no match is found, the packet is forwarded to software which translates again the packet into headers. If a matching entry is found in software, the entry is installed in the hardware FlowTable and the packet is forwarded through the Packet-Editor (Fig. 3.2). Otherwise, it is forwarded to the controller to define its action.

All the bugs in this packet forwarding are listed in Fig. 3.2. Some of these are also listed below:

1. In the header extractor (block 1 in Fig. 3.2), there can be hardware bugs related to extracting of some header fields (such as specific MAC address). In this case, the same set of packet-headers may be extracted incorrectly. Our mechanism can find the errors generated due to these hardware bugs.

2. In the match lookup in TCAM or SRAM (block 2 in Fig. 3.2), there can bugs related to specific TCAM bits[2]. For example, if a TCAM bit is supposed to be x, but it is always 1, then all packets with a 0 at that location will be ignored. If this issue is present in all TCAM entries, the errors can be reported correctly by our mechanism. However, if this issue is specific to some entries, it cannot be correctly identified by the mechanism. This is because test packets used in verification may be matched against an error-free or error-prone TCAM entry.

3. In the software part of packet translation (block 3 in Fig. 3.2), there may be some bugs related to the translation of some header fields. The errors due to these bugs can be detected by our mechanism.

4. In the hash table lookup in software (block 4 in Fig. 3.2), bugs may occur related to calculating the hash for specific headers. The errors due to these bugs can be identified using our mechanism.

5. In the Flow Entry addition from software to hardware (block 5 in Fig. 3.2), there may be bugs related to installing Flow Entries containing specific header fields. These errors can be also found through our mechanism.

3.3 Verification mechanism

Our mechanism uses a header field such as EtherType (or VLAN ID) for differentiating test packets from data packets and assumes that this header field (e.g., EtherType) is wildcarded in the Flow-Match Header part of Flow Entries.

The mechanism performs three steps for verification: (1) flow duplication, (2) test packet generation, and (3) matching errors identification. In the flow duplication step, the mechanism duplicates the Flow Entries from a FlowTable to another FlowTable. In the test packet generation step, the mechanism generates and transmits test packets that can match with the Flow-Match Header of duplicated Flow Entries. In the matching errors identification step, the mechanism calculates the matching errors either by reading the counters (statistics) of the duplicated Flow Entries or by comparing the sent and received test packets.

The flow duplication step can be performed by the controller, as it only needs to insert additional Flow Entries in the switches for verification. However, for test

[2]This bug can be efficiently tracked by a unit test of the TCAM memory modules.

packet generation and matching errors identification, either the controller or a VM (or server, see Introduction) can be used. It reduces the load of the controller (i.e., due to generation, transmission or reception of many test packets) for performing verification. We describe now all the steps in detail.

3.3.1 Flow duplication step

In the flow duplication step, for verifying FlowTable x (Fig. 3.3), the controller copies all the flows from FlowTable x to FlowTable y (x and y are $\geq$ 0 and $y > x$). The Flow-Match Header in Fig. 3.3 is an IP address field, but it can be any field (one or more) of the Flow-Match Header of Flow Entries. Fig. 3.3 shows that the Flow-Match Header of all the duplicated flows in FlowTable y is same as the Flow-Match Header part of the original flows in FlowTable x. However,

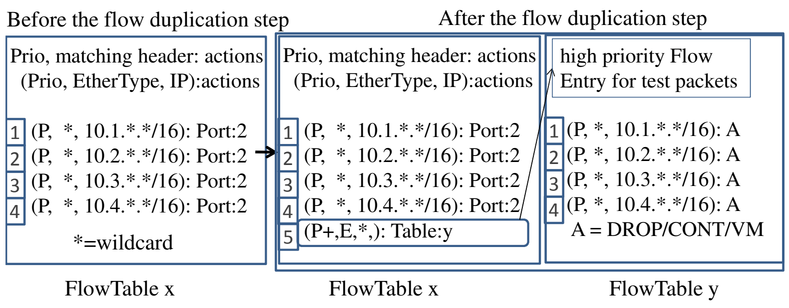

Before the flow duplication step
After the flow duplication step
Prio, matching header: actions
(Prio, EtherType, IP):actions
1 (P, *, 10.1.*.*/16): Port:2
2 (P, *, 10.2.*.*/16): Port:2
3 (P, *, 10.3.*.*/16): Port:2
4 (P, *, 10.4.*.*/16): Port:2
*=wildcard
5 (P+,E,*,): Table:y
high priority Flow Entry for test packets
1 (P, *, 10.1.*.*/16): A
2 (P, *, 10.2.*.*/16): A
3 (P, *, 10.3.*.*/16): A
4 (P, *, 10.4.*.*/16): A
A = DROP/CONT/VM
FlowTable x
FlowTable x
FlowTable y

Figure 3.3: Flow Entries before and after the flow duplication step. E is the EtherType of the test packets, P+ is the priority number higher than P. DROP, CONT or VM are actions for the duplicated FlowTable

the difference lies only in the action part. The action of all these duplicated flows is DROP, CONT, or VM. "DROP" means drop all the packets that match with the Flow Entry. "CONT" or "VM" means send all the matched packets to the controller or to the VM respectively.

Moreover, the controller inserts an additional Flow Entry (5^{th} entry in FlowTable x in Fig. 3.3) in FlowTable x to forward all the test packets to FlowTable y (action=Table:y). To match all test packets with this entry, the entry contains a higher priority (P+) number than the priority (P) number of the existing Flow entries in FlowTable x and the Flow-Match Header contains the Ethertype (E) of test packets and all other fields as wildcarded fields.

3.3.2 Test packet generation step

In this step, the controller or VM transmits test packets to the switch whose entries are needed to be verified. For this, the controller or VM can transmit all the test packets that can give a match with the Flow-Match Header of a Flow Entry. However, if all the fields of the Flow-Match Header are wildcarded, there is a need to transmits lots of test packets (e.g., 2^{32} different packets if there is a wildcard in all bits of the IP address field), increasing the bandwidth requirements and time for verifying the Flow Entries. To decrease the bandwidth requirement, we propose to transmit only those test packets that give a match with a partially wildcarded field and in a fully wildcarded field, we suggest to fill randomly generated values. This is proposed because if a field is partially wildcarded (such as wildcards in last 8 bits of IP address), there is a complex software implemented for matching a Flow Entry, as a part of a field is needed to be matched with the incoming packets [15]. This may lead to a matching error in any of these flows. However, if a field is fully wildcarded, matching implementation will be very simple (i.e., just ignore or do not match the header of the field [15]). Therefore, if these are tested with the limited number of headers, these can be considered as error free.

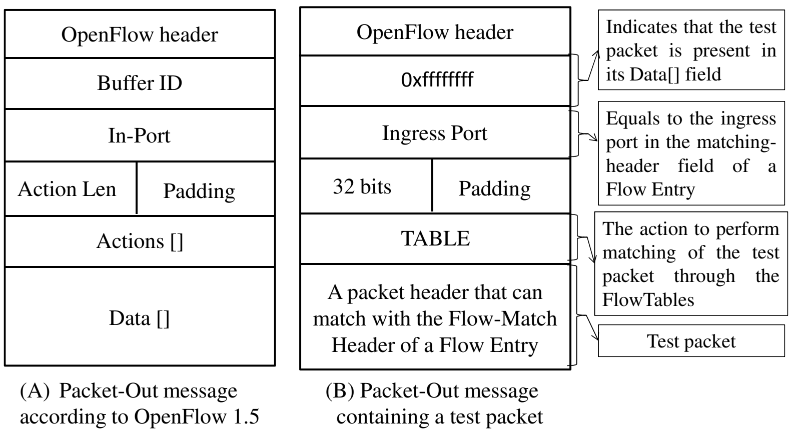

OpenFlow header
Buffer ID
In-Port
Action Len
Padding
Actions []
Data []
(A) Packet-Out message according to OpenFlow 1.5
OpenFlow header
0xffffffff
Ingress Port
32 bits
Padding
TABLE
A packet header that can match with the Flow-Match Header of a Flow Entry
(B) Packet-Out message containing a test packet
Indicates that the test packet is present in its Data[] field
Equals to the ingress port in the matching-header field of a Flow Entry
The action to perform matching of the test packet through the FlowTables
Test packet

Figure 3.4: Packet-out message for the test packet generation step

In our mechanism, the test packets are generated in the form of packet-out messages. Packet-out messages [12] are defined in OpenFlow to send packets from the controller through the FlowTables or to an outgoing port of a switch. We use these messages to transmit test packets to the switch (under verification). Fig. 3.4A describes all the fields of the packet-out messages according to OpenFlow version 1.5. Fig. 3.4B describes these fields for generating a test packet. In Fig. 3.4B, the packet-out message contains the ingress port of the Flow-Match Header in the

In-Port location and the test packet-header (i.e., all the other matching fields) in the Data[] location. As the action of the packet out messages is TABLE in Fig. 3.4B, it is matched against the Flow Entries of the FlowTables when it is received by the switch. Therefore, the matching errors can now be found using the Matching Error identification step.

3.3.3 Matching error identification

For this step, we describe two methods: (1) binary search and (2) packet-reception. The binary search method applies the well known binary search algorithm to find the matching errors. The packet-reception method receives the sent test packets and from the unreceived/received test packets, the method finds the matching errors. The advantage of the binary search method is that it reduces the upstream bandwidth to receive the test packets. However, the disadvantage is that it takes more time to find matching errors.

3.3.3.1 Binary-search method

For the binary-search method[3], the action of all the Flow Entries in FlowTable y is "DROP". The method is described in Fig. 3.5. Our explanation for the method is given for the Flow-Match Header as 10.1.1.* and the matching error is present in the packet that contains the IP address as 10.1.1.65. However, this explanation is applicable for more than one matching fields or errors.

To find the matching error in IP 10.1.1.65, the controller first transmits all test packets (i.e., containing the IP address from 10.1.1.0 to 10.1.1.255) to the switch (BS_1 in Fig. 3.5). The controller then finds the number of errors by subtracting the number of sent test packets (256) from the increase in the counters of the Flow Entry (i.e., under verification) of FlowTable y. As the Flow Entry cannot match one IP address (i.e., 10.1.1.65), the increase in the counters of the Flow Entry will be one less than the total sent test packets (i.e., 255). Therefore, the controller at this time knows that there is one matching error. However, it does not know that which flow (i.e., 10.1.1.65) cannot be matched through the Flow Entry. Therefore, to find this flow, the controller now transmits the first half of the test packets (i.e.

[3]In case of the binary search method, the well-known binary search algorithm is applied to find matching errors. However, there can be another method (known as linear search method) in which matching errors are found linearly i.e., test packets can be transmitted one by one and the counters of the Flow Entry (under verification) can be checked after each transmission for finding matching issues. For selecting a method (binary or linear), we can consider the following two parameters: (1) verification time and (2) bandwidth usage. The bandwidth requirement of the binary search method is more than the linear search method. However, the problem with the linear search method is that it needs to request counters from switches more times than the binary search method and the cost of requesting counters from switches is high in terms of time (the minimum time to update the counters in most of switches such as HP and Open vSwitch is 1 second). Therefore, compared to the binary search method, the linear search method will take more time to find matching errors.

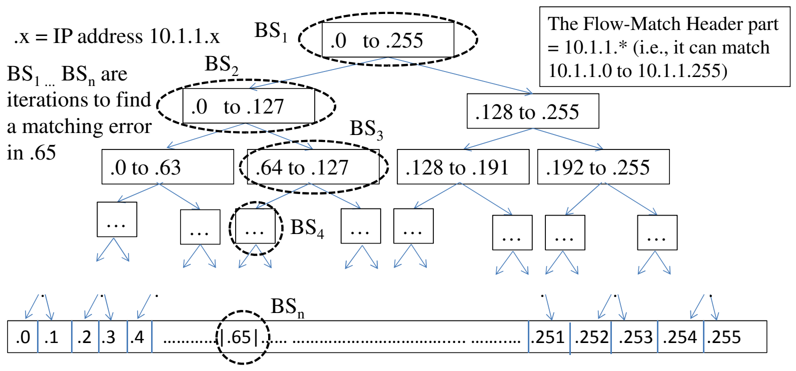


Figure 3.5: Binary-search Method

10.1.1.0 to 10.1.1.127, BS_2 in Fig. 3.5) and then finds the matching errors by the same formula (i.e., subtracting the number of sent packets from the counter increments). The controller will again find one error and therefore, it again sends the first half of the test packets (i.e. 10.1.1.0 to 10.1.1.63, the third row in Fig. 3.5) and finds no error. As there is no error in this first half, it now knows that the error is in the second half. Therefore, the controller now switches to the second half (i.e., 10.1.1.64 to 10.1.1.127, BS_3) and then process is repeated until it reaches to the last row of Fig. 3.5 (BS_n) in which it transmits only one test packet i.e., the test packet containing IP 10.1.1.65 and finds the error using the same formula. The controller then reports IP (10.1.1.65) as a matching error.

3.3.3.2 Packet-reception method

For this method, the action of all the Flow Entries in FlowTable y is CONT or VM. If this step is performed by the controller, the action is CONT. If this step is performed by a VM, the action is VM. Due to this action, the test packets are sent back to the controller or to the VM in the form of packet-In messages [12] after matching with the Flow-Match Header of a Flow Entry. If a test packet is not received back by the controller or the VM, it declares that it is due to a matching issue present in the respective Flow Entry and therefore, reports this matching issue.

To find matching with an incorrect Flow Entry, the packet-reception method depends on the flow cookie identifier [12] (Fig. 3.1), which is a unique 32 bit number associated with each Flow Entry in a switch. When a test packet is sent to the controller or VM back after matching a Flow Entry, the flow cookie identifier of the matched Flow Entry is copied in the packet-in message. The controller or VM receives this message. If it finds that the flow cookie identifier in the message is not same as the cookie identifier that is associated with the Flow Entry that must have a match with the test packet, the method declares that it is due to matching with an incorrect Flow Entry. This may happen because the received test packet has found a match with a Flow Entry that actually should not match with the test packet header (e.g., due to configuration errors or bugs in the switch implementation).

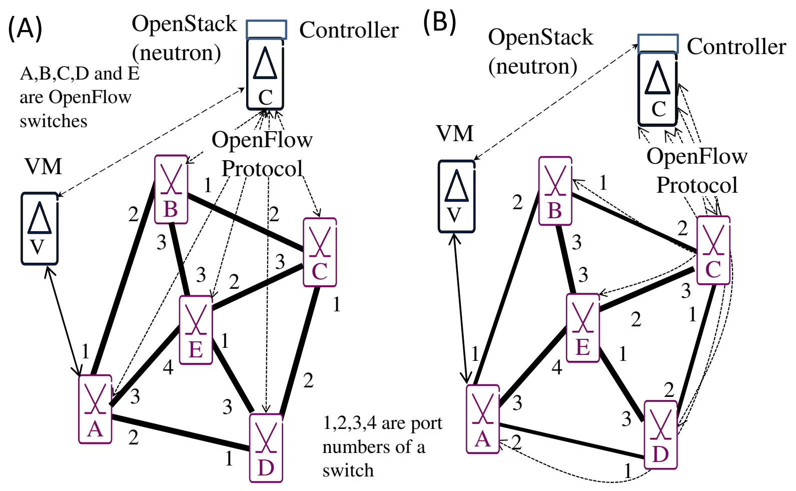
3.4 Out-of-band or in-band verification

Out-of-band means that control traffic is sent on a separate channel (or network). In-band means that control traffic is sent on the same infrastructure as the data plane [16]. As described earlier, the controller can perform all the steps of verification (Section 3.3) using out-of-band or in-band networks (without the VM part in Fig. 3.6A and Fig. 3.6B). However, in this case, the bandwidth requirement of the control network will increase due to the transmission of the large number of test packets for verification. Furthermore, if an in-band network is used (Fig. 3.6B) for the controller communication, the bandwidth requirement of the link between the controller and the switch (switch C in Fig. 3.6B) which connects the data network with the controller, will increase substantially as this link will be used for the communication between many switches and the controller. Nevertheless, this also increases the computational requirement of the controller to perform verification. Therefore, we propose that the steps – test packet generation and matching-error identification – which increase the computational and bandwidth requirement of the controller significantly, can be performed by a VM.

Instead of using additional controllers, we propose to use light-weight VMs for distributing the load of the controller. This is because the resources of a VM can be released once verification is completed, whereas in case of using additional controllers, these controllers may need to be present in the network all the time just for verification. This may be a wastage of the controller resources in the network.

For performing verification through VMs, we assume that there are some servers in the network that have capability to create VMs [9]. However, if a server has a capability to transmit test packets and can perform the verification activity, the server can be used directly for verification. In emulation, the controller creates VMs, makes a connection between a VM and a switch, and establishes paths between switches and the VM (Fig. 3.6). Currently, OpenStack neutron with the OpenDayLight [17] or Floodlight controllers [18] defines the API to create VMs from the controller.

We propose that the VM would be part of the control plane and should establish an OpenFlow session with the switches in order to perform the verification activity. As not every switch is directly connected to a cluster of servers capable of hosting VMs (we assume this to be only possible for some switches, which have for example a direct data center link), the VM needs to establish an OpenFlow session



(A) Out-of-band network for controller communication (B) In-band network for controller communication

Figure 3.6: VM connections with switches

path with these switches through other switches in the data plane. In this proposal, the VM works like an additional controller in the network. This is proposed because of the following two capabilities of a controller application:

1. The capability to generate test packets with an ingress port as a matching field
2. The capability to verify two or more switches at the same time.

The first capability is required because the Flow-Match Header part of a Flow Entry may contain an ingress port as a matching field [12]. Without having the first capability, the VM may need to send test packets through additional links in order to match the Flow Entry with the test packets, requiring more bandwidth for verification. This can be explained through Fig. 3.6. Suppose that switch A has a Flow Entry, containing port 1 (i.e., port AB) in the ingress port of the Flow-Match Header. For verifying this entry, the VM needs to send a test packet to switch A through path VADCBA. This requires bandwidth to be reserved in links AD, DC, CB, and BA for verification. However, if there is an OpenFlow session between the VM and switch A through path VA, the VM can directly send a test packet to switch A through link VA in the form of a packet out message that contains port 1 (or any port) in its ingress port field. This decreases the bandwidth required for verification.

The second capability is required because it is possible that VM needs to verify two or more switches at the same time to decrease the verification time. However, if there is no OpenFlow session between the VM and switches, the verification of two or more switches at the same time may be difficult to implement or if it is implemented, the verification time will increase significantly. This can also be explained through Fig. 3.6. In Fig. 3.6, the VM transmits test packets to B through A, as switch A is the only node connecting VM to the switch topology. If the VM wants to send the test packets to switch A and B simultaneously, switch A should have a way to distinguish which test packets are sent to it and which test packets are sent to switch B. This may be possible if the test packet specific entries are present in A to forward the test packets of switch B. However, as these entries are first needed to be verified before the entries in switch B, it will increase the verification time. However, if there is OpenFlow sessions between the VM and switches, the VM can send test packets to switch A and B at the same time by sending them through their OpenFlow session paths, decreasing the verification time.

3.4.1 OpenFlow session path selection in in-band networks

Suppose that the OpenFlow session paths between a VM and switches are along the data plane switches and contain the same subset of data plane links. In this case, as the VM can verify one or more switches at the same time (second capability, described above), the bandwidth requirement of these subset of links will increase significantly for verification. In addition, if bandwidth for verification is limited for a link, some switches have to wait for other switches to complete verification. This will increase the verification time, as some switches have to wait for other switches in the verification path to complete verification. Therefore, we propose to distribute the load of verification into many links (load balancing/load distributed approach) and therefore, the verification paths for different switches does not contain the same subset of links, decreasing the waiting time of switches.

For the load distributed approach, we assume that there is limited bandwidth (let say b) available in each switch link for verification. This bandwidth is assumed to be equal for each link. Furthermore, as the VM links (the link between VM and switch A in Fig. 3.6) are the links which will be used to create OpenFlow session paths for many switches, we assume that the VM links have enough bandwidth (equal to the bandwidth required to verify multiple switches at the same time) for verification.

We consider a network of N switches denoted by v_i connected through a link set e_{ij} and link cost c_{ij}, where e_{ij} is the link incident to nodes v_i and v_j, and $c_{ij}(\geq 1)$ is the cost associated with e_{ij}. In addition, VM has a direct connection with m $(1 \leq m \leq N)$ switches in the network. The edge between the VM and switch k is denoted by e_{vk} and the cost of the link between the VM and switch k is denoted by c_{vk}. The cost of each link (switch links and the link between the VM and a switch k) is 1 by default. However, the cost of a link c_{ij} including the VM links c_{vk} will increase depending on the load on the link due to selection of the link for OpenFlow session paths. Let P represents a path between the VM and switch t, where

$$x_{ij} = \begin{cases} 1 & \text{the path P traverses } e_{ij} \\ 0 & \text{otherwise} \end{cases}$$

We assume that P is a path that contains no cycle. This path is not required to be the shortest path between VM and t pair. In our mechanism we choose a path for establishing an OpenFlow session for which the cost $\sum\limits_{i,j} c_{ij}x_{ij}$ is minimal. If a session path for a switch is selected, which goes through a link e_{ij} (including a VM link), the cost of that link (e_{ij}) will increase by the load that the session path will generate on the link. This load is represented by the number of test packets that the session path needs to transmit for verifying Flow Entries. The number of test packets is equal to the number of packet-headers that a partially wildcarded field of Flow Entries in the switch can match (i.e., $\sum\limits_{\forall f} F_f$, where F_f is the number of different packet-headers that the Flow-Match Header of Flow Entry f can match). Suppose that the maximum value of the number of test packets that an OpenFlow session transmits is F_{Max}. Then, the cost of link e_{ij} will increase by:

$$c_{ij} = c_{ij} + \frac{\sum\limits_{\forall f} F_f}{F_{Max}} \tag{3.1}$$

For selection of a path to a switch, this cost will be taken into account.

3.4.2 VM placement

For VM placement, we rely on “betweenness centrality” [4] which is the number of paths from all nodes to all other nodes that crosses a given switch. We believe that this is a good indication where the VM should be placed in the network. Because if we place a VM in a switch (i.e., a direct connection between the VM and the switch) with the highest “betweenness centrality”, the VM will become close to many switches and OpenFlow session paths of many switches may contain different links. In this case, due to the second capability (discussed above), switches can be verified in the limited time. For example, if a VM is placed at switch E (switch E has the highest “betweenness centrality” in Fig. 3.6), the VM

[4]Betweenness centrality of node v can be expressed mathematically by: $\sum_{s\neq v\neq t} \frac{P_{st}(v)}{P_{st}}$. Here where P_{st} is the total number of shortest paths from node s to node t and $P_{st}(v)$ is the number of those paths that pass through v.

will become close to each switch and the session paths for switch A, B, C, D will contain different links from switch E, decreasing the verification time of these links for verification. However, in a network, there can be only some switches that have capability to place a VM (i.e., having a direct connection). Therefore, for VM placement, we consider only those switches to calculate the ”betweenness centrality” and place the VM in the switch that has the highest “betweenness centrality”.

Using our emulations, we show that the “betweenness centrality” is a good indication of the placement of VMs. The future work is to place more than one VM in the network such that verification time can be reduced. For this, we can partition the network into k partitions and place the VM in each partition with the switch having the highest “betweenness centrality”.

3.5 Emulation

We performed emulations on the Fed4fire testbed facility at iMinds [10]. The testbed is a generic test environment for advanced network, distributive software and service evaluation. It consists of 100 physical nodes interconnected by a non-blocking 1.5 Tb/s Force10 Ethernet switch. Each node has 4 CPU cores and 4 GB RAM. We generated emulated pan-European topologies using the nodes of the testbed and performed extensive verification experiments.

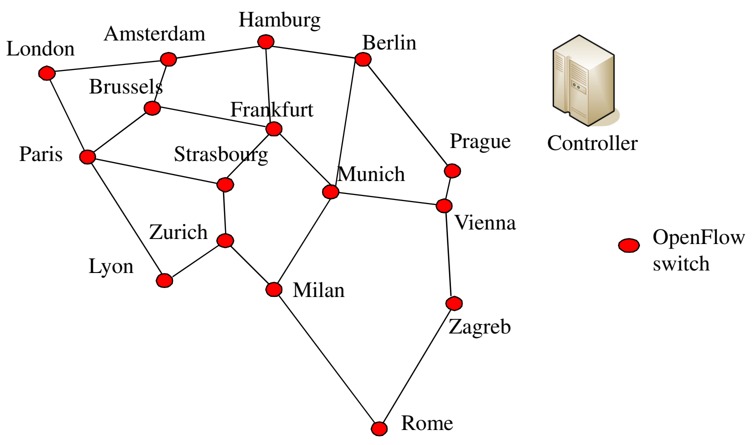


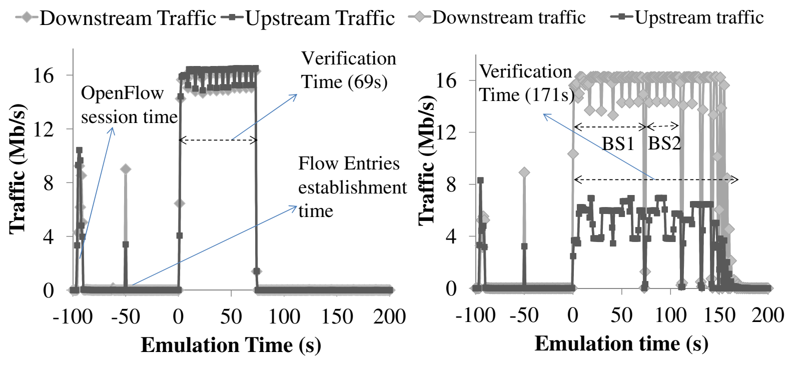
Figure 3.7: Pan European Topology

Fig. 3.7 shows one of the emulated pan-European topologies that contains 16 OpenFlow switches connected with each other in a mesh fashion. Each switch of the topology has also provided a dedicated interface to a switched Ethernet LAN (not shown in Fig. 3.7), which establishes an out-of-band connection with a single controller (the controller is shown in Fig. 3.7).

We perform three different ranges of experiments - (1) (out-of-band) controller-induced verification, (2) (in-band) VM-induced verification, (3) validation on multiple topologies - in our testbed. Each of these will be handled in more detail in the following subsections. For our emulation, we implemented the proposed verification mechanism in the Floodlight controller that uses OpenFlow version 1.3 [18] in its implementation. In addition, we used Open vSwitch [15] for running OpenFlow in the switches of the emulated topology. In the experiments, software matching errors are emulated by translating some packets of a flow to incorrect headers. In our emulations, we find these errors and find verification time. In all the experiments, multiple switches are verified at the same time.

3.5.1 Controller-induced verification experiment

In the controller-induced experiment, the controller makes an out-of-band connection with switches and performs all the steps of verification. Flow-Match Header errors are detected either via the packet-reception or via the binary-search method.



(A) Packet-Reception Method (B) Binary-Search Method

Figure 3.8: Traffic on the controller link (controller verification experiment)

Fig. 3.8 shows the emulation methodology using traffic captured on the controller when the binary-search or packet-reception method is used for verification. Traffic is shown from second -100 to 200, when 1 Mbps is available in between each switch and the controller for verification. In emulation, each switch in the emulated pan-European topology first establishes an OpenFlow session over the TCP session with the controller. The spikes in Fig. 3.8 from -100 to -90 seconds are due to traffic exchanged between the controller and switches to establish OpenFlow sessions. At emulation time -50 seconds, the controller adds

Flow Entries in each switch to forward data traffic. The Flow Entries contain the ingress port, source IP address and destination IP address as matching fields. The source and destination IP addresses are 24 bit addresses and therefore, the controller needs to transmit 65536 different test packets for verification. However, due to the generated bug, each Flow Entry is not able to match correctly 256 flows out of 65536 flows. At second 0, the controller starts verification of the Flow Entries. In Fig. 3.8, traffic generated due the verification of one Flow Entry is shown.

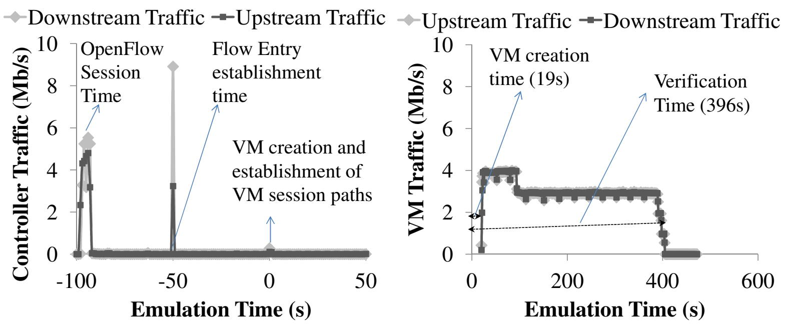
Fig. 3.8 shows that the verification time (time to find matching errors) using the packet reception method is 69 seconds (Fig. 3.8A) and using the binary-search method is 171 seconds (Fig. 3.8B). Fig. 3.8A shows that downstream and upstream traffic due to the packet-reception method are about 16.2 Mb/s. Fig. 3.8B shows that there is about 16 Mb/s downstream traffic and about 5 Mb/s upstream traffic using the binary-search method. In this method, downstream traffic is present because the controller sends test packets in the downstream direction in the form of packet-out messages to verify the Flow Entries of switches. The TCP session in switches then sends the acknowledgments of the sent traffic in the upstream direction. The upstream traffic in the binary-search method (Fig. 3.8) includes this acknowledgment traffic. Additionally, it includes the traffic sent by the switches in sending counters of Flow Entries. However, the total traffic in the downstream and upstream direction using the packet-reception method is only 0.2 Mb/s more than the traffic in the downstream direction using the binary-search method (i.e., 16 Mb/s). It means that there is only 0.2 Mb/s additional traffic generated by TCP for acknowledgments of test packets in the packet reception method. This traffic is very small compared to the acknowledgment traffic in the binary search method (5Mb/s). This is because of the push ACK functionality [19] of TCP in which TCP sends data in the acknowledgments of packets.

Fig. 3.8B also shows the iterations (BS1 and BS2) of the binary-search method. As the counter read interval of Open vSwitch in our implementation is 2 seconds, we see a downstream spike of 2 seconds after BS1, BS2 and so on.

3.5.2 VM-induced verification experiment

In the VM-induced verification, the controller triggers the creation of a VM (light weight Linux container) and connects it with one of the switches in the network (Hamburg in Fig. 3.7). We use the packet-reception method for calculating the matching errors in the network. We compare the verification time when our approach (i.e., load balancing approach, Section 3.4.1) or the shortest path approach is used to select OpenFlow session paths between switches and VM.

We now explain the emulation methodology of the VM-induced verification experiment using the traffic captured on the controller and the VM link. Traffic is



(A) Traffic on the controller link (B) Traffic on the link between VM and Hamburg

Figure 3.9: Traffic Intensity (VM-induced verification experiment)

shown when the VM is created in the Hamburg switch of the emulated topology (Fig. 3.7) and verification uses data plane links for transmitting or receiving test packets. The emulation methodology of this experiment is same as the controller verification experiment (Fig. 3.8) in which switches establish OpenFlow session paths with the controller from second -100 to -90 and at second -50, the controller adds Flow Entries in switches to forward data traffic. At second 0, the controller starts verification by copying Flow Entries to another table, creating a VM at the Hamburg switch, and establishing Flow Entries for making OpenFlow sessions between the VM and switches. We see small spike in Fig. 3.9 at second 0 due to this traffic.

In our emulation, the controller creates a VM using the RPC (Remote Procedural Call) commands. Fig. 3.9B shows that the time to create a VM is about 19 seconds. After creating the VM, the VM establishes OpenFlow session paths with all the switches in the network and starts verification. For verification, we reserved 1 Mb/s bandwidth in each switch link. In our mechanism, the controller controls the rate of test packets according to the bandwidth available in each link of its OpenFlow session. Additionally, the bandwidth between VM and the Hamburg switch is kept as 4Mb/s. Fig. 3.9B shows that the total verification time of a Flow Entry is 396 seconds.

3.5.3 Validation on multiple topologies

In the validation of multiple topologies experiments, different topologies are used for verification. The topologies are: core topology (CT), Basic Reference Topology (BT), and Ring topology (RT). The difference between BT and CT is that BT contains more switches than CT. BT contains 28 switches and CT contains

16 switches. The difference between BT and RT is in the degree of meshedness. RT has a lower meshedness than BT. All the other details of the topologies can be found in [16]. In these experiments, the VM is connected with one of the switches in the network in different experiments and the verification time is evaluated for each placement of the VM. In a network, there may be only some switches that have a direct connection with the VM (or the data center hosting the VM). However, for the completeness of the multiple topology experiments, we assume that each switch in the considered topology has this capability.

3.6 Results

In this section, we present the results gathered by performing all the experiments.

3.6.1 Controller-induced verification experiment

Fig. 3.10 depicts the verification time when the bandwidth between the controller and switches are varied. For transmitting test packets, the controller sets the rate of the test packets according to the bandwidth available between the controller and switches for verification. In case of the binary-search method, the verification time is calculated by subtracting the time when the last counter read reply message is received by the controller with the time when the first counter read request message is sent by the controller. In case of the packet-reception method, the verification time is calculated by subtracting the time when the first packet-out containing a test packet is sent by the controller with the time when the last packet-in containing a test packet is received by the controller. All the results are taken 50 times and the average is shown in Fig. 3.10.

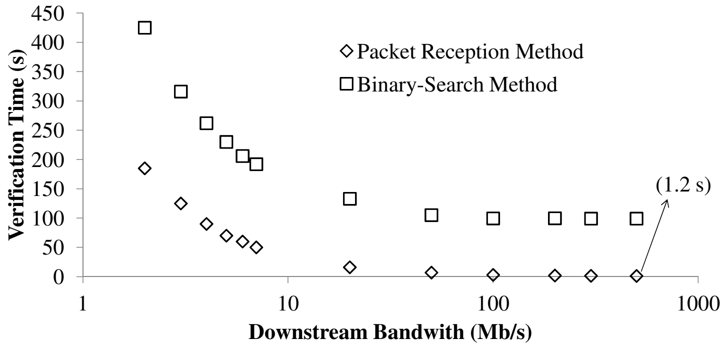


Figure 3.10: Verification time using the packet-reception and binary-search method (controller out-of-band network scenario)

As expected, Fig. 3.10 shows that the verification time of the Flow Entries decreases with the increase in the bandwidth reserved for verification. In this experiment, five Flow Entries are verified in each switch for Flow-Match Header issues. Fig. 3.10 also shows that the verification time in the binary-search method is longer than the verification time in the packet-reception method. This is because the binary-search method sends more number of test packets for verification through binary search iterations, leading to increase in the verification time. In addition, as the counter update interval is 2 seconds in our emulation, it further increases the verification time in the binary-search method. Furthermore, the binary search method performs the verification of the Flow Entries one by one. However, the packet-reception method can perform the verification all the Flow Entries at the same time, leading to decrease in the verification time. For the packet reception method, the minimum value of the verification time is 1.2 seconds in our emulations.

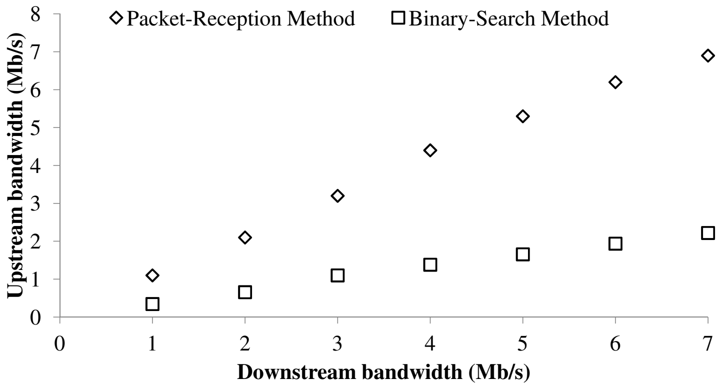


Figure 3.11: Upstream bandwidth usage (controller out-of-band scenario)

Fig. 3.11 shows that the upstream bandwidth usage in the packet-reception method is higher than the upstream bandwidth usage in the binary search method. This is because the packet-reception method sends back the matched test packets to the controller, leading to increase in the bandwidth requirements in the upstream direction. Furthermore, the binary search method drops all the test packets matched with a Flow Entry. However, we see in Fig. 3.11 that when the downstream traffic increases, the traffic in upstream direction due to the binary-search method also increases. These are due to the acknowledgments of test packets sent in the upstream direction by TCP.

Fig. 3.10 and Fig. 3.11 show the results when there are only 5 wildcarded Flow Entries (wildcards in the last 8 bits of source and destination IP address) in each switch for verification. In this case, verification is completed in limited time. However, when the number of Flow Entries is increased in switches, the verification time will increase significantly. Fig. 3.12 shows the results when the number of Flow Entries is increased from 5 to 400. In this experiment, 5 Mb/s of bandwidth is available in between each switch and the controller for performing verification.

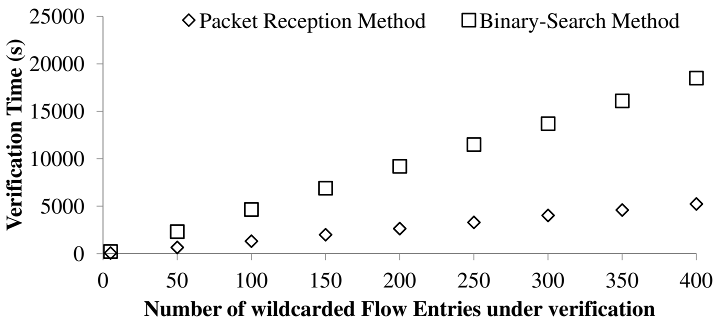


Figure 3.12: Verification time when 5 Mb/s bandwidth available in each controller link for verification

Fig. 3.12 illustrates that the verification time becomes significantly long as the number of Flow Entries increases. This is because if more Flow Entries are present for verification, more test packets are required to be sent to the switches. As the bandwidth is limited, the controller needs to wait long time to send all the test packets, increasing the verification time.

We see that the binary-search method leads to long verification time as compared to the packet-reception method (Fig. 3.10 and Fig. 3.12). However, as the bandwidth requirements of the binary-search method in the upstream direction are less compared to those of the packet-reception method (Fig. 3.11), the binary-search method can be used when upstream bandwidth is a bottle-neck.

3.6.2 VM-induced (in-band) verification experiment

For in-band verification, we assume that different queues are installed in OpenFlow switches for control and data traffic. Configuration of these queues is described in [16]. As traffic flows using different queues do not interfere with each other [16], we do not consider data traffic in our experiments.

Fig. 3.13 shows the results of the experiments when the bandwidth of each switch links is limited between 0.5 to 5.5 Mb/s (value is shown in the figure) for verification. The emulation methodology of these experiments is provided in Section 3.5.2. In these experiments, the VM makes an in-band connection with switches in the network and sets the rate of the test packets according to the verification bandwidth available in all the links along the OpenFlow session

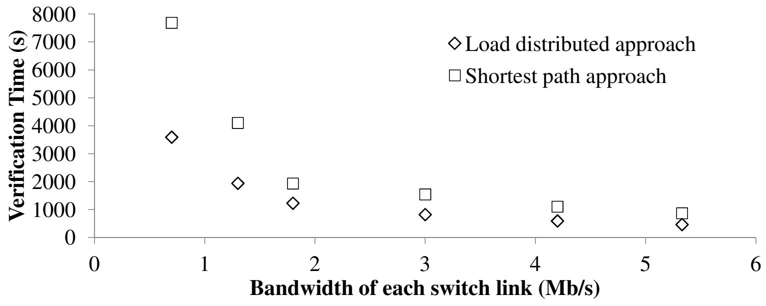


Figure 3.13: Limited bandwidth scenario (VM in-band control scenario)

paths. As the paths for verification for some of switches is through other switches in the network, the switches may have to wait for verification until the intermediate switches in the path perform verification. This leads to increase in the verification time in the VM-induced verification experiment as compared to the controller verification experiment (Fig. 3.10) in which the controller makes an out-of-band connection with the switches. However, an out-of-band control network is not always possible (for example, in widely distributed central offices in access networks). In this case, the controller may itself need to communicate with switches in the network using in-band control paths [16] (Fig. 3.6B). Even if an out-of-band control network is present, there may not be enough bandwidth in this network to perform verification. Therefore, in this case verification traffic may need to send through the data plane links by reserving some bandwidth over the data network for verification.

We compare the results of the load distributed/balancing approach (discussed in Section 3.4.1) with the shortest path approach (Fig. 3.13). In this experiment, the bandwidth is limited and therefore, if the same subset of links are used for verification paths of many switches (most probable case in the shortest path approach), it will increase the verification time (explained in Section 3.4). Fig. 3.13 shows that the verification time using the load distributed approach is shorter than the verification time using the shortest path approach (even though the path is shortest). This is because using the load balancing approach, the load is distributed among all the nodes in the network and the same subset of links are not used for many OpenFlow session paths. This leads to decrease in the waiting time of switches for performing verification. Additionally, it shows that if the bandwidth in each switch link for verification increases, verification can be performed in limited time.

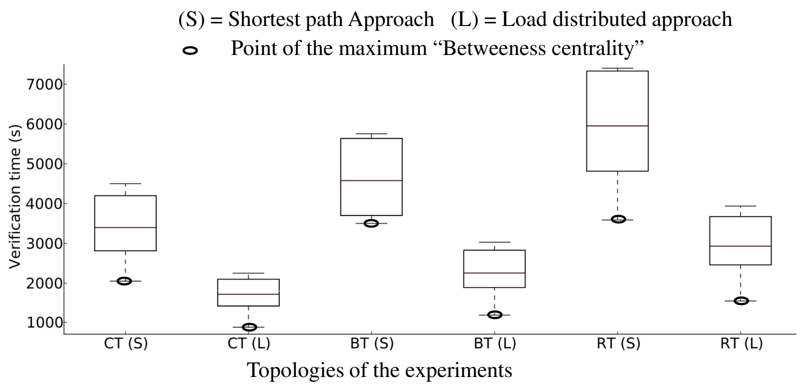


Figure 3.14: Verification time and switch centrality

3.6.3 Validation on multiple topologies

This experiment will evaluate how the verification time varies with the placement of a VM in a network for a range of different topologies, given that the bandwidth of each link between switches available for verification is restricted 1Mb/s. Fig. 3.14 shows the minimum value, lower quartile, median, upper quartile and the maximum value of the verification time when the VM is placed at the different locations. It shows that if the VM is not placed at the correct switch, the verification time could be as worse as the maximum value. Fig. 3.14 also confirms that the verification time is minimal for all topologies when the VM is placed at the switch containing the maximum value of "Betweenness centrality". Additionally, we see that the verification time has the order $CT < BT < RT$. This is because BT contains more switches than CT and hence, the VM needs to verify more switches, leading to an increase in the verification time. Additionally, as RT has a lower meshedness than BT, there can be many overlapping session paths links in RT than BT, which leads to an increase in the verification time in RT. Furthermore, as the bandwidth is limited, we see that the load distributed approach performs better than the shortest path approach.

3.7 Conclusions

In this article, we have proposed a mechanism for the verification of the Flow-Match Header part of Flow Entries in OpenFlow switches. The proposed mechanism might be executed within the OpenFlow controller(s), or within external devices or servers (e.g., on VMs), using the existing data network (in-band verification) or using an external control/verification network (out-of-band). The proposed mechanism was evaluated in extensive emulation experiments. The results illustrated that the verification time depends on the bandwidth available in the network for verification. If bandwidth is unlimited, verification can be achieved in a very short time interval. However, if bandwidth limitations exist, the verification time might increase significantly. Therefore, we proposed a load balancing approach to distribute the load induced by verification traffic among many links in the network. Our results indicate that the approach performs better compared to a load-agnostic shortest path strategy when limited bandwidth is available. Additionally, we evaluated the relationship between verification time and the placement of the verification functionality in the network (i.e., VM placement). Experiments validated that placing the verification functionality close to or at the node with maximal “betweenness centrality” is beneficial with respect to reducing the verification time of the entire process.

Acknowledgment

This research has received funding from the EU FP7 under agreement n^o 619609 (UNIFY).

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *OpenFlow: En- abling innovation in campus networks*, SIGCOMM Comput. Com- mun. Rev., vol.38, no.2, pp.6974, 2008.

[2] Bugs in OpenFlow switches [Online]. Available: http://osrg.github.io/ryu/certification.html.

[3] D. Kreutz, F.M.V. Ramos, P.E. Verissimo, C.E. Rothenberg, S. Azodolmolky, and S. Uhlig, *Software-defined networking: A com- prehensive survey*, Proc. IEEE, vol.103, no.1, pp.1476, 2015.

[4] P. Kazemian et al., *Header space analysis: Static checking for networks*, NSDI, 2012.

[5] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P.B. Godfrey, and S.T. King, *Debugging the data plane with anteater*, Proc. ACM SIGCOMM, pp.290301, 2011.

[6] A. Khurshid, W. Zhou, M. Caesar, and P.B. Godfrey, *Veriflow: Verifying network-wide invariants in real time*, SIGCOMM Comput. Commun. Rev., vol.42, no.4, pp.467472, 2012.

[7] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, *Automatic test packet generation*, IEEE/ACM Transaction Networking., vol.22, no.2, pp.554566, 2014.

[8] S. Sharma, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester, *Verification of aggregated flows in OpenFlow networks*, 2015 IEEE Conference on Computer Communications Workshops (IN- FOCOM WKSHPS), pp.78, 2015.

[9] A. Csaszar, W. John, M. Kind, C. Meirosu, G. Pongracz, D. Staessens, A. Takacs, and F.-J. Westphal, *Unifying cloud and car- rier network: EU FP7 project UNIFY*, 2013 IEEE/ACM 6th Inter- national Conference on Utility and Cloud Computing, pp.452457, 2013.

[10] M. Berman, P. Demeester, J.W. Lee, K. Nagaraja, M. Zink, D. Colle, D.K. Krishnappa, D. Raychaudhuri, H. Schulzrinne, I. Seskar, and S. Sharma, *Future internets escape the simulator*, Commun. ACM, vol.58, no.6, pp.7889, 2015.

[11] K. Suzuki, K. Sonoda, N. Tomizawa, Y. Yakuwa, T. Uchida, Y. Higuchi, T. Tonouchi, and H. Shimonishi, *A survey on Open- Flow technologies*, IEICE Trans. Commun., vol.E97-B, no.2, pp.375386, 2014.

[12] ONF OpenFlow features and specifications [Online]. Available: www.opennetworking.org.

[13] HP procurve 5400 zl series [Online]. Available: http://archive.openflow.org/wp/wp-content/uploads/2011/04/HP_Procurve_-OpenFlow_support.pdf.

[14] D.Y. Huang, K. Yocum, and A.C. Snoeren, *High-fidelity switch models for software-defined network emulation*, Proc. Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking HotSDN13, pp.4348, 2013.

[15] Open vSwitch [Online]. Available: http://openvswitch.org/.

[16] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, *In-band control, queuing, and failure recovery functionalities for OpenFlow*, IEEE Network, Vol. 30(1), pp. 116-124, 2016.

[17] OpenStack interface [Online]. Available: https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/opendaylight-an-open-source-sdn-for-your-openstack-cloud.

[18] Floodlight Controller [Online]. Available: http://sdnhub.org/releases/floodlight-plus-openflow13-support/.

[19] TCP RFC [Online]. Available: https://www.ietf.org/rfc/rfc793.txt.

4 In-Band Control, Queuing, and Failure Recovery Functionalities for OpenFlow

This chapter investigates bootstrapping, queuing, and fast failure recovery techniques for in-band OpenFlow networks in which control traffic (traffic to or from the controller) is sent on the same infrastructure (or the same network) used to transport data traffic. For bootstrapping, we propose a method with which OpenFlow devices can be bootstrapped without having any manual configurations. This chapter provides a brief description of the method, while Appendix A gives a detailed description. For queuing, we extend the queuing functionality of OpenFlow to add queues with different priorities. We used this extended queuing functionality for adding separate queues for control and data traffic in in-band networks and by serving the control traffic queue before the data traffic queue. For fast failure recovery, we propose restoration and protection techniques for control traffic, while utilizing the previously proposed restoration and protection schemes of out-of-band networks for data traffic (Chapter 2). In addition, prototyping details for a wide range of OpenFlow implementations are presented. Moreover, extensive experiments are performed to measure the suitability of the proposed techniques for OpenFlow.

⋆⋆⋆

Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester

Published in IEEE Network Magazine, January 2016.

Abstract In OpenFlow, a network as a whole can be controlled from one or more external entities (controllers) using in-band or out-of-band control networks. In this article, we propose in-band control, queuing, and failure recovery functionalities for OpenFlow. In addition, we report experimental studies and practical challenges for implementing these functionalities in existing software packages containing different versions of OpenFlow. The experimental results show that the in-band control functionality is suitable for all types of topologies. The results with the queuing functionality show that control traffic can be served with the highest priority in in-band networks and hence, data traffic cannot affect the communication between the controller and networking devices. The results with the failure recovery functionality show that traffic can be recovered from failures within 50 ms.

4.1 Introduction

In recent decades, the Internet has grown from being an experimental research network to a broadband commercial platform. At the same time, the Internet has been facing many technical challenges such as complexity and inflexibility to meet changing requirements. To solve these challenges, numerous research activities such as the Clean Slate Internet program [1] and the Future Internet [2] have been started to develop appropriate solutions. A major outcome of the former is the idea of decoupling the control plane from the data plane in Internet devices (e.g. switches/routers) and embedding the control plane into one or more servers, called controllers. This enables independent evolution of the control and data plane. In addition, an interface between the data and control plane has been proposed. The most prominent protocol implementing such an interface is the OpenFlow protocol [3].

The current research of OpenFlow focuses mainly on an out-of-band network (Fig. 4.1a) in which control traffic (traffic to or from the controller) is sent on a separate network [4]. Such an out-of-band network has the following main advantages:

- High security is provided for control/management information because a separate network is used for communication.
- Access to the switches is possible through the separate network even if there are failures in the data traffic paths.

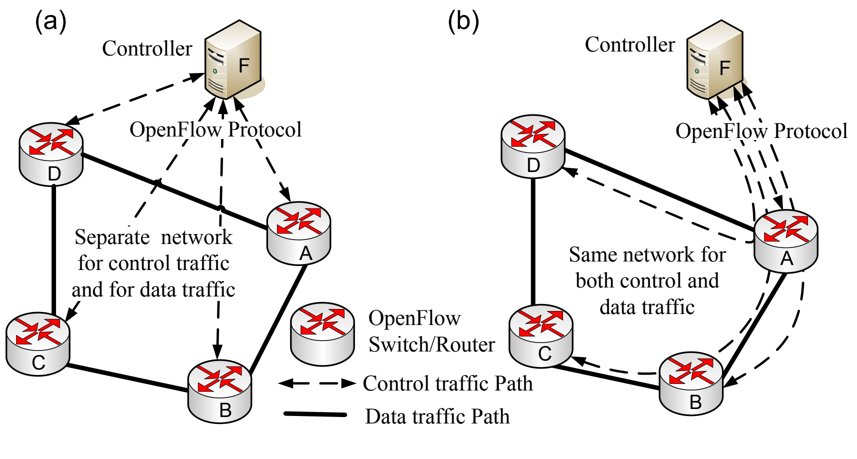


Figure 4.1: OpenFlow network: (a) out-of-band (b) in-band control.

However, these networks are expensive to build due to the requirement of a separate network. Also, building a separate network may not be feasible in some scenarios (e.g. widely distributed central offices in access networks).

To solve the above problems, OpenFlow is required to be implemented for an in-band control network (Fig. 4.1b) in which control traffic is sent on the same infrastructure as the data plane [5]. However, for such a network OpenFlow does not describe how a switch can establish a communication path (e.g. control traffic path in Fig. 4.1b) with the controller. Without configuring these paths automatically, operators may face many manual configuration problems such as going into the field to configure the switches. In this article, we implement a method (known as bootstrapping) that inserts this information automatically in in-band networks. We refer to this as *in-band control functionality*.

In in-band control networks, control traffic may compete with data traffic for network resources (e.g. bandwidth) [6] as both share the same infrastructure. Therefore, due to an increase in data traffic, the control plane operations (such as new service establishment, failure recovery, load sharing) may suffer significant delay, and the controller and switches may even disconnect. To solve these problems, we extend the in-band functionality by implementing separate queues for control and data traffic, and by serving the control traffic queue before the data traffic queue. We refer to this mechanism as *queuing functionality*.

In in-band control networks, failures in the data plane (switch or link failures) can affect both data and control traffic. As a loss in data traffic causes a disruption of service, and a loss in control traffic prevents any new service establishment from the switches affected by failures, failure recovery is important for both data and control traffic. For failure recovery, some networks offer carrier-grade quality (RFC 5654), meaning that a network should recover from failures within 50 ms. Therefore, we explore two well known techniques, restoration and path protection, for fast failure recovery of both control and data traffic. In restoration, an alternative path is established after a failure. In path protection, a disjoint alternative path is established before a failure, and when the failure is detected, traffic is redirected to the alternative path. For failure detection in restoration, loss-of-signal (LOS) can be used because it can detect failures in any forwarding port. However, as LOS cannot detect failures in any path in protection, bidirectional forwarding detection (BFD) (RFC 5880) can be used.

We proposed in-band control and failure recovery functionalities in [7] and [8] respectively. In this article, we extend these functionalities with queuing functionality and integrate BFD in OpenFlow switches for fast failure recovery. In addition, we report practical challenges for implementing these in existing OpenFlow packages, containing different OpenFlow versions. Furthermore, we implement these functionalities in one of the OpenFlow software packages and perform extensive experiments. The experiments with in-band control show that the implemented method is suitable for all types of topologies. The experiments with queuing show that data traffic does not affect the communication between the controller and switches, and the experiments with failure recovery show that carrier-grade quality can be achieved in OpenFlow.

The following section describes our proposed functionalities, the third section describes practical challenges, the fourth describes experimentation, and the final section concludes the article.

4.2 Functionalities for OpenFlow

4.2.1 In-band control functionality

For in-band control, each switch and the controller have to establish an OpenFlow session over a transport layer protocol such as TCP, SCTP, or UDP. As switches and the controller need a reliable connection between each other, TCP or SCTP are preferred over UDP. In addition, as not all the platforms support SCTP, TCP is mostly used for establishing sessions.

The method for in-band control may differ with the types of OpenFlow switches used in the network. Today, there are two types of OpenFlow switches: pure and hybrid [9]. Pure switches support only OpenFlow operations for forwarding packets. Hybrid switches support both OpenFlow and traditional switching operations (e.g. layer 2 Ethernet switching, layer 3 routing, and VLAN isolations) for forwarding packets, and are common with many manufacturers such as Brocade, Juniper, and Cisco. We implement a loop free in-band control method using hybrid switches. In this method, at the time of bootstrapping, a switch applies Ethernet switching operations to forward its own traffic and applies OpenFlow operations to forward control traffic of other switches.

We frame the following three challenges for implementing in-band control:

- Each switch needs to configure a unique IP address for itself.
- Each switch needs to know the IP address and transport layer port (e.g. TCP port) of the controller.
- Communication paths need to be established for the switches (B, C, and D in Fig. 4.1b) that are not directly connected with the controller.

To solve the first and the second challenge, the followings are performed:

- Each switch runs a DHCP (Dynamic Host Configuration Protocol) client.
- Each switch runs a hybrid stack to forward its own traffic (e.g. DHCP traffic).
- The DHCP server is configured with a vendor-specific option containing the IP address and transport layer port of the controller.
- Either the DHCP server is located on the controller or the DHCP server and the controller share IP (sub) networks (direct communication) with the same switch switch A in Fig. 4.1b).

To solve the third challenge, the controller establishes communication paths through the switches that have already established OpenFlow sessions.

Our in-band control method using DHCP solves the problems of configuring each switch with a unique IP address and other transport layer parameters (e.g. TCP port) of an OpenFlow session.

The four steps to perform in-band control are:

1. Notification of the required network parameters.
2. Establishment of a TCP session.
3. Establishment of an OpenFlow session.
4. Discovering the topology.

For the first step, each switch periodically sends DHCP messages to its neighbors until it receives a reply from the DHCP server. If a neighbor is the DHCP server, it replies to the switch (A in Fig. 4.1b). Otherwise, the neighboring switch may forward or drop the messages, depending on whether it has an OpenFlow session with the controller. In case the neighboring switch has the session, the controller allows the neighboring switch to forward the DHCP messages to the DHCP server.

When a switch knows its IP address and the IP address of the controller (using DHCP), it runs ARP to know the MAC address of the controller. After knowing the MAC address, the switch performs the second step.

In the second step, the switch establishes a TCP session with the controller. Either it establishes the session directly (in case of switch A) or the controller specifies a session path (shortest path) through the switches having an OpenFlow session.

In the third step, the switch instantiates an OpenFlow session with the controller [9].

In the fourth step, the controller discovers links of a switch after establishing the session with it. For this, the controller allows the switch to flood probe messages (Link Layer Discovery Protocol messages). From the received messages, the controller discovers links of the switch [10]. In addition, the controller discovers links of DHCP clients (for switches B, C, and D) and the DHCP server on reception of DHCP messages from them, and the same flooding mechanism (as probe messages) is exploited to infer the location of the DHCP server. In this case, instead of probe messages, DHCP messages are flooded.

4.2.2 Queuing functionality

A part of queuing functionality, i.e. the creation of queues, is out-of-scope for OpenFlow. However, with the OpenFlow protocol, a packet can be redirected through an already created queue. For the creation of queues, switches can rely on a separate protocol such as OF-Config (OpenFlow Configuration and Management Protocol) or OVS-DB (Open vSwitch Database Management Protocol) [11]. For the case when switches do not support these protocols, vendor specific options of the OpenFlow protocol can be used for the creation of queues. Many switches such as HP, Reference, Indigo, Trafficlab1.1, and Trafficlab1.3 (Table 4.1) allow queue creation through vendor-specific options. However, with these options, only a few types of queues (such as rate limiting queues) can be created. In this article, we propose to extend the vendor-specific option of switches to create queues with different priorities. In our proposal, the queue creation message of the vendor-specific option is extended to add a priority number, and therefore, on reception of this message, a switch can create queues having different priorities using switch traffic control commands (e.g. Linux traffic control commands in the Reference, Trafficlab1.1, and Trafficlab1.3 switches).

In queuing functionality, the aforementioned extension is used for creating different queues for control and data traffic. The control traffic queue is given the highest priority, and hence is served before any other queue. When all switch port information is received, the controller creates the control traffic queue on each port of the switch. For data traffic, the controller can create queues either in advance or on reception of data traffic. In addition, when the controller receives traffic to define its forwarding, the controller adds a forwarding entry to redirect control traffic to the control traffic queue and data traffic to the data traffic queue. For separating control and data traffic, the controller uses the source IP address, destination IP address, and transport layer parameters of an OpenFlow session in a forwarding entry of control traffic.

4.2.3 Failure recovery functionality

In OpenFlow, a switch sends an echo-request to the controller after an idle timeout. If it does not receive an echo-reply before an echo timeout, it declares failures. The switch then tries to establish a new session. If it fails, it waits for a backoff timeout to re-establish the session. As the minimum value of idle, echo, and backoff timeouts are 1 second, failure recovery cannot be achieved in milliseconds. Therefore, we implement two fast recovery techniques, restoration and path protection, for single failure scenarios in in-band networks.

For implementing restoration, the controller depends on a failure notification (PORT_STATUS [9]) instead of the echo timeout to declare failures. The controller receives the notification when a switch detects LOS and still has a connection with the controller. The challenge behind restoration is that the controller has lost communication with the affected switches and therefore it cannot establish paths from (or along) these switches.

To overcome the challenge, during bootstrapping the controller establishes a one-hop restoration path together with the working path for control traffic. In this path, the source switch floods its own traffic, only one neighbor (which is along the working path) forwards the traffic, and other neighbors just drop it. On the failure notification, the controller first makes a list of affected switches that can be restored first and then restores the affected switches according to the list. This is done because the restoration path of affected switches may be along the switches that are affected by the failure, and hence before establishing the path these switches are needed to be restored.

In addition to restoration, failure recovery can be achieved by protection. Protection removes the need of establishing an alternative path after a failure by installing it in advance. We implement 1:1 path protection in which the ingress switch redirects traffic to a pre-established disjoint alternative path when a failure is detected in the working path. For pre-establishing the path, the controller uses the group-table concept (fast-failover) [9] at the ingress switch and uses the flow-table concept in all other switches along the paths. With the group-table concept, two rules are kept for traffic forwarding. Before a failure, the ingress switch applies the first rule (which corresponds to the working path), and after the failure it applies the second rule, which corresponds to the alternative path.

In addition to control traffic, we apply the above restoration and protection techniques for data traffic. However, in the case of restoration of data traffic, the one-hop restoration paths are not established in advance, and after recovering control traffic, data traffic is restored.

In the case of protection of both control and data traffic, BFD can be used. In BFD, a pair of switches transmits BFD packets periodically between each other, and if a switch stops receiving the packets, the path between the switches is assumed to be failed.

OpenFlow does not define how to run BFD in the switches. Therefore, we propose to integrate BFD in the local networking stack of switches and add a vendor-specific extension in the OpenFlow protocol to run it through the switches. With this vendor-specific extension, the controller sends a message containing information about a BFD session (RFC 5880). Upon reception of this message, the switch runs the BFD session on the local networking stack, which allows the switch to send BFD packets through its local port (reserved port of the switch).

In protection, when the controller establishes the working and alternative path between two edge switches, the controller sends vendor-specific messages to edge switches to start a BFD session between them. In addition, the controller establishes a path for the BFD session, which follows the same path as the working path. Hence, when BFD detects the failure, the ingress switch declares the working path as a faulty path and therefore, the ingress switch can now apply the second rule.

4.3 Practical challenges

4.3.1 Evolution of OpenFlow specifications

Stanford University released specifications for OpenFlow known as version 1.0 and 1.1 in 2009 and 2011, respectively, and industrial players such as Deutsche Telekom, Google, Microsoft, and Yahoo! have shown substantial interest in this technology. These companies then formed ONF (Open Networking Foundation) to standardize and release the versions of OpenFlow according to their demands. Since then, six more versions (1.2, 1.3.0, 1.3.1, 1.3.2, 1.3.3, and 1.4.0 [9]) have been released publicly. Hence, the challenge is to choose which version to use for implementation of our functionalities. In addition, as OpenFlow is evolving quickly, not all the versions or all the enhancements of the released versions are implemented for OpenFlow.

4.3.2 Availability of required switch components

Table 4.1 shows the availability of the required components in existing implementations. The functions of these components are described below:

Table 4.1: Features required/present in different implementations of OpenFlow switches. VS is the vendor-specific extension, OVS-DB is the Open vSwitch database management protocol, and VER is the Open vSwitch version

	Software Component	**Reference switch** (License BSD) *www.openflow.org*	**Indigo switch** (License Eclipse) *www.openflowhub.org*	**Open vSwitch** (up to VER 1.11.0) (License Apache 2.0) *www.openvswitch.org*	**Trafficlab1.1** (License BSD) *https://github.com/TrafficLab*	**Trafficlab1.3** (License BSD) *https://github.com/CPqD*
In-band	DHCP Client	Yes	Yes	Not functional for in-band control	Not functional	Not functional
	NORMAL Stack	Yes	Yes	Yes	Yes	Not functional
	TCP Stack	Yes	Yes	Yes	Yes	Yes
	OpenFlow Stack	Yes	Yes	Yes	Yes	Yes
Queuing	Queue forwarding	Yes	Yes	Yes	Yes	Yes
	Queue creation	Yes (with VS)	Yes (with VS)	Yes (with OVS-DB)	Yes (with VS)	Yes (with VS)
	Queues with priorities	No	No	Yes	No	No
Failure Recovery	LOS failure detection	Yes	Yes	Yes	Yes	Yes
	BFD failure detection	No	No	No	No	No
	Group-table (fast-failover)	No	No	No	Yes	Yes

With a DHCP client [12], a switch can generate/receive DHCP messages from the local port. With the NORMAL stack [9], a source switch can forward its messages using L2 learning when it does not have an OpenFlow session with the controller. Using the TCP stack, a switch can establish a TCP session. Using the OpenFlow stack [9], a switch can establish an OpenFlow session. Using queue forwarding [9], traffic can be forwarded through queues. With the queue creation component, queues can be created in switches. Using queues having different priorities, queues can be served on a priority basis. Using LOS, a switch can detect failures in restoration. Using BFD, switches can detect failures in protection, and with the group-table fast-failover type, switches can change the actions of forwarding packets without contacting the controller.

The existing implementations for required components are Reference switch, Indigo, Open vSwitch, Trafficlab1.1, and Trafficlab1.3 (Table 4.1). Reference switch [3] is the first software release of OpenFlow version 1.0. Indigo is a hardware-switching release based on Reference switch. Open vSwitch is a production quality release of OpenFlow. Trafficlab1.1 is the extension of Reference switch to incorporate version 1.1, and Trafficlab1.3 contains version 1.3.0.

Table 4.1 shows that not all the required components are present in a single implementation. Therefore, the challenge is to integrate all the required components in one implementation. In addition, vendor-specific extensions of switches are required to configure queues with different priorities. Furthermore, BFD is not present in any implementations. Hence, it is needed to be either implemented fully or imported from any open-source implementations of BFD. Some modifications related to BFD are also required. The modifications are: running BFD sessions on the local networking stack; listening or sending BFD packets on the local port; and modifying a group-table entry on detection of a failure.

4.3.3 Availability of required controller components

The required controller components are:

- In-band control, which can run in-band functionality on the controller.
- Queuing, which can establish queues (with different priorities) in the switches.
- Failure recovery, which can implement restoration or protection for control and data traffic.

Currently, none of the available controllers (e.g. NOX, POX, Floodlight) implement these components.

For the implementation of these components, the available controllers can generate several events. The events, which are important for the required components, are:

- Switch-join, which is generated when a switch establishes an OpenFlow session with the controller.
- Switch-leave, which is generated when a switch disconnects from the controller.
- Port-config, which is generated when the controller receives all port information.
- Packet-in, which is generated when a packet is received to decide its forwarding action.
- Port-status, which is generated when an LOS is detected or repaired in one of the ports in a switch.

Using these events, all the proposed functionalities can be implemented in all the available controllers.

4.4 Experimental studies

Table 4.2: Emulated topologies

	Topologies		**#switches**	**#links**	**switch degree**		
					min	mean	max
1	Pan European Topologies	Core topology	16	23	2	2.88	4
		Basic reference	28	41	2	2.93	5
		Large topology	37	57	2	3.08	5
		Ring topology	28	34	2	2.43	4
		Triangular topology	28	61	2	4.36	7
2	Ring		100	99	2	2	2
3	Random Regular Graph		100	150	3	3	3
4	Balanced Binary Tree (height=5)		63	62	1	1.97	3
5	Star		100	99	1	1.98	99

We implemented our proposed functionalities in Traffliclab1.3 and in its compatible controller (NOX1.3). We used this switch because at the time of our implementation this was the only available soft switch containing the latest version and the group table concept. We implemented all the unavailable and non-functional software components (Table 4.1) in this switch using the mechanisms provided above.

The experiments are carried out using the emulated topologies described in Table 4.2. The pan-European topologies in Table 4.2 are basic reference topology (BT), core topology (CT), large topology (LT), ring topology (RT), and triangular topology (TT). The topologies that vary with the degree of meshedness are RT and TT, and the topologies in accordance with the number of switches are CT and LT. The other used topologies are ring, star, random regular graph, and balance binary tree. For experiments, one of the switches is physically connected with the controller, and the DHCP server is located on the controller.

For the in-band experiments, we use a single node of the iMinds island of the OFELIA testbed [13], and mininet [14] is used for emulating topologies. For all other experiments, a node of the island is dedicated to a single switch or the controller, and the topologies are generated using the emulab interface of the island.

4.4.1 In-band control experiments

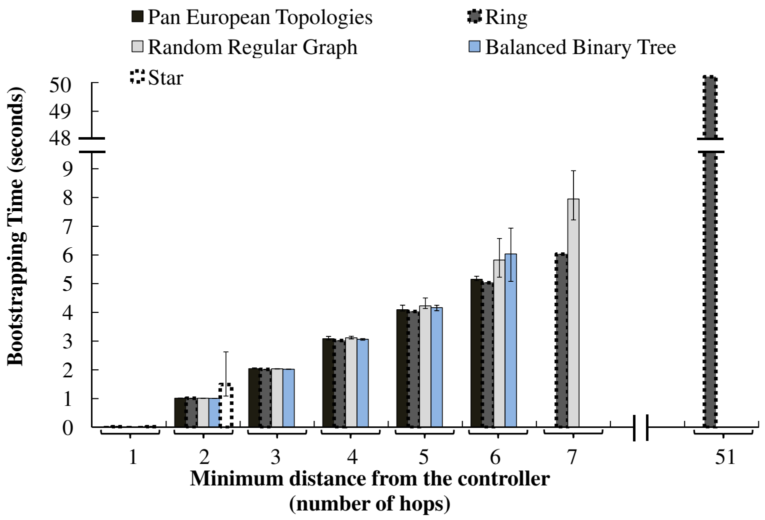


Figure 4.2: Bootstrapping time for all emulated topologies. The error bars show the minimum, average, and maximum values of the bootstrapping time.

In the case of in-band control experiments, the DHCP retransmit timeout is kept as 1 second (minimum value) and the bootstrapping time (the total time to establish OpenFlow sessions) of switches is calculated with respect to the distance from the controller. As the bootstrapping time for all pan-European topologies is approximately the same, we show the bootstrapping time of all these topologies by a single bar (Fig. 4.2). In addition, at the distance where there is no switch in the topologies, no bar is shown in Fig. 4.2.

For one-hop, the bootstrapping time is approximately zero because the switch at one-hop does not wait for the DHCP timeout to retransmit a DHCP request. As the distance from the controller increases, Fig. 4.2 shows an increase in the bootstrapping time, because the switches, which are n hop $(n > 1)$ away from the controller, are able to establish the session, if at least one of its neighbors has an OpenFlow session with it. When more switches are located at a certain distance from the controller (at distance 2 for star, at distance 6 for balance-binary tree, and at distance 6 and 7 for random-regular graph), we found a significant increase in the bootstrapping time, because in this case the in-band component of the controller receives lots of messages at about the same time. Until the controller replies, the messages stay in the packet-in buffer, increasing the bootstrapping time. In addition, as the buffer can overflow at some point, some of the messages have to be dropped. If a dropped message is a DHCP request, a switch waits for the next DHCP timeout to retransmit the DHCP request, and hence delays the bootstrapping time for an additional 1 second.

4.4.2 With queuing and without queuing experiments

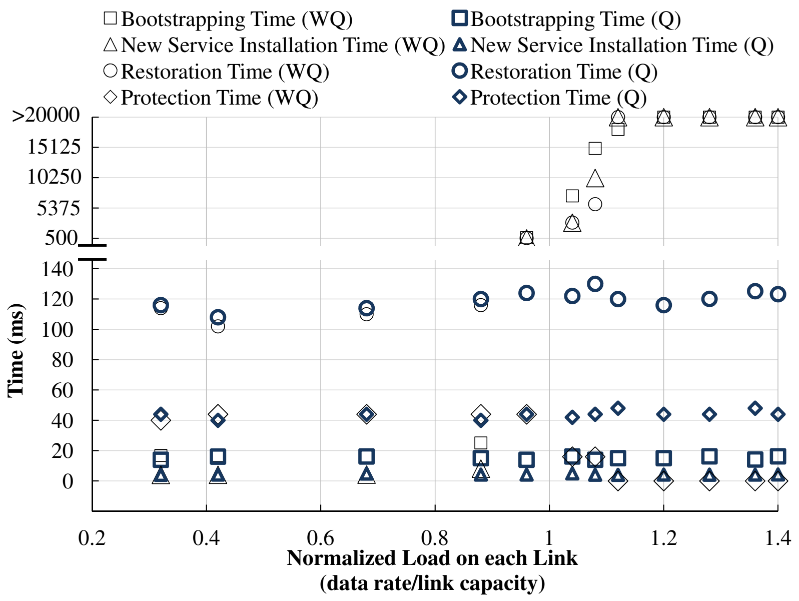


Figure 4.3: Impact of data traffic on control plane operations. WQ means in-band control without queuing functionality and Q means in-band control with queuing functionality.

In these experiments, the rate of data traffic (Poisson distributed on an average interval) is varied on each link of the CT topology, and the impact of data rate on control plane operations such as new switch connection (bootstrapping), new service installation, and failure recovery is calculated using queuing (Q) and without using queuing functionality (WQ). Each link of the topology is assigned a capacity of 10 Mb/s, and the size of data packets is 1000 bytes. All the results are calculated 50 times, and the average is shown in Fig. 4.3.

Figure 4.3 shows that under a low load ($load < 0.9$), bootstrapping, new service installation, restoration, and protection time is comparable for Q and WQ. However, at a high load ($load > 0.9$), due to congestion WQ takes a significantly longer time than Q for bootstrapping, new service installation, and restoration. In this case, as the load increases, switches drop more control and data packets. After dropping control packets, switches and the controller have to retransmit these packets after their timeouts, increasing the delay in completing bootstrapping, new service installation, and restoration. Moreover, at a $load > 1.04$, WQ has a lower protection time (less than 40 ms) than Q. This is because due to congestion, switches have dropped some BFD packets (sent interval = 20 ms and timeout = 40 ms) just before a failure, allowing BFD to detect failures faster than in a normal condition. Furthermore, after a $load \geq 1.08$, a large number of BFD packets drops in WQ due to congestion, and therefore BFD declares its timeout without the presence of the actual failure (link failure). This is the reason for zero protection time in WQ at a $load \geq 1.08$ as traffic is already on the protection path at the time of failure. The results also show that all control plane operations take significantly shorter time in in-band control with queuing (Q) in all load conditions. Indeed, queuing functionality circumvents the competition between control and data traffic by implementing separate queues.

4.4.3 Failure recovery experiments

We performed the following three types of failure recovery experiments for in-band OpenFlow using queuing functionality: (1) Control and data traffic, (2) Multiple topologies, and (3) Switches disconnection experiments.

In control and data traffic experiments, the failure recovery time is calculated for one of the combinations of restoration and protection for control and data traffic. In multiple topology experiments, the recovery time is calculated for different types of topologies, and in switch disconnection experiments, the recovery time is calculated to show the impact of the increased number of disconnected switches along the recovery path. In the experiments, a failure is given by disabling Ethernet interfaces, and for restoration, LOS is used to detect failures and the failure detection time is between 50 to 60 ms. For protection, BFD is used and the failure detection time is about 40 ms. All the results are calculated

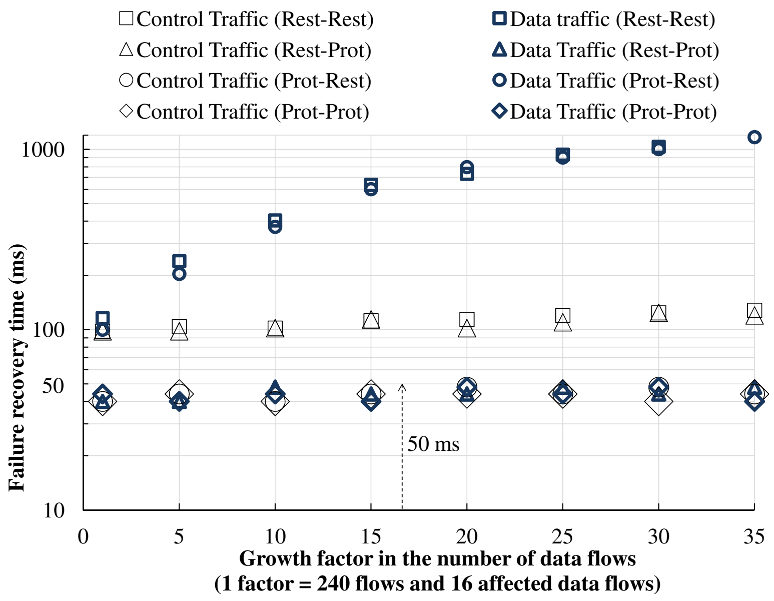


Figure 4.4: Recovery time of control and data traffic using all combinations of restoration and protection

50 times and the average is shown in Fig. 4.4.

In the control and data traffic experiments, the number of data flows (240 to 8400) is increased in the CT topology, and one of the combinations of restoration and protection is applied for control and data traffic. These combinations are:

- Restoration of both control and data traffic (Rest-Rest).
- Restoration of control traffic and protection of data traffic (Rest-Prot).
- Protection of control traffic and restoration of data traffic (Prot-Rest).
- Protection of both control and data traffic (Prot-Prot).

In all the combinations, Fig. 4.4 shows that restoration does not meet the carrier-grade requirement of 50 ms, while protection meets the requirement. In addition, the restoration time of data traffic (Rest-Rest and Prot-Rest) increases with the increase in the number of affected data flows, because as the number of affected data flows increases, a higher number of data traffic paths needs to be configured after the failure.

In the multiple topology experiments, different types of pan-European topologies (CT, BT, and RT) are used, and we found that the restoration time increases with the number of switches in a topology, because in our implementation the path calculation time grows as $O(n^2)$, where n is the number of switches. In addition, as the degree of meshedness increases, the restoration time decreases, because in this case fewer hops are required for the restoration path, and therefore the controller needs to configure fewer switches in the network. Furthermore, protection does not require controller intervention, and therefore it is far less dependent on the network topology.

In the switch disconnection experiments, ring topologies are used, and the restoration time follows a linear relationship with the number of affected switches along the restoration path. For protection, the recovery time is always within 50 ms and meets the requirement.

4.5 Conclusion and future work

In this article, we have explored OpenFlow for in-band control, queuing, and failure recovery functionalities, and have performed extensive experiments. The in-band control experiments conclude that the proposed method allows bootstrapping in all types of topologies. With this method, switches of emulated pan-European topologies have taken a maximum of 5 seconds to perform bootstrapping. The queuing experiments demonstrate that in-band control traffic can be served first before any other traffic, and hence it can avoid competition with data traffic for network resources. The failure recovery experiments conclude that restoration in OpenFlow does not allow achieving 50 ms recovery, and protection for both control and data traffic allows achieving recovery within 50 ms. In our results, we did not take into account the propagation delay. Among all the presented results, the restoration time may significantly increase with the increase in the propagation delay, further strengthening the conclusion of the article that restoration cannot meet the requirement of 50 ms. As future work, the effects of propagation delay can be studied to quantify the degradation of the restoration time with an increase in propagation delay.

Based on the presented emulation results, we believe that our functionalities can be applied in production networks. However, to improve the accuracy of results, our experiments can also be performed on real environment testbeds such as GENI (Global Environment for Network Innovations) or FIBRE (Future Internet testbeds/experimentation between Brazil and Europe). Using these testbeds, OpenFlow hardware switches can be used for experimentation and the topologies can be generated in real environment settings. In the experiments, the impact of real environment factors (e.g. hardware dependent parameters such as packet forwarding, processing, and queuing) on the results can be studied. For the bootstrapping time, we believe that this impact will be negligible, as the DHCP retransmit timeout (i.e. 1 second) dominates the bootstrapping time. For the restoration time, the impact can be significant as the restoration time is measured in milliseconds and a small variation due to real factors will influence the results. For the protection time, the impact will be negligible because only the ingress switch along the protection path is involved for the protection activity.

In this article, we have not explored security and controller failure issues for in-band OpenFlow. For security, there can be many concerns related to DHCP [12], transport layer [15], and OpenFlow messages. These concerns are security issues related to:

- TCP or DHCP requests from bad actors.
- DHCP messages from an unauthorized DHCP server.
- Denial of service from the DHCP server or the controller.
- Switch datapath ID conflicts.

Nevertheless, transport layer security (TLS) described in OpenFlow [9] can be applied in the bootstrapping phase. However, the problem is that OpenFlow does not provide any details of TLS operations. This could lead to interoperability issues. In addition, TLS has many technical barriers for operators. These are:

- Assigning controller certificates.
- Assigning switch certificates.
- Signing the certificates with a private key.
- Installing the keys and certificates into all network devices.

In future work, we will consider the aforementioned security issues and will explore controller failure solutions for in-band OpenFlow. To solve the controller failure issues, we will use two controllers. Hence, when one controller crashes, switches can rely on a backup controller to take actions.

Acknowledgments

This research has received funding from EU FP7 under agreement n^o 317576 (CityFlow), 258457 (SPARC) and 258365 (OFELIA).

References

[1] J. Rexford, *Future Internet Architecture: Clean-Slate Versus Evolutionary Research*, Communications of the ACM, Vol. 53, no. 9, September 2010, pp. 36–40.

[2] J. Pan, S. Paul, and R. Jain, *A survey of the research on future internet architectures*, IEEE Communications Magazine, Vol. 49, no. 7, July 2011, pp. 26–36.

[3] N. McKeown, T. Andershnan, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, *OpenFlow: Enabling innovation in campus networks*, ACM Computer Communication Review, Vol. 38, no. 2, April 2008, pp. 69–74.

[4] R. Ahmed and R. Boutaba, *Design considerations for managing wide area software defined networks*, IEEE Communications Magazine, Vol. 52, no. 7, July 2014, pp. 116 - 123.

[5] C. Tu, P. Wang, and T. Chiueh, *In-Band Control for an Ethernet-Based Software-Defined Network*, ACM SYSTOR, 2014, pp. 1–11.

[6] P. Skoldstrom and K. Yedavalli, *Network virtualization and resource allocation in OpenFlow-based wide area networks*, IEEE ICC, June 2012, pp. 6622–6626.

[7] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, *Automatic bootstrapping of OpenFlow networks*, IEEE LANMAN, April 2013, pp. 1–6.

[8] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, *Fast failure recovery for in-band OpenFlow networks*, DRCN, March 2013, pp. 44–51.

[9] OpenFlow specifications [Online]. Available: https://www.opennetworking.org/sdn- resources/onf-specifications.

[10] A. S. Tan, O. Karakaya, A. Ulas, M. Parlakisik, O. Kupusoglu, O. Erhan, E. Lokman, *Automatic topology discovery in software defined networks*, SIU, April 2014, pp. 939- 942.

[11] S. Sharma, D. Staessens, D. Colle, D. Palma, J. Goncalves, M. Pickavet, L. Cordeiro, P. Demeester, *Demonstrating resilient quality of service in Software Defined Networking*, IEEE INFOCOM WKSHPS, May 2014, pp. 133–134.

[12] S. Duangphasuk, S. Kungpisdan, S. Hankla, *Design and implementation of improved security protocols for DHCP using digital certificates*, IEEE ICON, December 2011, pp. 287–292.

[13] M Sune, L Bergesio, H Woesner, et. al., *Design and implementation of the OFELIA FP7 facility: the European OpenFlow testbed*, Computer Networks, Vol. 61, March 2014, pp. 132–150.

[14] V. Antonenko and R. Smelyanskiy, *Global Network Modelling Based on Mininet Approach*, HotSDN, 2013, pp. 145–146.

[15] P. Casas, J. Mazel, and P. Owezarski, *Knowledge-Independent Traffic Monitoring: Unsupervised Detection of Network Attacks*, IEEE Network, Vol. 26, no. 1, January 2012, pp. 13–21.

5
CityFlow, Enabling Quality of Service in the Internet: Opportunities, Challenges, and Experimentation

This chapter proposes the introduction of Quality of Service (QoS) techniques for the Future internet enabled with OpenFlow. For implementing QoS techniques, we explore the queuing functionality proposed for OpenFlow in the previous chapter and implement a framework with which a high priority user connected to an OpenFlow network is dynamically allocated a right of way between any two endpoints, on multiple autonomous systems, for a given application flow. The framework is evaluated in a wide range of network scenarios for a city with a population of 1 million inhabitants on a large-scale experimental facility in Europe. The scenarios include both data and control traffic scalability, as well as, failure recovery. For failure recovery, we do not focus on fast failure recovery (Chapter 2 and Chapter 4), but we focus on the mechanisms with which high quality can be achieved for high-priority users in all scenarios including failure scenarios. This chapter presents a brief description of the QoS mechanism proposed for providing high QoS under failure conditions, while Appendix C gives a detailed description about the mechanism.

⋆ ⋆ ⋆

Sachin Sharma, David Palma, Joao Goncalves, Dimitri Staessens, Nick Johnson, Charaka Palansuriya, Ricardo Figueiredo, Luis Cordeiro, Donal Morris, Adam Carter, Rob Baxter, Didier Colle

Submitted to Computer Networks, November 2015.

Abstract By including a dynamic *right of way* in the Future Internet, services can be delivered more efficiently and effectively. In this article, we propose an OpenFlow enabled Internet infrastructure, using virtual path slicing, so that any user connected to an OpenFlow network substrate is dynamically allocated a corresponding *right of way*. This approach allows an interference-free path, from other traffic, between any two endpoints, on multiple autonomous systems, for a given application flow (e.g., WebHD Video Streaming or HD Video to Video). Additionally, we propose an operational model for the Future Internet and extend the virtual path slice engine, which enables virtual path slicing over the Internet, to support future Internet technologies such as OpenFlow. The proposed architecture was evaluated in distinct reference network-scenarios for a city with a population of 1 million inhabitants, emulating xDSL (Digital Subscriber Line), LTE (Long-Term Evolution) and Fibre networking scenarios. The obtained results confirmed the suitability of the proposed architecture between multiple autonomous systems, considering both data and control traffic scalability, as well as resilience and failure recovery.

Keywords

OpenFlow; Quality-of-Service; Internet; Border Gateway Protocol

5.1 Introduction

In recent years, Internet traffic from content provider companies (e.g., Skype, Google, Netflix, Akamai, Facebook) has increased drastically due to the exponential usage of over-the-top applications by users. This traffic is expected to keep increasing, as content providers launch more and more over-the-top applications. However, telecommunication companies, who bear the operational and maintenance cost of the Internet infrastructure, are not interested in investing on additional infrastructure capacity to provide the required bandwidth for these applications without an adequate return on network capital employed.

Since no content provider has created a successful business model for large scale quality of service (QoS) over the Internet [1], it is very difficult for Internet infrastructure owners to get profit from the growing demand for bandwidth and quality. Currently, the Internet works on a best-effort basis and content owners, who obtain revenue for their applications, can inject traffic into the Internet at an originating Autonomous System (AS) and expect it to be carried over the Internet to a destination autonomous system without sharing revenue with infrastructure owners. Differentiation is one possible solution. In addition, it would be interesting for content provider and users to open a guaranteed pipe over the Internet using differentiation.

In the CityFlow project[1], we propose a differentiated Internet based on virtual path slicing (VPS) and Software Defined Networking (SDN) technologies, such as OpenFlow. Using SDN [2], the control plane can be separated from the data plane of Internet devices and can be embedded into one or more external servers called controllers. Using VPS [3], telecommunication companies can enable a *right of way* for users' traffic over the Internet without interference from best-effort traffic. In this article, VPS is enabled using the virtual path slice engine, which is a commercial product of Redzinc, Ireland[2] and it is extended to support SDN technologies.

For the differentiated Internet, we propose an operational model for the Internet and give opportunity for infrastructure owners, content providers, and users to benefit from it. In the operational model, we propose that traffic engineers of the Internet infrastructure dimension their networks into two categories: (1) aggregated traffic, which is historically the best-effort Internet, and (2) personalized flows, which are treated as high-priority traffic. For treating traffic as a personalized flow, the infrastructure owners can charge users or their content providers. Our model can raise questions regarding network neutrality [4]. Network neutrality means that no bit of information should be prioritized over another on the Internet. This is a complicated regulatory issue, and a full discussion is out of scope of this article. Nevertheless, it is worth mentioning that technical methods used in our model are based on the requirements of reasonable network management practices including transparency, non-blocking, and no unreasonable discrimination [5].

We test our model using a wide range of large-scale multi autonomous signaling experiments that are performed on the OFELIA testbed (OpenFlow in Europe Linking Infrastructure and Applications). The OFELIA testbed [6] is a large-scale experimental facility in Europe. One of the experiments is also performed on the public Internet using the Amazon cloud facility. Our experiments mimicked the conditions that would be required for WebHD Video Streaming and HD Video to Video. All the experiments are performed by taking into account the key Internet technologies (4G/xDSL/Fibre) for a mid-sized European city of around 1 million inhabitants.

Section 5.2 presents the proposed CityFlow model, architecture and

[1]The CityFlow project: https://www.cityflow.eu/

[2]www.redzinc.net

components, followed by the presentation of the defined CityFlow network scenarios in Section 5.3. Section 5.4 provides details about the performed experimentations and the obtained results are discussed in Section 5.5. Finally, in Section 5.6, concluding remarks and future directions are presented.

5.2 QoS model for the Internet

In the CityFlow project, deployment of a virtual path slice over the Internet is influenced by the research performed by the EuQoS project [3] which promises to provide end-to-end QoS over heterogeneous networks. The essential principal of the research is that bandwidth resources are managed in an on-path off-line manner. By on-path we mean that resource management follows the forwarding path of the IP packets, across multiple autonomous systems, as determined by BGP (Border Gateway Protocol). By off-line we mean that the resource management is implemented in software off-line from the network elements that are responsible for packet forwarding. Along the path, capacity management is implemented only at choke points which are mostly the interconnection points and the edges.

5.2.1 VPS engine overview

Our model uses the VPS engine[3] to setup virtual path slices over the Internet. This engine contains three main interfaces (shown in Fig. 5.1): (1) the first one is to receive requests from users, (2) the second one is for inter-carrier domain communication and (3) the third one is to communicate with the infrastructure networks. The first interface is implemented to receive requests from users to reserve *a right of way* in the Internet. The implementation of the second interface within the VPS engine (i.e., for inter-carrier domain) is largely influenced by the initial work done in the the Internet Engineering Task Force (IETF) for Next Steps in Signalling (NSIS) [7]. The implementation of the third interface was developed in order to communicate with OpenFlow-based infrastructure networks, detailed in the next subsection.

The VPS engine contains also a MySQL Database, translation layer component and invocation controller, responsible for storing all the information about invocations, topology and resources. The translation layer component is responsible to translate the input of the VPS client to the NSIS protocol input and the invocation controller is responsible to reserve resources in networks for establishing a *right of way* for services.

[3]Commercial product from Redzinc

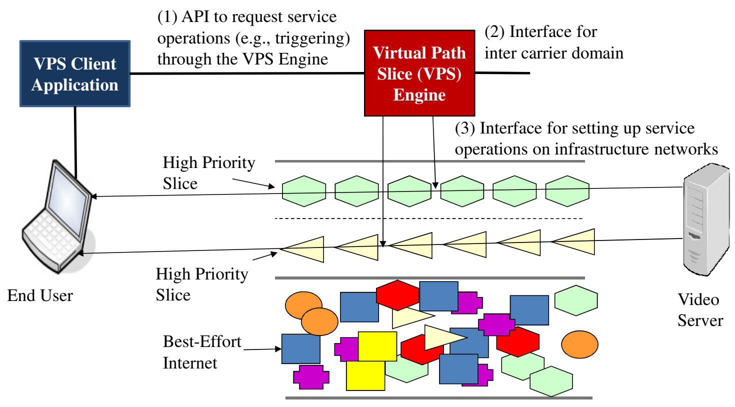

(1) API to request service operations (e.g., triggering) through the VPS Engine
VPS Client Application
Virtual Path Slice (VPS) Engine
(2) Interface for inter carrier domain
(3) Interface for setting up service operations on infrastructure networks
High Priority Slice
High Priority Slice
Best-Effort Internet
End User
Video Server

Figure 5.1: VPS engine overview

5.2.2 Components of the proposed model

The architecture devised by CityFlow was envisaged for future OpenFlow networks, enabling them with the possibility to dynamically configure paths with guaranteed traffic performance. Motivated by the separation between data and control planes followed by the software-defined networking paradigm, additional business intelligence is included on top of the control plane, which in turn enforces the necessary decisions on the data plane. Fig. 5.2 depicts a high-level perspective of this approach.

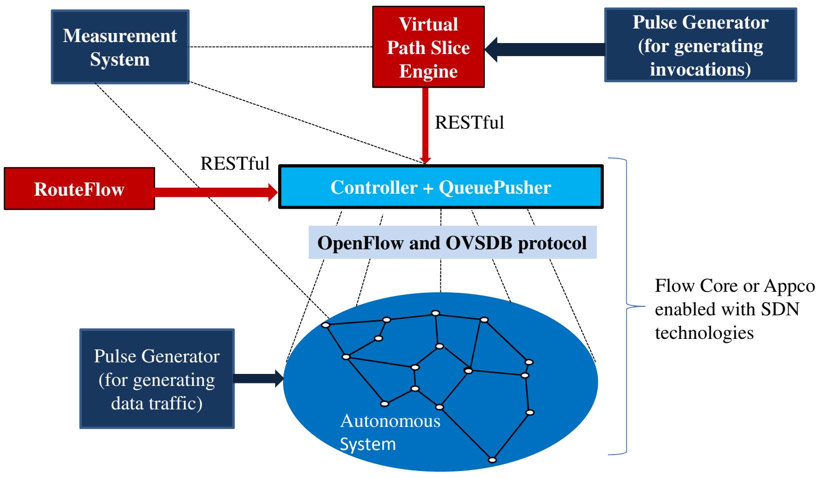

Measurement System
Virtual Path Slice Engine
Pulse Generator (for generating invocations)
RESTful
RESTful
RouteFlow
Controller + QueuePusher
OpenFlow and OVSDB protocol
Flow Core or Appco enabled with SDN technologies
Pulse Generator (for generating data traffic)
Autonomous System

Figure 5.2: CityFlow's architecture and components

In particular, in CityFlow, the Virtual Path Slice (VPS) engine is the entity responsible for the business intelligence and manages the relationship with the remaining CityFlow components, namely RouteFlow [8] and the controller extended with the QueuePusher module [9]. RouteFlow is used for running routing protocols such as BGP in the OpenFlow networks. The QueuePusher module[4] is used to set up queues for providing high QoS to the personalized flows.

Since the beginning of this project several platforms have been developed for SDN. In fact, even though during the development of this work the Floodlight controller was chosen due to its northbound API, nowadays, other controllers provide similar characteristics (e.g. the Open Daylight controller). Nonetheless, the conclusions presented by this article are independent of these factors and could be verified with the latest available SDN solutions.

In the proposed architecture, the communication between the components within the control plane is based on a RESTful interface, so that it can easily ported between different software solutions, while interactions with OpenFlow switches and the controller are supported by the OpenFlow and OVSDB (Open vSwitch Database Management) protocols. Other supporting tools such as Pulse Generator and CityFlow's measurement system are also developed for enabling the scenarios presented in Section 5.3. The pulse generator tool is developed to transmit high rate of control or data traffic to cover 1 million city population. The measuring tool is developed to gather the measurement data from the system.

5.2.3 Operational model for the Internet

A high level view of our proposed operational model is shown in Fig. 5.3. The essential idea is to segment the network capacity into a Best-Effort (BE) and a high priority (HP) domain. As capacity grows an operator can make a policy decision regarding the proportion to be allocated to the BE or HP domain. Initially, the HP domain might have a low share of capacity, but as demand grows, and BE becomes constrained, new capacity could be allocated to the HP domain. This can be implemented by using an aggregated queue in a gateway network element dimensioned for x% (can be 50%) of the capacity for traffic painted as BE using DiffServ code points. The remaining capacity (i.e., HP) can be painted as Expedited Forwarding or Assured Forwarding using DiffServ code points.

Content or application providers (appcos) with multiple applications in the area of Internet of Things, eHealth, Consumer, Cloud and Social Media can use the slicing mechanism (queuing mechanism shown in Fig. 5.2) through VPS to obtain a slice of bandwidth in the HP domain across multiple autonomous systems and to the consumer connected via optic fibre access or 4G/5G radio access network. In

[4]The source code of QueuePusher is available at:
https://github.com/OneSourceConsult/floodlight-queuepusher

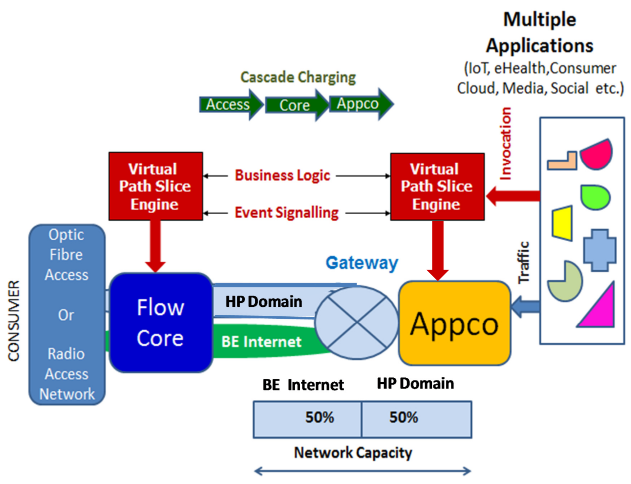


Figure 5.3: Conceptual operational model for the Internet

exchange for obtaining guaranteed bandwidth to users, the applications provider who will be able to drive new business models (e.g., 4K webTV) can expect to receive a charge for conveying the guaranteed traffic. This can be implemented using cascade charging on a wholesale basis, from access-to-core to applications provider.

The concept model in Fig. 5.3 is based on the separation of control from the data plane and the inclusion of features for business engagement. We add some business logic and event signalling between the remote “appco” application traffic source and the flow core at traffic sink where the consumer is located. A gateway distributes the application traffic on the BE or HP domain based on DSCP and/or MPLS EXP marking. The flow core allocates a discrete flow in the distribution and access network onwards to the consumer. While the consumer has a contract relationship (e.g., Netflix contract) with the appco. Cascade charging from the access to the core to the appco enables all infrastructure stakeholders in the traffic pathway share in the economic activity.

Our model works based on the requests from the user application to the VPS Engine as the user request determines the end points themselves. The OpenFlow controller, which in the case of the CityFlow project is Floodlight, tells the VPS Engine (c.f. Fig. 5.2 and Fig. 5.3) where those endpoints are connected in the data plane (i.e. the switch). RouteFlow by running BGP determines the output ports of the end points for delivering the user application. The VPS engine then replicates the existing best-effort flow rules and creates a new flow on the output port of the ingress switch. This new flow is ‘painted’ with the DiffServ code point for expedited forwarding. In parallel with this, a new flow is assigned to a queue that is given a scheduler rate corresponding to the bandwidth for the associated virtual path slice.

From a path point of view, the virtual path slice model follows the path determined by BGP between different autonomous systems and OpenFlow areas. The bandwidth of the slice is determined by the rate in the shaper of policer on the ingress switch. The model is relevant for a mixed topology including legacy IP routers and new OpenFlow switches. Connection Admission Control (also known as RACF – Resource Admission Control Function) is implemented at the ingress. A count is taken of the allocated capacity and a "busy tone" is implemented if a threshold is reached.

5.3 Reference scenarios for experimentation

The CityFlow experimentation was defined considering a target population of 1 million inhabitant, representing a mid-sized city as a reasonable and practical dimension – not too large and not too small – implementable on the OFELIA testbed. We have analyzed the network infrastructure of Brussels, population 1.1 million, in order to obtain reference scenarios for mobile (LTE), xDSL, and Fibre. Unfortunately, Brussels has currently no fibre access network deployed, thus in order to have a more future-proof reference network, we add Fibre-To-The-Home (FTTH) data from other European cities of similar size (e.g., Cologne). Starting from real data gives us a realistic scenario from which we can base our experiments. In reference scenarios, we use the ACG study [12] to design LTE, FTTH, and DSL networks for our experimental city, Flowville. As a city can contain mobile, DSL, and FTTH networks simultaneously, we also present an integrated model that combines these networks.

For mobile networks, we collect data from real sources i.e., BIPT (Belgian Institute for Postal Services and Telecommunications). According to BIPT, there are 958 radio base stations in Brussels. These stations provide wireless access to all users in the city. In addition, there are three large mobile operators: (1) Base, (2) Mobistar, and (3) Proximus. For each of the mobile operators, we consider a latest radio access technology, e.g., LTE. For each LTE, we place 958 base stations in the access rings of Flowville. According to the ACG Study, a maximum of 25 radio base stations operated over a ring can be connected with a pre-aggregation site. Therefore, there can be a maximum of 39 pre-aggregation sites (958/25=39) for Flowville. In addition, as there can be a maximum of 16 pre-aggregation sites per one aggregation site (ACG Study), there can be a maximum of three aggregation sites (39/16=3) in the aggregation network. Moreover, in order to connect these three aggregation sites to the core, we require two core locations [12].

In xDSL scenarios, DSLAMs (Digital Subscriber Line Access Multiplexers) are used to connect multiple customer Digital Subscriber Lines (DSL) to an aggregation network. Currently, Brussels has only VDSL (Very high bit rate

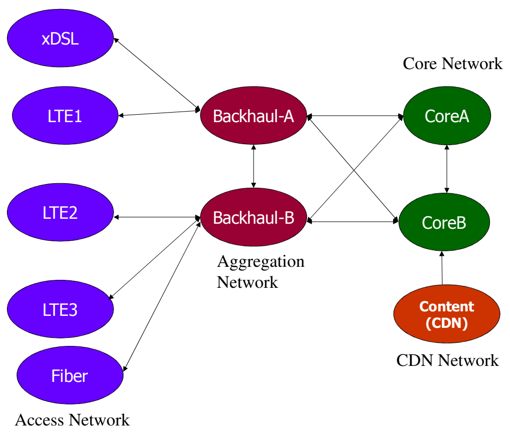


Figure 5.4: CityFlow reference city (Flowville, integrated model)

Digital Subscriber Line) technology and 59.29% of Brussels population use this technology. In our design, we assured that 90% of DSLAMs in Flowville are with 8 line cards and 10% of DSLAMs are with 3 line cards. Therefore, the number of DSLAMs required for Flowville is 1026, as one DSLAM line card can serve a maximum of 48 households [13]. As a DSL-based access/aggregation network is operating over rings, we can use similar architecture as the LTE case. Therefore, there can be a maximum of 42 pre-aggregation sites (1026/25=42) and three aggregation sites (42/16=3) in the aggregation network for Brussels.

In FTTH scenarios, we assume that 20% of population will use FTTH connections. In FTTH, we assume Passive Optical Networks (PONs), which consist of Optical Line Terminals (OLTs) and Optical Network Units (ONUs). The OLT resides in the central office, and the ONU resides in the customer's premises. For PON, there can be 48 subscriptions per OLT [14]. As there is an average of 2.06 persons per household, Flowville requires 11,518 OLTs to cover the total population. For 20% of population, Brussels requires around 2300 OLTs. As OLTs resides in the central offices and one central office can have up to 500 OLTs, we require 5 central offices to cover 20% of Flowville's population. If we consider 3% growth rate in FTTH adoption, we will require seven central offices. Assuming that one central office is directly connected with one pre-aggregation site, we require seven pre-aggregation sites and one aggregation site for Brussels.

In Brussels, there are two operators in the Backhaul network: (1) Telenet (Backhaul-A in Fig. 5.4) and (2) Belgacom (Backhaul-B in Fig. 5.4). These operators connect the access network to the core network. Therefore, for the integrated model, we connect one LTE and one DSL or Fibre network to one of the Backhaul operators, and the remaining to the second Backhaul operator (Fig.

5.4). Here, LTE 1, LTE 2 and LTE 3 are the access network scenario from three mobile operators in Flowville. In this design, all core locations are connected with the CDN (content delivery network) servers. The goal of a CDN server is to serve content to end-users with high availability and high performance.

5.4 Experimentation

In this section, we report the experimentation methodology for setting city experiments on the OFELIA testbed.

5.4.1 Software used for experimentation

Due to emerging importance of SDN, a large number of OpenFlow switches and controllers are currently available. In particular, software-based switches such as Open vSwitch (OvS), Trafficlab 1.1, Indigo, CpQd could have been used for CityFlow experimentations. This also includes a large number of controllers such as Open DayLight, NOX, POX and Floodlight. As previously mentioned, Floodlight was chosen as the reference controller for the CityFlow project due to its RESTFul API and associated high performance, which was not rivaled by any other controller when this research work was conducted. Moreover, OvS was chosen over the reference OpenFlow switch implementation due to its production quality and stability.

5.4.2 Topology setup on the OFELIA testbed

The OFELIA testbed has 10 islands located in different places in Europe (Germany, Belgium, Switzerland, Spain, UK, Italy and others.). The iMinds island, which is located in Belgium, has resources to perform large-scale emulations and has the ability to emulate multiple AS experiments. Therefore, most of the CityFlow experiments are performed on the iMinds island. However, one of the experiments is also performed on different islands in which some of islands worked like autonomous systems of the Internet.

5.4.2.1 Topology setup on the iMinds island

The iMinds island has a limitation that there can only be a maximum of 100 physical nodes in an experiment with a maximum of 6 interfaces per node. Therefore, for the experiments, we converted the reference Flowville scenarios to an experimental setup (Fig. 5.5), which could be implemented in the iMinds island.

The access networks in the island are implemented by nodes USER1, USER2 and USER3. In these nodes, multiple access clients (the numbers are described

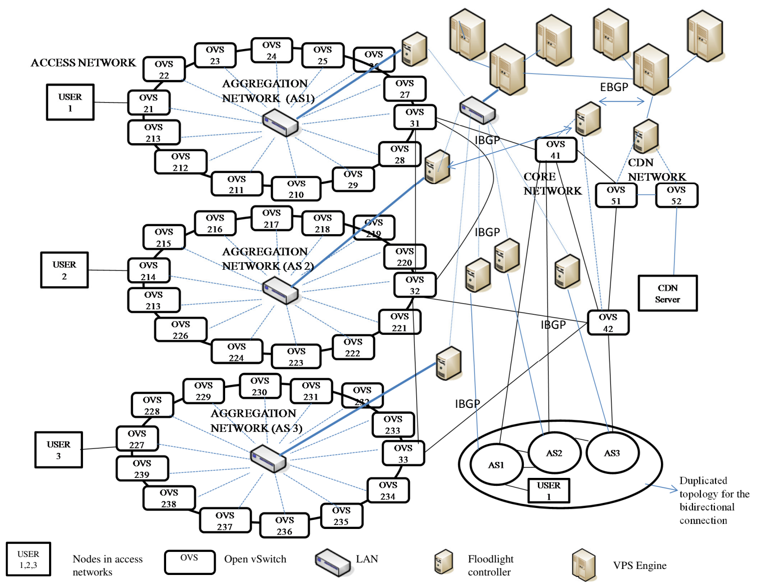


Figure 5.5: Flowville for CityFlow experimentation on the OFELIA testbed

in Section 5.3) are emulated using virtual interfaces. There are three ASs (AS1, AS2 or AS3) that represent aggregation networks, one AS represents the core network and one AS represents the CDN network, running OvS for forwarding user traffic. Each of these ASs is connected with a separate Floodlight controller. Similarly, OvSs representing the aggregation networks and CDN AS nodes are also connected with the VPS Engine through the connected Floodlight controller. For load sharing, we use three VPS Engines (shown in Fig. 5.5). In addition, for bidirectional experiments (i.e., user to CDN and CDN to user), we duplicate the access and aggregate networks as shown in Fig. 5.5.

5.4.2.2 Topology setup in multiple islands

All the islands have real OpenFlow switches and virtual machines (VMs) to perform experiments. As users do not have low-level access in OpenFlow switches, we chose virtual machines present in each island to perform CityFlow experiments. Additionally, we extended the Redzinc lab[5] at Dublin to the OFELIA testbed. Therefore, the Redzinc lab also worked like an additional OFELIA island in our experiments. We configured 4 virtual machines (VM) at each of the following islands: TUB (Berlin), ETH (Zurich) and Dublin islands. Our objective

[5] www.redzinc.net

was to setup an OpenFlow environment on the virtual machines of each island and to make virtual machines of each island behave like an autonomous system. For topology creation of the autonomous system, we needed to setup virtual interfaces on top the interfaces of the virtual machines in different islands. For creating virtual interfaces to create the required topology, we used GRE TAP interfaces on the top of interfaces of virtual machines. Additionally, VPS, Floodlight Controller and Open vSwitch are installed on the virtual machines of the islands.

5.4.3 Scale of test platform

Establishing very large topology emulation in the OFELIA testbed was a challenging task due to: (1) installation and running of software on many nodes, collecting the debugging data, and parsing the debugging data. To overcome the challenge, we made Linux images in which OvS, Floodlight, RouteFlow, and VPS are already installed. In addition, we made scripts to automatically run OvS, Floodlight, RouteFlow in the large-scale experiment of CityFlow. For collecting and parsing the debugging data, we built a measurement system[6] (Fig. 5.2).

Another challenge for test platform was to configure RouteFlow. Before running RouteFlow, an administrator needs to devote significant amount of time in configurations. For a large topology (typically for 100 switches), it may take many hours to configure RouteFlow. To overcome the issue, we proposed and implemented a framework to automatically configure RouteFlow [10]. In this framework, we use an additional controller module, which runs a topology discovery module to know network configurations. The network configurations are then sent to RouteFlow[7].

5.4.4 Implementation of test harness

In order to provide a realistic test of the topology and technology we implemented a test harness designed to provide high volume control plane (CP) traffic and to trigger appropriate traffic generators for data plane (DP) traffic. This allowed a stress test of the VPS server alone (by not enabling DP traffic) and of the whole CityFlow stack. We were able to supply the test harness with recipes for different traffic mixes; for example different rates of CP traffic where we could control the duration, inter-arrival time and magnitude of the traffic. We are also able to overlay more than one traffic recipe and be selective about which recipes trigger associated DP traffic[8].

[6]The source code of the measurement system is available at www.cityflow.eu

[7]The source code of RouteFlow configuration is available at:
https://github.com/routeflow/AutomaticConfigurationRouteFlow

[8]The source code of the test harness is available at www.cityflow.eu

5.5 Results

We performed four different experiments in CityFlow: (1) data traffic, (2) control traffic, (3) failure recovery, (4) multiple islands. The first three experiments are performed on the iMinds island, the fourth one is performed on multiple islands of OFELIA. Additionally, the control traffic experiment (the second experiment) was also performed on the Amazon cloud facility.

5.5.1 Data traffic experiments

In order to understand how the data plane responds in different situations, using the setup previously presented, data traffic was rate-limited and forwarded through OpenFlow switches dynamically configured by the VPS. In particular, this experiment aims at emulating and analyzing the performance of typical video streaming, being shaped (resorting to queues) at the edge of the source network.

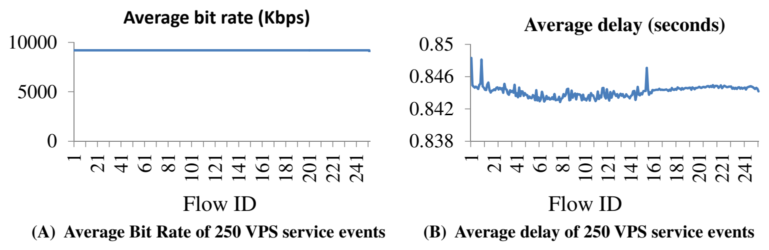


Figure 5.6: Data traffic experiments

Having considered two variations of the same setup, the first version of the experiment measures the performance of traffic where 13.75Mbps of UDP packets, resembling a typical HD video stream, are sent from source to destination, being inserted into a queue configured with maximum bandwidth rate at 10Mbps for each of 250 invocations. The route of the flow is same as the route discovered by routing protocols in the current Infrastructure. The chosen limit for the created queue has around 30% less bandwidth, allowing the impact of this queue to be noticeable throughout the experiment. The purpose is to understand how traffic injected in both ways (unidirectional and bidirectional), such as interactive video between source and destination, impacts the overall system performance. The considered traffic pattern consisted the injection of 250 flows with a duration of 45s per flow. The results for measured bit rate and delay are shown in Fig. 5.6. Another important aspect of this experiment is creation and installation of queues and their associated flows, which implies the creation and deletion of 250 queues per receiving site (500 queues in the bidirectional scenario). The results of queue creation and removal (with average throttled speed) are depicted in Table 5.1.

Table 5.1: Data traffic experiment results

Average Queue Installation time	Average queue removal time	Source speed	Capacity	Throttled Average speed
48.3 ms	452ms	13.75Mbps	10Mbps	9.2Mbs

The obtained results revealed that the VPS engine was able to cope with the amount of requested invocations, efficiently issuing the respective queue and flow management. Moreover the process of creating or installing queues revealed to be quite efficient, taking on average less than 50 milliseconds. On the other hand, the queue removal or deletion process took unexpectedly more time (approx. 450ms, Table 5.1). After a close analysis we were able to conclude that this was due to internal OvS database verifications for consistency, regarding the installed flows and queues for each QoS entry. Regarding the flow management operations there was no significant limitation or variation, presenting a very good performance in data traffic scenarios. These results also showed that the expected traffic shaping introduced by the create queues successfully limits the amount of transmitted data. The registered average value is around 9.2Mbps, which is lower than the expected 10Mbps.

5.5.2 Control traffic experiments

For this experiment, a city of 1 million inhabitants was considered and assumed that the service provider has a penetration in such a market of 20%, giving a possibility of 200,000 users. Two busy periods during the day were considered: a mid-morning period driven by enterprise traffic and an early evening period driven by domestic traffic (2 hours of duration each). In the busy period, it was considered that 75% of the users are active. We consider two cases in this experiment: (1) baseline and (2) expansive. In the baseline case we consider that each customer demands 1 event during the busy hour. This baseline case equates to a requirement to handle 75,000 events during the busy hour. In the expansive case we consider that each customer demands 3 events during the busy hour. This expansive case equates to a requirement to handle 225,000 events during the busy hour.

We then consider what engineering headroom is needed for expansion. In a voice network today, service growth is low as voice is a mature service, so the systems operate with a low headroom, but in the Internet with the rapid arrival of new services, growth can be quite fast. So in the expansive case we consider a headroom factor of 2. In a production network it typically takes 6 to 12 weeks to upgrade capacity so a headroom of 2 seems reasonable to cope with a 100% increase in demand. Therefore, for the expansion case, the VPS Engine needs to handle 450,000 events during the busy hour.

In the iMinds testbed we emulate the baseline case and in the Amazon cloud facility, we increase the number of invocations and show that how many invocations can be handled by the VPS Engine for the expansion case.

5.5.2.1 Experiment on the iMinds testbed

With the purpose of assessing the VPS controller, and its associated software stack performance, this experiment submits it to a high-load of signaling requests (i.e., mimics a high number of users). For this purpose a pulse generator was employed to create signaling pulses, in a two-hour interval distributed according to a Poisson distribution and with random service duration of between 3 minutes and 30 minutes. This was employed for a closer approximation to a real world scenario, focusing on the measurement of the Busy Hour Flow Invocation (BHFI) indicator, which determines the number of simultaneous flows supported by the VPS controller.

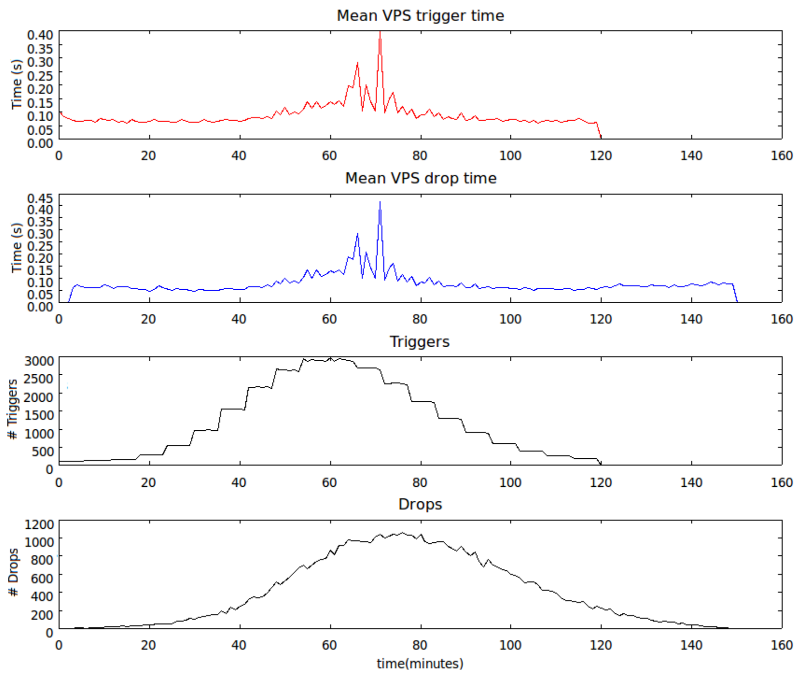


Figure 5.7: 75000 Busy Hour Flow Invocations on the Virtual Path Slice Engine

The obtained results (Fig. 5.7) revealed that even under high-load, and for prolonged period, the VPS engine is capable of handling triggers in under 400ms for the baseline case (with the arrival of invocations in Poisson distribution). The VPS trigger time in Fig. 5.7 is the time to trigger allocation of resources for the invocation requests, and the VPS drop time is the time to delete the resources of the invocations. From the results, it is concluded that the VPS Engine can scale to a high volume of flow invocations and terminations, to support a busy hour flow invocation capacity of 75000 events on mid-range servers.

5.5.2.2 Experiment on the Amazon cloud facility

An initial investigation into cloud deployment was carried out to examine the ability of the VPS to handle high rate of requests, find out suitable deployment configurations and to document the scalability and useful characteristics.

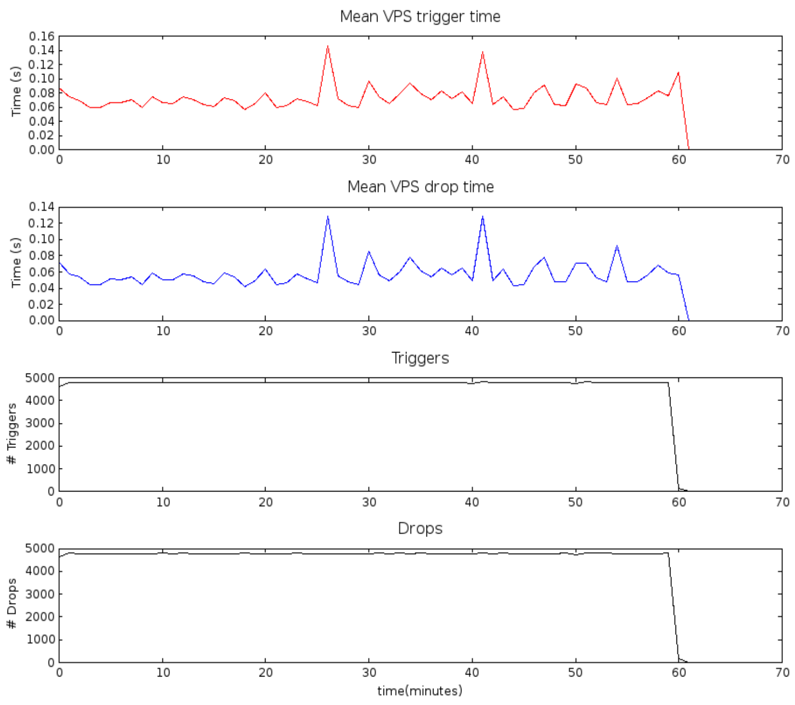


Figure 5.8: Scaled experiment for high volume invocations on Amazon

In this experiment, the VPS engine showed the capability to handle up to 4800 requests per minute (see Fig. 5.8). That is, up to a total of 288000 requests in a one-hour period that would constitute a Busy Hour, well above the estimated 75000 requests in the baseline situation. The results show a BHFI capacity of 288000 on one server, which is 64% to our target of 450,000 in the expansion situation. This number, however ultimately depends on the performance of the associated OpenFlow controller and network hardware. One, two, and three server Cloud deployments were used as the basis of comparisons with the rest of the deployments. In one server deployment, VPS is deployed in one VM and the test harness is deployed in another VM, both of which are in the same Amazon availability zone.

In the two-server deployment scenario, one VPS server contains all components - including the MySQL database and the other VPS server contains just Translation Layer and Invocation Controller components and has to communicate to the remote VPS server with the common database component to coordinate the scheduling. These components are placed behind an Amazon Elastic Load Balancer. Surprisingly, response times when the two VPS servers were used via the Amazon ELB were more than the single-server scenario. This indicates that the communication time between the VPS sever containing just the processing components and the remote common database may be impacting performance. The results obtained from the two-server deployment of VPS indicated that it may be better to deploy the common database and the components of VPS that control access to this database to a separate server. This would mean that each VPS server will be performing processing tasks alone and have to access a separate remote server to gain access to the common database which holds the scheduling data. This should give a similar communication costs to the two VPS servers and allow investigation of scalability with comparable processing and communication costs to each VPS server involved. In the three-server deployment scenario, the response times for requests did not show any improvements over testing against a single VPS server.

5.5.3 Failure recovery experiments

For failure recovery experiments, we implemented a framework [15] with which high quality of service can also be achieved in failure conditions. In this framework, under failure conditions, the controller reroutes traffic to a failure-free path gained through BGP. Regarding the implementation, we did not focus on fast failure recovery [16], but instead, we focus on scenarios in which high-priority should be provided to high-priority over best-effort users. We conducted emulations on the iMinds islands with different rate of traffic and one of the links between aggregation and core networks is failed (or made down). With results (Fig. 5.9), we were able to conclude that when there is enough bandwidth of the failure recovery path, neither high-priority nor best-effort traffic gets affected after re-routing traffic to a failure-free path. On the other hand, when there is limited available bandwidth, best-effort traffic experiences packet loss in order to meet the requirements of high-priority traffic. Finally, when the total amount of bandwidth is insufficient even for high-priority traffic, despite

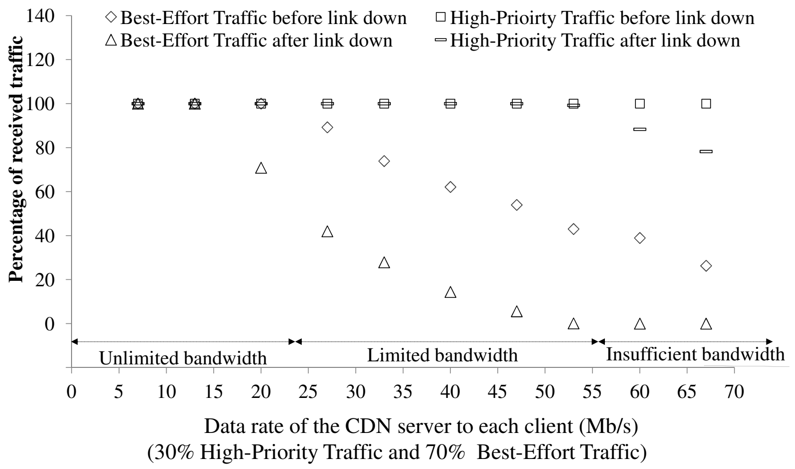


Figure 5.9: Failure recovery experiment results

registering some losses it still maintains its priority above best-effort traffic, which causes no interference.

5.5.4 Multiple island Experiment

We performed experiments for the multiple island scenarios described in Section 5.3. The topology of the experiments is shown in Fig. 5.10.

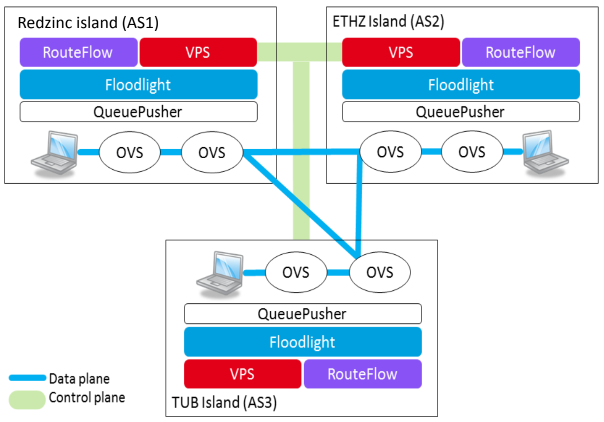


Figure 5.10: Multiple islands experiment on the OFELIA testbed

For this, we calculated response time (time taken to response a user request), CPU usage, and memory usage of VPS. All the tests used the ETHZ island as the source, having the services triggered by the test harness from a user machine installed on the aforementioned island. To validate the experiments, two tests were performed: (1) triggers originating from the ETHZ island were performed, having the Dublin island as destination during 1 hour duration; (2) triggers originating from the ETHZ island were performed, having the TUB island as destination during 1 hour.

Table 5.2: VPS machine CPU usage and memory usage

Indicator	Dublin island (Average)	TUB island (Average)	ETH Zurich (Average)
CPU usage	33%	40%	35%
Memory usage	21%	15%	41%

Table 5.2 presents the CPU usage and memory usage of the VPS machine during the span of experiments.

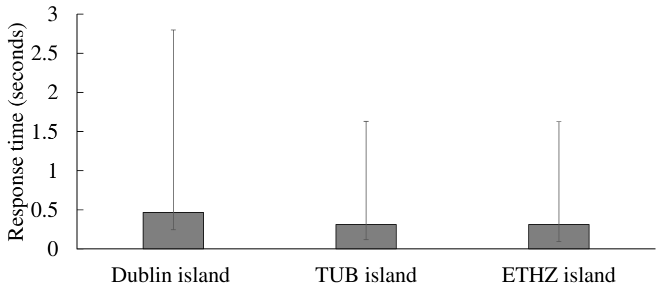


Figure 5.11: Response time. Error bars show the minimum and maximum response time

Fig. 5.11 shows the response time in different islands. It shows that the response time in these islands is significantly longer than the iMinds islands (see results in previous subsections). Due to the fact that the machines present on these OFELIA islands, as well as the Dublin island, are underpowered (i.e., virtualized) when compared with the machines on iMinds island could justify the increase in response time. Additionally, the results obtained against TUB and ETH islands show a small difference between them.

5.6 Conclusions

In this article we have presented the CityFlow project's proposal of dynamic OpenFlow capable public networks, capable of addressing the challenge of the current and future Internet data-traffic growth. Large-scale experiments for a city of one million inhabitants were performed in order to demonstrate the feasibility of this proposal, considering scenarios involving multiple autonomous systems. These experiments were undertaken on multiple islands of the OFELIA testbed emulating different technology networks.

Based on the obtained results, we have confirmed that an OpenFlow network with virtual path slice coordination between multiple autonomous systems is feasible and is able to achieve all of the proposed objectives. These results motivate future research, upgrading the used network emulation platform, based on OpenFlow software-switches, into a a production network for validating the proposed architecture with OpenFlow-enabled hardware switches. Moreover, this conclusion further reveals that support of quality of service queues on commercial OpenFlow switches would be required in the future.

Acknowledgment

This work was conducted within the framework of the FP7 CityFlow project (agreement no. 317576), which is partially funded by the Commission of the European Union.

References

[1] Aref Meddeb. *Internet QoS: Pieces of the Puzzle*, IEEE Communications Magazine, vol. 48 (1), pp. 86-94, 2010.

[2] N. McKeown, T. Andershnan, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *OpenFlow: Enabling innovation in campus networks*, ACM Computer Communication Review, Vol. 38(2), pp. 69-74, 2008.

[3] T. Braun, M. Diaz, J. E. Gabeiras, Staub, *End-to-End Quality of Service Over Heterogeneous Networks*, Springer, 2008.

[4] S. Jordan, *Implications of Internet architecture on net neutrality*, ACM Transactions on Internet Technology, vol. 9(2), Article 5, 2009.

[5] Federal Communications Commission FCC 10-201. [Online]. Available: https://apps.fcc.gov/edocs_public/attachmatch/FCC-10-201A1.pdf.

[6] M. Sune, L. Bergesio, H. Woesner et. al., *Design and implementation of the OFELIA FP7 facility: the European OpenFlow testbed*, Computer Networks, Vol. 61, pp. 132-150, 2014.

[7] R. Hancock, G. Karagiannis, J. Loughney, S. Van den Bosch, *Next Steps in Signaling (NSIS): Framework*, RFC 4080, IETF, 2005.

[8] C. Esteve Rothenberg, Marcelo R. Nascimento et.al, *Revisiting Routing Control Platforms with the Eyes and Muscles of Software-Defined Networking*, HotSDN, Helsinki, Finland, Aug 2011.

[9] D. Palma, J. Goncalves, B. Sousa, L. Cordeiro, P. Simoes, S. Sharma, D. Staessens, *The QueuePusher: Enabling Queue Management in OpenFlow*, EWSDN, 125-126, 2014.

[10] S. Sharma, D. Staessens, D. Colle, M. Pickavet, P. Demeester, *Automatic configuration of routing control platforms in OpenFlow networks*, ACM SIGCOMM Computer Communication Review, Vol. 43(4), pp. 491-492, 2014.

[11] B. Pfaff and B. David, *The Open vSwitch Database Management Protocol*, RFC 7047, 2013.

[12] M. Kennedy, *A TCO Analysis of Ericsson's Virtual Network System Concept Applied to Mobile Backhaul*, ACG Research Inc., 2012.

[13] J. P. Pereira, *Telecommunication Policies for Broadband Access Networks*, in Proceedings of the 37th Research Conference on Communication, Information and Internet Policy, pp. 1-13, 2009.

[14] M. V. Wee, K. Casier, K. Bauters, S. Verbrugge, D. Colle, M. Pickavet, *A modular and hierarchically structured echno-economic model for FTTH deployments*, ONDM, pp. 1-6, 2012.

[15] S. Sharma, D. Staessens, D. Colle, D. Palma, J. Goncalves, M. Pickavet, L. Cordeiro, and Piet Demeester, *Demonstrating Resilient Quality of Service in Software Defined Networking*, IEEE INFOCOM, pp. 133-134, 2014.

[16] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and Piet Demeester, *OpenFlow: Meeting carrier-grade recovery requirements*, Computer Communications, Vol. 36(6), pp. 656-665, 2013.

6 Inter-Burst Segregation Protocol guaranteeing loss-free packet-switched networks

This chapter proposes the inter-burst segregation protocol (IBSP) for guaranteeing zero packet-loss in packet-switched networks. For implementation, the protocol requires three buffers at each port of network devices and an additional functionality (i.e., transmission/processing of end-of-frames after periodic intervals). The protocol is implemented and evaluated using traditional packet-switched networks. For the implementation of the protocol in OpenFlow, three buffers per port can be created using the queuing functionality described in Chapter 4 and Chapter 5. However, the additional functionality (i.e., transmission and processing of end-of-frames) is not currently supported by OpenFlow switches/routers. In future, OpenFlow switches/routers can be extended to add this functionality.

⋆ ⋆ ⋆

Sachin Sharma, Didier Colle, Wouter Tavernier, Mario Pickavet, and Piet Demeester

Submitted to IEEE Communications Letter, February 2016.

Abstract Traditional packet-switched networks suffer from a number of limitations (such as packet-loss, jitter, and operational efficiency). The current

solutions (IEEE 802.17 for optical fiber and IEEE 802.1Qbb for Ethernet) to overcome the limitations are distance dependent. We address these shortcomings and propose a novel protocol called Inter-Burst Segregation Protocol (IBSP), which guarantees zero packet loss and low jitter. The protocol is implemented and tested in Network Simulator-3 (NS-3) and in the DPDK (Data Plane Development Kit) platform implemented by INTEL. The results confirm that using IBSP, bandwidth can be used almost upto 100% without having any packet loss. In addition, the jitter using IBSP is low and bounded.

6.1 Introduction

TDM (Time Division Multiplexing) based networks, such as SONET/SDH (Synchronous Optical Networks/Synchronous Digital Hierarchy), deliver high Quality of Service (QoS) in terms of packet-loss, jitter, and bandwidth. This is because resources (i.e., bandwidth) are statically divided into different time slots in periodic time intervals (i.e., timeframes) and each user is allocated a unique time slot (in each timeframe) to transmit data traffic. However, the problem is that if a user (for example, transmitting bursty traffic such as media traffic) does not have traffic to transmit, its allocated time slot (i.e., resources) gets wasted. This problem is solved by packet-switched networks in which resources are dynamically shared with many users. However, due to accumulated congestion, these networks cannot guarantee QoS in terms of packet-loss, jitter, and bandwidth.

Many standards, such as IEEE 802.17 for optical fiber [1] and IEEE 802.1Qbb (approved in 2011) for Ethernet [2], have been proposed to decrease the packet loss. In these standards, a control packet is sent back to the sender to notify about congestion and hence, to control its traffic rate (or burstiness). The issue here is that if the control packet is lost or there is a large distance between the congested and sender node, the former may suffer buffer overflow and packet-loss can happen [3, 4]. Furthermore, QoS mechanisms such as DiffServ (differentiated services) are proposed to deliver QoS to high-priority traffic in packet-switched networks. However, Diffserv can only work for (small) fractions of high priority traffic [5]. In addition, it does not guarantee zero packet-loss.

Moreover, traffic shaping (e.g., leaky bucket) at the ingress nodes is proposed to control burstiness in a packet-switched network [6]. However, due to many factors (e.g., interference among different flows), burstiness may still increase within the network and packet-loss can happen.

In this article, we first describe the possibility of packet-loss (i.e, increased burstiness) in packet-switched networks (Section 6.2) and then propose a novel protocol, Inter-Burst Segregation Protocol (IBSP), to guarantee zero packet-loss (Section 6.3). In IBSP, each node controls burstiness by separating different bursts of each source. The basic principle of IBSP is taken from SONET/SDH, where there are periodic time intervals (i.e., timeframes) and the idea is to make sure that traffic in different timeframes on an incoming port is transmitted in separate timeframes on an outgoing port. Using this principle, different bursts of a source can be separated by transmitting them in separate timeframes (i.e., time intervals).

Compared to SONET/SDH, packet-switched networks have an advantage that if a flow does not have traffic to transmit in a time interval, the allocated resources may be used by other flows (e.g., best-effort flows). We implement IBSP using three buffers at each port. The solution is implemented in Network Simulator-3 (NS-3) and in DPDK (Data Plane Development Kit) provided by INTEL. Our results show that using IBSP almost all the bandwidth can be used without having any packet loss. Using IBSP, the jitter is low and bounded.

6.2 Packet-loss in packet-switched networks

In this section, we present a worst case network scenario (Fig. 6.1) in which packet-loss can happen, although the offered traffic load in the network is less than (or equal to) the load that can be handled by the network. Therefore, we assume that the link capacity (in bits/s) between devices (e.g., routers or hosts) is at least equal to the average of incoming rates of different data transfers from customers. The rate of a data transfer depends on the bandwidth assured in a service level agreement (or contract) with the customer.

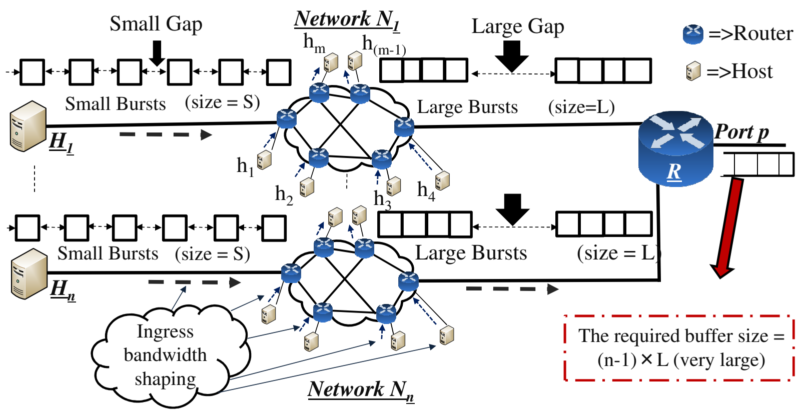


Figure 6.1: Worst case network scenario of lossy packet-switched networks

In Fig. 6.1, host H_i ($1 \leq i \leq n$) is directly connected with network N_i which is connected with Router R. Router R is then connected with other routers (not shown in Fig. 6.1) through port p. As shown in Fig. 6.1, N_i may contain many routers and hosts (h_j, where $1 \leq j \leq m$), and therefore, bursts of H_i may have to suffer interference from bursts of other hosts connected to the network. In addition, bursts from all H_i have to collide at router R. To illustrate packet-loss, we derive the required buffer size of router R at port p.

In Fig. 6.1, traffic shaping such as leaky bucket is applied at all the ingress nodes (i.e., H_i and h_j). Leaky buckets shape traffic such that burstiness is bounded and on top of that it limits traffic by dropping (or marking) excess traffic beyond the contract or SLA (service level agreement). In case of a well-dimensioned non-overbooked network, using leaky buckets the average load will not exceed what the network can handle.

In the network scenario (Fig. 6.1), bursts originating from H_i are small in size (S) and when a burst from H_i passes through N_i, due to interference from bursts of other hosts (h_j), the burst of H_i may have to wait in the network until the bursts of other hosts are transmitted. Therefore, it may happen that the current burst catches up with previous bursts of H_i, which have been queued somewhere in the network and hence, makes a large burst (size = L) at the end of N_i. The value of L can be represented as Eq. 6.1, i.e., as a function of the size of bursts originated from H_i (i.e, S), the average number of interfering flows per hop (i.e., f), the average delay (i.e., d) occurred per interfering flow, the number of hops traveled (i.e., h), and the time interval at which bursts are transmitted by H_i (i.e., t).

$$L = F(S, f, d, h, t) \tag{6.1}$$

Let T be a time interval in which a burst of size L is transmitted from router R through port p (i.e., $L = bitrate \times T$, where $bitrate$ is the bandwidth ($bits/s$) of port p). As network scenarios for all N_i are same in Fig. 6.1, router R receives bursts (i.e., each having size L) from n hosts (i.e., H_1 to H_n) at interval T. So, the total size of incoming bursts at R is $n \times L$. As the size of an outgoing burst at interval T is given by L, the buffer size requirement of port p is given by:

$$B_p = n \times L - L = (n-1) \times L \tag{6.2}$$

Eq. 6.2 illustrates that if L is very large (i.e., many bursts of hosts catch up with previous bursts), router R requires a very large buffer in order to guarantee zero packet loss. The presence of large buffers may result into unnecessary latency and poor performance (bufferbloat problem [7]). However, in the absence of large buffers, packet-loss will happen.

6.3 Inter-Burst Segregation Protocol (IBSP)

This section provides a description about our proposed protocol, which guarantees loss-free packet-switched networks.

6.3.1 Our approach

Like SONET/SDH, IBSP transmits data (i.e., a stream of packets) in timeframes and in each timeframe, data from multiple users are transmitted. The data reserved (or transmitted) for a user in a timeframe depends on the bandwidth assured in an SLA with the user. Unlike SONET/SDH, IBSP has an advantage that resources reserved for a user in a timeframe can be used by other flows (such as best-effort flows), when the user does not have packets to transmit.

By transmitting packets in timeframes, IBSP controls burstiness (i.e., separates bursts of each source) in each device and hence, prevents making a large burst (e.g, a burst of size L in Fig. 6.1). Using IBSP, bursts (size S) of H_i always remain separated from their previous bursts in Fig. 6.1. For controlling burstiness, IBSP proposes three buffers (of equal size) in each port, and needs to perform ping-pong buffering [8] for filling and emptying buffers. In ping-pong buffering, while one buffer is being filled, the other buffer is emptied. The difference between our mechanism and ping-pong buffering is that the latter uses two buffers, while our mechanism uses three buffers (the reason of using three buffers is explained in the next subsection). However, the main objective of both the mechanisms is similar (i.e., filling and emptying are not performed on the same buffer at the same time). The followings are the three main activities of IBSP:

1. **Timer-start activity**: Each node starts a timer of a fixed duration (referred to as timeframe). This duration is assumed to be equal for all nodes in a network.

2. **Timer-expiration activity**: Each node notifies the expiration of the timeframe to its neighboring nodes by transmitting a control packet (end-of-frame, EOF).

3. **Inter-burst separation activity**: Each node ensures that bursts in different timeframes on an incoming port are transmitted in separate timeframes on an outgoing port.

The timer-start activity can be implemented by running a periodic timer that measures the duration of timeframes. The timeframe size is fixed and same for each node in the network.

The timer-expiration activity can be performed by transmitting an EOF when the duration of a timeframe expires. The EOFs are used by neighboring nodes to know that all the packets received after an EOF on an incoming port belong to a different timeframe. Therefore, in the inter-burst separation activity, packets of different timeframes on an incoming port can be transmitted in separate timeframes on an outgoing port.

The inter-burst separation activity starts at the beginning of the timer-start activity and completes at the end of the timer-expiration activity. During this

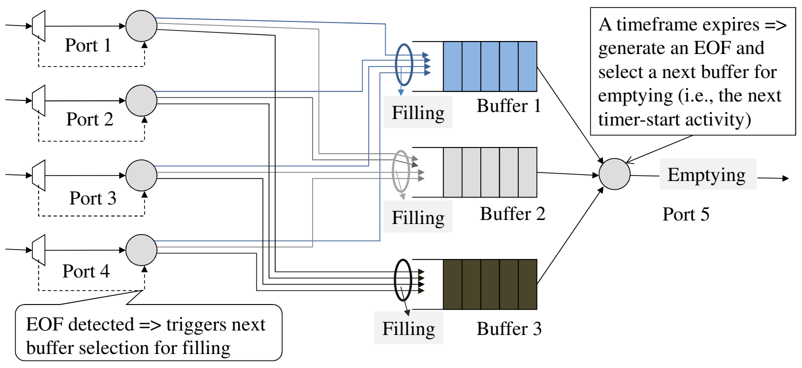

Port 1
Port 2
Port 3
Port 4
EOF detected => triggers next buffer selection for filling
Filling
Buffer 1
Filling
Buffer 2
Filling
Buffer 3
A timeframe expires => generate an EOF and select a next buffer for emptying (i.e., the next timer-start activity)
Emptying
Port 5

Figure 6.2: Filling and emptying buffers. Each buffer (i.e., buffer 1, 2 and 3) has a capacity to accommodate one timeframe of bursts. In addition, bursts of different timeframes are in separate buffers.

activity, each node performs two tasks: (1) fill a buffer and (2) empty a buffer. For filling a buffer of an outgoing port (see Port 5 in Fig. 6.2), IBSP depends on EOFs received from incoming ports (see Port $1, 2, 3, 4$ in Fig. 6.2). By default, in beginning, IBSP selects a buffer, which is not currently used for emptying. Then, each time when an EOF is received on a port, the next buffer (chosen in a round-robin fashion) is selected for filling packets from that port.

For emptying a buffer, IBSP depends on the timer-start activities. At the triggering of the first timer-start activity, IBSP selects one of the buffers (in each port) for emptying. It then selects the next buffer (chosen in a round-robin order) at the occurrence of the next timer-start activity (see at Port 5 in Fig. 6.2). This process is then repeated forever.

In IBSP, the duration of timeframes is assumed to be equal for all nodes in a network and is given by T_f (Eq. 6.3). In Eq. 6.3, there are $F(p)$ flows to be transmitted from port p in a network, and each flow f_i has allocated an amount $s(f_i)$ (in bits) of data to be transmitted in each timeframe. $s(f_i)$ depends on the contract (SLA) with the user of flow f_i.

$$T_f = \max_{\forall p\, in\, a\, network} \left(\frac{\sum_{i=1}^{F(p)} s(f_i)}{bitrate}\right) + padding \tag{6.3}$$

The $padding$ time (shown in Eq. 6.3) is described in Section 6.3.3 and $bitrate$ is the bandwidth (in bits/s) of each port. Furthermore, each buffer in IBSP is assumed to have a capacity to accommodate data of the timeframe duration (i.e., $T_f \times bitrate$).

6.3.2 Justification of using three buffers

IBSP proposes three buffers (instead of two) in each outgoing port. However, if input is exactly aligned with output (i.e., EOFs from incoming ports are received exactly at the same time when the timer-start activities are triggered by IBSP), IBSP only requires two buffers. Fig. 6.3A illustrates such a case. It shows that two buffers are sufficient for implementing IBSP, as filling and emptying tasks (described above) can be performed on separate buffers all the time.

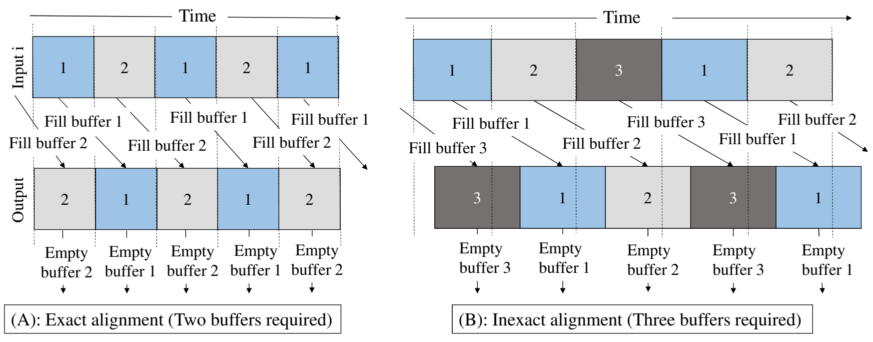


Figure 6.3: Buffers for implementing IBSP. 1, 2, and 3 are the buffer numbers and all rectangles are timeframes.

The problem is that it is not possible to make inputs aligned exactly with the output (i.e., due to the clock difference between two nodes, propagation delay, etc.). Because of this issue, it may happen that a buffer (in case of two buffers) is filled (EOF is received from an input) and the output is still transmitting from the other buffer (i.e., the current timeframe is not yet expired). So, at this time, no buffer can be used for filling. To solve the issue, we propose the third buffer in each port (Fig. 6.3B). Buffers are numbered 1, 2, and 3 in Fig. 6.3B. Therefore, when an EOF is received while emptying buffer 1, the input selector is advanced from buffer 2 to 3. In addition, when an EOF is received while emptying buffer 2, the input selector is advanced from buffer 3 to 1. Furthermore, when an EOF is received while emptying buffer 3, the input selector is advanced from buffer 1 to 2.

6.3.3 Solutions to the issues of our approach

There are still two additional issues that our approach needs to resolve. These issues may occur due to the clock difference between two nodes. This clock difference can be very marginal (e.g., can be expressed in parts per million, i.e., ppm).

1. **The next selected buffer for filling is the buffer that is currently used for emptying**

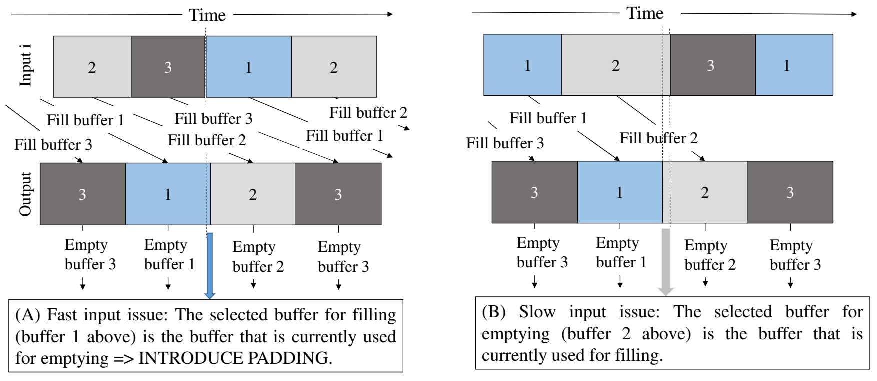


Figure 6.4: Issues that may occur due to clock difference between different nodes

This issue may occur due to the fast clock of a neighboring node. In Fig. 6.4A, when the input selects buffer 1 for filling, the output is still emptying this buffer. To solve this issue (i.e., input and output should not be on the same buffer), we propose *padding* (i.e., no data to be allocated) at the end of each timeframe. Therefore, when the issue occurs, the node should immediately expire the current timeframe and should start the next timer-start activity (i.e., emptying the next buffer). As we propose that a node should not transmit any data during the padding time, the expiration of the timeframe before the actual timer expires will not cause any packet-loss. The *padding* time can be equal to the size needed to accommodate the clock difference due to the fast input. As the clock difference can be expressed in ppm, a very small padding per timeframe is needed in IBSP.

2. **The next selected buffer for emptying is the buffer that is currently used for filling**

This issue may occur due to the slow clock of a neighboring node. In Fig. 6.4B, when the output selects buffer 2 for emptying, the input is still filling this buffer. To solve this issue, we propose that the input should immediately switch to the next consecutive buffer for filling (in Fig. 6.4B, the next consecutive buffer is 3), when the issue occurs. In this case, when sufficient padding is present at the end of the received timeframe, no data will be received anymore before the receipt of the upcoming EOF, which will advance the buffer for this input once more. Thus, virtually a timeframe at the output will not contain any data for this input port (slow input port).

6.3.4 Delay and jitter using IBSP

In IBSP, as each node first fills a buffer (duration of 1 timeframe) and then empties it, the delay per hop using IBSP can be of 1 timeframe duration. Furthermore, as due to slow input the buffer for filling is forcefully moved to the next buffer, the maximum delay per hop could be 1 timeframe longer than the normal case. Similarly, the jitter (end-to-end) can be within 1 timeframe and is 1 timeframe longer than the normal case when the slow input catches up with fast output (does not happen very often). As the size of timeframes can be short (e.g., $125us$), the jitter will be low using IBSP.

6.4 Experimental study

We performed NS-3 simulations for verifying the proof-of-concept of IBSP, and then performed emulations on a high performance platform (i.e., DPDK) to check the suitability of IBSP in a real environment. The worst-case scenario (shown in Fig. 6.1) is tested. The worst-case scenario is tested because it is difficult emulate such a scenario in real settings. The topology for N_i in Fig. 6.1 is depicted in Fig. 6.5. A chain of routers ($R_j, 1 \leq j \leq n$) is present in N_i in a linear fashion. Router R_j in Fig. 6.5 is also connected with host h_j through 4 ports (p_1, p_2, p_3, p_4), emulating 4 different flows generating collided bursts to bursts from H_i.

6.4.1 Simulations

In the simulations, host h_j transmits bursts to its neighboring host h_{j+1} through all of its ports and therefore, bursts of H_i (shown in Fig. 6.1) have to compete with bursts of h_j in the router's outgoing port (p_5) for forwarding. In order to create interference, each h_j in our simulation starts transmitting bursts such that these bursts reach to R_j just before the bursts of H_i. Therefore, bursts of H_i have to wait in R_j until bursts of h_j are transmitted. The propagation delay and bandwidth of each link in our simulation is $100us$ and 1 Gb/s respectively. In addition, the packet size is 624 bytes.

The bursts of H_i in our simulation have to pass through 100 routers to reach the destination node. In addition, bursts of H_i reach at router R after traveling 96 routers of N_i. There are five such H_i in our simulation. The $ontime$ of bursts originated from H_i and h_j is kept as $24us$. As there can be five small bursts in each router (including R_j and R), the timeframes in IBSP should be at least $120us$ (i.e., $5 \times 24us$). We keep $5us$ as the padding time. Therefore, the duration of timeframes in our experiment is $125us$. We also perform simulations of legacy packet-switched networks where only a single buffer is used. To make a fair comparison between the legacy packet-switched networks and the networks using IBSP, the size of the buffer in the legacy packet-switched networks is kept three

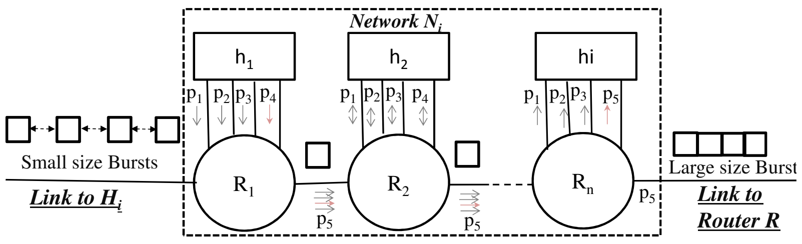


Figure 6.5: Topology of Network N_i in Fig. 6.1

times the single buffer size in IBSP. The bandwidth (BW) usage in our experiment is measured in router R at port p.

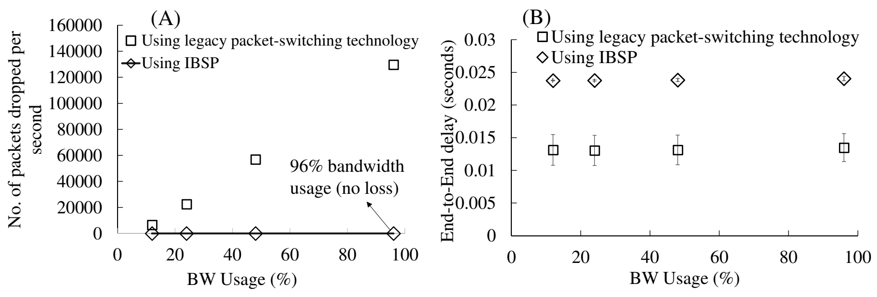


Figure 6.6: Zero packet-loss (see A) and low jitter (error bars in B) using IBSP

We run different experiments for a different bandwidth usage to see the impact on packet-loss, delay, and jitter. Fig. 6.6A illustrates that there is packet-loss in legacy packet-switched networks even at a low bandwidth usage (i.e., 12%). It then increases with the increase in the bandwidth usage. However, there is no packet-loss using IBSP. Moreover, the jitter is significantly low using IBSP (Fig. 6.6B).

6.4.2 DPDK emulations

For implementation, IBSP requires a platform which can process packets (incoming and outgoing) at line rate (i.e., very fast). This is possible by implementing IBSP in: (1) hardware or (2) high performance software. We have chosen the second option and hence, implemented IBSP in DPDK and have tested it in the FIRE testbed facility provided by iMinds [9].

The emulation scenario is similar to the scenario used for simulations. The difference lies in the number of routers in the linear chain and the size of bursts

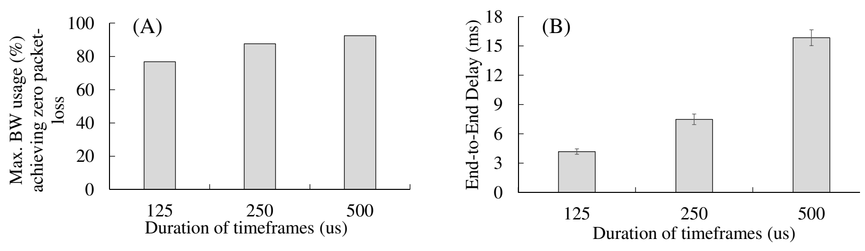


Figure 6.7: Emulation results using DPDK implementation of IBSP

generated by hosts. There are 20 routers in emulations and the burst size is chosen according to the timeframe size and the bandwidth usage shown in Fig. 6.7. In our implementation, due to timing inaccuracy in software, the padding time may include the time needed to process EOFs or the other packets. Therefore, we need to put additional padding in each timeframe to guarantee zero packet-loss. Fig. 6.7A confirms that using IBSP, the maximum of 76% of bandwidth can be used without having any packet-loss for short timeframes $(125us)$. However, when the timeframe size is increased, the bandwidth can be used more than 90%, without having any packet-loss. Fig. 6.7B illustrates that the end-to-end delay increases, as the timeframe size increases. In addition, the jitter increases, when the size of timeframes increases.

6.5 Conclusions

In this article, we have proposed a novel protocol, the inter-burst segregation protocol (IBSP), to guarantee zero packet-loss (with low jitter) in packet-switched networks. The results confirmed that traditional packet-switched networks cannot guarantee zero packet-loss, although the bandwidth usage is very low. However, using IBSP, zero packet-loss can be guaranteed, even though nearly all the bandwidth is consumed in the network. In addition, the jitter using IBSP is low.

Acknowledgment

This work was partly funded by ICON project - MECaNO (grant no. 130646).

References

[1] *IEEE Standard for Resilient packet ring (RPR) access method and physical layer specifications*, IEEE 802.17: RESILIENT PACKET RINGS, pp. 1–712, 2011 (Revised version).

[2] *IEEE standard for local and metropolitan area networksmedia access control (MAC) bridges and virtual bridged local area networks*, IEEE Std 802.1Qbb-2011, pp. 1–40, 30 2011.

[3] S.-A. Reinemo, et. al., *Ethernet for high performance data centers: On the new IEEE datacenter bridging standards*, IEEE Micro, vol. 30, no. 4, pp. 42–51, 2010.

[4] G. Rodrguez de los Santos et. al., *Buffer Design Under Bursty Traffic with Applications in FCoE Storage Area Networks*, IEEE Communications Letter, Vol. 17(2), pp. 413–416, 2013.

[5] T. Brans et. al., *End-to-End Quality of Service Over Heterogeneous Networks*, Springer, 2008.

[6] T. G. Orphanoudakis et. al., *Leaky-bucket Shaper design based on time interval grouping*, IEEE Communications Letter, Vol. 9 (6), pp. 573-575, 2005.

[7] J. Gettys et. al., *Bufferbloat: dark buffers in the internet*, Communications of the ACM, Vol. 55(1), pp. 57–65, 2012.

[8] Y. Joo et. al., *Doubling Memory Bandwidth for Network Buffers*, IEEE INFOCOM, pp. 808–815, 1998.

[9] M. Berman et. al., *Future internets escape the simulator*, Communications of the ACM, Vol. 58(6), pp. 78–89, 2015.

7 Concluding remarks and future directions

"I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right"

–Albert Einstein

Current operational networks are the results of over 40 years development, beginning with the ARPANET (in the late 1960s) and the formulation of the TCP/IP protocol suit (in early 1970s). At present, network devices (of operational networks) contain two elements: control plane and data plane, and a proprietary interface (i.e., closed implementation) to communicate between them. We are now moving towards the next major change in networking with the introduction of Software Defined Networking (SDN). OpenFlow is the current open de-facto SDN standard for communication between the control plane and data plane. In reality, OpenFlow is accepted by network industries and is feasible to implement, making it possible to move the control plane implementation to a separate network element (i.e., external servers) called controller. Moreover, it allows taking one control plane implementation and use it to steer different data plane implementations. This is useful because it prevents vendor lock-in (at least on the hardware side). In addition, network devices have become much simpler and inexpensive, as these do not have to deal with complicated and distributed information and decision-making (control plane). Furthermore, OpenFlow can also accelerate innovations for services, as it is easier to prototype them in software.

In this dissertation, we performed research on how OpenFlow networks can be adapted to be suited for future communication services. The research aimed at providing fast failure recovery, verification, automatic bootstrapping, high quality-of-service, and loss-free packet-switching solutions to OpenFlow.

For the fast-failure recovery research, we investigated how carrier-grade quality can be achieved in an OpenFlow network. To achieve carrier-grade quality, the network should be able to recover from a failure within 50 ms. In fact, in traditional networks, carrier-grade quality is achieved by first designing a network topology with failures in mind in order to provide alternate paths upon a failure. The next step is to add the ability to detect failures and react to them using a proper recovery mechanism. In OpenFlow, we first investigated how failures can be detected, which may be present due to link and node failures, and then two failure recovery mechanisms - restoration and protection - were proposed. Furthermore, extensive experiments were performed in a wide range of emulated network topologies in real settings on the testbed facility provided by iMinds. From the results of restoration experiments, we conclude that it is difficult for restoration to meet carrier-grade recovery requirements in a large scale network serving many flows. This is because due to the centralized nature of OpenFlow, the controller has to transmit lots of messages in the network to establish alternative paths upon a failure. This increases the load of the controller at the restoration time. In addition, the restoration time may increase significantly with the increase in the propagation delay, further strengthening the conclusion of our research that restoration cannot meet the requirement of 50 ms. In order to meet the carrier-grade requirement, we conclude that protection, where recovery actions are taken by OpenFlow devices themselves without contacting the controller, does not suffer from limitations of centralized control (i.e., load and propagation delay). From the results of protection experiments, it has been concluded that protection is a way in OpenFlow to meet carrier-grade recovery requirements.

In the next study, we proposed a mechanism to detect failures that can be present due to flow-matching errors in the OpenFlow data plane. The mechanism transmits test packets to find flow-matching errors. The study concludes that the verification time depends on the bandwidth available in the network for verification. If bandwidth is unlimited, verification can be achieved in a very short time interval. However, if bandwidth limitations exist, the verification time might increase significantly.

We also performed research about bootstrapping and quality of service (QoS) in OpenFlow. For bootstrapping, we proposed a method with which OpenFlow devices can be bootstrapped without having any manual configurations (in in-band and out-of-band networks). The research concludes that the proposed method allows bootstrapping in a minimal time, making it suitable for large networks. For QoS, we implemented a framework to achieve high quality service to high-priority users in OpenFlow networks. We implemented quality of service in the paths chosen by standard routing protocols (OSPF and BGP) in OpenFlow. The framework is tested in real settings on a large-scale European testbed facility (i.e., in the OFELIA testbed) by taking into account reference scenarios for a city of 1 million users. The results conclude that an OpenFlow network is able to achieve high QoS for high-priority users in all conditions including failure conditions.

Finally, we conclude that packet-switched networks (including OpenFlow networks) cannot guarantee zero packet-loss, although the overall bandwidth usage is low. Therefore, we proposed the inter-burst segregation protocol (IBSP) for guarantying zero packet loss. Based on the results obtained from the experiments with IBSP in simulations and in a high performance platform (i.e., DPDK), we confirmed the suitability of IBSP in packet-switched networks for guaranteeing zero packet-loss. However, for implementing IBSP in OpenFlow, additional functionality (i.e., transmission/processing of end-of-frames after periodic intervals) in OpenFlow devices is required. Therefore, in future these functionalities/extensions can be added in OpenFlow to guarantee zero packet-loss in OpenFlow networks.

7.1 Future directions - deploying SDN into operational networks

In this section, we provide future research directions and argue that where this PhD research fits into a broad scope of SDN.

7.1.1 Transition from legacy networks to SDN

We have seen that SDN has a number of advantages (such as designing a flexible network, fostering innovations, and reducing complexity). Therefore, currently researchers are finding ways to deploy SDN into operational networks [1]. The transition from a current legacy network to a completely software defined network may take long time. Hence, SDN devices may need to communicate with legacy devices in a network. Therefore, further research could be focused on:

1. How many of existing legacy devices of a network need to be upgraded to use the SDN technology?
2. How important is the actual placement of the legacy devices that have to be upgraded to SDN?

In fact, in order to deploy SDN devices in a network where there are some (or more) legacy devices, SDN should be able to run protocols that are currently used in legacy networks. These networks generally run IP routing protocols (such as

OSPF and BGP) to route packets towards destinations. To run such protocols in SDN networks, RouteFlow (discussed in Chapter 5) [2] can be used. In addition, we provided a framework (in Chapter 5 and Appendix B) which can automatically configure Routeflow with minimal configurations. This framework can be used in SDN to deploy its devices together with legacy devices.

7.1.2 Performance concerns

As a single controller can control many network devices in SDN, the future research can be focused on how many devices of a network can be controlled through a single controller, meeting all requirements (such as performance, e.g., latency, throughput, and reliability requirements). For a small network, a single controller may be enough to control all the devices of the network. However, as the network size increases, more events/requests will be sent to the controller and therefore, it may not be able to handle all the requests. The following three topics could be researched to overcome these problems: (1) inserting proactive flows, (2) deploying multiple controllers for a network device, (3) deploying multiple controllers where a part of a network is controlled by one controller and the other part is controlled by another controller. All these solutions are discussed in detail in following paragraphs.

The future research could be focused on how proactive schemes (i.e., establishing flows before a packet arrives at a device) can be deployed in OpenFlow networks. Proactive approaches do not insert additional delay in forwarding. In our research, we have shown that for issues such as failure recovery, these proactive schemes (i.e., protection) are more important than reactive schemes (e.g., restoration) because these schemes (i.e., proactive) do not require a high-speed controller. Implementing such a controller (i.e., high performance controller) only for restoration will probably not make sense because the high-speed controller is required momentarily at the time of a failure. Establishing proactive flows (i.e., protection mechanisms) in network devices for failure recovery will be more cost-efficient (Chapter 2 and Chapter 4). It may slightly increase the bandwidth requirement at flow setup time due to extra protection information to be sent to devices, but highly decrease the bandwidth requirements during failures, as alternative paths (established as proactive flows) to recover from failures are already established.

As discussed above, multiple controllers for a device can be deployed in a network for sharing the load of a controller. It can be further researched that if multiple controllers are present per device, which type of requests should be forwarded to which controller (for sharing the load). The simple mechanism could be that all the incoming TCP packets are forwarded to one controller and all the incoming UDP packets are forwarded to another controller. In future, all these mechanisms could be researched. In Chapter 3, we deployed an additional controller in a virtual machine to perform verification activities (for sharing the load with the controller). In operational networks, implementing such techniques would be useful to overcome performance concerns of the controller.

Furthermore, controllers can be deployed in a distributed manner where a part of a network is controlled by a separate controller. This is similar to our research on deploying quality of service mechanisms on multiple autonomous systems (in Chapter 5). In our research, we deployed one controller per autonomous system and for communication between different controllers, we used a traditional packet switching network protocol (i.e., BGP). As OpenFlow currently does not define protocols for controller-to-controller communications, new protocols could be researched for controller-to-controller communication. This is because some networks may not be running BGP for communicating between controllers of different domains.

Other issues for deploying OpenFlow networks are to handle controller failure and controller placement scenarios. In this dissertation (Chapter 2 and Chapter 4), we presented future directions on how controller failure scenarios can be deployed in OpenFlow networks, i.e., we can deploy two controllers (one as a master and the other as a slave) and when the communication with the master controller fails, network devices can rely on the other available controller in the network. However, there are still some concerns on how state synchronization (or consistency) between two controllers (i.e., master and slave) can be performed [3]. For other issues such as controller placement problems (e.g., in an in-band network), we can place a controller from where it becomes close to all devices in the network (i.e., at the center of the network). These placement algorithms are briefly discussed in Chapter 3.

The above future directions are related to controller issues. From the data plane side, there can be issues related to TCAM memory size. As this memory is costly, OpenFlow devices may contain a FlowTable with a limited size. The limited size of FlowTables can create many problems. For example, when a FlowTable becomes full, the device must first remove an old entry before inserting a new entry. If the entry, which is removed, represents an active flow in the network, the device has to send any subsequent packets from that flow to the controller, as they do not match any forwarding entries. This may increase the load on the controller. In addition, some of these packets can be dropped in the controller (or in the packet-in buffer), resulting in a significant drop in flow throughput.

The other issues related to the data plane side are for congestion (or packet-loss). We have shown in Chapter 6 that a packet-switched network cannot guarantee zero packet-loss. The same issue is present in an OpenFlow network. Therefore, we proposed inter-burst segregation protocol to guarantee zero packet-loss in packet-switched networks and described some extensions (i.e., transmission/processing of end-of-frames after periodic intervals) to implement it in OpenFlow networks. The future research could be focused on how these extensions can be implemented in OpenFlow.

7.1.3 Support for Network Function Virtualization

Although Software Defined Networking and Network Function Virtualization (NFV) are closely related terms, they are not really dependent on each other. NFV is similar to traditional server virtualization mechanisms, but focuses on networking services (or network functions such as network address translation, firewall, intrusion detection, domain name service and caching). Within NFV, there are virtualized network functions (VNFs). A VNF may consist of one or more virtual machines (running different software and processes) on top of commercial off-the-shelf (COTS) programmable hardware, switches and storage, or even cloud infrastructure (instead of having custom hardware devices for each network function). By leveraging virtualization technologies, NFV decouples network functions from proprietary hardware.

In fact, NFV and SDN are mutually beneficial. While NFV can be implemented without SDN (i.e., without the separation of the control plane from network devices), usage of SDN can simplify NFV for automatic configuration, facilitating compatibility with existing deployments, and enabling operation and maintenance procedures.

In [12], many challenges for NFV (i.e., performance, reliability, manageability, and security) have been discussed. Regarding performance, a great concern has been shown on how performance (such as throughput and latency) will be affected when virtualized network functions will be deployed on general-purpose servers. It is argued that using NFV it may be difficult to achieve the same performance as dedicated hardware. For manageability, the concerns are about instantiation of VNFs in the right locations at right time, and therefore, it includes dynamic allocation of hardware resources for them.

In traditional networks, network functions (deployed on dedicated hardware) devices can achieve strict requirements (e.g., reliability requirements) because they are specifically dedicated to perform these tasks. To meet the same requirements (i.e., reliability), NFV needs to build resilience into software (or software running on virtual machines). Therefore, it may be difficult to meet the same requirements (or performance) in NFV as traditional devices. In addition, for security threats, it is easier for attackers to replace one virtual network function (VNF) with another than one hardware box with another. Therefore, when deploying VNFs, operators need to make sure that security features of their networks are not be affected. In addition, VNFs may be run in the network infrastructure that are not owned by network operators directly. Therefore, new security threats can also be introduced when the underlying infrastructure and storage are shared, for example, when running network functions in a VM that shares physical resources with other VMs (or network functions).

SDN together with NFV does not overcome all the above concerns, however it simplifies some of concerns such as manageability and reliability. As the SDN controller has a global view of the underlying infrastructure (e.g., topology and network resources), it can simplify the manageability task of NFV. In addition, for reliability, NFV together with SDN can propose new algorithms to recover from failures. For this, our work on failure recovery for SDN is the first step to perform failure recovery in these networks. In addition, for automatic configuration of network functions, our research on bootstrapping and automatic configuration of RouteFlow is useful.

7.1.4 Troubleshooting

Troubleshooting a network is always a complex and difficult task. In traditional networks, engineers and developers have to use tools, such as ping, traceroute, and tcpdump, for debugging a network. In fact, troubleshooting is usually manual and therefore, time consuming in traditional networks. However, due to a high degree of flexibility and programmability offered by SDN, it is currently opening up automatic ways for developing tools to debug, troubleshoot, verify and test networks [4], [5], [6], [7], [8], [9], [10].

Early debugging tools for SDN/OpenFlow, such as ndb [4] and OFRewind [5], make it easier to discover the source of network problems such as faulty device firmware. Like the well-known gdb software debugger [11], ndb provides basic debugging actions such as breakpoint and packet backtracing. A breakpoint shows the history of function calls leading to that breakpoint, while a packet backtrace in ndb lets us define a packet breakpoint (e.g., a dropped packet or a packet filter) and then shows the sequence of forwarding actions seen by that packet leading to the breakpoint. These actions help programmers to debug networks in a similar way to traditional software. In contrast, OF-Rewind works in a different way. It enables record and replay network events, in particular control messages events. The motivation behind OFRewind is that by recording and replaying, many of the bugs can be reproduced (or detected).

Other debugging tools for OpenFlow are: NICE [6], FlowChecker [7], OFTEN [8], VeriFlow [9], and ATPG [10]. NICE [6] verifies controller programs to detect an incorrect behavior by automatically generating a streams of packets under many possible events. FlowChecker [7], OFTEN [8], and VeriFlow [9] are three examples of tools to verify correctness properties of a system. While the former two are based on offline analysis, the latter is capable of online checking of network invariants. Verification parameters include security issues, reachability issues, loops, black holes, etc. The next tool, i.e., ATPG, is proposed for automatic verification of all flows by transmitting test packets in networks for finding action faults in Flow Entries. However, this tool does not verify (or verifies partially) the matching header part of aggregated flows for matching issues. In this dissertation, we proposed a mechanism to verify all the aggregated flows for matching issues (Chapter 3). Like ATPG, the mechanism transmits test packets in a network to verify aggregated flows. There can be two reasons for matching errors in aggregated flows: (1) bugs (software or hardware) in OpenFlow switch implementation and (2) errors in FlowTable configuration. The objective of the verification mechanism is to find this incorrect or no matching and hence, to find the packet-headers that cannot be matched or can be matched incorrectly with the matching-header part of a Flow Entry.

As OpenFlow/SDN accelerates innovations (i.e., many features can be introduced in short timeframes), it may give rise to many other troubleshooting challenges (not discussed above) depending on the features introduced. Therefore, new troubleshooting mechanisms may be researched in future to overcome these challenges. The next challenge could be that where to place troubleshooting functionalities in SDN networks to get the maximum benefits (such as short troubleshooting time and minimum required resources).

7.1.5 Security concerns

Traditional networks have natural protection against security threats. This is because of the closed (proprietary) nature of network devices, their fairly static design, and the decentralized nature of the control plane. A common SDN standard (e.g., OpenFlow) among vendors and clients increases the risk of threats, by the possible introduction of faults in SDN networks. In [13], many threats for SDN networks are researched. These attacks may be present due to vulnerabilities in switches, attacks in the controller to the switch communications, and vulnerabilities in the controller. Some of security threats are also discussed in Chapter 4. In these threats, security problems of establishing an OpenFlow session are discussed.

References

[1] D. Levin, M. Canini, S. Schmid, and A. Feldmann, *Incremental SDN Deployment in Enterprise Networks*, Proceedings of the ACM SIGCOMM 2013 Conference, Vol. 43, No. 4, pp.473 – 474, 2013.

[2] C. Esteve Rothenberg, Marcelo R. Nascimento, Marcos R. Salvador, Carlos Corrła, Sidney Lucena, and Robert Raszuk. *Revisiting Routing Control Platforms with the Eyes and Muscles of Software-Defined Networking*, ACM

SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), pp. 13-18, 2012.

[3] S. H. Yeganeh, A. Tootoonchian, Y. Ganjali, *On scalability of software-defined networking*, in Communications Magazine, IEEE, Vol. 51(2), pp. 136 – 141, 2013.

[4] N. Handigol, B. Heller, V. Jeyakumar, and N. McKeown, *Where is the Debugger for My Software-defined Network?*, ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), pp. 55 – 60, 2012.

[5] A. Wundsam, D. Levin, S. Seetharaman, A. Feldmann, *OFRewind: Enabling Record and Replay Troubleshooting for Networks*, Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, 2011.

[6] M. Canini, D. Venzano, P. Pereni, D. Kosti and J. Rexford, *A NICE way to test openflow applications*, 9th USENIX conference on Networked Systems Design and Implementation, 2012.

[7] E. Al-Shaer and S. Al-Haj, *FlowChecker: configuration analysis and verification of federated openflow infrastructures*, SafeConfig, pp. 37–44, 2010.

[8] M. Kuzniar, M. Canini D. Kostic, *OFTEN Testing OpenFlow Networks*, European Workshop on Software Defined Networks, pp. 54-60, 2012.

[9] A. Khurshid, and W. Zhou, M. Caesar, Matthew, Godfrey, P. Brighten, *VeriFlow: Verifying Network-wide Invariants in Real Time*, HotSDN, pp. 49–54, 2012.

[10] H. Zeng, P. Kazemian, G. Varghese and N. McKeown, *Automatic Test Packet Generation*, CoNEXT, pp. 241–252, 2012.

[11] GNU GDB: [Online]. Available: https://www.gnu.org/software/gdb/.

[12] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, *Network Function Virtualization: Challenges and Opportunities for Innovations*, IEEE Communications Magazine, Vol 53(6), pp. 90-96, 2015.

[13] D. Kreutz, F. Ramos, P. Verissimo, *Towards Secure and Dependable Software-Defined Networks*,ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), pp. 55-60, 2013.

A
Automatic bootstrapping of OpenFlow networks

In this Appendix, the work presented in Chapter 4 about automatic bootstrapping has been presented in detail.

⋆ ⋆ ⋆

Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester

Published in IEEE LANMAN, April 2013.

Abstract OpenFlow decouples the control plane functionality from switches, and embeds it into one or more servers called controllers. One of the challenges of OpenFlow is to deploy a network where control and data traffic are transmitted on the same channel (in-band mode). Implementing such an in-band mode is complex, since switches have to search and establish a path to the controller through the other switches in the network (bootstrapping). In this paper, we propose a method that facilitates this automatic bootstrapping of switches. In this method, the controller establishes its own control network through the neighbor switches that are connected to it by the OpenFlow protocol. We measure suitability of the proposed method by performing bootstrapping experiments in different types of topologies: linear, ring, star and mesh topologies. The experimental

results show that the proposed method allows bootstrapping in a minimal time, which makes it suitable even for a large network.

A.1 Introduction

There is sometimes need to define behavior of networks in a custom manner. Historically, this was possible only by proprietary hardware that was prohibitively expensive or impossible to obtain by researchers and experimenters. The need of this functionality exists in order to run wide-scale projects implementing new experimental protocols. Therefore, in the field of networks, OpenFlow [1] which controls networks freely by software located at one or more servers (so called controllers), has caught attention of many research communities. OpenFlow is developed in a clean-slate future internet program by Stanford University, which aims to offer a programmable network to test new protocols in current Internet platforms. The core idea of OpenFlow is to decouple the control plane functionality from network switches, and to embed it into one or more servers called controllers. This makes switches/routers inexpensive. In addition, this imparts flexibility, as the control plane functionality is moved to the controllers, while only forwarding is required to be done in hardware.

OpenFlow is based on the fact that most modern routers/switches contain a proprietary FIB (Forwarding Information Base) which is implemented in hardware using TCAMs (Ternary Content Addressable Memory). OpenFlow provides the concept of a FlowTable that is an abstraction of the FIB. Additionally, it provides a protocol to program the FIB via adding/deleting/modifying entries in the FlowTable. This is achieved by one or more controllers that communicate with the OpenFlow switches using the OpenFlow protocol (Fig. A.1). The switch/router that exposes its FlowTable through the OpenFlow protocol is called an OpenFlow switch/router.

An entry in the FlowTable consists of: (1) a set of packet fields to match with incoming packets (called as flow), (2) statistics which keep track of matching packets per flow, and (3) actions which define how packets should be processed. When a packet arrives at an OpenFlow switch, it is compared with the Flow Entries in the FlowTable. If a match is found, the actions specified in the matching entry are performed. If no match is found, the packet (a part thereof) is forwarded to the controller. Thereafter, the controller makes a decision on how to handle the packet. It may return the packet to the switch indicating the forwarding port, or it may add a Flow Entry directing the switch on how to forward packets with the same flow.

In OpenFlow, control messages (e.g., messages to add Flow Entries in switches) are required to be exchanged between the controller and switches. These messages can be exchanged either in an in-band or in an out-of-band mode. In the case of an in-band mode, control messages are sent on the same channel used to

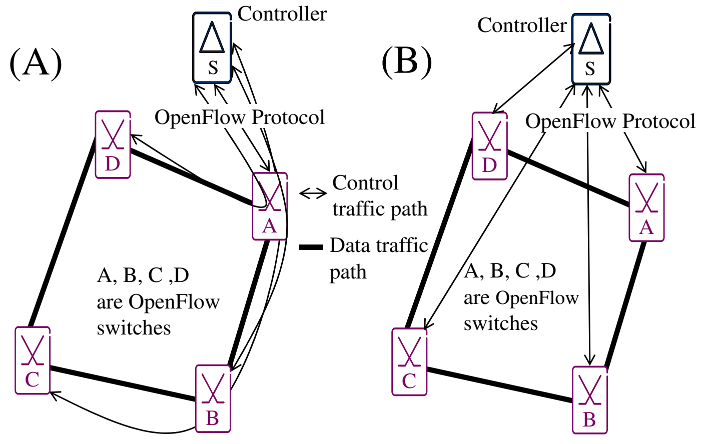


Figure A.1: OpenFlow network: (A) In-Band Mode (B) Out-of-Band Mode

transport data traffic, whereas in the case of an out-of-band mode, control messages are sent on a different channel. As shown in Fig. A.1, in the in-band mode, switches B, C and D share the same channel for control and data traffic, and in the out-of-band mode, switches A, B, C and D use a different channel for control and data traffic. The out-of-band mode is simpler and easier to design because the controller is directly connected (physically) to each of the switches through a separate network. However, the out-of-band mode might not be possible in some scenarios, for example, a widely distributed central offices in access networks. In addition, due to the requirement of a separate network, the out-of-band mode is expensive to build in a real network.

In the in-band mode, switches do not need a separate network for control traffic. OpenFlow defines a virtual port (reserved) in a switch called as local port, which enables remote entities (e.g. controller) to interact with the switch via an OpenFlow network (in-band mode). OpenFlow, however, does not describe how control traffic paths can be established in the OpenFlow network. This task is especially challenging in the case of the in-band mode, since switches need to establish these paths through the other switches in the network. Establishing control paths is important because an OpenFlow session needs to be established through these paths (bootstrapping). In this paper, we propose a method that facilitates this automatic bootstrapping of switches. In this method, the controller establishes its own control network through the switches that are connected to it by the OpenFlow protocol.

The proposed method is emulated using different types of topologies, which vary with different scales (degree, number of nodes, and distance from the controller). The emulation results show that the proposed method allows bootstrapping in a minimal time. It shows that the scalability and simplicity of the method make it suitable even for a large network.

The rest of the paper is organized as follows: section A.2 presents our approach of bootstrapping, section A.3 describes the emulation environment and results, and finally section A.4 concludes.

A.2 Bootstrapping of OpenFlow networks

This section is divided into three parts. The first part gives an overview of our bootstrapping approach. The second part describes OpenFlow mechanisms and the messages that are used for bootstrapping. The third part gives the bootstrapping approach in detail.

A.2.1 Overview of our bootstrapping approach

Bootstrapping of a switch in an OpenFlow network requires at least two steps:

1. Assignment of connection identifiers for connecting the switch to the controller. The connection identifiers required are at least the IP address of the local port and the IP address of the controller. The other identifiers can be MAC and transport layer parameters. Transport layer parameters may include a transport layer protocol and a port number.

2. Instantiation of an OpenFlow session with the controller.

The first step can be accomplished by using protocols such as DHCP (Dynamic Host Configuration Protocol), OF-CONFIG (OpenFlow management and configuration protocol) [2] or ARP (Address Resolution Protocol). ARP can allow switches to know the MAC address of the controller. DHCP or OF-CONFIG can assign a unique IP address to a switch (i.e., local port), and can allow it to know the other identifiers (IP address of the controller and transport layer parameters) to connect with the controller.

We used ARP and DHCP to accomplish the first step. In the case of DHCP, we assumed that either the DHCP server is located in the controller node or it is the neighbor of the switch that is directly connected (physically) to the controller through a separate network.

For bootstrapping, each switch runs a DHCP client and keep on flooding DHCP messages to its neighbors until it receives a reply from the DHCP server. If a neighbor is the DHCP server, it replies to the switch. In the case the neighbor is a switch connected to the controller by the OpenFlow protocol, the controller allows the switch to forward DHCP messages to the DHCP server. In the case the neighbor switch is not connected to the controller, the messages are dropped.

Once a switch has an IP address and it knows the other identifiers by the DHCP protocol, the switch learns the MAC address of the controller by the ARP protocol. If the transport layer protocol between the switch and the controller is TCP (Transmission Control Protocol), the switch then establishes a TCP connection with the controller. The switch is able to establish the connection in at least one of the following cases: (1) the controller is directly connected (physically) to the switch, (2) a neighbor switch has an OpenFlow session with the controller.

When a switch has a transport layer connection with the controller, the switch instantiates an OpenFlow session (the second step). The OpenFlow session can be established along the same path used to establish the transport layer connection. Bootstrapping of an OpenFlow network completes when each switch in the network has an OpenFlow session.

A.2.2 OpenFlow mechanisms and messages in bootstrapping

We used the local and normal stack of OpenFlow to implement the bootstrapping approach mentioned in the previous subsection. By the local stack, we refer to the local networking stack of an OpenFlow switch, which can be used to communicate with remote entities (DHCP server or controller). In order to communicate with remote entries, OpenFlow defines a local port. The local port allows the local networking stack to send or receive packets to or from remote entities. In addition, we run a DHCP client, a TCP/IP stack, and an OpenFlow stack in the local stack to perform bootstrapping.

In the case of the normal stack, an OpenFlow switch can also forward packets using Ethernet switching technologies such as MAC learning. In the case of MAC learning, MAC addresses are learned through the source MAC address and the incoming port of a packet. In the case the destination address is an unknown address or the broadcast address, the packet is flooded. In the case the destination address is already learned, the packet is forwarded through the learned port. This mechanism is used in our bootstrapping approach when a switch is not connected with the controller and the switch has to forward its own control traffic (e.g., DHCP messages or messages to instantiate an OpenFlow session) without contacting the controller.

In order to perform bootstrapping, we also used some of the messages of the OpenFlow protocol. These messages are Hello, Feature-Request, Feature-Reply, Packet-In, Packet-Out and Flow-Mod messages. With a Hello message, a switch and the controller match a version of the supported OpenFlow protocol. In the case the version matches, the controller requests the features of the switch by sending a Feature-Request message. Upon receipt of the Feature-Request message, the switch replies the controller by sending a Feature-Reply message. These messages (Hello, Feature-Request and Feature-Reply) are used to instantiate an OpenFlow session. The other messages such as Packet-In, Packet-Out and Flow-Mod messages are used to control packet forwarding in switches. In the case a switch needs to transmit an unknown packet, the packet is first sent to the controller in a Packet-In message. In the case the controller needs to send a packet through a port of a switch, the packet is sent to the switch in a Packet-Out message. In the case the controller needs to add a Flow Entry in a switch, the controller sends a Flow-Mod to the switch.

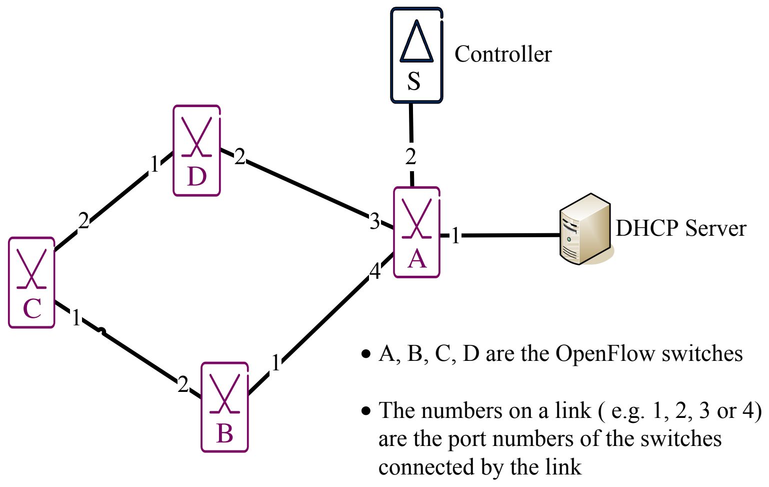

Controller
S
D
A
C
B
DHCP Server
1
2
3
4
• A, B, C, D are the OpenFlow switches
• The numbers on a link (e.g. 1, 2, 3 or 4) are the port numbers of the switches connected by the link

Figure A.2: A topology to describe bootstrapping

A.2.3 Detailed bootstrapping

In this section, we describe bootstrapping in detail by taking an example of an OpenFlow network shown in Fig. A.2. In Fig. A.2, the DHCP server and the controller are directly connected (physically) with switch A. For bootstrapping, DHCP is enabled with Option 43 [3]. This option allows a programmer to program vendor-specific information in the DHCP server. In our case, vendor-specific information is the controller IP address and transport layer parameters. The DHCP clients in our bootstrapping approach request this information by sending a "vendor class identifier" in a DHCP Discover message.

The controller in our approach maintains a topology database which contains IDs (switches IDs, the controller ID and the DHCP server ID) and links information. We assigned the ID of a switch equal to the MAC address of the switch local port. At the initial stage when no switch is connected with the controller, the topology database has only the controller and the DHCP server as IDs. During bootstrapping, the controller gathers a topology of neighbor switches by transmitting a special kind of probe messages from (or to) the recently connected switch. The format of these probe messages is inline with the Link Layer Discovery Protocol (LLDP) [4]. Topology gathering is important in our approach because during bootstrapping the controller needs to find a path to any switch or to the DHCP server.

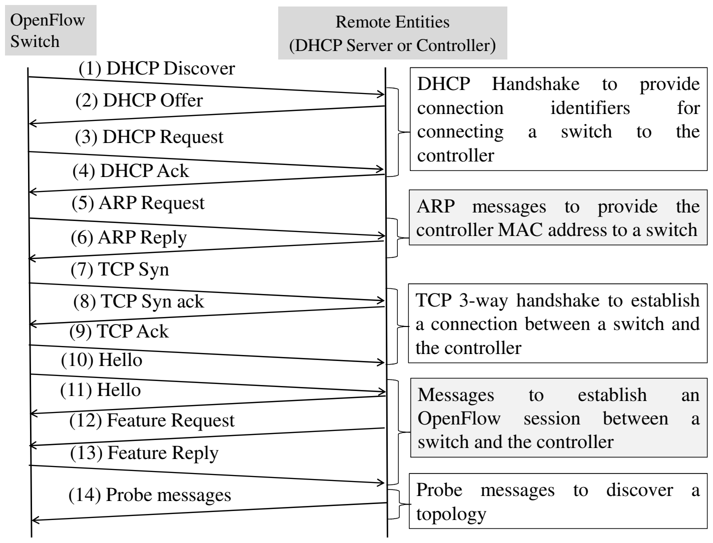

OpenFlow Switch
Remote Entities (DHCP Server or Controller)
(1) DHCP Discover
(2) DHCP Offer
(3) DHCP Request
(4) DHCP Ack
(5) ARP Request
(6) ARP Reply
(7) TCP Syn
(8) TCP Syn ack
(9) TCP Ack
(10) Hello
(11) Hello
(12) Feature Request
(13) Feature Reply
(14) Probe messages
DHCP Handshake to provide connection identifiers for connecting a switch to the controller
ARP messages to provide the controller MAC address to a switch
TCP 3-way handshake to establish a connection between a switch and the controller
Messages to establish an OpenFlow session between a switch and the controller
Probe messages to discover a topology

Figure A.3: Message exchange between a switch and the controller

In bootstrapping, each switch and remote entities exchange a sequence of messages. This sequence is shown in Fig. A.3. In order to exchange these messages, a switch uses the local and the normal stack of OpenFlow.

We now explain bootstrapping of switches $A, B, C,$ and D (shown in Fig. A.2). The first message exchanged by each of the switches is the DHCP Discover message (shown in Fig. A.3). At the stage, when the local port of a switch does not have an IP address, a DHCP client in the local networking stack transmits a DHCP Discover message from its local port.

The DHCP Discover message transmitted from the local port (switch own control traffic) is handled by the normal stack of the same switch. As the destination MAC address of a DHCP Discover message is the broadcast address, the normal stack of each switch floods the DHCP Discover message through all its outgoing ports (e.g., $1, 2, 3,$ or 4 in Fig. A.2).

A.2.3.1 Bootstrapping of switch A

At the stage when no switch is connected with the controller, all DHCP Discover messages are dropped except the one that is transmitted through port 1 of switch A (see Fig. A.2). The messages are dropped in our bootstrapping approach because neighbor switches or the controller that have received these messages have no way to forward unknown traffic. The DHCP Discover message of switch A, which is not dropped, reaches to the DHCP server. Upon receipt of the DHCP Discover message from the DHCP client of switch A, the DHCP server returns a DHCP Offer message (message (2) in Fig. A.3). The DHCP Offer message contains an unleased IP address and the string containing vendor-specific information. Switch A receives this message through port 1. As the destination MAC address of the DHCP Offer message is the MAC address of the local port (switch A's own control traffic), the message is handled by the normal stack of switch A. Thereafter, the normal stack forwards the message to the local port. The message reaches to the DHCP client of switch A via the local port.

The DHCP client of switch A now stores vendor-specific information from the DHCP Offer message, and responds by transmitting a DHCP Request message (message (3) in Fig. A.3) through the local port. The DHCP Request from the local port is handled by the normal stack of switch A. The normal stack then floods the DHCP request message through all its ports because the destination MAC address is the broadcast address. The DHCP Request, which is transmitted through port 1 of switch A, reaches to the DHCP server. The DHCP server acknowledges the DHCP Request by sending a DHCP Ack message (message (4) in Fig. A.3). Upon receipt of the DHCP Ack, the normal stack of switch A transmits the Ack to its local port, and thereby, the DHCP client receives this message. Thereafter, the DHCP client assigns the IP address to the local port. The local networking stack of switch A now parses the vendor-specific information (stored at the time of the DHCP offer message), and it knows the controller IP address and transport layer parameters to connect with the controller.

The local networking stack of switch A now transmits an ARP request message (message (5) in Fig. A.3) through the local port to know the controller MAC address. The ARP request is now flooded by the normal stack of switch A. The controller receives this request through port 2 of switch A. Upon receipt of the ARP request, the controller returns the MAC address in an ARP reply message (message (6) in Fig. A.3). Switch A receives this reply through port 2. The normal stack of switch A then forwards the reply to its local networking stack by sending it to the local port. In addition, MAC learning mechanism in switch A learns the output port (i.e., port 2) to reach the controller.

Assuming TCP as a transport layer protocol in the vendor specific information, the local networking stack of switch A starts a TCP connection upon receipt of the ARP reply. In this TCP connection, the local networking stack sends a TCP syn message (message (7) in Fig. A.3) from the local port of switch A. The normal stack of switch A then sends it through port 2 (learned port for the controller). Upon receipt of the TCP Syn, the controller sends a TCP Syn Ack (message (8) in

Fig. A.3) to switch A. When switch A receives the Syn Ack, the normal stack of switch A forwards it to the local port. The local networking stack acknowledges then the Syn-Ack by sending a TCP Ack (message (9) in Fig. A.3). At this stage, switch A has a TCP connection with the controller.

After establishing the TCP connection, the OpenFlow stack in the local stack of switch A instantiates an OpenFlow session by sending a Hello message (message (10) in Fig. A.3) to the controller through the local port. The normal stack then sends the Hello message via port 2. Upon receipt of the Hello message, the controller replies back with the Hello message (message (11) in Fig. A.3). The controller then sends a Feature request message (message (12) in Fig. A.3). Upon receipt of the Feature Request message, the OpenFlow stack of switch A sends a Feature Reply message (message (13) in Fig. A.3) through the local port. The normal stack sends this message to the controller via port 2. The controller receives the Feature Reply message, and declares an OpenFlow session with switch A. In the Feature reply message, switch A has sent all its attributes/parameters including the MAC address of its local port. Henceforth, the controller adds the MAC address of the local port as the ID of switch A in its topology database.

At this time the controller does not know how switch A (this is the only switch present in the topology database) is connected with the controller. To know this, the controller sends a probe message to switch A. Upon receipt of the message, switch A treats this as unknown traffic, and sends this back to the controller as a Packet-In message. The Packet-In message includes the ID of switch A and the incoming port of the probe message (i.e., port 2) in its message. Upon receipt of the Packet-In message, the controller now finds that the Packet-In message is generated by switch A and there is no path in its topology database to reach from switch A to the controller. Therefore, the controller adds a link in its topology database such that switch A is connected to the controller through port 2. In order to take control over control traffic of switch A, the controller at this time may add two Flow Entries in switch A. The first entry can be for the flows containing the destination MAC address as the MAC address of the local port. The second entry can be for the flows containing the incoming port as the local port and the destination address as the controller address.

A.2.3.2 Bootstrapping of switches B, C and D

When switch A established a session with the controller, switch B, switch C and switch D are still in the initial phase of transmitting DHCP Discover messages. As switch B and switch D in Fig. A.2 are directly connected (physically) with switch A, DHCP Discover messages from switch B and switch D reach at switch A. switch A has now an OpenFlow session with the controller. Therefore, upon receipt of the DHCP Discover messages, switch A sends these messages to the controller in Packet-In messages. Let us take the case when the DHCP Discover message from switch B reaches to switch A. The Packet-In message in this case includes the ID of switch A and the incoming port of the DHCP message, i.e., port 4 in its message. Upon receipt of the Packet-In message, the controller finds that the message in the Packet-In is the DHCP message (because the message has the destination transport layer port equals to 67) and the source of the DHCP message (i.e the local port of switch B) is not present in its topology database. Therefore, the controller adds the ID of switch B in its topology database. In addition to the ID, the controller adds a link in its topology database such that switch B is connected to switch A through the incoming port of the DHCP message (i.e., port 4).

The controller does not know at this time the location of the DHCP server, therefore, it sends a Packet-Out message to switch A to flood the DHCP Discover message from all ports of switch A except the incoming port of the DHCP Discover message (port 4). Upon receipt of the DHCP Discover message from port 1 of switch A, the DHCP server sends the DHCP offer message to switch B. The DHCP Offer message is now received by switch A through port 1. However, switch A does not know how to handle this message. Therefore, it sends the message to the controller in the Packet-In message. This Packet-In message includes the incoming port of the DHCP Offer message (port 1) and ID of switch A in its message. Upon receipt of the Packet-In message, the controller calculates a path from switch A to the destination of the DHCP offer message (i.e., switch B). As the controller knows the path to switch B through port 4 of switch A, the controller sends a Packet-Out message to switch A to forward the DHCP Offer message via port 4. In addition, the controller finds that this DHCP message is from the DHCP server (because the message contains the transport port of the source as 67). Therefore, the controller adds a link in its database such that switch A is connected to the DHCP server through the incoming port of the DHCP Offer message (i.e., port 1).

Upon receipt of the DHCP offer message, the normal stack of switch B handles this message, and forwards this to its local port. Switch B now exchanges the other messages (the message (1) to (13) in Fig. A.3) to instantiate an OpenFlow session. In this cases, all messages of switch B go through switch A.

Note that the controller at this time (the time when the switch B has exchanged all messages (1) to (13)) does not have complete information about the links of switch B. In our case, the controller does not know which port of switch B connected to port 4 of switch A. Therefore, the controller transmits probe messages through each port of switch B after having the OpenFlow session with it. A probe message contains the ID and the outgoing port of switch B from where the probe message has to be transmitted. The probe message of switch B, which is sent from port 1 of switch B, reaches to switch A through port 4. As the probe message is unknown traffic for switch A, it is sent to the controller in a Packet-In message. Upon receipt of the Packet-In message, the controller now parses the probe message, and finds that the message is sent from port 1 of switch B. As the incoming port of the Packet-In is port 4 of switch A. The controller updates the link of switch A in its topology database such that port 4 of switch A is connected to switch B by port 1. After this, the controller adds Flow Entries in switch A and switch B for the control traffic of switch B.

Like switch B, switch D in our bootstrapping approach also establishes an OpenFlow session with the controller. Like the same way, the controller transmits probe messages from switch D after having the session with it, knows the link connecting switch D to switch A, and adds Flow Entries in switch A and switch D for the control traffic of switch D.

At the time switches A, B and D have OpenFlow sessions with the controller, switch C may be in the initial stage of transmitting the DHCP Discover messages. In the case of switch C, the DHCP Discover messages are received by switch B and switch D. switch B and switch D have now OpenFlow sessions with the controller. Therefore, the DHCP messages from switch B and switch D will be sent to the controller in Packet-In messages. Let the Packet-In message from switch B first reaches the controller. Upon receipt of the Packet-In message from switch B, the controller adds switch C in its database and adds a link in its database such that switch B is connected to switch C through port 2. In the case the controller does not know the port of switch B along the calculated path to the DHCP server (because the controller may have not transmitted/received a probe message giving information about the link between switch B and switch A), the controller replies switch B to drop the message. In the case the controller knows the port of switch B along the calculated path to the DHCP server, it replies to B to forward the message along the path.

In the case of the Packet-In message from switch D, the controller adds a link in its topology database such that switch D is connected to switch C through port 1. Like the DHCP Discover message from switch B, the controller replies switch D to forward the message to the DHCP server along the available path. Thereafter, two DHCP Discover messages from switch C may reach to the DHCP server (the one from the path C-D-A and the other from the path C-B-A). Upon receipt of these messages, the DHCP server replies to only one DHCP Discover message by sending a DHCP Offer message. Therefore, at the end one DHCP Offer message reaches to switch C. switch C now exchanges all other messages (shown in Fig. A.3) with the DHCP server and the controller. The messages exchanged are similar to the sequence of messages exchanged at the time of bootstrapping of switches A, B and D. After exchange of the messages, switch C will have an OpenFlow session with the controller.

The controller now gathers information about all links of switch C by transmitting probe messages from all ports of switch C. After this, for the control traffic of switch C, the controller adds the Flow Entries along a calculated path (shortest) from switch C to the controller.

A.3 Emulation environment and results

In this section, we describe the testbed, topologies, methodology and results of the bootstrapping experiments.

We performed emulation on our virtual-wall testbed which is a generic test environment for advanced network, distributive software and service evaluation. We created linear, ring, star and mesh topologies in our testbed nodes to perform boostrapping in OpenFlow networks. The topologies were created by using Linux processes in different network namespaces. In all topologies, we connected the controller and the DHCP server to one of the switches present in the topology. In the case of a star topology, we connected the controller and the DHCP server to the central switch connecting all the other switches in the star network. The number of switches connecting the central switch is varied and the effect on the bootstrapping time is shown in the results. In the case of mesh topologies, we used topologies that were developed within the COST 266 action project [6]. In this project, a basic reference topology (BT topology) and variations of the BT topology, suited for a pan-European network, were designed. The variations of the BT topology were Core Topology (CT), Large Topology (LT), Ring Topology (RT) and Triangular Topology (TT). These were obtained by varying the total number of nodes and the degree of meshedness. The CT topology and the LT topology differ with respect to the number of nodes. The BT consists of 28 nodes, the CT consists of 16 nodes and the LT consists of 37 nodes. The other derived topologies contained the same number of nodes as the BT, but the difference lies in the degree of meshedness. The maximum degree of nodes in these topologies is 7. We performed the bootstrapping experiments on all these topologies.

There are many extensions of the OpenFlow protocol. Some of the extensions have been released publicly in the form of versions. The OpenFlow 1.0 version that is developed by Stanford is called as the reference switch [7]. This reference switch contains the DHCP client software in its implementation. However, this software is abandoned in the higher versions. We integrated this DHCP client software in the OpenFlow 1.1 version (developed by Ericsson), and used this for our implementation. In addition, many OpenFlow controllers are also available for controlling OpenFlow networks. These are NOX, Beacon, Onix, Floodlight, Helios and Maestro. We implemented our bootstrapping approach in the NOX controller (developed by Ericsson [8]) and used this in our emulation.

In our emulation, the DHCP server is enabled with two options: ping checked enabled and ping check disabled. The ping check may be required to verify address availability before offering it to a client. We tested our bootstrapping approach with both the options and calculated the bootstrapping time. In the case of ping check enabled, the DHCP server pings an IP address before offering it to the DHCP client. In the case the DHCP server does not receive a reply of a ping until a certain time (1 second in our case), it offers the IP address to the client. In the case of ping check disabled, the DHCP server offers an IP address to the DHCP client without pinging the IP address. For the transmission of the DHCP Discover message, we kept the retransmission time of the DHCP Discover messages (the time if a DHCP client does not receive a reply of the DHCP Discovery message) equals to 1 second. The DHCP client in an OpenFlow switch changes this value to a random interval between 0.90 to 1.10 second.

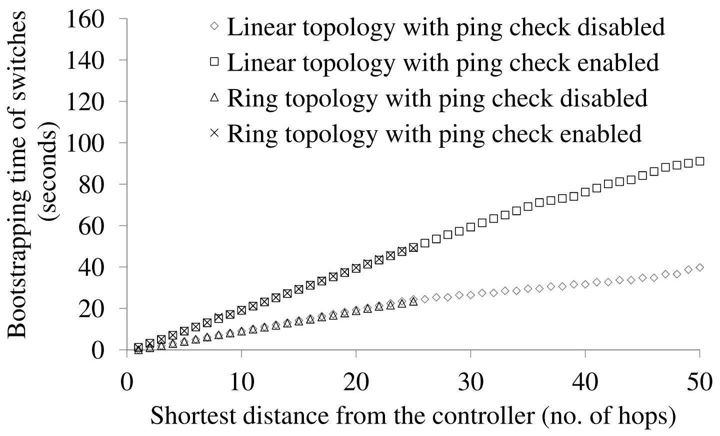


Figure A.4: Bootstrapping experiments on linear and ring topologies

We now show the results of the experiments performed on different topologies. Fig. A.4 shows the results of the experiments performed on linear and ring topologies. The results show a linear relationship between the bootstrapping time and the shortest distance (number of hops) from the controller. In the case of ping check enabled, the bootstrapping time of switches is delayed by an additional time. This is because the DHCP server waited 1 second before offering an IP address to each of the switches. In the case of the linear topology of 50 nodes, bootstrapping of all the switches took approximately 40 seconds with the ping check disabled option and 91 seconds with the ping check enabled option. In the case of the ring topology of 50 nodes, bootstrapping took approximately 23 seconds with the ping check disabled option and 50 seconds with the ping check enabled option.

Fig. A.5 shows the results of the experiments performed on the star topologies. The results show that until the degree of the central switch is 30, bootstrapping took approximately 1 second in the case of the ping check disabled option and 2 seconds in the case of the ping check enabled option. After this, the bootstrapping time increases with the increased degree of the central switch. This is because as

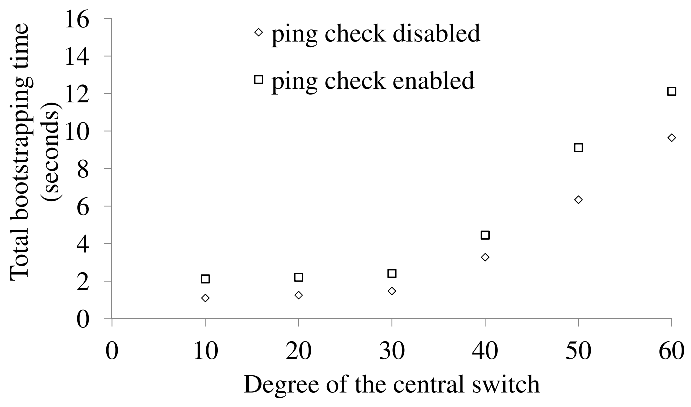


Figure A.5: Bootstrapping experiments on star topologies

the degree of the central switch increases, more messages will be buffered in the packet-in buffer of the central switch. A message remains in the buffer until the controller responds on a forwarding decision of the message. This led to overflow of the packet-in buffer, and resulted into drop of some of the messages. In the case a DHCP Discover message drops, bootstrapping in our emulation will take additional 1 second to retransmit the next DHCP Discover message. In the case a TCP syn message drops, the TCP stack will take an additional time to retransmit the TCP syn message. This additional time increases exponentially in TCP with the number of Syn messages dropped [9].

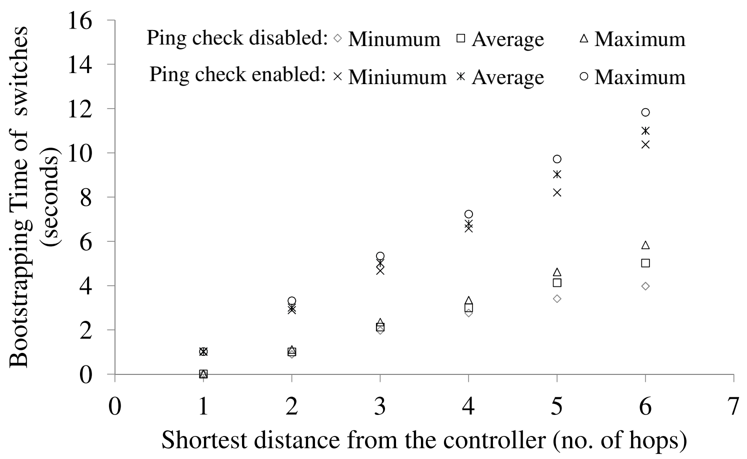


Figure A.6: Bootstrapping experiments on mesh topologies

Fig. A.6 shows the results of the experiment performed on the mesh topologies developed in the COST 266 action project. The figure shows the minimum, the average, and the maximum time of bootstrapping. In emulated mesh topologies, we found a linear relationship between the bootstrapping time and the minimum distance from the controller. With the ping disable option, our method took 6 seconds (approximately), and with the ping enable option, our method took 12 seconds (approximately) to bootstrap the emulated mesh networks.

A.4 Conclusions

In this paper, we have proposed a method that facilitates automatic bootstrapping in an in-band case of OpenFlow networks. We have performed extensive experiments on different types of topologies, and have shown that the proposed method allows automatic bootstrapping in a minimal time. In our emulation, bootstrapping took a maximum of 12 seconds to discover the OpenFlow network created by well known pan-European topologies developed in the COST 266 action project.

In this work, bootstrapping of OpenFlow networks is performed by using existing auto-configuration mechanisms such as DHCP. However, with the recent addition of OF-config to the OpenFlow architecture, there is an additional interface available, dedicated for configuration tasks. OF-config is based on NETCONF (network configuration protocol) [10], a transactional protocol that uses remote procedure calls on top of a secure transport channel to manage configurations on remote devices. Hence, in future work we will use OF-config, and will compare it with DHCP for auto-configuration of OpenFlow switches.

There are two topics that can enhance the work performed in this paper: (1) consideration of multi-controller networks, and (2) failure recovery in the in-band case of OpenFlow networks. In [11] [12], we performed a failure recovery experiment for the in-band case of an OpenFlow network, and achieved failure recovery within a reasonable amount time.

Acknowledgment

The research leading to these results has received funding from the EU FP7 programme under grant agreement n^o 258457 (SPARC) and n^o 258365 (OFELIA).

References

[1] N. McKeown et al., *Openflow: Enabling innovation in campus networks*, ACM Computer Communication Review, 2008.

[2] OF-Config [Online]. Available: https://www.opennetworking.org/standards/of-config.

[3] S. Alexander et al., *DHCP Options and BOOTP Vendor Extensions*, RFC 2132, 1997.

[4] IEEE standard 802.1AB [Online]. Available: http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf.

[5] OpenFlow Switch Specification: Version 1.0.0 (Wire Protocol 0x01)[Online]. Available: www.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[6] S. D. Maesschalck et al., *Pan-European Optical Transport Networks: An Availability-based Comparison, Photonic Network Communications*, Vol. 5, Issue 3, pp. 203-225, 2003.

[7] OpenFlow reference switch implementation [Online]. Available: http://www.openflow.org/.

[8] Ericsson OpenFlow and NOX Controller Software [Online]. Available: https://github.com/TrafficLab.

[9] V. Paxson et al., *Computing TCP's Retransmission Timer*, RFC 2988, 2000.

[10] R. Enns et al., *Network Configuration Protocol*, RFC 6241, 2011.

[11] S. Sharma et al., *Fast failure recovery for in-band OpenFlow networks*, DRCN, 2013.

[12] S. Sharma et al., *A demonstration of automatic bootstrapping of resilient OpenFlow networks*, IFIP/IEEE Integrated Network Management Symposium (IM), 2013.

B
Automatic configuration of routing control platforms in OpenFlow networks

In this appendix, automatic configuration of routing control platforms used in Chapter 5 is presented.

⋆⋆⋆

Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester

Published in ACM SIGCOMM Computer Communication Review, August 2013.

Abstract RouteFlow provides a way to run routing control platforms (e.g. Quagga) in OpenFlow networks. One of the issues of RouteFlow is that an administrator needs to devote a lot of time (typically 7 hours for 28 switches) in manual configurations. We propose and demonstrate a framework that can automatically configure RouteFlow. For this demonstration, we use an emulated pan-European topology of 28 switches. In the demonstration, we stream a video clip from a server to a remote client, and show that the video clip reaches at the remote client within 4 minutes (including the configuration time). In addition,

we show automatic configuration of RouteFlow using a GUI (Graphical User Interface).

B.1 Introduction

OpenFlow decouples control plane functionality from forwarding functionality of switches, and embeds it into one or more servers called controllers. In OpenFlow networks, RouteFlow [1] provides a way to run routing control platforms (e.g. Quagga). It executes switches' (OF-A, OF-B, OF-C and OF-D in Fig. B.1) control logic through virtual machines (VM-A, VM-B, VM-C and VM-D in Fig. B.1) which mirror a physical topology. Each virtual machine (VM) runs a routing control platform (e.g., Quagga) and is dynamically interconnected with other VMs.

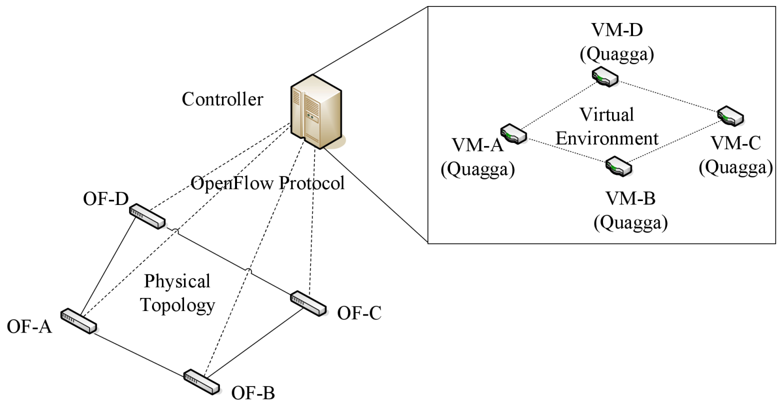


Figure B.1: RouteFlow Design

Currently, configurations of RouteFlow are not automatic. Before running RouteFlow, an administrator needs to devote a lot of time in configurations: (1) creating VMs, (2) creating mapping between a VM and an OpenFlow switch, (3) creating mapping between VM interfaces and switch interfaces, and (4) writing routing configuration files (e.g., ospf.conf, zebra.conf) for each VM. For a large topology (typically for 1000 switches), it may take many days to configure RouteFlow.

We propose a framework to automatically configure RouteFlow. In our framework, we use an additional controller which runs a topology discovery module [3] to know network configurations. The network configurations are then sent to RouteFlow using configuration messages. Using these messages, RouteFlow configures itself.

For this demonstration, we use an emulated pan-European topology of 28 switches. In the demonstration, we stream a video clip from a server to a remote client. This video clip reaches at the remote client within 4 minutes (including the configuration time). This is quite optimal compared to the time consumed in manual configurations. In addition, we show automatic configuration of RouteFlow by showing configurations of VMs in a GUI.

B.2 Automatic configuration of RouteFlow

In this section, we introduce our framework and present the results of the experiments performed on the OFELIA testbed [2].

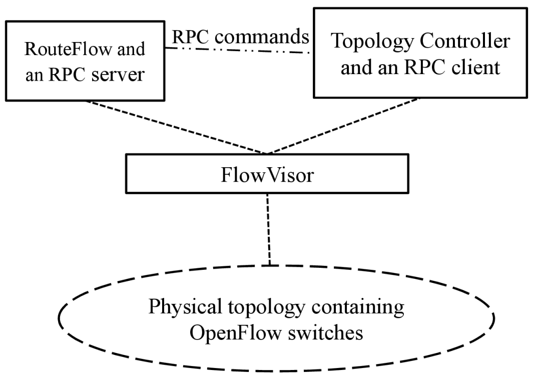

RouteFlow and an RPC server
RPC commands
Topology Controller and an RPC client
FlowVisor
Physical topology containing OpenFlow switches

Figure B.2: Framework for automatic configuration of Routeflow

Fig. B.2 shows five different components of the proposed framework: (1) RF-controller, which runs RouteFlow without any manual configuration of VMs, (2) Topology controller, which gathers topology information (switches and links information) by sending probe messages in the physical topology [3], and contains a very small part of configurations from the administrator (e.g., a range of IP addresses for the virtual environment), (3) RPC (remote procedural call) client, which collects configuration information from the topology controller, and sends it to a server called RPC server, (4) RPC server, which resides in the RF-controller and configures RouteFlow on reception of configuration messages from the RPC client, (5) FlowVisor, which acts as a proxy server between a switch and controllers (the topology controller and the RF-controller in our framework).

In our framework, we used different controllers (e.g., topology controller) for gathering topology information and running RouteFlow. This is done to share the load of automatic configuration of RouteFlow.

At the initial stage, the RF-controller does not have any configurations i.e. there are no virtual machine to run Quagga. On detection of a new switch, the topology controller sends a configuration message to the RPC client, which then forwards it to the RPC server. This configuration message contains the ID of the switch and the number of switch ports. Upon receiving of the message, the RPC server creates a VM with an ID identical to the switch ID and the number of ports equivalent to the switch ports.

On detection of a new link, the topology controller computes unique IP addresses (from the range of IP addresses) for the corresponding VM interfaces, and sends this information to the RPC server through the RPC client. The RPC server then configures IP addresses of the VM interfaces.

Additionally, the RPC server writes routing configuration files (e.g. ospf.conf, zebra.conf, bgp.conf) using the information present in the configuration message sent by the RPC client.

B.3 Results of automatic configuration experiments

We perform experiments to automatically configure RouteFlow that uses OSPF (Open Shortest Path First) as a routing protocol. The experiments are performed on ring topologies with different number of switches. These topologies are generated on a node of the OFELIA testbed by using Linux processes in different network namespaces. In each Linux process, we run Open vSwitch 1.4.1 implementation [4]. Separate nodes of the testbed are used to run FlowVisor, the topology

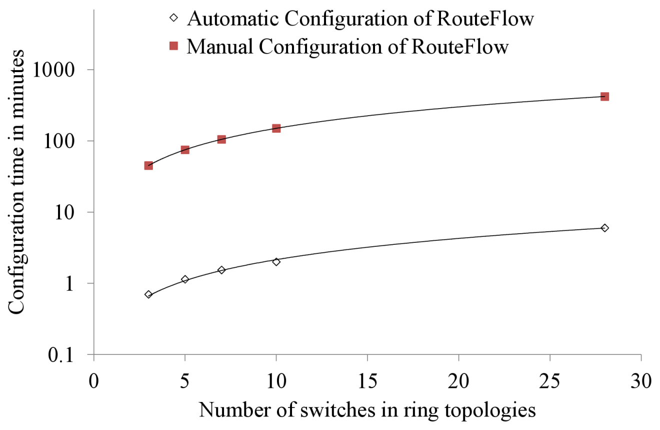


Figure B.3: Configuration Time

Fig. B.3 shows the time of automatic and manual configurations of RouteFlow.

We calculate the time in manual configurations based on personal experience. In manual configurations, we assume that the administrator takes 5 minutes in creating a VM (writing VM configurations, installing Linux distributions and routing packages like Quagga), 2 minutes in creating mapping between switch interfaces and VM interfaces, and 8 minutes in writing routing configurations for a VM. The figure shows that there is a large difference between automatic and manual configurations of RouteFlow.

B.4 Demonstration setup

We demonstrate the proposed framework to automatically configure RouteFlow in OpenFlow networks. For the demonstration, we connect two laptops using an Ethernet cable. The first laptop contains the RF-controller, the RPC server, the RPC client, the topology controller and the FlowVisor. The second laptop contains an emulated OpenFlow network topology, which is a pan European topology [5] consisting of 28 nodes. The clients and servers are connected with the nodes of this topology.

In the demonstration, we show automatic configuration of RouteFlow by showing switches with red and green colors in a GUI. The color of a switch remains red until it is configured by the RPC server. Otherwise, it changes to green. Note that a switch is considered as configured when it has a corresponding VM.

At the start of the experiment, we stream a video clip from a server to a remote client. At this point, there is no virtual machine present in the RF-controller. However, thanks to the proposed framework, the VMs are created and the routing protocol is enabled within a very short time, and the video clip reaches (after around 4 minutes) at the remote client.

Acknowledgment

The research leading to these results has received funding from the EU FP7 programme under grant agreement n^o 317576 (CityFlow) and n^o 258365 (OFELIA).

References

[1] C. E. Rothenberg et al., *Revisiting Routing Control Platforms with the Eyes and Muscles of Software Defined Networking*, HotSDN, pp. 13–18, 2012.

[2] OFELIA Testbed [Online]. Available: http://www.fp7-ofelia.eu/.

[3] Topology Discovery module [Online]. Available: https://github.com/noxrepo/nox-classic/wiki/Discovery.

[4] Open vSwitch [Online]. Available: http://openvswitch.org/.

[5] S. D. Maesschalck et al., *Pan-European optical transport networks: an availability-based comparison*, Photonic Network Communications, Vol. 5(3), pp. 203-225, 2003.

C
Demonstrating resilient quality of service in Software Defined Networking

In this Appendix, a resilient quality of service framework proposed for OpenFlow is presented. This framework is explained briefly in Chapter 5.

⋆ ⋆ ⋆

Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester

Published in IEEE INFOCOM, August 2014.

Abstract Software defined Networking (SDN) such as Open-Flow decouples the control plane from forwarding devices and embeds it into one or more external entities called controllers. We implemented a framework in OpenFlow through which business customers receive higher Quality of Service (QoS) than best-effort customers in all conditions (e.g. failure conditions). In the demonstration, we stream video clips (business and best-effort customer's traffic) through an emulated OpenFlow topology. During the demonstration, we trigger a failure in the paths of video clips and show an effectively higher QoS for business customers when compared against best-effort customers. This is demonstrated by simply watching the video clips at the receiver.

C.1 Introduction

Nowadays, providing users with a guaranteed Quality of Service (QoS), meeting the service level agreements, is of paramount importance. However, implementing such a QoS system is challenging in the current Internet. This is because in the current Internet, each forwarding device runs its own control plane software to make the forwarding decisions, lacking a broader picture of network resources, and there is no standard protocol available to configure QoS parameters in the forwarding devices. In this environment, a QoS provider (e.g., bandwidth broker in a single autonomous system) uses vendor-specific protocols to configure QoS parameters. However, not all forwarding devices support all of these protocols.

The Software Defined Networking (SDN) approach, such as OpenFlow [1], is one of the emerging Future Internet technologies in which control plane software is removed from all forwarding devices (switches or routers) of a network and is embedded into one or more external entities called controllers. In OpenFlow, the available network resources can be known by simply requesting the controller and in addition, there are standard protocols (OpenFlow configuration protocols [1, 2]) available to configure QoS parameters.

We implemented a QoS framework in OpenFlow, which divides different types of traffic (business and best-effort traffic) into different flows (services), configures priority queues, and redirects different flows to a suitable priority queue. Upon a failure, our framework reconfigures the network and provides high QoS to the business customers. In future versions of OpenFlow, namely since version 1.3, flow-related meters can also be used in this framework.

In the demonstration, we stream video clips (business and best-effort traffic) in an emulated OpenFlow pan-European topology, and show that business customers achieve high quality of service than best-effort customers using our framework. In addition, during the demonstration, we trigger a failure in the paths of video clips and show an effectively higher QoS for business customers as compared to best-effort customers.

C.2 Resilient QoS framework for OpenFlow

In our framework, we use the OpenFlow protocol together with the OVSDB (Open vSwitch Database Management Protocol) configuration protocol [2] to provide high QoS. The OpenFlow protocol is used to divide different types of traffic into different flows and to redirect these flows through a suitable priority queue. The configuration protocol is used to configure suitable priority queues in the OpenFlow routers (or switches). Both of these protocols are used between the controller and the OpenFlow switches. As the current controllers such as Floodlight do not support the OVSDB protocol, the Floodlight controller is extended to support this feature. In addition, for communication with a QoS provider, we use the Northbound API (Application Program Interface) of the controller and for routing, we use a standard routing protocol (OSPF, Open Shortest Path First). Furthermore, for running OSPF in OpenFlow, we rely on our previously presented framework [3] for RouteFlow [4].

Starting on an OpenFlow router, three queues are configured on each port of the OpenFlow router. The first queue has the highest priority and therefore, traffic from this queue is forwarded first, then from the second queue, and so on. The first queue is configured to traverse control traffic, the second queue is configured to traverse business traffic, and the third queue is configured to traverse best-effort traffic. The traffic is called business traffic, if the TOS (Type of Service) field of the traffic is enabled. The traffic is called best-effort traffic, if the TOS field is not enabled. The edge OpenFlow router enables the TOS field of business traffic.

When the controller, running the RouteFlow framework, discovers a new routing entry for an OpenFlow Router, the controller establishes two corresponding Flow Entries on the router. With the first flow entry, business traffic (TOS field enabled) is traversed through the second queue (configured above), and with the second flow entry, best-effort traffic (TOS field disabled) is traversed through the third queue (configured above).

When a QoS provider receives a request to reserve a bandwidth from a business customer, a confirmation regarding the availability of network resources is performed through the NorthBound API of the controller. If the resources are available on the path retrieved by OSPF, a rate limiter queue (Q) having the same priority as the second queue is configured on the edge router. Moreover, in order to enable the TOS field of business traffic and to redirect this traffic to the rate limiter queue, a forwarding entry is established on the edge router.

Upon a failure, the flow entries on the affected paths are re-established and the edge router reconfigures its rate limiter queues appropriately, along the available alternative path.

C.3 Results and discussions

In order to assess the described framework, experiments were performed on the OFELIA testbed facility provided by iMinds [5]. Fig. C.1A represents an emulated pan-European topology. Each switch in the topology also makes an out-of-band connection with a single controller. For emulation purposes, Open vSwitch was used as OpenFlow software and RouteFlow with our QoS framework was used as controller software. The bandwidth capacity of each link in the topology was limited to 50 Mb/s. In the experiments, each server sent both business traffic (30%) and best-effort traffic (70%) to all other servers in the topology using DITG (Distributed Internet Traffic Generator) [6]. In order to assess the framework, the

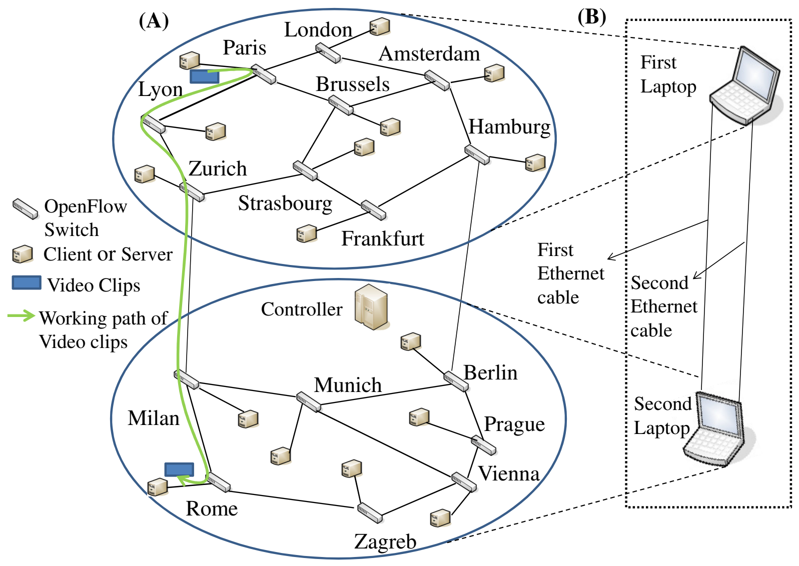


Figure C.1: (A) Emulated pan-European topology (B) Portable testbed

rate of both traffic sources, following a Poisson distribution, was varied and one of the links was torn down. Afterwards, the effects on QoS of these operations on business and best-effort traffic were thoroughly analyzed. Regarding failure recovery, we do not focus on providing fast-failure recovery [7] but instead, we focus on the scenarios in which high-priority traffic always gets a higher precedence than best-effort traffic.

Three distinct scenarios of business traffic were analyzed: low data-rate (less than 2.4 Mb/s of business traffic from each server to other server); medium data-rate (between 2.4 and 7 Mb/s); and high data-rate (more than 7 Mb/s). For the low data-rate scenarios, neither business nor best-effort customer traffic received the degraded service. This was because the failure-free path (before and after the link down) had enough bandwidth to accommodate both business customer and best-effort traffic. In the medium rate scenarios, only best-effort traffic received the degraded service. This was because the failure-free path (before and after the link down) had only enough bandwidth for business traffic. As a result, some of the best-effort traffic had to drop in order to meet the requirements of the business traffic. Finally, for the high data rate scenarios, business traffic had also received the degraded service after the link down. This was because in this scenario, some of links in the failure-free path after the link down had not the enough bandwidth to accommodate all the business customer's traffic. Therefore, some of business traffic was also dropped. In these links, we observed about 0 Mb/s best-effort traffic.

C.4 Demonstration on portable testbed

With the portable testbed (two laptops, Fig. C.1), we show the working of our QoS framework using an emulated pan-European topology. With the Mininet software [8], the half of the topology is emulated on the first laptop and the other half is emulated on the second laptop. The connection between the emulated topologies on different laptops is done using two Ethernet cables, shown in the figure. The controller, which runs our framework, is located on the second laptop. The controller controls all the switches of the topologies including the switches on the first laptop by the second Ethernet cable.

The link and traffic characteristics in the demonstration is replicated from the scenarios presented in the previous subsection. Hence, DITG is used to send business and best-effort traffic from each server. In addition, the server connected to Paris (which is present on the first laptop) streams two video clips – one as business traffic and the other as best-effort traffic – to the server connected to Rome (which is present on the second laptop). These video clips follow the path through the first Ethernet cable of the laptops.

In the demonstration, we show all the three scenarios presented in the previous subsection by simply watching the video clips of business and best-effort traffic on the second laptop. These three scenarios are shown by varying business and best-effort traffic (DITG traffic) from each server of the topology. For a failure condition of these scenarios, during the demonstration, we remove the first Ethernet cable of the laptops (the working path of video clips) and show switching of the video clips from the first Ethernet cable to the second Ethernet cable. After the failure, we show that business traffic always gets better QoS than best-effort traffic.

Acknowledgment

This research has received funding from the EU FP7 under agreement n^o 317576 (CityFlow), and n^o 258365 (OFELIA).

References

[1] OpenFlow and OF-ConFig: https://www.opennetworking.org/.

[2] B. Pfaff et al., *The Open vSwitch Database Management Protocol*, IETF, 2013 (http://tools.ietf.org/html/draft-pfaff-ovsdb-proto-04).

[3] S. Sharma et al., *Automatic configuration of routing control platforms in OpenFlow networks*, ACM SIGCOMM, Vol. 43(4), pp. 491-492, 2013.

[4] RouteFlow code [Online]. Available: https://sites.google.com/site/routeflow/.

[5] OFELIA testbed [Online]. Available: Available: http://www.fp7-ofelia.eu/.

[6] A. Botta et al., *A tool for the generation of realistic network workload for emerging networking scenarios*, Computer Networks, 2012.

[7] S. Sharma et al., *OpenFlow: Meeting carrier-grade recovery requirements*, Computer Communications, Vol. 36(6), pp. 656-665, 2013.

[8] Mininet Software [Online]. Available: http://mininet.org/.