SDN for Large Scale IoT Networks

Subhrendu Chattopadhyay

SDN for Large Scale IoT Networks

Thesis submitted in partial fulfilment
of the requirements for the degree of

Doctor of Philosophy

by
Subhrendu Chattopadhyay

under the supervision of
Prof. Sukumar Nandi

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, INDIA
March 30, 2021

Intentionally Left Blank

वितर्कविचारानन्दास्मितानुगमात सम्प्रज्ञातः ।

“The absolute knowledge can be attained after thinking, reasoning and assimilation”

Dedicated to
All my teachers

Who taught me to assimilate the information to convert it into knowledge.

Declaration

I declare that

1. The work contained in this thesis is original and has been done by myself under the general supervision of my supervisor.
2. The work has not been submitted to any other Institute for any degree or diploma.
3. Whenever I have used materials (data, theoretical analysis, results) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.
4. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Subhrendu Chattopadhyay

Place: IIT Guwahati
Date: March 30, 2021

Subhrendu Chattopadhyay
Research Scholar
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Guwahati, INDIA 781039

Certificate

This is to certify that the work contained in this thesis entitled "SDN for Large Scale IoT Networks" is a bonafide work of Subhrendu Chattopadhyay(Roll No. 146101002), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and is worthy of consideration for the award of the degree of Doctor of Philosophy of the Institute.
The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree or diploma.

Place: IIT Guwahati
Date: March 30, 2021

Prof. Sukumar Nandi
Thesis Supervisor
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Guwahati, INDIA 781039

Intentionally Left Blank

Acknowledgements

This thesis would not have been complete without the support of my thesis supervisor *Prof. Sukumar Nandi* and my mentor *Dr. Sandip Chakraborty.* Both of them have provided their support, knowledge and guidance to help me in my research. I have been working with both of them from my early days of M.Tech. I am truly grateful to them for putting me back in track during the trying times of my PhD. I have also learned many things about life while having a discussion with them. Their constant support and positive criticism have influenced my attitude, nature and personality in many ways.

I am very thankful to my doctoral committee members *Prof. Diganta Goswami*, *Prof. Ratnajit Bhattacharya* and *Dr. Tamarapalli Venkatesh* for understanding my works and providing valuable comments to improve the work. I extend my gratitude to the thesis reviewers, *Dr. Antony Franklin* from Indian Institute of Technology Hyderabad and *Prof. Ashutosh Dutta* from John Hopkins Whiting School of Engineering . I express my sincere thanks to former *Head of the Department Prof. S.B. Nair* and *Prof. S.V. Rao* and current *Head of the Department Prof. Jatindra Deka* for providing a nice research environment in the *Department of Computer Science and Engineering, IITG* and support my research work in many ways. During my early stages of research I enjoyed discussion with *Dr. Sushanta Karmakar* and *Dr. Arnab Sarkar* who helped and motivated me in a lot of ways. I am forever grateful for their support and time.

I am also very grateful to *Tata Consultancy Services India Private Limited* for awarding me the research fellowship that gave me extremely good opportunities to broaden my research activities and interact with eminent researchers in the world, both from the industry as well from the academia. I thank *Mr. Sachin Parkhi*, *Program Manager of TCS Research Scholar Program*, for extending their helps and supports in technical and official activities.

I had the privilege to collaborate with *Dr. Sushanta Karmakar*, *Dr. Samar Shailendra*, *Prof. Soumya Kanti Ghosh* and *Dr. Abhinandan S Prasad* whose interest and knowledge has enriched me a lot. I also enjoyed working with *Dr. Niladri Sett*, *Mr. Soumyajit Chatterjee* and *Mr. Shubabrata Nath.* I express my sincere gratitude towards *Mr. Debasish Naskar* for designing the logos and cover page of this thesis.

I would also like to express my hearty gratitude to *Prof. Gautam Barua* and *Prof. Gautam Biswas* the past Directors of the institute, *Prof. T. G. Sitharam* the present Director of the institute, all the Deans and other officials of IIT Guwahati, whose collective efforts have made this institute a place for world-class studies and research. I am thankful to all the faculties and the staffs of the Department of Computer science and Engineering for extending their cooperation in terms of technical and official supports. I thank the research scholars, M. Tech and the B. Tech students of this institute, with whom I have closely worked. I am sorry for not to mention all of their names, however, I have learned a lot from them during the course of our discussions.

I am forever thankful to *Rahul, Soumadip, Akash, Mandar* and *Ujjal* for the countless coffee sessions and round of discussions on various topics of interests which I will be missing in my upcoming days. My sincere thanks to my lab-mates *Pranav, Madhurima, Pradeep, Debanjan, Kangan* and *Sikha* for providing me a healthy work environment and helping me out during my needs. I am thankful to *Bennith and Suraj* who helped me to develop experimentation frameworks.

Last, but not least, I probably would not have done a PhD were it not for my parents. From the childhood they have encouraged me to raise questions and indulged my curiosities. Many of the things I appreciate in research can be traced back to an idealized version of what my parents said while I was growing up. While, I often disagree with them on issues, everything I do is influenced by them. My wife *Piyali* also helped me in writing and proof reading this thesis. Her constant support has helped me a lot in writing this thesis.

Finally, there are several collaborators and anonymous reviewers who are not listed out explicitly, but who had an impact on my experience in grad school: people who enabled me to solve new problems, find new topics, and just meet new people. This acknowledgement was getting too long to list everyone, but know that all of you made a huge difference.

Subhrendu Chattopadhyay

Place: IIT Guwahati
Date: March 30, 2021

Subhrendu Chattopadhyay
Research Scholar
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Guwahati, INDIA 781039

Abstract

Internet of things (IoT) is one of the rapidly growing network technologies which has the potential to serve millions of devices. Such a large scale IoT network (LSiN) requires network management to efficiently serve the end-user applications. The modern network management systems are expected to identify the time varying traffic pattern and take suitable actions to ensure fine grained network management. Taking these dynamic decisions require programmability in the network, where the programmable network management system can be used to deploy evolutionary protocols rapidly based on the objective. However, traditional Internet architecture suffers from lack of flexibility due to absence of programmability. Software-Defined Network (SDN) has emerged to provide a flexible architecture for network control and management. Additionally SDN provides opportunities to cater ever increasing bandwidth demand by fine tuning the network resources. The objective of this thesis is to design a distributed scalable SDN orchestration framework which is suitable for handling the dynamic nature of LSiN. In this thesis we explore the performance improvement of IoT applications by utilizing SDN. Subsequently we explore various deployment and architectural design related issues of SDN.

Modern IoT and hand-held devices are equipped with multiple interfaces. To leverage the bandwidth capacity of multiple interfaces several multi-path transport layer protocols exist which provide bandwidth aggregation. The first contribution in this thesis enhances the performance of IoT applications by proposing SDN control plane application `SDN-MPTCP` for Multipath TCP (MPTCP), where MPTCP is one of the popular multipath transport protocols. In this work we find that the performance of MPTCP has a strong correlation with the selected paths, and SDN can assist in path selection of MPTCP. During the performance improvement by employing SDN we understood that the deployment of SDN over an existing LSiN increases the capital expenditure (capex) of the system. Moreover, the centralized nature of the SDN control plane becomes a single point of failure for the network operation. Therefore, in this next work we investigate the SDN deployment challenges for LSiN. As mentioned earlier, SDN requires deployment of SDN supported hardwares. In this work, we utilized Network Function Virtualization (NFV) for development of `FLIPPER`. `FLIPPER` enables deployment of SDN like network management over existing Commercial off-the-shelf (COTS) devices of LSiN by converting them into Policy decision and enforcement points (PDEPs). `FLIPPER` provides a scalable, flexible, fail-safe and distributed "*self-stabilized*" architecture. In the next contribution we use `FLIPPER` to design `Aloe` orchestration framework which utilizes the in-network or In-network processing (In-network processing) platforms of LSiN to achieve "*servicification*" of SDN control plane. `Aloe` promises "*plug-and-play*" and "*zero touch deployment*" along with light-weight, fault-tolerant and auto-scalable network management platform for LSiN. Through exhaustive experimentation over an in-house test bed and Amazon web service (AWS) platform we find that `Aloe` can significantly improve performance of various IoT applications. During this study we also observed that various end-user applications targeted for LSiN require Virtual Network Function (VNF) based Service Function Chaining (SFC) depending on the network service access policy. However, dynamic deployment of VNFs and traffic steering through those VNFs to preserve the SFC ordering is difficult in a LSiN which spans across multiple administrative domains. In the next contribution we propose `Amalgam` which incorporates SFC management with `Aloe` to ensure scalability and dynamic SFC. Based on the NP-hard nature of VNF placement problem, `Amalgam` also proposes a greedy heuristic for VNF placement which ensures fast flow initialization and provides performance improvement for short-duration flows. As a whole, this thesis provides auto-scalable and fault-tolerant distributed architecture and orchestration framework for LSiN network management to provide fine-grained network control. We have compared the proposed solutions with the state-of-the-art works. Based on the experimental results we found that the proposed solutions can provide significant performance improvement for short duration flows while incurring lower resource consumption of the system.

Contents

List of Figures

List of Tables

Intentionally Left Blank

List of Algorithms

List of Abbreviations

ACL	Access Control Lists
API	Application Programming Interfaces
ARIMA	Autoregressive Integrated Moving Average
AWS	Amazon web service
BALIA	Balanced Linked Increase Algorithm
COTS	Commercial off-the-shelf
CTLR	Controller State
DAL	Device Abstraction Layer
DCN	Data Center Networks
DHT	Distributed Hash Table
DHT-NIB	Distributed Hash Table based NIB
DHT-flipper	Distributed Hash Table based flipper
DNS	Domain Name Server
DPI	Deep Packet Inspector
DTMC	Discrete Time Markov Chain
element wise product	Hadamard product
EM	Element manager
HOL	Head of Line
ICN	Information centric network
ID	identifier
IoT	Internet of things
IETF	Internet Engineering Task Force
ISPN	Internet Service Provider Network
l2tp	Layer Two Tunneling Protocol
LIA	Linked Increase Algorithm
LLDP	Link Layer Discovery Protocol
LAN	Local area network
M2M	Machine to Machine
MAPE	Mean Average Percentage Error
MEC	Mobile Edge Computing
MILP	mixed integer linear program
MIS	Maximal Indemendent Set
MPTCP	Multipath TCP
NAT	Network Address Translation
NFVI	NFV infrastructure
NFVO	NFV orchestrator
NFV	Network Function Virtualization
NF	Network Functions
NIB	Network Information Base
NOS	Network Operating System
NSAL	Network Service Abstraction Layer
OS	Operating System

OOS Out of order segments

ODL Open Day light

OFS OpenFlow supported switch

OLIA Opportunistic Linked Increase Algorithm

OVS Open virtual switch

P2NM PushToNode Module

PDEP Policy decision and enforcement point

PMM Path Manager Module

PRIO Priority

MPTCP Multipath TCP

QoS Quality of Service

RMM Resource Management Module

RE Resource Enforcer

RM Resource Monitor

REST REST API

RTT Round Trip Time

SCI Service Chain Identifier

SCM Service Chain Manager

SDC Service Deployment Controller

SDM State Discovery Module

SDN Software-Defined Network

SFC Service Function Chaining

SNC Super Network Controller

SNMP Simple Network Management Protocol

SS-MIS self-stabilizing MIS

TCP Transmission Control Protocol

TCAM Ternary Content-Addressable Memory

TMM Topology Management Module

VIM Virtualized infrastructure manager

VMM VNF manager module

VM Virtual Machines

VNFM VNF Manager

VNF Virtual Network Function

WAN Wide area network

capex capital expenditure

opex operational expenditure

pmf probability mass function

e Euler-Mascheroni constant

In-network processing in-network or In-network processing

LSiN large scale IoT network

OSS/BSS Operations and Business Support Systems

μ**S** micro-service

μ**C** micro-controller

μ**MM** μC Manager Module

μ**PM** μC Placement Module

μ**CaaS** μC as a Service

Chapter 1

Introduction

Internet of things (IoT) refers to an interconnecting infrastructure to integrate everyday used embedded computing devices. Recently IoT is being used for improving the quality of life [1]. In an IoT, the number of end-users in a single network can reach up to a million [2] very easily. Due to this, the global Machine to Machine (M2M) traffic is estimated to reach 51% [3] of the total traffic demand in 2023. Therefore, IoT is estimated to grow as a major technology in the near future. In this thesis, we focus on large scale IoT network (LSiN) . Like IoT, LSiN also spans from backbone network to edge devices. We identify LSiN as a Wide area network (WAN) which is a subset of IoT. We make the following key assumptions to segregate LSiN from IoT in this thesis.

- The LSiN contains millions of heterogeneous resource constraint Commercial off-the-shelf (COTS) devices. Each device can have multiple interfaces. The traffic generated by the LSiN devices is mainly short-flows [4].
- Since it is difficult to deploy such a vast network while maintaining single administrative domains, LSiN spans across multiple administrative domains.
- By looking at the momentum of virtualization technologies used presently [5], we firmly believe that virtualization can be an inherent technology used in the future LSiN.

Since the user generated network traffic pattern is time variant in nature, to optimize the potential of the costly physical resources, the network behaviour requires periodic customization of the device configurations. To ease the customization, network management systems exist in the literature [6, 7]. However, most of the modern networks like LSiN require evolutionary protocols and internet architectures to cater the diversified user needs. For example, many enterprise systems utilize private cloud, public cloud or a mix of both to reduce the cost of operations based on the traffic demand to reduce the cost of operation. To ensure the smooth dynamic transition between these architectures the network administrator requires a evolutionary design. Managing such a system with traditional network management approaches result in complicated configuration and, minor inconsistency in configuration that results in a significance drop in performance. Additionally overhead of a new protocol deployment requires significant development and testing time due to the complicated configurations. These issues can be avoided easily by implementing programmable network. Software-Defined Network (SDN) [8, 9, 10, 11] can realize evolutionary network management by implementing programmable network which can ensure easy and rapid deployment of new protocols or network architectures.

SDN plays a significant role in handling dynamic demands of network management [12] where traditional approaches generally struggle. SDN has been developed to ensure dynamic management of network and it relies on "*control plane*" and "*data plane*" separation where control plane responsibilities are assigned to dedicated devices called "*controller*". SDN controllers maintain a logically centralized view of the network and provide programmability through standard Application Programming Interfacess (APIs). Therefore, SDN has the potential to assist system administrators in defining and enforcing dynamic network-wide policies.

However, available SDN oriented solutions for existing "*backbone networks*" like "*enterprise network*" [13, 14, 15], Internet Service Provider Network (ISPN) and Data Center Networks (DCN) [16, 17, 18] does not suit well in case of LSiN. The salient differences between existing backbone networks and LSiN are as follows.

1. Unlike existing backbone networks, LSiN does not use costly hardware. Therefore, performance improvement of traffic generated in LSiN is difficult. LSiN can be extended to mobile devices also. For example, IoT applications can use the idle resources of mobile devices. In such cases, LSiN provides performance improvement by aggregating resources from multiple such devices that require fine-grained network control over a highly distributed platform.

2. LSiN requires higher degree of scalability than DCN or ISPN. On the other hand, SDN supported devices are costly, and it isn't easy to replace all the existing pieces of equipment at a single sweep. Therefore, reduction of capital expenditure (capex) and operational

expenditure (opex) are serious concerns in case of a LSiN.

3. Since LSiN can be composed of resource constraint devices and mobile devices, the system is highly dynamic and failure-prone in nature, which rarely happens in the case of DCN/ISPN.
4. The existence of heterogeneous devices results in diversified traffic demands. Fine-grained management of these traffic classes requires various types of network-oriented services apart from simple quality of service (QoS) management and route selection challenges.

1.1 Motivation for This Thesis

In this thesis, we identify some of the issues related to SDN oriented network management of LSiN. Our research is primarily based on the following questions.

Question 1.1.1 How to improve performance of the applications in an LSiN?

Due to the increase in integrated sensors in smart-phones and other hand-held devices, mobile devices have become one of the essential parts of LSiN deployment [19]. Improvement in mobile traffic can significantly improve the quality of LSiN user application performance. Since, the application layer performance is highly dependent on the transport layer performance. Therefore, we aim to find the issues in transport layer protocols used in mobile devices.

Modern mobile devices are usually equipped with multiple hardware interfaces. Avaialability of multiple interfaces can be exploited by aggregating the available bandwidth at all interfaces. The aggregation of bandwidth can be used to satisfy the ever increasing traffic demand. Multipath TCP (MPTCP) [20, 21, 22, 23, 24] is a transport layer protocol primarily used in data-center and enterprise networks. Usually the hosts used in data-center and enterprise network are equipped with multiple interfaces. MPTCP provides the support for bandwidth aggregation in such cases via concurrent usage of different interfaces by creating multiple subsockets.

MPTCP initiates multiple sub-flows via different interfaces to aggregate the bandwidth. However, in a network, the path characteristics (such as bandwidth, delay, loss rate, jitter, etc.) of the underlying sub-flows can be significantly different and time-varying. This diversity adversely affects MPTCP performance. Additionally, the time-varying nature of the path characteristic further compounds the problem as it is difficult to estimate them apriori. The difference in end-to-end path characteristics of each active sub-flow may lead to an increase in out of order delivered segments at the receiver side. This increase in out of order delivery leads to Head of Line (HOL) blocking at the receiver side [25]. HOL blocking also results in delays and increases packet drops, which increase the number of retransmission timeouts. Currently, MPTCP uses Round Trip Time (RTT) as a measure of path characteristics. However, in the case of MPTCP, one segment and its acknowledgment might follow different paths which leads to unreliable measure of path characteristics. On the other hand, the effect of MPTCP congestion control and segment scheduling is discussed in the literature [26, 27, 28, 29] also depends on the path characteristics. This issue can be avoided by modelling the MPTCP behaviour based on the available end-to-end semantics which to the best of our knowledge none of the prior works tried. The absence of such formal model becomes necessary for minimizing HOL blocking and designing of an intelligent path identification method to increase transport layer performance.

Question 1.1.2 How to choose a suitable design of SDN for LSiN management?

Apart from the transport layer performance issue, the biggest challenge in LSiN is to maintain the scalability of the network. Let us consider the following scenario where the network administrator of an LSiN wants to dynamically update bandwidth distribution policies based on network usage statistics. The network is connected with multiple network service providers, and therefore she needs to update the configuration at different edge routers and gateways. With traditional network devices, like layer 3 switches, this task is tedious. Even a minor configuration inconsistency among the edge routers and gateways may lead to severe network under-utilization or bandwidth imbalance. Further, the system is also not scalable for dynamic updates of network configuration policies.

SDN [15] can help in dynamic network configuration update. SDN uses a centralized controller to convert system administrator defined policies to device configurations and apply those configurations in the targeted networking devices. By using the programmable controller, SDN separates the network control plane from the data plane. The SDN control plane takes care of all the control functionalities (like forwarding decision) based on the network parameters and installs the control decisions to the data plane devices. In contrast, the data plane is only responsible for forwarding packets based on the configuration parameters set by the control plane.

Although SDN has revolutionized dynamic network management aspects, it requires specific hardware that can understand the instructions given by the SDN controller. Therefore the critical question is: *How much effort and cost does one need to convert an existing network infrastructure to an SDN supported one?* An SDN supported hardware is much costlier than a COTS network device, which requires huge capex to replace existing infrastructure by SDN supported infrastructure. Deployment of SDN supported equipments incrementally can be one way to avoid this extra cost. On the other hand, there are existing SDN control plane architectures [30, 31, 32] which propose interoperability between the SDN and non-SDN devices. However, in both the cases fine grained network control can be ensured. Therefore, we require a technique that can transform COTS device into an SDN supported Policy decision and enforcement point (PDEP) device in order to reduce the cost of deployment. On the other hand, the use of COTS devices as PDEPs can increase the failure rate, which increases opex. By ensuring fault and partition tolerance the opex can be reduced which motivated us to understand a suitable design of SDN that can satisfy the above mentioned challenges of SDN deployment over LSiN.

Question 1.1.3 Can SDN harness the dynamic nature of LSiN?

Apart from the fault, and partition tolerance and capex related issues, the dynamic nature of the LSiN is also difficult to manage. Due to the rise of IoT, rapid proliferation of LSiN has made the network architecture complicated and challenging to manage for service provisioning and ensuring security to end-users. Simultaneously, with the advancement of edge-computing, in-network or In-network processing (In-network processing), and "*platform-as-a-service*" technologies, end-users consider the network as a service platform for deployment and execution of myriads of diverse applications dynamically and seamlessly over the network. Consequently, network management is becoming increasingly difficult in today's world with this intricate service-oriented platform overlay on top of the inherently distributed "*TCP/IP*" network architecture. The cost-effective and logically centralized control plane of SDN is useful for monitoring, controlling, and deploying new network services. Nevertheless, managing edge and in-network processing over an LSiN platform are still challenging even with an SDN based architecture [33].

Primary requirements for supporting edge and In-network processing over an LSiN are as follows: (1) Platform should be agile to support the rapid deployment of applications without incurring additional overhead for In-network processing [34]. The use of In-network processing also ensures the scalability of the system [35]. (2) In-network processing often requires dividing a service into multiple microservices and deploying the microservices at different network nodes for reducing application response time with parallel computations [36]. However, such microservices may need to communicate with each other, and therefore flow-setup delay from the in-network nodes need to be very low to ensure near real-time processing. (3) Percentage of short-lived flows are high for "*things*" centric LSiN [37]. This property of LSiN requires a reduction of flow-setup delay in the network. (4) Failure rates of unmanaged LSiN devices are in-general high [38]. Therefore, the system should support a fault-tolerant or fault-resilient architecture to ensure liveness.

Although SDN supported edge computing and In-network processing have been widely studied in the literature for the last few years [39, 40, 33] as a promising technology to solve many of the network management problems associated with LSiN, they have certain limitations. The logically centralized control plane of SDN becomes a bottleneck when each flow initiation requires communication between switches and the controller. The bottleneck scenarios can be avoided by using a distributed SDN control plane. In such a case, placement of controllers is vital for the reduction in flow performance of LSiN, where most of the flows are short-lived. On the other hand, static deployment of controllers is not adequate to provide fault-tolerance to LSiN, where most of the devices show "*plug-and-play*" nature. Therefore, we found that the design of an SDN control plane that reduces control plane bottleneck and caters to "*plug-and-play*" devices of an In-network processing framework deployed on top of LSiN are very much necessary.

Question 1.1.4 How to create a management framework for rapid deployment and performance enhancement of "middlebox" dependent traffic?

Since the LSiN can provide a large number of heterogeneous applications, it requires different network-centric services like Network Address Translation (NAT), "*proxy*", "*firewall*". In literature, these services are termed as "*middlebox*" applications. End-user application performance depends on locations and performances of the middlebox applications. Therefore, the management of middlebox applications becomes important in LSiN. The source/destination oriented routing protocols are insufficient for steering traffic through these middleboxes. The problem intensifies when middleboxes are deployed using virtualized platforms (termed as Virtual Network Function (VNF)) where locations of middleboxes can change dynamically [41, 42, 43].

Depending on the type of applications, a single flow may require services from multiple VNFs. In such cases, the order of execution is also essential. An ordered set of VNFs for a particular flow is known as Service Function Chaining (SFC). Since LSiN can span across multiple administrative domains, the development of a unified, scalable framework for the management of SFCs and traffic steering through them is not an easy feat. It isn't easy to design a VNF management framework that is scalable and still capable of providing QoS requirements of the traffics. To satisfy the QoS demand in an LSiN where the number of short-duration flows is significantly high, scheduling and placement of VNFs in the actual devices require quick convergence. Furthermore, the entire framework must comply with the "*plug-and-play*" nature of the LSiN devices.

1.2 Contributions

In this thesis, we propose solutions to the issues mentioned earlier by developing SDN control plane applications and orchestration frameworks for LSiN or similar systems. The proposed solutions presented here investigates several challenges of LSiN (and/or like scalability, incremental deployment issues, transport layer protocols, network management systems, and service chaining management. The step-by-step contributions of this thesis are as follows.

1.2.1 Improvement of MPTCP Performance

The first major contribution of our thesis is an intelligent dynamic path management scheme for MPTCP traffics that optimizes the traffic performance as mentioned in Question 1.1.1. To develop this path manager, we rely on the SDN control plane which provides a logically centralized view of network topology parameters by periodically obtaining statistics from all its data plane devices [10]. This centralized view makes it feasible to optimize the end-to-end performance of MPTCP by selecting a suitable active set of MPTCP sub-flows. In order to identify a suitable active set, we provide a formal model of MPTCP by using an irreducible and aperiodic Discrete Time Markov Chain (DTMC). The proposed formal model provides an estimation of MPTCP throughput and receiver buffer length based on end-to-end path characteristics (latency, available bandwidth, etc.) of the sub-flows. We use this estimation mechanism to develop an SDN control plane application named as `SDN-MPTCP`. The performance of the proposed solution is compared with various baselines. During the evaluation period, we suffered from the lack of real

LSiN experimental facility. This challenge motivated us to investigate the deployment challenges of an SDN enabled LSiN infrastructure.

1.2.2 SDN Deployment Over LSiN

The primary challenge to design a suitable SDN control plane for LSiN infrastructure is to reduce thecapex as mentioned in Question 1.1.2. Therefore, in this thesis we design `Flipper`. `Flipper` is somewhere in-between traditional architecture and SDN based architecture, where COTS routers dynamically change their roles from a conventional network router to an Network Information Base (NIB) and participate in PDEP functionalities. `Flipper` reduces capex by using COTS devices with the help of Network Function Virtualization (NFV) [44][1]. We also propose a distributed self-stabilizing NIB placement algorithm which reduces the opex by ensuring fault and partition tolerance. We also provide formal proofs to ensure the linear convergence of the proposed algorithm. The performance of `Flipper` is analysed from both simulations through a synthetic network environment and real implementation over an emulation platform using "*network name-space*". Our implementation of Flipper provides proof-of-concept support of the new architecture while comparing performance with existing methods in terms of flow initiation delay.

1.2.3 Providing Plug and Play Support

We extended the `Flipper` principles to develop `Aloe`, which is a fault tolerant SDN orchestartion framework for dynamic In-network processing platforms. `Aloe` is custom built to cater the "*plug-and-play*" devices (Question 1.1.3) of LSiN. `Aloe` primarily serves two purposes: (a) easy and improved management of LSiN application generated flows (b) without increasing additional capex. To implement these two features, `Aloe` exploits the capabilities of In-network processing platforms and proposes "*servicification*"[2] of control plane. `Aloe` ensures auto-scalability which is desired for a large scale network like LSiN. Additionally, our proposed framework preserves the `Flipper` properties like fault-tolerance and linear time convergence which help to reduce the flow initiation time significantly. We found significant performance improvement for various end user applications in our experimental set-up using an in-house test bed and a large scale Amazon web service (AWS) platform.

---

[1] At the time of this research NFV was less popular

[2] Servicification is defined as "transformation of existing system into one or more discrete services" [45]

1.2.4 Enhancing Capability of SDN Managed LSiN Using "middlebox" Application Management

`Aloe` is further extended into `Amalgam` to combat the issues given in Question 1.1.4. `Amalgam` couples distributed SFC management and SDN enabled traffic steering framework. `Amalgam` can extend its services over multiple administrative domains by exploiting in-network processing [46, 47] architecture. `Amalgam` ensures fine-grained QoS. Moreover, `Amalgam` is compatible to cater the "*plug-and-play*" nature of the devices without compromising operation, where, the plug-and-play devices may join and leave the platform dynamically. The coupling of VNF placement and traffic steering in `Amalgam` ensures dynamic service chaining during an on-going session. To evaluate performance of `Amalgam` we develop an emulation framework *MiniDockNet* for VNF deployment using "*docker*" [48] over in-network processing, as the existing network name-space oriented mininet [49] emulator is not sufficient for in-network processing. The performance of the proposed framework is compared with some of the existing works, which shows that `Amalgam` can provide better end-to-end delay than it's predecessors for short-duration flows.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 provides background and preliminaries to understand various technical aspects of this thesis. Chapter 3 proposes `SDN-MPTCP` which is an SDN oriented framework to improve the performance of MPTCP. In Chapter 4 we analyze the capex-opex trade-off and propose `Flipper` which is a scalable control plane architecture suitable for LSiN. Chapter 5 describes proposed `Aloe` orchestration framework. `Aloe` is an extension of proposed `Flipper` and is capable of handling the dynamic nature of the LSiN. In Chapter 6 we analyse the SFC management issues to propose `Amalgam`. `Amalgam` solves the VNF placement and traffic steering problem over multiple administrative domains in LSiN. Finally, we conclude the thesis and suggest possible future works in Chapter 7.

Chapter 2

Background

In this thesis, we contribute towards several aspects of network management of large scale IoT network (LSiN). Each chapter presents a particular research problem, and a literature survey related to the problem is present in the corresponding chapters itself. Instead of providing a diverse literature survey, we present background, definitions, and brief descriptions of some of the existing works used in this section.

2.1 Large Scale IoT Network (LSiN)

LSiN is a special case of Internet of things (IoT). Therefore, we describe LSiN in the context of IoT.

2.1.1 Internet of Things (IoT)

IoT [50] is one of the popular and wide spread technology to connect "*things*" and provide intelligence to the real world problems. However, most of the definitions available for IoT[51, 52] is context dependent. In order to generalize, we define IoT, based on the desired characteristics [53] of an IoT system as follows.

- **Things:** IoT systems are composed of "*things*", where "*things*" refers to any physical object/device relevant to an user/application.
- **Sensing/Actuation capable:** "*Things*" can be capable of sensing/actuation to interact with the physical world and eligible to bring smartness to users/applications.

- **Programmable:** "*Things*" are programmable devices, which means they can exert multiple behaviors based on users' behest.
- **Communication Capable:** The things must be "*communication capable*" through standard interfaces and inter-operable communication protocol.
- **Connected Through Internet:** The things must be connected through the Internet. As the name suggests, the system can not only be a Local area network (LAN).
- **Uniquely Identifiable:** Since the things are connected via the Internet, unique identification for each entity is a necessity.
- **Accessibility:** Ideally IoT devices should be accessible "*anytime*" and from "*anywhere*". However, this may not be required for most of the IoT applications. Therefore, in context of IoT, the "*things*" must provide accessibility only "*when and where it is required*" instead of "*all the time and globally*".

Based on the characteristics, we define IoT system as given in Definition 2.1.

Definition 2.1 *IoT is defined as a network of "*Uniquely Identifiable*", "*Sensing/Actuation capable*", "*communication capable*", "*heterogeneous*" and "*programmable*" "*things*" which are "*connected through Internet*" in such a way that the things can be accessed "*when and where it is required*"*

As mentioned earlier, IoT is one of the basic building blocks for making smart systems like Smart Cities [54, 55], Smart Homes [56], Factory/office automation [57], Intelligent transportation systems [58] etc. The scale of IoT is increasing rapidly as the availability of low-cost networked components, and the need for automated monitoring keeps increasing.

In addition to the IoT, we define LSiN as a system having the the following characteristics

- **Short-flow heavy:** The end user applications running over the system generates mostly short duration flows[1].
- **Multiple administrative domain:** The system spans across multiple administrative domains.
- **Large scale:** The system can potentially scale upto millions of devices.

---

[1] In this thesis, we identify a flow as a short-flow which has a life time of less than a minute[37]

- **Multiple interfaces:** The used Commercial off-the-shelf (COTS) devices have multiple interfaces.
- **Resource constraint:** The used COTS devices are resource constrained in nature. Therefore, the system heavily utilizes micro-service architecture.
- **Use of virtualization:** The system relies on virtualization for resource isolation and micro-service deployment.

Aggregation of resources in an LSiN eco-system provides better utilization of resources and reduces the "***capex/opex***" significantly by employing multiplexing on top of the same physical hardware. Resource aggregation depends on the types of resources being shared. In this thesis, we focus only on the following two types of resource aggregation methods; (a) Bandwidth aggregation and (b) Computational Resource Aggregation.

2.1.2 Bandwidth Aggregation

Traffic demand for LSiN is about to consume 70% of the total users in 2023 [59]. The amount of bandwidth demand is also expected to increase accordingly. To support bandwidth-hungry applications running at the end-hosts, bandwidth aggregation is necessary. The majority of participating end-host devices of an LSiN are mobile and equipped with multiple interfaces that can be exploited for bandwidth aggregation. Several bandwidth aggregation methods (See Table 2.1) at transport layer [60, 61] are developed to exploit capabilities of multiple interfaces. Among the existing multipath protocols, `MP-SCTP` [62] uses message-oriented multiple streams, unlike byte-oriented `TCP`. The use of multiple streams between source-destination pairs provides reliability and bandwidth aggregation. However, a `MP-SCTP` connection can acquire greater bandwidth than a competing TCP flow over a single bottleneck link. Thus `MP-SCTP` shows "*unfriendliness*" towards existing TCP connections. The recent adoption of `MP-SCTP` named as `MSTCP` [63] extends use of multiple streams over Software-Defined Network (SDN) controlled network to provide fine-grained path control. However, `MSTCP` also suffers from TCP friendliness issues. In a LSiN with multiple flows, TCP unfriendliness is very much undesired [64]. On the other hand, separate congestion window management of `MSTCP` and `MP-SCTP` increases out-of-order delivery [61]. At the same time, path qualities used by streams have a high degree of disparity. A Data Center Networks (DCN) targeted multipath transport protocol `MP-DCCP` [65] uses dead-

Table 2.1: Popular Bandwidth Aggregation Protocols

Protocol	**Key Benefits**	**Issues**
`MP-SCTP` [62]	Uses multiple streams to increase reliability	TCP friendliness
`MSTCP` [63]	Stream based congestion control, independent congestion window for each stream	TCP friendliness, out-of-order delivery
`MP-DCCP` [65]	Deadline aware delivery, targeted towards data-center applications	unreliable and out-of-order delivery [61]
`NMCC` [66]	Ignore friendliness constraints (of LIA) if there is no competing flow	Not compatible in LSiN
`MPTCP` [64]	Sub-socket oriented multipath to ensure TCP friendliness and responsiveness to network changes	Provides better performance than the rest [69, 70]
`MP-Quic` [71]	User space multi-path using UDP	Very recent and partially implemented [69]

lines to schedule the packets among multiple streams to ensure timeliness of each flow. However, `MP-DCCP` suffers significantly due to its unreliable delivery when adopted to LSiN. `NMCC` [66] suffers from compatibility issues as it is developed to exploit capabilities of the "*`ICN`*"/"*`PSI`*". To solve existing challenges "*`MPTCP`*" [64] uses multiple sub-flows in the place of multiple streams. `MPTCP` ensures TCP friendliness by employing an appropriate congestion control mechanism. On the other hand, `MPTCP` explicitly manages responsiveness of the sub-flows in the presence of network change events, which makes it very compatible for LSiN bandwidth aggregation [67]. A very recent and popular user space transport layer implementation of `MP-Quic` can be suitable for LSiN. However, in this thesis, we use `MPTCP` since it has a stable source and has been adopted by the industry [68]. On the other hand, comparison of `MPTCP` and `Multipath Quic` [69] reveals that `MPTCP` can provide slightly higher throughput. Similar experimental comparison shows that, `MPTCP` performs slightly better [70] than `MP-SCTP`.

a Architecture of MPTCP

Multipath TCP (MPTCP) is standardized by the Internet Engineering Task Force (IETF) [72]. The primary design consideration for MPTCP is to use multiple available interfaces at the

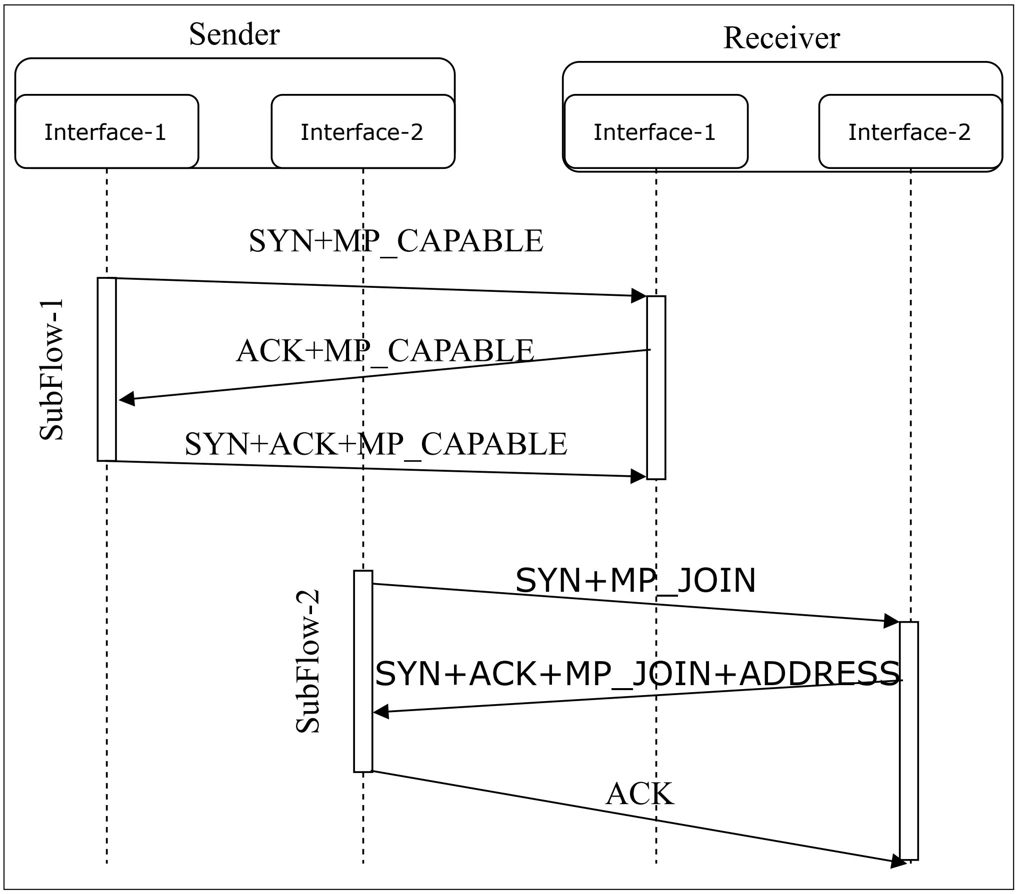


Fig. 2.1: MPTCP Event Timing

host device to transmit data in concurrent fashion. By the doing so, MPTCP can aggregate bandwidth for better throughput and error resilience. A typical connection establishment scenario for two sub-flow MPTCP is provided in Fig. 2.1. The primary sub-flow is initiated with a "*MP_CAPABLE*" flag along with standard "*SYN*" segment by sender. Upon receiving "*MP_CAPABLE + ACK*" from receiver, the sender initiates secondary sub-flow with "*MP_JOIN*" segment as shown in Fig. 2.1.

Current Linux kernel implementation of MPTCP [73] consists of three modules: *Path Manager*, *Segment Scheduler*, and *Congestion Control Mechanism*, as shown in Fig. 2.2. The *Path Manager* module manages the available sub-flows between the end hosts. Currently, MPTCP has proposed two choices of path manager:

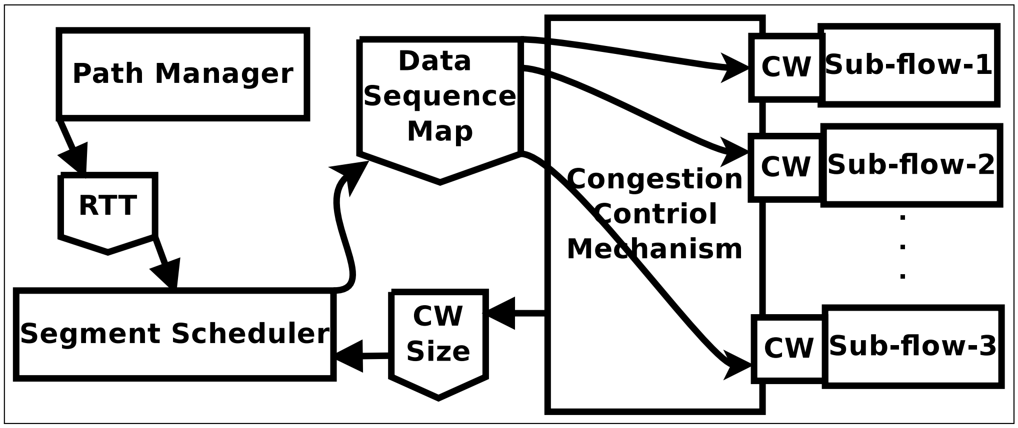


Fig. 2.2: MPTCP Architecture

- **Full-mesh:** This creates $N \times M$ sub-flows for a MPTCP connection with sender and receiver having N and M ports respectively. This is the default path manager.

- **ndiffports:** This arbitrarily selects k sub-flows among all available sub-flows.

Congestion Control mechanism controls the congestion window for each sub-flow separately. Several congestion control algorithms like Linked Increase Algorithm (LIA), Opportunistic Linked Increase Algorithm (OLIA), Balanced Linked Increase Algorithm (BALIA), etc. [74] have been proposed for MPTCP. Performance improvement of MPTCP by employing congestion control techniques are discussed in [75, 76, 27]. Peng *et.al.* [26] have shown that a congestion control mechanism design depends on a trade-off between responsiveness towards network changes and fairness towards other transport layer protocol. According to their work, `LIA` is unfair to TCP. On the other hand, `OLIA` is unresponsive towards network changes. Therefore, Peng *et.al.*has proposed a TCP `New-Reno` based balanced linked adaptation a.k.a `BALIA` [26], which balances this trade-off by introducing a normalized responsiveness factor. Once the congestion window size for each path is decided, the segment scheduler takes responsibility for scheduling segments to the individual sub-flows. "*Round-Robin*" and "*lowest RTT First*" are existing segment scheduling strategies described in the MPTCP standard.

One of the segment scheduler's core purposes is to reduce the out-of-order packets at the receiver. Based on our pilot study, we found that, despite different segment scheduler, MPTCP performance is adversely affected by the increasing disparity in active sub-flow characteristics.

Ou *et.al.* [77, 28] have considered to tackle Head of Line (HOL) blocking and proposed a joint congestion control and segment scheduling mechanism.

However, the existing segment schedulers use Round Trip Time (RTT) based approach to estimate receiver buffer size, which is not a reliable estimate for a lossy and dynamic network. Moreover, a segment and its acknowledgment might follow a different path. Therefore, segment scheduling does not perform well in the case of a dynamic network. In their work, Zhou *et.al.* [76] have shown that MPTCP provides near-optimal experience when the active sub-flows have similar path characteristics. We emphasize this issue and provide a detailed literature survey of `MPTCP` problems exclusive to this issue in Section 3.2.

2.1.3 Computational Resource Aggregation

Apart from bandwidth aggregation of communication resources, a large amount of resource constraint devices in LSiN provide the opportunity to aggregate computational resources (like CPU, Memory, etc.). The aggregation of resources can provide an alternative computational platform for time-critical processing. Unlike bandwidth aggregation, which provides resource aggregation at the end hosts, computational resource aggregation aggregates the entire network's resources. In this thesis, we denote this platform as in-network or In-network processing (In-network processing) platform. One of the primary objectives of LSiN is to allow embedding of intelligence inside the network. In-network processing helps to achieve that embedding.

"*INP*" [78, 79] existed in the literature for a long time. Recent advancemnet of IoT and "*cloud computing*" has exploited the use of In-network processing [80, 81]. In this thesis, we define In-network processing in context of LSiN as follows.

Definition 2.2 *In-network processing refers to a distributed system where residual resources of the networking equipments can be used for data processing.*

Use of In-network processing processing can significantly reduce capital expenditure (capex) and operational expenditure (opex) by multiplexing the existing hardware. In-network processing creates a resource pool from the residual resources from the platform's devices to provide computational services without employing cloud or any expensive technologies. In-network processing is also capable of providing quick responses to the delay-sensitive end-user applications [82, 83, 84], as it requires less communication overhead to access an In-network processing device than the traditional server/cloud infrastructure. Services associated with the user applications are sub-divided to create "*micro-services*" to cater resource demands of the user applications. The "*micro-services*" are light-weight so that they can be executed inside the resource constraint LSiN devices. Depending on nature and topological position of the execution devices [85], In-network processing architectures can be categorized into 3 platforms as shown in Fig. 2.3.

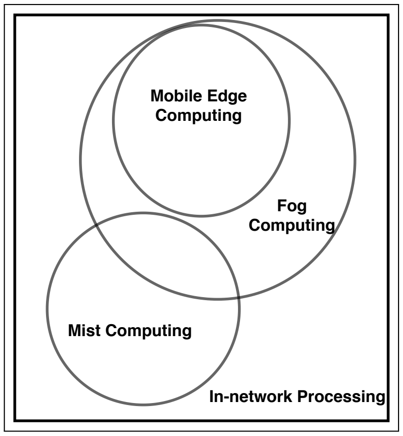


Fig. 2.3: Classification of In-network Processing

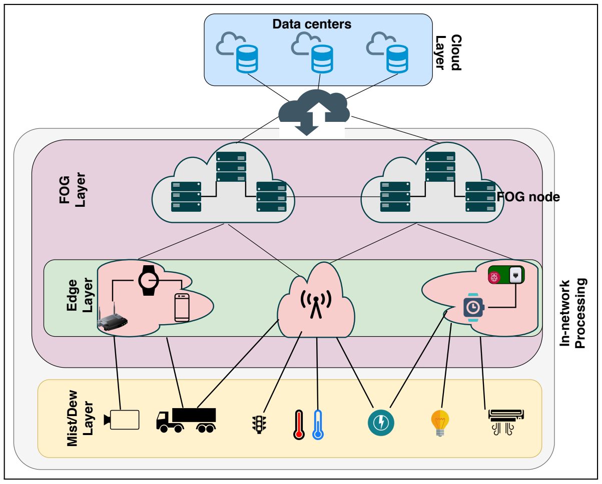


Fig. 2.4: In-network Processing Architecture

Mobile Edge Computing: In the case of Mobile Edge Computing (MEC) [86], micro-services are executed in the edge devices of a network. An edge device is part of an enterprise network that is directly connected to host/client devices. In this thesis, we identify MEC platforms are In-network processing platform where "*micro-services*" are executed in the edge devices of the mobile network infrastructure. A service deployed in a MEC platform can significantly reduce delay and jitter of the end-user application [87] since the service is deployed as close as possible to the hosts. Therefore, latency-sensitive services are most suitable for MEC. Additionally, MEC can reduce cloud management costs. Although MEC can provide significant performance improvement for real-time and mobility dependent services, it is not capable of handling huge data volume generated by LSiN.

Fog Computing: "*Fog computing*" [88, 34] extends the computational capabilities of MEC by including all the possible devices having idle resources. As per the standard [89], fog computing is a system-level horizontal architecture that distributes resources and services of computing, storage, control, and networking anywhere along the continuum from "*Cloud*" to "*Things*", thereby accelerating the velocity of decision making.

Mist/dew Computing: "`Mist/dew computing`" [88] extends the capabilities of the Fog computing by including the end host devices (IoT devices). Management of Mist/dew computing platforms requires a highly scalable and dynamic management platform.

In this thesis, we refer to a combined architecture of MEC, Fog, and Mist computing architecture, an In-network processing architecture. A pictorial representation of this integrated architecture is shown in Fig. 2.4, where the end devices can form a mist computing resource pool, and rest of the network components participate in the Fog computing architecture. The resource pool of In-network processing has two significant advantages. (a) The services of end-user applications executing over In-network processing has substantial performance benefits; (b) The network-related services can be hosted over the In-network processing platforms to improve overall quality of the network service. While advantages of the former concept are easy to apprehend, the later idea requires an explanation that has been provided in the next section.

2.2 Network Function Virtualization (NFV) / Virtualized Network Function (VNF)

Initial network developments and researches were to develop protocols to overcome the challenges. Though very successful, the protocol centric research is not adequate [42] for a modern rapidly changing network where the network layer requires frequent customization. To ensure customizability, the system must go through an evolutionary design. "The current Internet architecture and business relationships that have developed among various stakeholders have become a serious obstacle to its continuing evolution and growth"[2]. Network Function Virtualization (NFV) emerged to overcome the rigidity of network infrastructure. The formal definition of NFV is as follows.

Definition 2.3 *NFV utilizes virtualization techniques to deploy network-related services/functions on top of the networking/general-purpose hardware.*

---

[2] `https://www.arl.wustl.edu/netv/main.html`

For example routing[3], load balancing[4], intrusion detector [90] can be implemented using NFV. NFV is widely used in "*`ETSI-MANO`*" [91] for deployment of Virtual Network Function (VNF) over LSiN. The "*ETSI-MANO*" standard, can be applied for management of network services over In-network processing platforms. Thus, resource aggregation can improve performance of the network. The "*ETSI-MANO*" standard consists NFV orchestrator (NFVO) which maintains data repositories for possible configurable resources and set of permissible configurations (viz. NFV Instances, NFV resources and network service catalogue, VNF catalogue respectively) through VNF Manager (VNFM) module. VNFM interacts with the Element manager (EM) and NFV for configuration of NFV resource accounting, fault management of NFV and configuration of NFV. The physical resources of the infrastructure which are also refereed as NFV infrastructure (NFVI)), are managed using Virtualized infrastructure manager (VIM). VIM can be implemented as an agent residing in each devices. Overall management system is connected to Operations and Business Support Systems (OSS/BSS) which also provides a generic interface to other management systems. The OSS/BSS becomes useful in presence of a network control plane (like SDN).

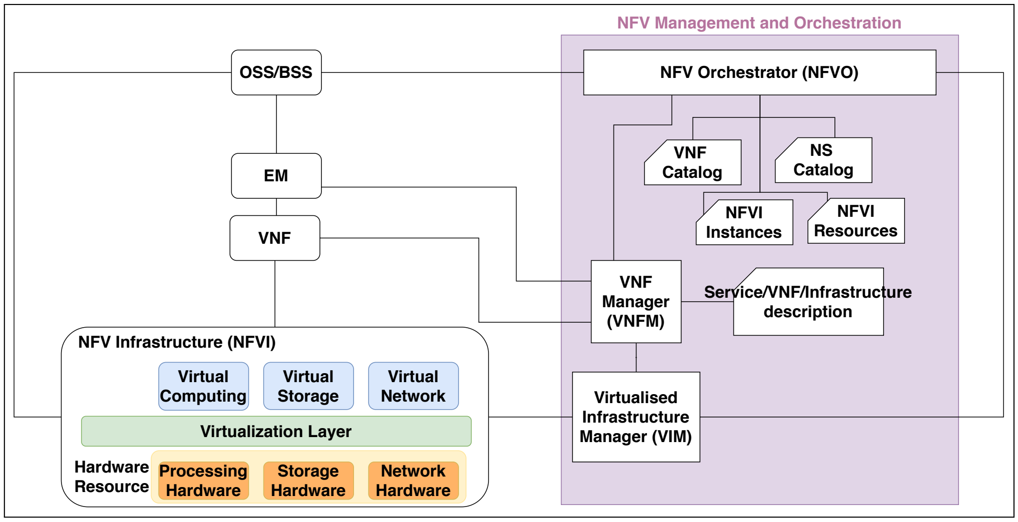


Fig. 2.5: ETSI MANO Reference Architecture

---

[3] `https://www.cisco.com/c/en/us/solutions/service-provider/network-functions-virtualization-nfv/index.html#~virtual-routers`

[4] `https://www.sdxcentral.com/articles/contributed/delivering-load-balancing-services-data-center-nfv-technology/2017/01/`

The “`ETSI-MANO`” specification is capable of providing differentiated networking service were dependent on pre-defined policy the traffic can enjoy multiple networking services. To enable this, “`ETSI-MANO`” facilitates dynamic deployment of NFVs over In-network processing where the traffic is “`steer`”-ed through an appropriate set of NFVs. This technique of distributing of fine-grained services and traffic steering through the NFVs is termed as Service Function Chaining (SFC). We define SFC as follows.

Definition 2.4 *SFC represents an ordered set of NFVs through which a particular traffic class needs to be steered.*

However, network management over NFV capable In-network processing suffers from manageability issues, mainly due to the following reasons.

Optimal utilization of resources: The differentiated network services implemented using NFV can not provide optimal services. For example, the optimal bandwidth aggregation mechanism of a multi-homed device requires the sharing of end-to-end path metrics to utilize the capacity of the interfaces fully. However, this currently not possible without the help of an external entity.

Monitoring and controlling issues: Deployment of a VNF for a particular type of network service requires knowledge of the traffic path. On the other hand, depending on the deployment location of the VNF, the traffic route may require adjustment to receive the services provided by the VNF. This close coupling between VNF placement and routing is presented as a joint optimization problem in existing literarture [92, 93, 94]. Traditional Simple Network Management Protocol (SNMP) driven network monitoring platforms are not suitable for providing fine-grained control over such infrastructure [95].

Programmatic traffic handling: Due to an increase in the number of users, deployment of a LSiN requires support for heterogeneous applications. These heterogeneous applications require different types of network customization is required for providing support to the applications. To scale the system and provide rapid deployment, programmatic handling of traffic is necessary [96].

Dynamic deployment of NFV: Implementation of SFC through the dynamic deployment of NFVs requires traffic profile-specific dynamic routing, which is difficult to attain in traditional destination centric routing mechanisms. Management of the traffic profile-specific dynamic path management requires a programmable network architecture.

To overcome these challenges NFV very often utilizes the SDN [97, 98] for management of network[5]. NFV and SDN are independent of each other, are considered as complementary technologies [6]. Basic concepts of SDN are described in the following section.

2.3 Software-Defined Network (SDN)

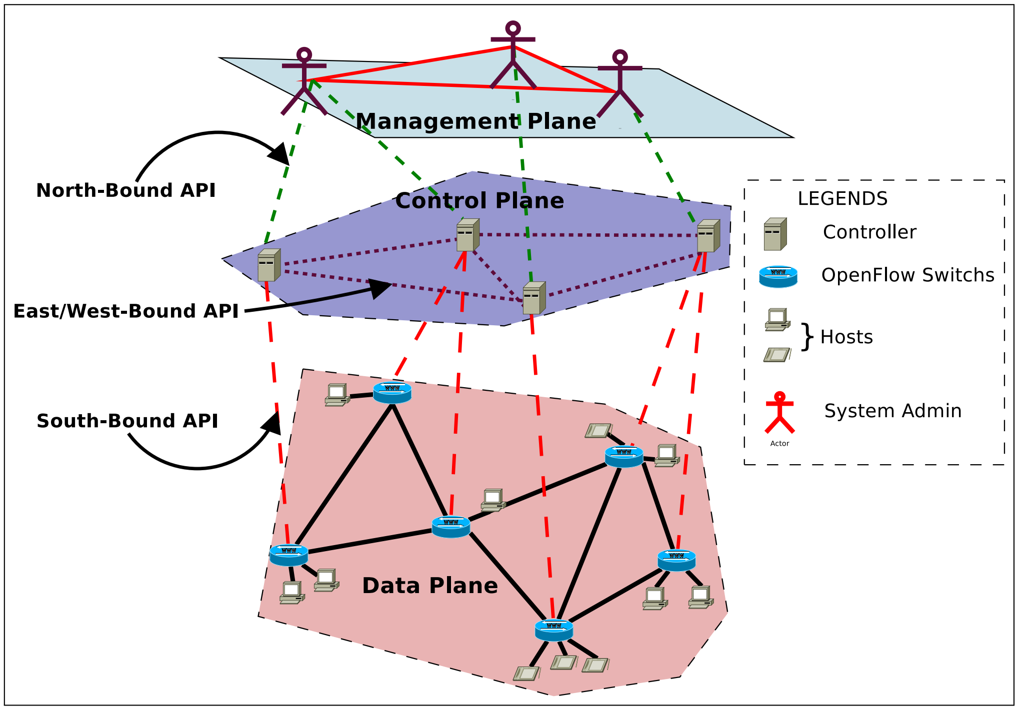


Fig. 2.6: SDN Architecture Overview

SDN emerged as an unified network management alternative to the traditional layer oriented network management techniques [6, 7]. SDN provides freedom from the traditional complex, static and vendor specific architectures [99, 100, 101] that are prone to mis-configuration [102, 103]. SDN decouples the network operation into multiple planes based on the objectives to simplify network management. SDN [15] is defined as follows.

---

[5] In fact we could not find any paper on NFV that does not employ SDN

[6] `https://datatracker.ietf.org/meeting/93/materials/slides-93-edu-openflow-9`

1. **Data and control plane separation:** Controlling overhead is removed from network devices. Each device acts as a forwarding element. Forwarding devices form "*data plane*". "*Control plane*" is defined as the part of the network which is responsible for signalling traffic and routing.
2. **Controller based decision:** Control logic is performed by a separate device called "*controller*". A controller provides a standard programming interface with the help of a logically centralized abstract network view.
3. **Flow-based decision:** Unlike traditional IP based networking SDN uses forwarding decisions are based on "*flow*"(s). A "*flow*" is broadly defined as a set of packets with similar packet header fields.
4. **Programmable network:** Functionality of the network can be programmed with the help of Network Operating System (NOS). It is task of the NOS to interact with underlying data plane devices.

SDN data plane consists of switches. Switches can communicate with one or multiple controllers. A typical SDN architecture looks like Fig. 2.6 [104] where controllers act as decision-makers and data plane executes the decision taken by controllers.

2.3.1 SDN Architecture

Each switch is capable of executing actions (like forward, drop, meter, change header, etc.) on each packet of a specific "*flow*". As per [104], data plane is composed of two sub-plane; (a) Forwarding plane and (b) Operational plane. The "*forwarding plane*" consists of a "*flow table*". Flow table contains a list of flow identifiers, and corresponding actions require to be applied to that particular flow. Whenever a packet enters a switch, the forwarding plane of the switch executes action(s) given in the flow table. On the other hand, "*operational plane*" keeps track of the device states, like available ports, port status, queues, etc. The device states are input to the "*control plane*". The control plane is connected to the forwarding plane via "*control plane south bound API*". Whenever a new flow enters the system (i.e., "`packet_in`" event), the forwarding plane consults the control plane since the flow-table does not contain any action regarding that particular flow. Similarly, the control plane intervention is requested in case of topology change. Control plane requires network states for identification of actions in case of

Table 2.2: Example of Existing Implemented SDN Components

Forwarding plane	ForCES [105], OpenFlow [106],Yang Model [107], SNMP MIBs [108], P4 [109], etc.
Operational plane	ARP [110], LLDP [111],
Control plane south bound API	OpenFlow [112]
Management plane south bound API	OF-CONFIG [113]
Control plane	RCP [11], Routeflow [114], SoftRouter [115] etc.
Management plane	OVSDB [116], NETCONF [117], SNMP [118], ForCES [119] etc.
NSAL	XML/JSON, RPC [120], REST [121], CORBA [122], NETCONF [117] etc.
East/West API	REST [121], XML/JSON, RPC [120], No-SQL [123] etc.

any such events are generated. The network state is a collection of the device states provided by Device Abstraction Layer (DAL). Therefore, device state change signifies network state change. Whenever the device state changes, the "*management plane*" is invoked by the operational plane through "*management plane south bound API*", as shown in Fig. 2.7. Corresponding change in the network state inside DAL required due to switch state change is handled by the management plane. Both management plane and control plane are connected to "*application plane*" via "*north bound API*". Application plane can provide interfaces to interact with the system administrators or policymakers through pictorial and/or programmatic interfaces. The application plane may require multiple services from the control plane and/or management plane to provide a uniform interface. Therefore, the north bound interface is managed through Network Service Abstraction Layer (NSAL). Table 2.2 provides some of the existing literature for each component of the SDN plane. To maintain scalability, SDN control plane can be implemented in a distributed fashion also [11, 124, 18]. To implement a distributed control plane, the control planes of different controllers must interact with each other. To ensure inter-controller communication, "*East/west bound API*" is used.

Based on the physical connection between data plane and control plane, the SDN control

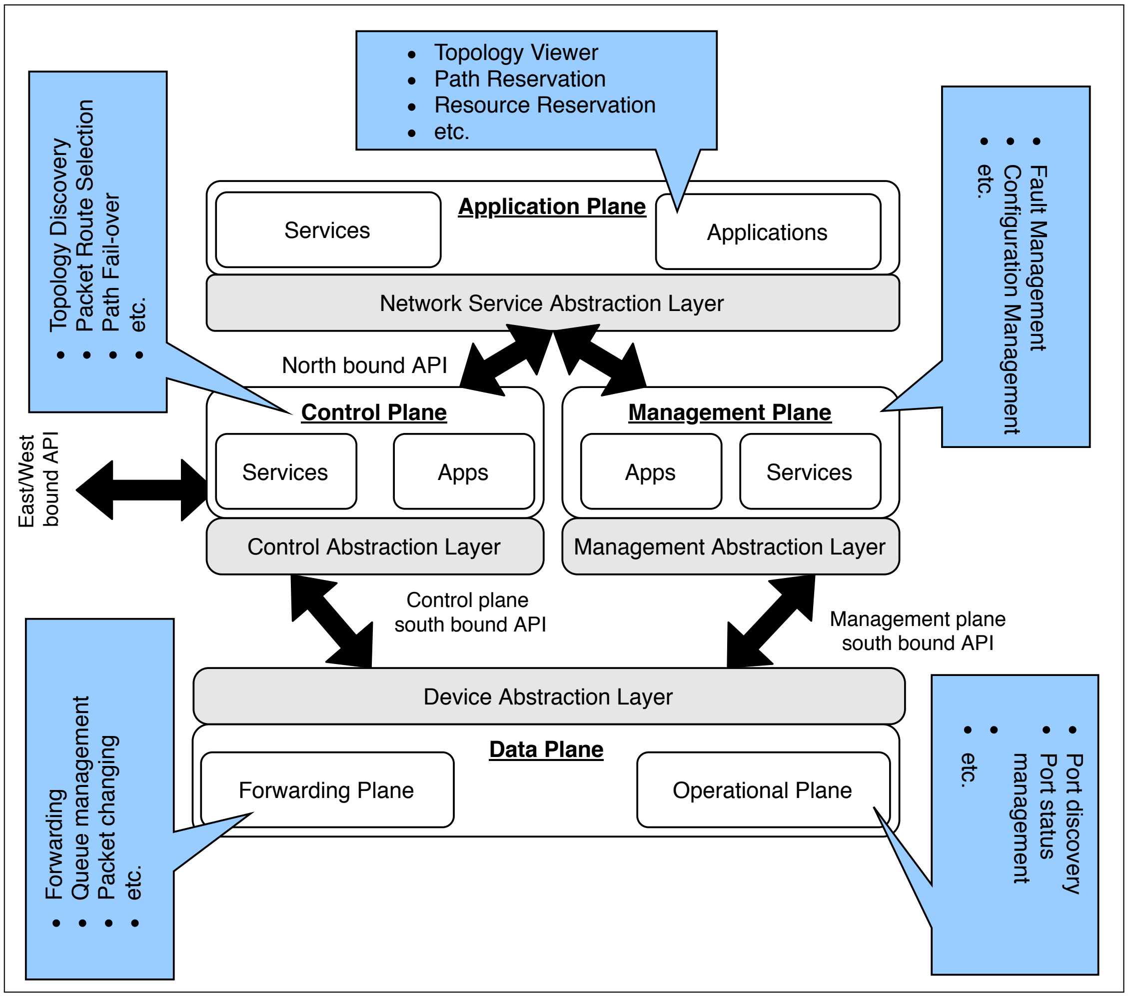


Fig. 2.7: SDN Architechture

plane can be either "*in-band*"[7] or "*out-of-band*"[8]. In case of "*in-band*" control plane, the controller/(s) are part of the network topology and uses same communication channels for sending control messages. On the other hand, "*out-of-band*" control plane uses dedicated control channels for communication with the controllers. Although "*out-of-band*" control plane provides quick initiation of flows, deployment of such plane incurs high "*capex*". Therefore, apart from "*DCN*", "*out-of-band*" controllers are rarely used. Use of same communication channel reduces the cost of deployment from "*in-band*" control planes, but in the presence of high data traffic, the

---

[7] `https://en.wikipedia.org/wiki/In-band_control`

[8] `https://en.wikipedia.org/wiki/Out-of-band_control`

control plane performance is largely affected [125, 126]. Additionally, the control plane failure probability also increases in such cases. LSiNs connects a huge amount of user equipment; thus, expected generated data traffic volume is reasonably high [3]. On the other hand, deployment of "*out-of-band*" control plane is practically infeasible for any large scale infrastructure due to added capex. In this thesis, our proposed solutions are primarily targeted towards the "*in-band*" control plane where even single failure can jeopardize the operation of the entire network. Whereas due to relative reduction in capex in-band control planes are the primary choice for LSiN. Irrespective of the control plane choices, one of the primary concerns of SDN is deployment. Since SDN is not compatible with legacy devices and requires SDN supported hardware, deployment of SDN over an existing large scale platform has always been an issue.

2.3.2 Deployment of Data plane

Based on the internal architecture, data plane devices/switches are categorized into two broad classes as follows.

Hardware Switch: Eventhough "*Hardware switch*"es are costly (increases capex), can deliver faster packet processing performance by utilizing Ternary Content-Addressable Memory (TCAM). To reduce cost of deployment capex `Panopticon` [32, 31] presents a network hypervisor to enable SDN over legacy devices by deploying hardware switches in the strategic location in the network topology. Hong *et.al.* [127] have proposed identification of such strategic loactions to minimize the capex and overall link utilization. This deployment of SDN switches in the non-SDN legecy platform is termed as "*hybrid SDN*" [128]. However, `WPE` [129] have shown that, incremental deployment of SDN using "*hybrid SDN*" can comprise secure operation of network. To increase the SDN data plane coverage while maintaining capex is the objective of low cost hardware switch `Zodiac` [130],`Pica8` [131]. In order to increase performance of the data plane devices, several optimization techniques have been proposed as listed in Table 2.3. The most critical issue with these low-cost hardware switches is that they do not configure all "*`OpenFlow`*" supported parameters (e.g., metering). Whereas, Goto *et.al.* [135] have shown a significant increase in packet loss probability and packet processing delay based on a queuing theory-based model if the packet arrival rate increases.

Software Switch: To overcome the capex problem of hardware switches, data planes are im-

Table 2.3: Optimizing Techniques for SDN Data Plane Devices/Switches

Existing Works	Key Contributions
[132]	Proactive flow installation to avoid race conditions in asynchronous in-band control based on RTT estimation
`FastRule` [133]	Find flow table entry position in "*`TCAM`*" to minimize the update time using binary indexed tree
[134]	Selection of aggregated flows and pre-installation of wild carded flow rule to reduce controller response time

plemented using software which can multiplex capabilities of the existing general-purpose hardware. The key aspects of some of the existing software switches are presented in Table 2.4. To exploit the reduction of capex, "*`Swift`*" [140] proposed ways to enable SDN control in commodity wifi "*`AP`*" by using a separate Open virtual switch (OVS). Although the proposed approach employs a non-evasive technique, `Swift` increases communication delay and controller communication overhead. In[141], the authors have deployed an OVS driven framework over COTS devices to enable seamless vertical handover. Their work

Table 2.4: Existing Software Switches

Existing Works	Key Contributions
`Lagopus` [136]	Intel DPDK [137] based OpenFlow1.3 [106] switch, high-performance for 10Gbps bandwidth interface, Use of polling based design instead of interrupt driven architecture, implements lock free queues, batch packet processing, carrier grade
`Bofus` [138]	Simplistic design suitable for prototyping, single threaded event handling, separated packet parsing, role based multi-controller set-up
`P4` [109]	Programmable data plane, removes header fields and protocol dependence, uses "*P4*" language for dynamic packet parsing,
`OVS` [139]	very popular, used both in for commercial and test-beds, support for wide varaity operating systems, user-space implementation, event driven architecture

has implemented a communication mechanism between the end host and the control plane to improve the end host performance. This technique also shows that the evolution of data plane deployment also requires an evolution in the control plane.

2.3.3 Deployment of Control plane

In order to solve incremental deployment of SDN `Fibbing` [142] proposed injection of fake "*`LSA`*" packets to realize SDN control plane on top of a traditional network platform. `SNDp` [143] have proposed deployment of SDN supported data plane devices in border nodes of the network domains to deploy SDN incrementally. However, all these approaches can not ensure fine-grained flow routing and monitoring. `SCMon` [144] have proposed a NFV oriented path monitoring mechanism.

Deployment of full SDN architecture can be useful for attaining fine-grained control but increases centralized control plane overhead. To avoid the controller becoming a bottleneck, decentralized control planes have been proposed in [152] where multiple controllers are deployed to handle issues related to the control plane. Decentralized control plane provides fault resilience in the control plane either by maintaining controller replica instances (e.g. `ONOS` [18], `Ravana` [145], `B4` [149])or by partitioning the network into multiple sub-networks (e.g. `Onix` [44], `Hyperflow` [146], `Kandoo` [124] etc.). Based on the role distribution, decentralized SDN control planes are categorized into two categories; (a) distributed SDN and (b) hierarchical control plane. Distributed control planes partition the network into several sub-networks, and the switches of the same sub-networks are allowed to be controlled by a controller. The control plane decisions involving multiple sub-networks are taken distributively by the controllers. In such a scenario, the controller directly connected to a switch is called "*local controller*" of that switch, where rest of the controllers are called "*remote controllers*" of that particular switch. Similarly, all switches connected directly to a controller are called "*local switches*", and rest of the switches are denoted as "*remote switches*". Although distributed control planes are highly fault-resilient and scalable, the distributed decision making process requires consensus and state consistency preservation among the controllers, which increase control plane overhead. To reduce the control plane overhead, hierarchical controllers segregate the control plane information in order of access frequency. Where management of frequently access/required data (e.g., topology, metering, local routing policy, etc.) are responsibilities of a "*leaf controller*", management of less frequent information (e.g., global routing policy, quality of service (QoS), etc.) are managed by the "*upper layer controllers*". Key aspects of some of the popular existing decentralized SDN control planes are listed in Table 2.5. However, most of the existing control plane designs are primarily targeted for DCN and/or Internet Service Provider Network (ISPN); they are not suitable for dynamic networks like LSiN.

2.4 SDN Controlled LSiN

Since, existing SDN control planes are not sufficient for LSiN, innovations are required for network management in LSiN. In this section, we discuss some of the existing works where the interplay between SDN and LSiN has been exploited to improve end-user performance. We segregate this discussion into two aspects; (a) Deployment of SDN exploiting the capabilities of LSiN, and (b) Use of SDN for network control of LSiN deployment.

2.4.1 SDN Deployment Using LSN

Although the deployment of SDN has been discussed in the previous section, in this section, we want to address some of the approaches of SDN deployment over LSiN. The significant difference in SDN deployment in managed networks (e.g., service provider networks, DCN) and LSiN is the deployment of SDN of LSiN is difficult due to the use of heterogeneous resource-constrained devices and non-standard topologies. Therefore, very few of the existing works (given in Table 2.6) can exploit the capabilities of LSiN. The exploitation of native NFV support of LSiN can ease the deployment of SDN. Both the control plane and data plane of SDN can take advantage of NFV to reduce the capex/opex of softwarizing network. However, executing NFVs over resource constraint LSiN devices requires failure management. On the other hand, existing replica based fault resilience used in SDN control plane is not sufficient in case of LSiN where there exists only a limited number of auxiliary paths between a switch and its controllers. Finally, the network management of LSiN requires "*in-band*" control plane due to the lack of a dedicated path between controllers and switches. Therefore, the data plane performance hugely impacts the performance of the control plane. This problem alleviates the short flow heavy traffic pattern of LSiN, where the control plane flow initiation delay significantly affects the end-to-end performance of the application.

2.4.2 LSiN Control Using SDN

As the size and users of LSiN increase, variety of traffic also increases in LSiN. Management of diversified traffic requires fine-grained control, which SDN promises. Although, there are multiple existing SDN control planes to ensure security [157] for LSiN. `DPIDIt` [158] proposes the use of a covert timing channel to ensure secure communication between switch-controllers when there exist malicious switches. `CENSOR` [159] provides a flow event monitoring to ensure security and communication reliability of SDN. Although SDN can provide security, management of LSiN using SDN is very difficult when the network is dynamic [160]. Whereas, the conversion of a traditional network into a SDN enabled network is challenging [142]. `SIMECA` [161] proposes a lightweight data and control plane deployment for "*5G*" enabled LSiN. However, the problem of controlling a dynamic network still a challenge in LSiN network management. In a dynamic network, devices can join and/or leave the eco-system rapidly. The dynamic network is prone to failure; therefore, providing fault-tolerance and partition tolerance is one of the primary objectives of SDN control plane. Further, ensuring only a little involvement of the system administrator can reduce the joining overhead of a device, which can significantly help in the auto-scaling of LSiN.

Apart from the scalability issues, the network management becomes even difficult in presence of SFC. `ElasticSFC` [162] and `EvoVNFP` [163] proposed VNF deployment and traffic steering through VNFs over a centralized SDN. A few of of the existing methods are given in Table 2.8. However, SFC management becomes difficult in presence of header modifying VNF. An detailed analysis regarding the deficiencies of existing SFC management is presented in Section 6.2.

2.5 Summary

This chapter presented a brief overview of the LSiN, NFV/VNF, and SDN architecture and technologies, which are used in this thesis. We also describe some of the problems related to our thesis in this chapter. Our primary contribution lies in exploiting these seemingly different architectures to help each other ensure end-user performance. Our contributions start in the next chapter.

Table 2.5: Popular Distributed SDN Control Plane

Architecture	Existing Works	Key Contributions
Distributed	`Onix` [44]	Distributed NIB
	`ONOS` [18]	Consistent network state in local ONOS instance, state information exchange to maintain global consistency
	`Ravana` [145]	Introduces master and slave controller architecture to provide fault-resilience from controller crash events, slave controllers maintain replicated copies of their respective master controller, can not handle an arbitrary number of failures.
	`Hyperflow` [146]	Local controller decides flow actions for the switches attached to them, publish-subscribe framework for instructions to remote switches and state consistency among the controllers.
	`Elasticon` [147]	Provides scaling of control plane by ensuring load balancing to the controller, novel consistency preserving barrier based hand-off mechanism for dynamic change of controller-switch association
	`TOPSIS` [148]	Proposes an ILP for path finding in case of link failure, proposed ILP finds the least energy consuming path, Implements periodic path searching for minimization of failure probability and energy consideration
Hierarchical	`Kandoo` [124]	2-layer hierarchical design, store frequent event in "*leaf controller*", less frequent informations are kept in "*upper layer controllers*".
	`B4` [149]	Each Wide area network (WAN) site is managed by a "*leaf controller*", "*root controller*" provides traffic engineering services, "*leaf controller*"(s) are connected to each other by gateways
	`CuttleFish` [150]	Identification of frequently accessed state information and dynamic off-loading of states
	`HiDCoP` [151]	3-layer hierarchical design, "*middle layer*" is used for managing communication between "*leaf controllers*" through gateway management, uses master-slave controller fail-over mechanism

Table 2.6: Deployment of SDN using LSiN

Existing Works	Key Contributions
[153]	distributed SDN over Information centric network (ICN)
`MEC-SDN` [154]	Proposed an hierarchical SDN control plane where the "*leaf controllers*" are executing in the mobile edge devices
`SDN-IoT` [155]	Proposed NFV based architecture for deployment to realize SDN enabled gateways
"*`Barista`*" [98]	distributed SDN controller brewing framework, distributed and selective event handlers for each component by using NFV
`BLAC` [156]	Controller technology agnostic load balancing framework

Table 2.7: Management of LSiN using SDN

Existing Works	Key Contributions
`DPIDIt` [158]	State model for switch-controller association and vulnerabilities, proposes use of covert timing channel for switch-controller communication
`CENSOR` [159]	Proposes security and communication reliability over SDN enabled IoT
[160]	Experimentally compared performance of topology discovery mechanisms of existing SDN control planes over a dynamic LSiN, they have found that, change of topology events introduces huge amount of jitter in the end-user application
`Fibbing` [142]	Partial SDN control over traditional network devices, lacks fine-grained control
`SIMECA` [161]	Proposes a lightweight data and control plane for "*`5G`*" enabled edge cloud

Table 2.8: Management of LSiN using SDN under SFC

Existing Works	Key Contributions
`ElasticSFC` [162]	VNF deployment and bandwidth allocation to enable auto scaling using centralized SDN over single administrative domain.
`EvoVNFP` [163]	Taboo search for VNF placement, single controller, single domain
[164]	Multi objective optimization based on link load and communication delay, heuristic for single domain
`BFPR` [165]	centralized SDN, Datacenter network, Bloom filter to encode path of each packet for path tracing, in-network packet header modification,
[166]	Distributed path optimization over multiple administrative domain, Service provisioning by utilizing domain to node abstraction

Chapter 3

SDN-MPTCP: MPTCP Sub-Flow Management Over large scale IoT network (LSiN)

3.1 Introduction

Modern day devices are usually equipped with multiple hardware interfaces that can be leveraged to satisfy the demand for increasing traffic by aggregating the available bandwidth at all interfaces. Multipath TCP (MPTCP) [72] has been proposed in the literature as an end-to-end protocol for data-center and enterprise networks with the availability of multi-interface networking devices, which provides the support for bandwidth aggregation via concurrent usage of different interfaces by creating multiple sub-sockets. MPTCP initiates multiple sub-sockets via different interfaces to aggregate the bandwidth.

Has been published in

[T.1] Subhrendu Chattopadhyay, Sukumar Nandi, Samar Shailendra, and Sandip Chakraborty. "Primary Path Effect in Multi-Path TCP: How Serious Is It for Deployment Consideration?" In: *Eigthteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc).* 2017, p. 36

[T.2] Subhrendu Chattopadhyay, Samar Shailendra, Sukumar Nandi, and Sandip Chakraborty. "Improving MPTCP Performance by Enabling Sub-Flow Selection over a SDN Supported Network". In: *Fourteenth International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob).* 2018

The current Linux kernel implementation of MPTCP [73] consists of three major modules: *Path Manager*, *Segment Scheduler*, and *Congestion Control Mechanism.* The *Path Manager* module manages the available sub-flows between the end hosts. Currently, MPTCP has proposed two choices of path manager. **(a) Full-mesh path manager** creates sub-sockets for between all available pair of interfaces. **(b) ndiffports** selects k sub-flows among all available sub-flows, where k is a user defined parameter. MPTCP *Congestion Control* module manages congestion window for each sub-flow separately. Several congestion control algorithms like Linked Increase Algorithm(Linked Increase Algorithm (LIA)) [75], Opportunistic Linked Increase Algorithm(Opportunistic Linked Increase Algorithm (OLIA)) [27], Balanced Linked Increase Algorithm(Balanced Linked Increase Algorithm (BALIA)) [64] etc. [74, 167] have been proposed for MPTCP. Once the congestion window size for each path is decided, segment scheduler takes responsibility of scheduling segments for the individual sub-flows. *Round-Robin* and "*lowest RTT First*" are the two available segment scheduling strategies described in the MPTCP standard.

The primary task of a segment scheduler is to reduce out of order packets at the receiver. However, in a network, path characteristics (such as bandwidth, delay, loss rate, jitter etc.) of the underlying sub-flows can be significantly different as well as time varying. Differences in end-to-end path characteristics of each active sub-flow may lead to an increase in out of order segments delivered at the receiver. Li *et.al.* [168] have tried to limit receiver buffer in order to decrease out of order segment delivery by employing network coding. However, it has been found that, their implementation violate MPTCP basic principle of do no harm objective [169]. Whereas, Guo *et.al.* [170] and Lim *et.al.* [171] focuses on the segment scheduling mechanism to avoid out of order segment generation. However, segment scheduling alone can not reduce out of order delivery and may lead to Head of Line (HOL) blocking at the receiver side [25]. HOL blocking increases delays and packet drops. So, number of spurious retransmission also increases. Therefore, Cao *et.al.* [25] proposes a receiver buffer aware path selection mechanism. However, like most of the transport layer protocols, their implementation uses Round Trip Time (RTT) as a measure of path characteristics. In case of MPTCP, one segment and its acknowledgment might follow different paths. So, RTT is not a faithful estimate of a path at sender side. Therefore, relying on simple RTT driven path characteristics leads to severe performance degradation in MPTCP performance. In this work, we provide a short experimental study to show that, MPTCP provides near-optimal experience, when the active sub-flows have similar path characteristics, as in those cases, RTT provides a good estimation. However, the difference in delay, effective bandwidth, and loss rate can significantly increase the number of out of order segments at the receiver [172]. Therefore, we argue instead of relying on the RTT, MPTCP must rely on end to end path characteristics. Based on the end to end semantics, MPTCP path management module must choose a set of sub-flows which can avoid HOL blocking by reducing out of order delivery [173].

Therefore, we propose `SDN-MPTCP`, a Software-Defined Network (SDN) [15] aided intelligent dynamic path management scheme. SDN provides a logically centralized view of network topology parameters to the application protocols by periodically obtaining statistics from all its data plane devices [10]. This makes it feasible to optimize end-to-end performance of MPTCP by selecting a suitable active set of MPTCP sub-flows. We consider a SDN controlled LSiN, where network switches are connected with a SDN controller that can estimate sub-flow characteristics based on end-to-end path properties. As SDN cannot obtain information about receiver buffer evolution as well as prediction of aggregated MPTCP throughput based on the sub-flow properties, building up a SDN aided path manager application is non-trivial. Therefore, in this work, we propose an estimation mechanism to predict the MPTCP aggregated throughput for a set of sub-flows with their end-to-end path characteristics (latency, available bandwidth, etc.). Unlike prior works our proposed model provides aggregated throughput for a given set of sub-flows. Consequently, we take a two-stage approach in this work. We formulate an irreducible and aperiodic Discrete Time Markov Chain (DTMC) to model the aggregated throughput prediction of a MPTCP flow with the end-to-end path characteristics of a given set of sub-flows (Section 3.5). Based on the predicted throughput from the estimator model, we develop an optimization framework to find out the optimal set of sub-flows that can maximize the aggregated throughput for a given MPTCP flow (Section 3.6). The SDN controller executes this optimization framework and schedules the sub-flows accordingly. Finally, we evaluate the performance of the proposed mechanism and compare it with various baselines.

3.2 Related Works

Initial design and development of MPTCP targets aggregation of bandwidth via multiple interfaces[174]. Despite of having significant potential, deployment of MPTCP is not very popular till date, apart from some available SDN enabled research testbeds [175, 176]. Mehani *et.al.* [177] have found that only 0.1% of their gathered service domains uses MPTCP end points. According to them, two major reasons behind low adaptation of MPTCP are, (a) unreliable performance, and (b) network management overheads. Performance improvement of MPTCP by employing congestion control techniques are discussed in [75, 76, 27]. Peng *et.al.* [26] have shown that performance of MPTCP is guided by a trade-off between TCP friendliness and responsiveness towards network changes. They have proposed a congestion control technique (named "BALIA") which can balance this trade-off criteria. Ou *et.al.* [77] have considered to tackle HOL blocking and proposed a joint congestion control and segment scheduling mechanism. In another work [28], they have proposed a segment scheduling mechanism to avoid HOL blocking. However, the existing segment schedulers uses RTT based approach to estimate receiver buffer size which is not a good estimate for a lossy and dynamic network. Moreover, a segment and its acknowledgement might follow different path. Therefore, segment scheduling does not perform well in case of dynamic network.

3.3 Preliminary Experiments

In order to show the insufficiency of the RTT driven path management we conduct some pilot study experiments. Our experimental setup is built over "*Mininet*" network emulator platform where two hosts are connected via two parallel paths, Path A and Path B, through two different interfaces of each host. The end hosts as well as all the network switches in the path use Linux based operating system, and we use Linux tool `iperf` to generate the network traffic. The Linux kernel at the hosts are configured with `MPTCP` V0.90[1]. All experiments are carried out for two cases – one by selecting Path A as the primary sub-flow and the other with Path B as the primary sub-flow. We define a path as a primary sub-flow if the connection is initiated through that particular path. The bandwidth, delay and loss rate for Path B is kept constant at 10 Mbps, 15 ms and 0%, respectively. We perform three experiments by varying the parameters {bandwidth, delay, loss rate} for Path A – (a) **Exp 1 (delay difference)**: $\{10Mbps, 250ms, 0\%\}$, (b) **Exp 2 (bandwidth difference)**: $\{5Mbps, 15ms, 0\%\}$, and (c) **Exp 3 (loss rate difference)**: $\{10Mbps, 15ms, .5\%\}$.

---

[1] https://www.multipath-tcp.org

3.3.1 Effect on Transport Layer Throughput

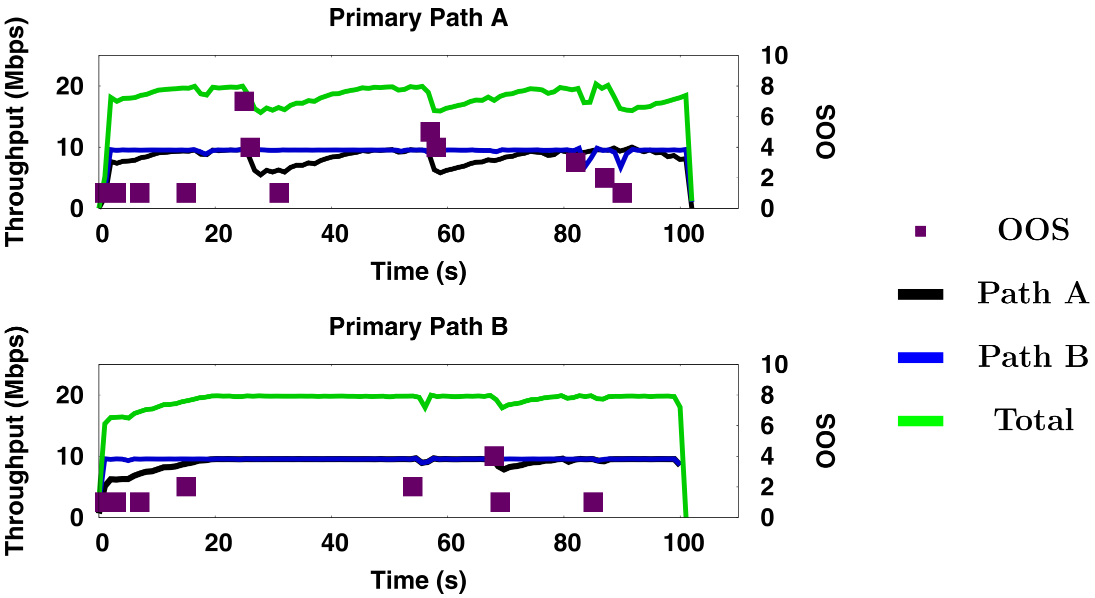


Fig. 3.1: Effect of Delay (Exp 1)

Fig. 3.1 represents the difference in throughput when there exists a significant delay difference between the primary sub-flow and secondary sub-flow. The overall throughput reduces significantly in case of Primary Path A (slower path). The results also show that, the number of Out of order segments (OOS) generated are significantly higher in case of slower primary sub-flow (i.e. Path A). MPTCP retransmission due to OOS is handled by resending the segments via the same path for two times, and after that, the segment is assigned to a different sub-flow. A retransmission due to three duplicate acknowledgement affects severely in case when the primary sub-flow is Path A. The primary sub-flow takes longer time to converge to the highest attainable bandwidth due to higher RTT. Retransmission in primary sub-flow due to OOS further worsen the convergence. This phenomenon is absent in the case when the primary sub-flow is Path B, as generation of OOS can be quickly mitigated by retransmitting the segment via that sub-flow due to less end-to-end delay. Due to this behavior, the selection of primary sub-flow is more significant in case of delay deference, even for longer flows. Fig. 3.2 shows the impact of flow duration on the MPTCP performance for the three experimental scenarios. As flow duration increases, delay difference has more impact on the throughput difference based on the primary

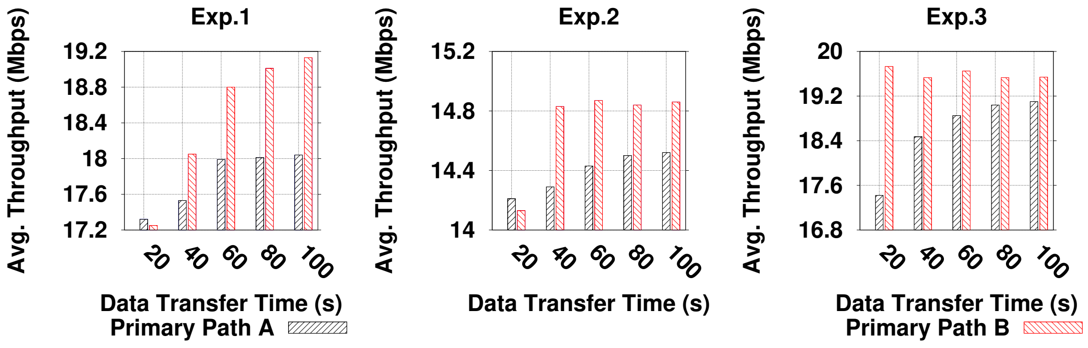


Fig. 3.2: Effect of Flow Duration

sub-flow selection.

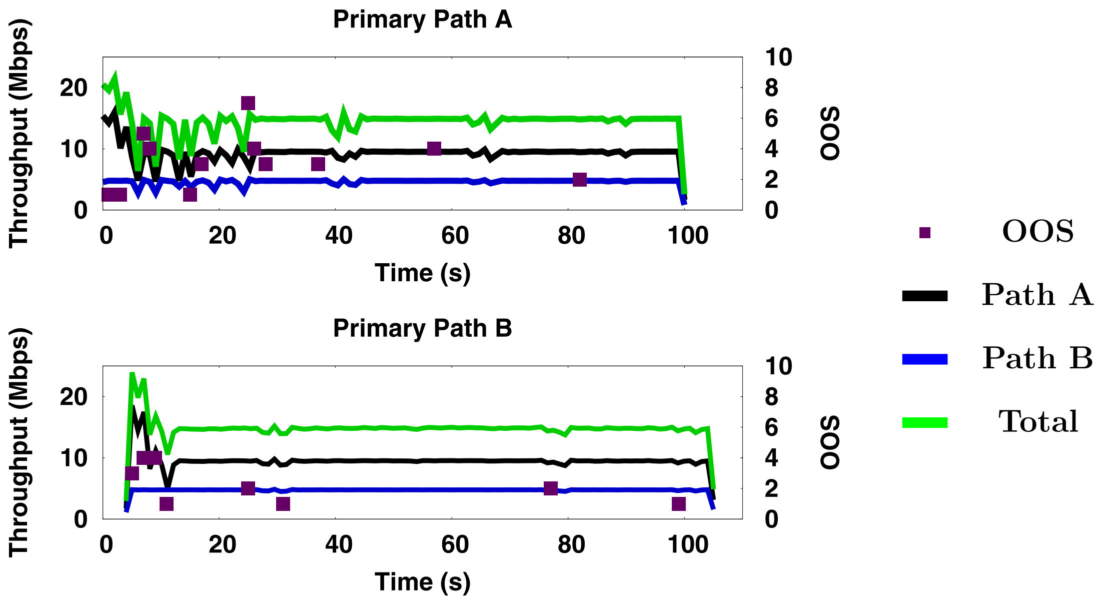


Fig. 3.3: Effect of Bandwidth (Exp 2)

In case of a bandwidth difference in between the primary and the secondary sub-flow, we see a similar effect (Fig. 3.3). This result also suggests that choosing a higher bandwidth path as the primary sub-flow can significantly reduce the generation of OOS. Fig. 3.4 shows the results when primary sub-flow has higher loss rate than that of secondary sub-flow. The results suggests that, choosing a high loss rate path as the primary sub-flow can reduce the overall

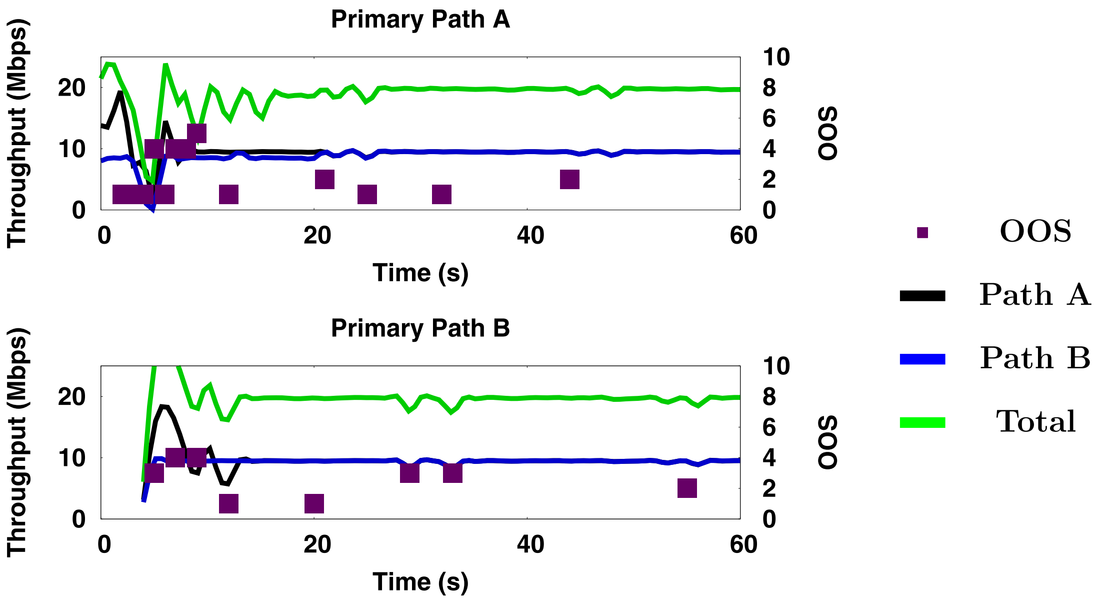


Fig. 3.4: Effect of Loss Rate (Exp 3)

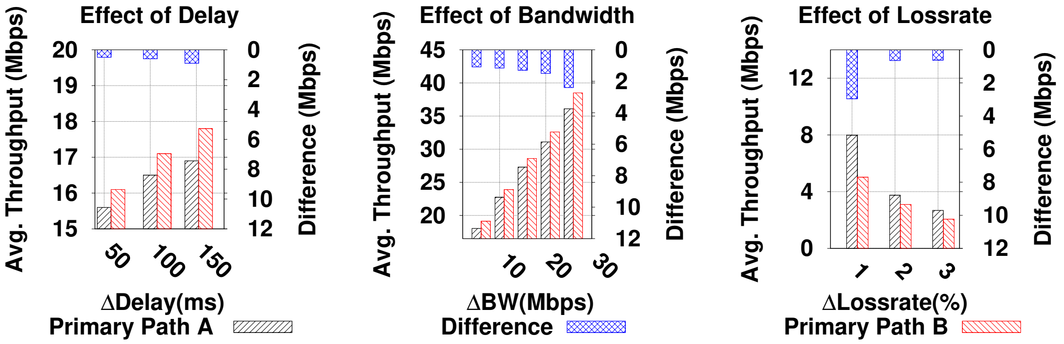


Fig. 3.5: Throughput variation

throughput. Further from Fig. 3.2, we observe that the impacts of bandwidth difference and loss rate difference are more in case of short duration flows.

3.3.2 Impact of Parametric Difference Between Two Paths

Next we analyze how the delay, bandwidth and throughput difference between the two paths impact the MPTCP throughput based on primary sub-flow selection. Fig. 3.5 represents the change in aggregated throughput with the change in delay, bandwidth and loss rate difference between the two paths. From these figures, it is clear that the increase in variability between the two paths have significant impact on the throughput performance based on the selection of primary sub-flow. The impact is significant under certain cases. For example, in case of loss rate (Exp 3), selecting a low loss rate path as the primary sub-flow can improve the MPTCP throughput as high as 60%, as we can observe from Fig. 3.5.

3.3.3 Summary of Observations

From the experimental results, we observe that a primary sub-flow with lower bandwidth, higher delay and/or lower loss rate gives rise to OOS at the receiver (Fig. 3.1, Fig. 3.3 and Fig. 3.4). Such an increase in OOS increases the number of "*triple-duplicate ACK*"s at the sender causing the reduction in the congestion window. This unduly reduction in congestion windows adversely affects the aggregated throughput of the network. The main reason behind the increase in OOS is, traditional RTT based congestion control algorithms are not suitable for disparate path characteristics between sender and the receiver.

It can also be observed from Fig. 3.2 that the effect of delay disparity between paths is more detrimental than the effect of bandwidth disparity between paths. This is due to the very nature of congestion control algorithm and its direct dependence over the RTT. Moreover, the effect of sub-flow selection is visible for both the short flows as well as the long flows. In the next section we provide a formal model to capture this sub-flow selection procedure.

3.4 Network and System Model

The objective of this work is to develop a solution for dynamic sub-flow management while considering end-to-end path characteristics. The problem is to identify a set of sub-flows from all available paths between a source-destination pair of a MPTCP flow, such that (i) the overall MPTCP aggregated throughput is maximized, and (ii) the receiver buffer size is always limited by a certain threshold to avoid HOL blocking problem. However, obtaining sub-flow characteristics (like receiver buffer evolution) under a dynamic scenario is non-trivial over a complete distributed network management framework, and therefore we leverage on SDN based network management concept in this work. We consider a centralized SDN controlled LSiN, where the network switches are connected with a SDN controller. The controller can estimate sub-flow characteristics based on end-to-end path properties.

Although a SDN controller can monitor end-to-end path characteristics like latency and available bandwidth, obtaining the information about receiver buffer evolution as well as the prediction of aggregated MPTCP throughput based on the sub-flow properties are non-trivial. Therefore, we need to build up an estimation mechanism to predict MPTCP aggregated throughput for a set of sub-flows with their end-to-end path characteristics (latency, available bandwidth etc.). To the best of our knowledge, prior works on MPTCP do not model aggregated throughput for a given set of sub-flows. Consequently, we take two-stage approach in this work as follows.

1. We formulate an irreducible and aperiodic DTMC to model aggregated throughput prediction of a MPTCP flow with end-to-end path characteristics of a given set of sub-flows (Section 3.5).

2. Based on predicted throughput from the estimator model, we develop an optimization framework to find out an optimal set of sub-flows that can maximize aggregated throughput for a given MPTCP flow (Section 3.6). The SDN controller executes this optimization framework and schedules the sub-flows accordingly.

3.4.1 Network and System Model

We assume a network as a undirected graph $G = \{V, E\}$, where vertices represent network switches and hosts, and edges represent physical connectivity between them. Let $\mathcal{S}$ be the set of all node disjoint sub-flows between a pair of sender-receiver. A MPTCP flow is a collection of sub-flows; therefore, we represent $\mathcal{S} = \{S_1, S_2 \ldots S_n\}$, where S_k represents k^{th} sub-flow, and n represents the total number of node-disjoint sub-flows between the corresponding sender-receiver pair. A sub-flow $S_k = \{v_1^k, v_2^k \ldots v_{n_k}^k\}$ is equivalent to an ordered set of vertices, where each $v_i^k \in V$ such that $v_1^k, v_2^k \ldots v_{n_k}^k$ forms a path of hop count of n_k between the sender-receiver pair. As a consequence, we use the terms "path" and "sub-flow" interchangeably, where "sub-flow" represents a MPTCP connection whereas "path" indicates the underlying network path between the sender-receiver pair. Let $e_{ij}^k \in E$ denotes an edge between two nodes v_i^k and v_j^k. Let B_{ij}^k and L_{ij}^k represent bandwidth and loss rate of e_{ij}^k. We further assume that propagation and queueing delay of e_{ij}^k follow independent normal distribution with mean D_{ij}^k and standard deviation Θ_{ij}^k.

We consider that end-to-end path characteristics of S_i can be represented by following three tuples: $q_i = \{b_i, Pr_i(X = r), l_i\}$, where b_i and l_i represent bandwidth and segment loss probability of S_i, respectively. Note that in this section, we use terms "*packet*" and "*segment*" interchangeably. $Pr_i(X = r)$ represents probability mass function (pmf) for RTT of S_i being r. For the sake of simplicity we assume that, $\forall i : Pr_i(X = r)$ follows independent truncated normal distribution with mean μ_i and standard deviation σ_i. Therefore, $Pr_i(X = r) = \Psi(\mu_i, \sigma_i, 0, \infty; X = r)$ where $\Psi(X = r; \mu, \sigma, a, b)$ represents cumulative probability density function of a random variable X having mean as μ and standard deviation σ such that $\forall X : a \leq X \leq b$. By using addition rule of normal distribution, we get $\mu_i \approx 2\sum\limits_{jk}(D^i_{j,k})$ and $\sigma_i^2 \approx 2\sum\limits_{j,k}(\Theta^i_{jk})^2$. However, we use notation $Q_i = \{b_i, l_i, \mu_i, \sigma_i\}$ for easy representation. Here, Q_i signifies path characteristics of S_i. For ease of representation we use $\vec{Q} = \{Q_i\}$.

Each sub-flow maintains a separate congestion window. The size of congestion window of S_i at time t is represented as $w_i(t)$. We use $\mathcal{T}$ and $\mathcal{R}$ to signify steady state throughput and receiver buffer size of MPTCP connection between intended sender-receiver pair. Our objective is to estimate value of $\mathcal{T}$ and $\mathcal{R}$ in terms of Q_i that represents the underlying path characteristics of the MPTCP sub-flows for a sender-receiver pair.

3.5 Impact of MPTCP Sub-flow Selection on Throughput Performance – An Estimation Model

In this section, we use an irreducible and aperiodic DTMC model to estimate values of aggregated steady state throughput ($\mathcal{T}$) and receiver buffer size ($\mathcal{R}$) based on path characteristics q_i. Considering n number of possible sub-flows $\{S_1, S_2, ..., S_n)$, states of a MPTCP flow can be represented through the congestion window across those n different sub-flows. Therefore, a state in the system can be represented as $\{w_1, w_2, ..., w_n\}$ where w_i is the congestion window value of the sub-flow S_i. We make the following assumptions,

- The change in congestion window of a path is triggered based on a discrete event system by observing corresponding RTT of the underlying path.
- The congestion window updates at different paths are mutually independent and identically distributed.

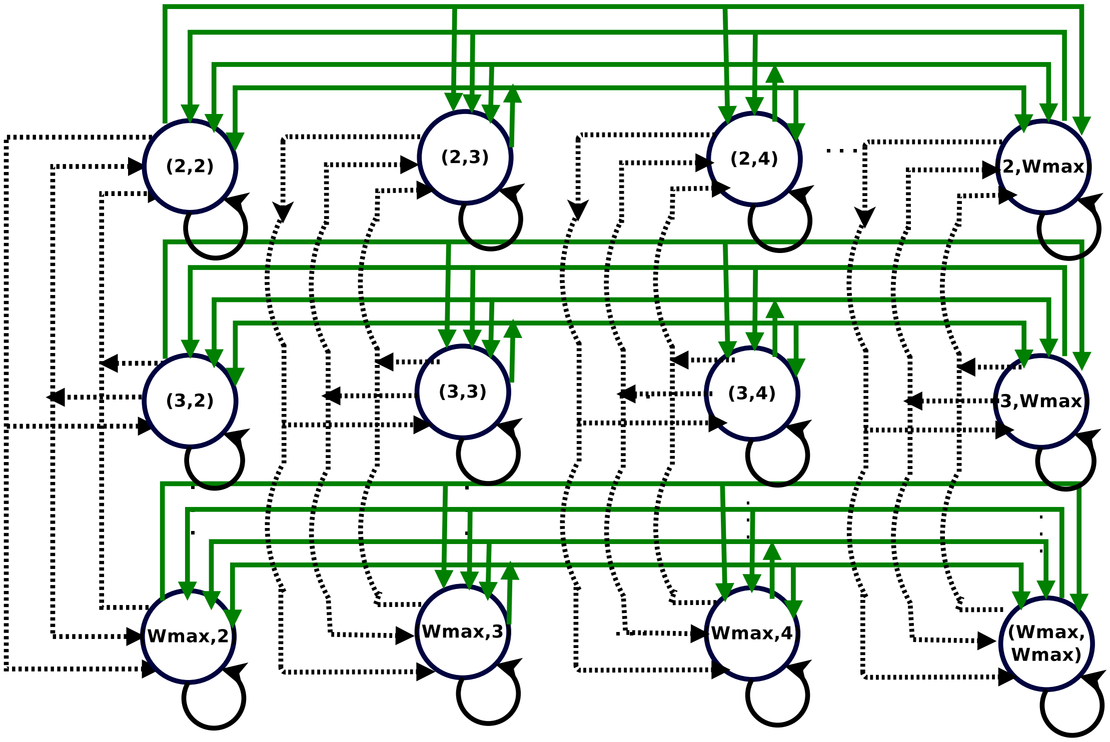


Fig. 3.6: Markov Model for a MPTCP with 2 Sub-Flows

Therefore, congestion window evolution of i^{th} sub-flow of a MPTCP flow can be represented as a stochastic process $w_i(t)$. Accordingly, we develop a n dimensional irreducible and aperiodic discrete time "*Markov*" model, where a state of the system is represented by n-tuple $\{w_1, w_2, ..., w_n\}$, where each $w_i \in [2, W_{max}]$, W_{max} being the maximum congestion widow value. An example DTMC for $n = 2$ is shown in Fig. 3.6. The state transition is allowed when a segment is either received successfully or it is lost in the transmission. Transition triggering events are handled on a sub-flow level. Therefore, we assume that state transition triggered by sub-flow k alters only k^{th}-element of the state variable. We term this property of our model as "*single path transition*" property. Let us denote the state transition probability from state $(w_1, \cdots, w_r, \cdots, w_n)$ to $(w_1, \cdots, w_r', \cdots, w_n)$ by $P_{(w_1,\cdots,w_r,\cdots,w_n);(w_1,\cdots,w_r',\cdots,w_n)}$. Without any loss in generality we use notation $P_{(w_r;w_r')}$ to indicate the transition probability $P_{(w_1,\cdots,w_r,\cdots,w_n);(w_1,\cdots,w_r',\cdots,w_n)}$. All popular MPTCP congestion control algorithm (like BALIA [26]) adapts congestion window size of a sub-flow based on RTT estimation along that sub-flow. Therefore state transition probabilities of the proposed DTMC depend on RTT estimation. At this stage we ask this question: *What can be RTT estimate (r_i) at path (sub-flow) S_i, that can trigger a change of congestion window size to w'_i from w_i?*

3.5.1 Estimation of RTT for Congestion Window Size Adaptation

Although any MPTCP congestion control algorithm can be used for our modeling purpose, we use BALIA [26] as a representative case. As shown in [64], BALIA congestion control algorithm can be represented using the family of equations given by sub-eq. (3.1a) (for successful segment transmission) and sub-eq. (3.1b) (for a transmission failure), where $Y_i(t) = \frac{w_i(t)}{r_i}$ and $\alpha_i(t) = \frac{\max\limits_k\{Y_k(t)\}}{Y_i(t)}$. In this case, r_i represents measured RTT of S_i.

$$w'_i = \begin{cases} \dfrac{Y_i(t)}{r_i\left(\sum_k Y_k(t)\right)^2}\left(\dfrac{1+\alpha_i(t)}{2}\right)\left(\dfrac{4+\alpha_i(t)}{5}\right) & \text{Success} \quad (3.1a) \\ \frac{w_i(t)}{2}\min\{\alpha_i(t), 1.5\} & \text{Failure} \quad (3.1b) \end{cases}$$

Based on above estimation of congestion window size as given for BALIA congestion control algorithm, our objective is to find out r_i that can trigger a congestion window size of w'_i.

Let $\sum\limits_k Y_k(t) = (C_{-i}(t) + Y_i(t))$ and $\max\limits_k\{Y_k(t)\} = Y_m(t)$. Therefore, $\alpha_i(t) = \frac{w_m(t)r_i}{r_m w_i(t)}$. So, sub-eq. (3.1a) simplifies to sub-eq. (3.2a) and sub-eq. (3.1b) reduces to sub-eq. (3.2b). From this point onwards, we use w_i,w'_i and C_{-i} instead of $w_i(t)$, $w_i(t+1)$ and $C_{-i}(t)$ for notational simplicity.

$$w'_i = \begin{cases} \dfrac{w_i}{r_i^2\left(C_{-i}+\dfrac{w_i}{r_i}\right)^2}\left(\dfrac{4r_m^2 w_i^2+5r_i w_i w_m r_m+r_i^2 w_m^2}{10r_m^2 w_i^2}\right) & (3.2a) \\ \frac{w_i}{2}\min\{\frac{w_m r_i}{r_m w_i}, 1.5\} & (3.2b) \end{cases}$$

By solving sub-eq. (3.2a), we get

$$\begin{aligned} r_i =& \frac{5}{4}\left(\frac{-w_m w_i r_m}{2r_m^2 w_i w'_i} + C_{-i}w_i\right) \\ &\pm\sqrt{\left(\left(\frac{w_m w_i r_m}{2r_m^2 w_i w'_i} - 2C_{-i}w_i\right)^2 - \frac{8}{5}\left(\frac{w_m^2}{10r_m^2 w'_i i} - C_{-i}^2\right)\right)} \end{aligned} \tag{3.3}$$

Let $\vec{W} = \{w_1, w_2, ...w_n\}$ and $\vec{R} = \{r_1, r_2, ..., r_n\}$. From Eq. (3.3), we can observe that r_i is a function of $\vec{W}$, $\vec{R}$ and m. Note that here S_m is the path for which $Y_k(t)$ is maximum. we denote $r_i = f(m, \vec{W}, \vec{R})$ where $f(.)$ is the corresponding function as given in Eq. (3.3). We consider two cases – (i) Y_k is maximum for the current path S_i under consideration ($\max\{Y_k\} = Y_i$, and $m = i$), and (ii) Y_k is maximum for some other path S_m such that $m \neq i$. Therefore,

$$r_i = \begin{cases} f(i, \vec{W}, \vec{R}) & \text{if } \max\{Y_k\} = Y_i \\ f(m, \vec{W}, \vec{R}) & \text{otherwise} \end{cases} \tag{3.4}$$

By substituting $m = i$ in Eq. (3.3), we derive Eq. (3.5).

$$f(i, \vec{W}, \vec{R}) = \frac{w_i \pm \sqrt{w_i^2 + \frac{12w_i}{5w_i'} + 1.6}}{2C_{-i}} \tag{3.5}$$

Similarly, sub-eq. (3.2b) can be simplified also as follows.

$$r_i = \begin{cases} \dfrac{2w_i' r_m}{w_m} & w_i' \leq \dfrac{3w_i}{4} \text{ and } \max\{Y_k\} = Y_m & \text{(3.6a)} \\ 0 \geq r_i < \infty & \text{Otherwise} & \text{(3.6b)} \end{cases}$$

Now we can argue that given $\vec{W}$ and $\vec{R}$, the required RTT r_i can be calculated as per Eq. (3.4), sub-eq. (3.6a) and sub-eq. (3.6b). Therefore, we proceed for estimating state transition probabilities of the proposed DTMC based on this RTT estimation.

3.5.2 Estimation of State Transition Probabilities

According to Eq. (3.4) and sub-eq. (3.6a), transition events are:

1. SS_i**:** If the segment is delivered successfully via S_i, there can be two possible cases as follows:
   (a) SS_{max_i}**:** This transition event is triggered if $m = i$, that is $\max\{Y_k\} = Y_i$ for path S_i after successful delivery. As per the definition of Y_k, $Y_k \propto \frac{1}{r_i}$. Therefore, $\max\{Y_k\} = Y_i$ represents $\min\{r_k\} = r_i$.
   (b) SS_{max_m}**:** If $m \neq i$, then $\exists m \in \{1, 2, ..., i-1, i+1, ..., n\} : \max\{Y_k\} = Y_m$. This event is complement event of SS_{max_i}.
2. SL_i**:**If the segment is delivered successfully via S_i, there can be two possible cases as follows:
   (a) SL_{max_i}**:** This transition event is triggered if there is a segment loss reported, and $\max\{Y_k\} = Y_i$. In this case, according to sub-eq. (3.6b), the value of this event does

not depend on r_i. Therefore, we consider $0 \geq r_i \geq \infty$. In such cases only allowed sub-event is $w'_i = \frac{3w_i}{4}$. To signify this event, we use an indicator variable $\Gamma(w_i, w'_i)$ as given in Eq. (3.7).

$$\Gamma(w_i, w'_i) = \begin{cases} 1 & \text{if } 4w'_i = 3w_i \\ 0 & \text{Otherwise} \end{cases} \tag{3.7}$$

(b) SL_{max_m}**:** If $m \neq i$, then $\exists m \in \{1, 2, ..., i-1, i+1, ..., n\} : \max\{Y_k\} = Y_m$. This event is complement event of SS_{max_i}. Whenever this event is triggered, transition of $w'_i > \frac{3w_i}{4}$, becomes impossible (see, sub-eq. (3.6b)). Therefore, we only consider here sub-event $w'_i \leq \frac{3w_i}{4}$. To notify this sub-event, we use a separate indicator variable $\Delta(w_i, w'_i)$ as given in Eq. (3.8).

$$\Delta(w_i, w'_i) = \begin{cases} 1 & \text{if } 4w'_i \leq 3w_i \\ 0 & \text{Otherwise} \end{cases} \tag{3.8}$$

Now from the above set of events, we can say $pr(SL_i) = l_i$ and $pr(SS_i) = (1 - l_i)$, where $pr(E_i)$ denotes the probability of event E_i. Based on the set of events, we simplify the transition probability $P_{(w_i;w'_i)}$ by repeatedly applying law of total probability as given in Eq. (3.9).

$$P_{(w_i;w'_i)} = pr(SS_i)pr(w'_i|SS_i) + pr(SL_i)pr(w'_i|SL_i) \tag{3.9}$$

where,

$$pr(w'_i|SS_i) = pr(w'_i|SS_{max_i})pr(SS_{max_i}) + pr(w'_i|SS_{max_m})pr(SS_{max_m})$$

and,

$$pr(w'_i|SL_i) = \Gamma(w_i, w'_i)pr(SL_{max_i}) + \Delta(w_i, w'_i)pr(w'_i|SL_{max_m})pr(SL_{max_m})$$

It can be noted from sub-eq. (3.2b) that with BALIA, new congestion window (w'_i) should be less than or equals to $\frac{3}{4}$th of original congestion window (w_i) when a segment loss occurs. The indicator variable $\Gamma(w_i, w'_i)$ ensures this and accordingly we compute $pr(w'_i|SL_i)$. Now both the events SS_{max_i} and SL_{max_i} are equivalent to the event of i^{th} sub-flow having minimum r_i. According to our conjecture, $\forall i : Pr_i(X = r)$ are independent and identically distributed. Therefore, $pr(SS_{max_i}) = pr(SL_{max_i}) = \mathscr{Z}$ reduces to Eq. (3.10).

$$\mathscr{Z} = \int_{r=0}^{\infty} Pr_i(X = r) \prod_{k \neq i} Pr_k(X < r)dr \tag{3.10}$$

Similarly, $pr(SS_{max_m}) = 1 - pr(SS_{max_i})$ and $pr(SL_{max_m}) = 1 - pr(SL_{max_i})$ and other conditional probabilities can be calculated as follows – (a) $pr(w_i'|SS_{max_i}) = pr(X < f(i, \vec{W}, \vec{R}))$, (b) $pr(w_i'|SS_{max_m}) = pr(X < f(m, \vec{W}, \vec{R}))$, and (c) $pr(w_i'|SL_{max_m}) = pr(X < \frac{2w_i' r_m}{w_m})$.

This way we obtain transition probability from state $(w_1, w_2, ..., w_i, ..., w_n)$ to state $(w_1, w_2, ..., w_i', ..., w_n)$ $(P_{(w_i;w_i')})$ based on Eq. (3.9).

3.5.3 Estimation of Average MPTCP Throughput

We now compute average throughput of a MPTCP flow considering the data transfer rate through all its sub-flows. Let us consider that, $\vec{\Pi} = \left[\pi_{(2,2,...2)}, \pi_{(2,2,...,2,3)}, \ldots, \pi_{(W_{max_1}, W_{max_2}, ..., W_{max_n})}\right]$ be the stationary probability distribution vector of states for the given DTMC. Therefore, by using "*Markovian property*", stationary distribution of this DTMC can be calculated as per the following system of equations.

$$\pi_{w_1,...,w_n} = \sum_{k_1=2}^{W_{max_1}} \pi_{k_1,...,w_n} P_{(k_1;w_1)} + \ldots + \sum_{k_n=2}^{W_{max_n}} \pi_{w_1,...,k_n} P_{(k_n;w_n)} \tag{3.11}$$

We also have the normalization equation from the DTMC, which can be represented as follows.

$$\sum_{w_1=2}^{W_{max_1}} \sum_{w_2=2}^{W_{max_2}} \cdots \sum_{w_n=2}^{W_{max_n}} \pi_{w_1,w_2...w_i,...,w_n} = 1 \tag{3.12}$$

Let us define a "*round*" as interval between two successive state transition events. If the system is currently under state $(w_1, w_2, ..., w_n)$, then the total number of segments that can be sent is calculated as $\sum_{j=1}^{n} w_j$. Therefore, average number of segments sent by a state $(w_1, w_2, ..., w_n)$ is $\pi_{(w_1,w_2,...,w_n)} \sum_{j=1}^{n} w_j$. Consequently, the average number of segments that can be sent in one round (denoted as $Avg_C(\vec{Q})$ for a given configuration $\vec{Q} = \{q_1, q_2, ..., q_n\}$) is expressed as Eq. (3.13).

$$Avg_C(\vec{Q}) = \sum_{\forall w_i} \left(\pi_{(w_1,w_2,...,w_n)} \sum_{j=1}^{n} w_j \right) \tag{3.13}$$

Now we have to compute average time for a "*round*". Average time for a "*round*" includes (a) total data transmission time (time to transmit $Avg_C(\vec{Q})$ number of segments), (b) time to receive "*acknowledgments*" for the transmitted segments, and (c) time for "*retransmission*" of lost segments. We assume a x-"*duplicate acknowledgment*" scheme, where a segment is "*retransmitted*" if the sender receives x number of consecutive "*duplicate acknowledgments*". Assume that segment size is s_s and acknowledgment size is a_s. Then for a given $\vec{Q}$, average time required for one "*round*" ($Avg_T(\vec{Q})$) is computed as follows.

$$\mathcal{G}(\vec{Q}) = \max_x \left\{ \frac{((w_s^{avg} + x) \times s_s)}{b_s} + \rho \frac{w_s^{avg} \times a_s}{b_s} \right\} \tag{3.14}$$

In this case w_s^{avg} represents average number of segments sent by a MPTCP sub-flow S_s and ρ is RTT of that sub-flow.

Therefore, using Eq. (3.13) and Eq. (3.14), average throughput is calculated as,

$$Avg_{Th}(\vec{Q}) = \frac{Avg_C(\vec{Q})}{\mathcal{G}(\vec{Q})} \tag{3.15}$$

3.5.4 Estimation of Receiver Buffer Size

The receiver buffer occupancy increases mainly due to out of order segment delivery. We define segment seg_i as a "*key*" segment if all other segments $seg_j : j > i$ reach to the destination before seg_i. All seg_j must wait at the receiver buffer for the key segment, in order to ensure reliable delivery. Therefore, occupancy of receiver buffer depends on the event that seg_j is successfully delivered before seg_i. We denote seg_i^{max} as the segment which stays in the queue for longest time for a key segment seg_i. So, receiver buffer length (RL) can be expressed as Eq. (3.16), where $\Delta(seg_k, seg_l)$ denotes arrival time difference between seg_k and seg_l.

$$RL = |\Delta(seg_i^{max}, seg_i)| \times \text{throughput} \tag{3.16}$$

Subsequently, we can approximate average receiver buffer length ($E_{RL}(\vec{Q})$) for a given configuration $\vec{Q}$ based on [178]:

$$E_{RL}(\vec{Q}) \approx (\max_k(r_k) - \min_k(r_k)) Avg_{Th}(\vec{Q}) \tag{3.17}$$

3.5.5 Model Verification

To verify correctness of our proposed DTMC based model, we have compared average throughput and receiver buffer length with emulation results obtained using "*Mininet*" [49]. The test topology is given in Fig. 3.7. All switches given in the topology (Sw1-Sw6) are SDN switches. Emulation links are configured to have $20ms$ delay. Path S1 and S2 have bottle neck bandwidth

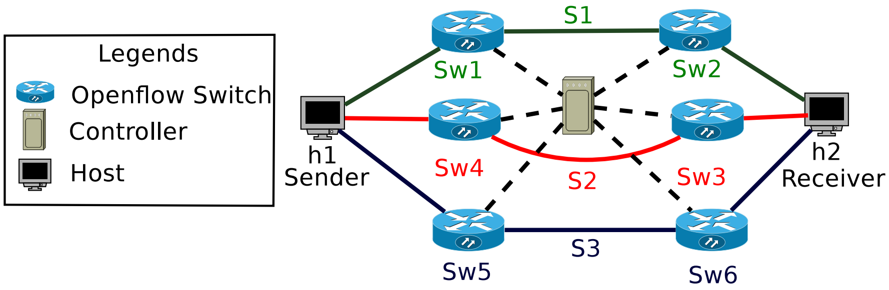


Fig. 3.7: Topology Structure for Experiments

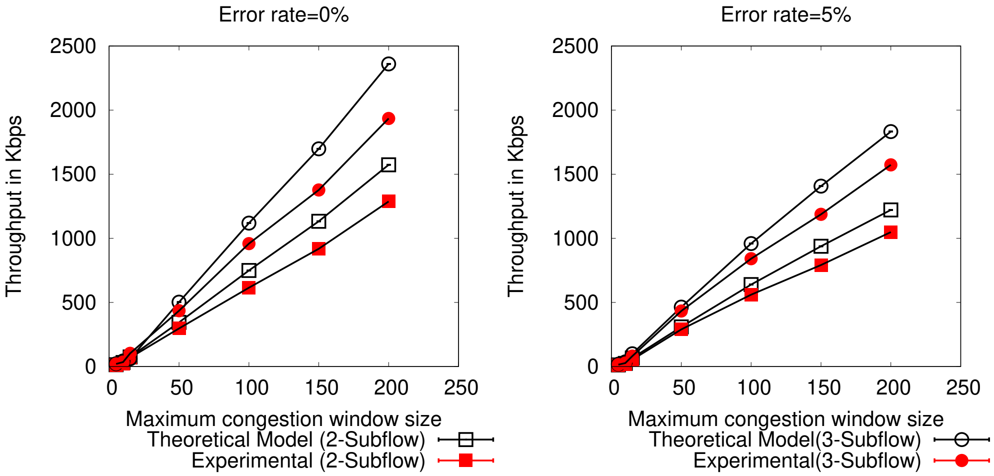


Fig. 3.8: Throughput Comparison

of $8mbps$ and S3 is configured with $18mbps$ of bandwidth. Results are obtained for two different loss rates (0% and 5%). Fig. 3.8 shows effect of maximum congestion window size with average throughput for two and three active sub-flows. Fig. 3.9 represents effect of maximum congestion window size on length of the receiver buffer. We observe that our proposed model can predict behavior of MPTCP reasonably well as the average prediction error of the proposed model has been found as 9.19%. Therefore, in the next section, we present the sub-flow scheduling problem

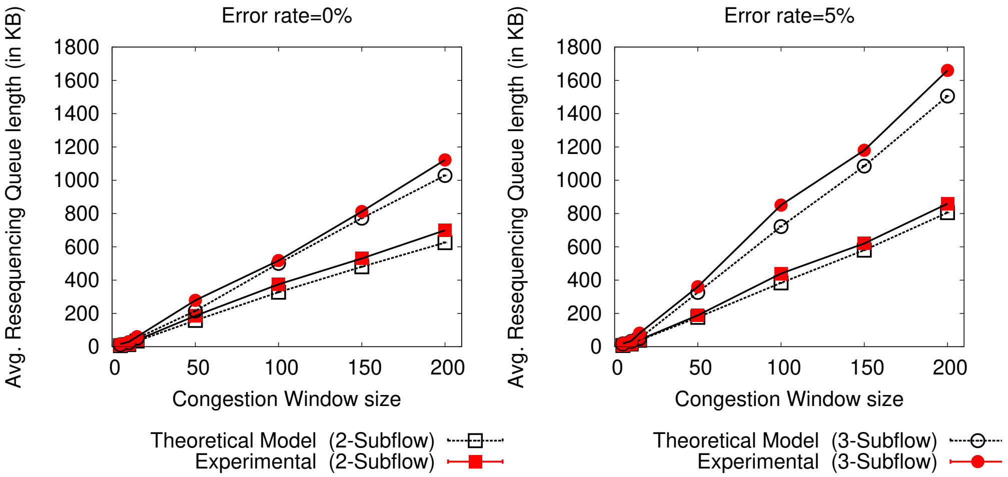


Fig. 3.9: Receiver Buffer Size Comparison

based on this estimation model.

3.6 Sub-Flow Selection based on Performance Estimation from DTMC

The objective of sub-flow selection problem, for a given MPTCP connection and a set of all available sub-flows $\mathcal{S} = \{S_1, S_2 \ldots S_n\}$, is to select a subset of $\mathcal{S}$ for optimizing the average throughput. However, optimal average throughput can increase receiver buffer size, which in turn might deteriorate overall performance. Therefore, sub-flow selection problem must limit receiver buffer size to a certain threshold (RL_{max}). Length of the receiver buffer length depends upon congestion control algorithm and scheduling mechanism. Therefore, in the previous section, we propose a mathematical model to estimate receiver buffer length in Eq. (3.17) in presence of BALIA congestion control. The proposed model also provides average throughput (Eq. (3.15)). Based on the estimated values, sub-flow selection problem can be formulated as a mixed integer linear program (MILP).

Given $\mathcal{S}$, power set of $\mathcal{S}$ ($\wp(\mathcal{S})$) provides all the possible configurations. Let, $\vec{I}$ be an indicator vector of all possible sub-flow configuration such that $\vec{I} = \{\forall k \in \wp(\mathcal{S} : I^k)\}$. We define $I^k \in \mathbb{R}^n$

for a given configuration $k \in \wp(\mathcal{S})$ as per Eq. (3.18).

$$I_j^k = \begin{cases} 1 & \text{if } S_j \in k \\ 0 & \text{otherwise} \end{cases} \tag{3.18}$$

Let, $\vec{Q} = \{q_1, q_2, ..., q_n\}$ be path quality matrix of all available sub-flows such that $q_i = \{b_i, l_i, \mu_i, \sigma_i\}$. Therefore, effective path quality matrix of only active sub-flows (X^k) for k^{th} configuration can be expressed as $X^k = \vec{Q} \circ I^k$ where $\circ$ denotes the Hadamard product (element wise product) between two matrices of same dimension. We denote the average throughput of all active sub-flows in k^{th} configuration as $Avg_{Th}(X^k)$. $Avg_{Th}(X^k)$ can be calculated using Eq. (3.15). Eq. (3.17) can be used to calculate estimated receiver buffer length ($RL(X^k)$) for the k-th configuration. Now we can represent optimal sub-flow selection problem as an optimization problem as given in Eq. (3.19).

$$\begin{aligned} &\max_k Avg_{Th}(X^k) \\ &\text{subjected to, } RL(X^k) \le RL_{max} \end{aligned} \tag{3.19}$$

This optimization problem is equivalent to "0-1 *knapsack problem*" [179], where $Avg_{Th}(\vec{X})$ and $RL(\vec{X})$ can be treated as the capacity of the knapsack. 0-1 knapsack problem is known to be *NP*-hard. Therefore, we propose a greedy heuristic Algorithm 1 to solve Eq. (3.19).

We define effective bandwidth of a sub-flow as $b_i(1 - l_i)$. Our proposed heuristic should be able to increase effective bandwidth. However, as per Eq. (3.17), the length of receiver buffer inversely proportional to effective bandwidth. Eq. (3.17) also reveals that, with increase in RTT, delay between key segment and the rest of segment increases. Therefore, we can conclude that RTT of a sub-flow is directly proportional to the length of receiver buffer length. So, the proposed heuristic is built upon these two governing factors. We apply linear scalarization to find the best possible sub-flow. Our proposed heuristic ensures that a sub-flow with high effective bandwidth and low RTT gets higher priority of selection if that sub-flow does not increase estimated receive buffer length than RL_{max}.

To implement the heuristic, we exploit SDN capabilities for accumulating $\vec{Q}$. In case of SDN supported infrastructure, an SDN controller may periodically gather individual port statistics such as link bandwidth, loss rate and approximate delay for each data plane device. The gathered statistics can provide an estimate of end-to-end characteristics. Upon receiving a MPTCP connection-open request, the controller finds set of n paths between source-destination

Algorithm 1: Heuristic for sub-flow selection

```
Input: $\vec{Q}$
Output: $\vec{I}$
$\forall i : I_i \leftarrow 0$;
Sort $\vec{Q}$ based on $T_i \leftarrow b_i(1 - l_i) + \frac{1}{\mu_i}$;
Find $\max_i(T_i)$; $I_i \leftarrow 1$;
foreach $j \in (2, 3 \cdots n)$ do
    $\vec{X} \leftarrow \vec{Q} \circ I$;
    $\mathcal{A} \leftarrow Avg_{Th}(\vec{X})$; $\mathcal{R} \leftarrow RL(\vec{X})$;
    if $\mathcal{R} \leq RL_{max}$ then
        $I_j \leftarrow 1$;
return $\vec{I}$;
```

pair based on the underlying routing protocol. The value of n depends on number of network interfaces available and the path manager used by the end hosts. According to the full-mesh path manager, all of n paths should be used as active sub-flows. After initial path selection and sub-flow identification, the controller periodically calculates end-to-end quality of sub-flow S_i (as denoted by Q_i). Upon calculating $\vec{Q}$, the controller uses Algorithm 1 to calculate set of active sub-flows as $\mathcal{S}_{active} = \{\forall i, I_i \neq 0 : S_i\}$. This $\mathcal{S}_{active}$ is relayed back to the path manager which activates corresponding sub-flows.

3.7 Implementation Details and Performance Evaluation

In this section, we discuss the performance of the proposed sub-flow selection mechanism in previous section. We have emulated an SDN environment through *Open vSwitch* [139] via the Mininet [49] emulation platform at the Department of Computer Science and Engineering, IIT Guwahati.

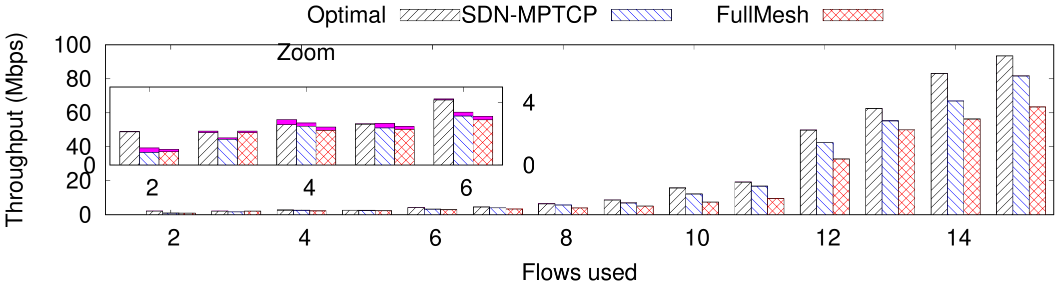


Fig. 3.10: Comparison of Throughput

3.7.1 Implementation Methodology

We have used open-source MPTCP kernel module [73] in our testbed. To integrate with the SDN "`POX`" controller, our developed module[2] uses "`UDP`" to communicate. Upon detecting change, "`POX`" recalculates the active sub-flow set and pushes to the host as a recommendation in "`JSON`" format. The recommendation identifies sub-flow set by network addresses. Upon receiving the request, the path manager module translates network addresses to sub-flow IDs and selects the corresponding sub-flows as active. Accordingly, it notifies the congestion control module about these changes.

3.7.2 Competing Heuristics

It can be noted that to the best of our knowledge, existing literature have not worked on MPTCP sub-flow selection problem. As discussed earlier, MPTCP kernel implementation has two variants of sub-flow selection or path manager algorithm – *Full-mesh* and *ndiffports*. Although *ndiffports* progresses in the direction of sub-flow selection, but it uses a naive implementation of random sub-flow selection, which does not work well in practice. Therefore, we consider the *Full-mesh* path manager as the competing heuristic of our proposed protocol. Further, we compare performance of the proposed methodology with optimal performance, as computed by enumerating over all possible combination of paths.

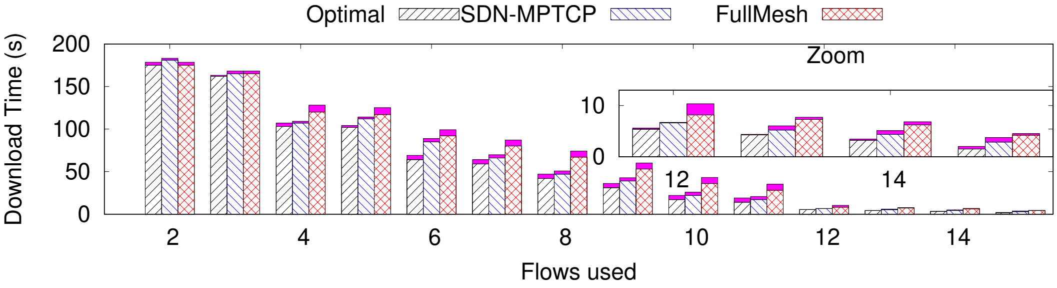


Fig. 3.11: Comparison of Flow Completion Time

3.7.3 Topology Details and Emulation Results

Topology – We choose a topology which is similar to the one given in Fig. 3.7. We configure 15 parallel paths between a sender and a receiver with the end-to-end parameters as follows. We increase bandwidth of these paths from 1 Mbps to 15 Mbps with a step of 1 Mbps. The delay is increased from 10 ms to 150 ms with a step of 10 ms, whereas the path loss increases from 0% to 15% with a step of 1% per path. The sender generates MPTCP supported "*HTTP*" flows destined towards receiver host.

Average file download time and aggregated throughput – Figs. 3.10 and 3.11 shows performance comparison of the proposed scheme with the Full-mesh path manager and off-line optimal path manager in terms of download time of a 100MB file over standard "*HTTP*" protocol and average aggregated throughput. The emulation results reveal that, Full-mesh path manager performs quite well for up to 3 sub-flows. However, increase in the number of sub-flows increases download time significantly and reduces average throughput. Performance of the proposed methodology is considerably well compared to the Full-mesh path manager, and also very close to the optimal performance as we observe for our emulation scenarios.

Effect on congestion control parameters – To understand why the proposed methodology significantly boosts up performance of MPTCP, we explore evolution of several parameters that control MPTCP congestion control mechanism. As given in Fig. 3.12, a significant reduction in out of order segments can be observed in case of our proposed methodology in comparison to the Full-mesh path manager. As shown in Figs. 3.13 and 3.14, our proposed path manager also significantly reduces number of retransmitted segments and lost segments by selecting effective

---

[2] `https://github.com/subhrendu-subho/SDN_pathmanager`

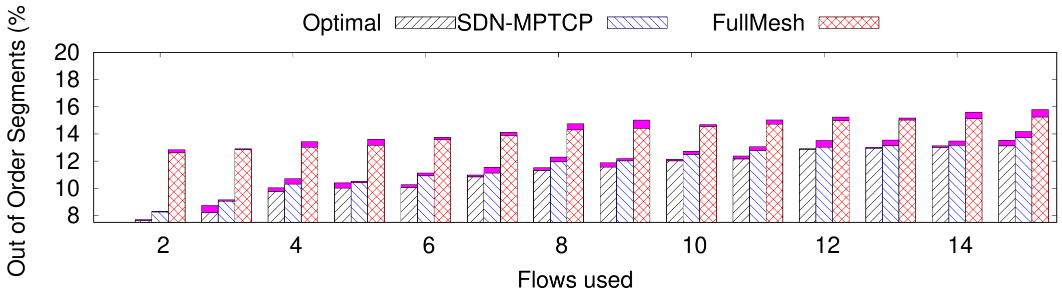


Fig. 3.12: Out of Order Segments

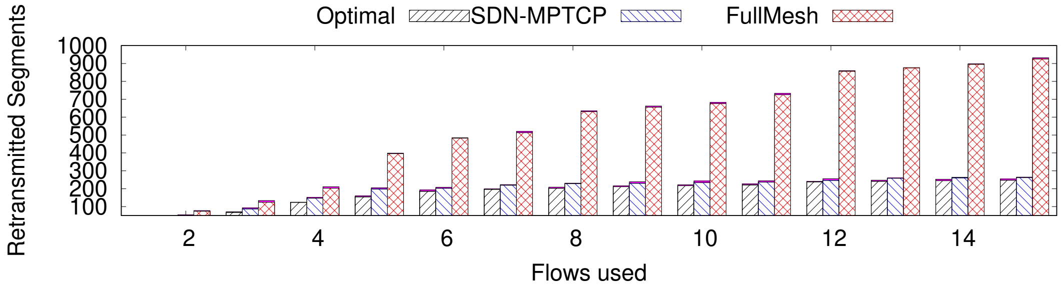


Fig. 3.13: Retransmitted Segments

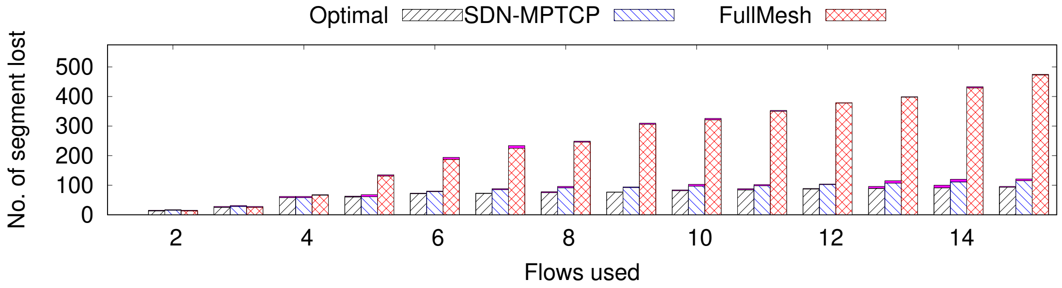


Fig. 3.14: Lost Segments

sub-flows.

Analysis of congestion window evolution – The reason behind effectiveness of the proposed sub-flow management mechanism can be justified by the help of congestion window progression analysis. For this purpose we analyse and compare two particular run instances Fig. 3.15 to explain the behaviour. The Fig. 3.15 shows progression of the congestion window for a particular run where the Full-mesh path manager uses 4 sub-flows. For similar scenario,

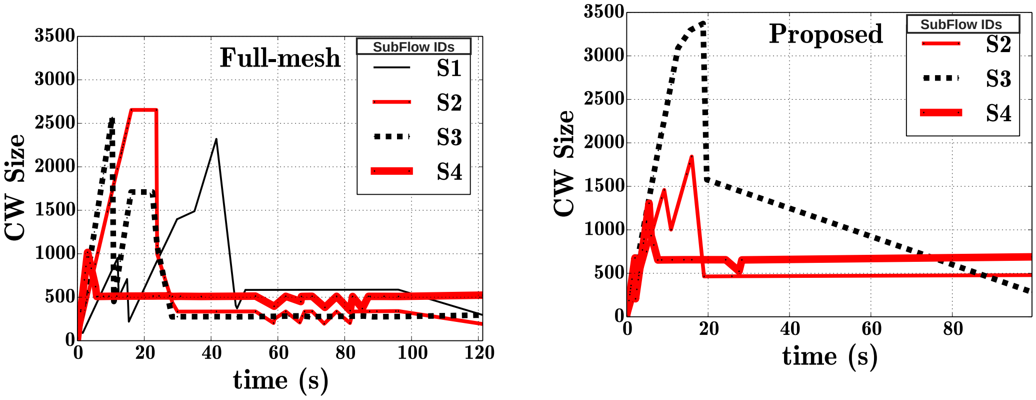


Fig. 3.15: Congestion window size

the proposed path manager uses only 3 sub-flows which can be observed in Fig. 3.15[3]. As argued earlier, due to reduction of lower bandwidth path, the proposed methodology can reduce the number of retransmit events along with out of order segments. Therefore, it can help all the sub-flows to converge to their steady state congestion window size quickly.

Impact on RTT – On the other hand, our proposed scheme also reduces receiver buffer size. Therefore, the sub-flows experience lesser delay compared to the Full-mesh path manager, and observe reduced RTT, as shown in Fig. 3.16. As a result, overall performance improves significantly with the help of the proposed sub-flow management module hooked with the standard MPTCP kernel.

---

[3] The sub-flow ($S1$) is not used by the proposed framework

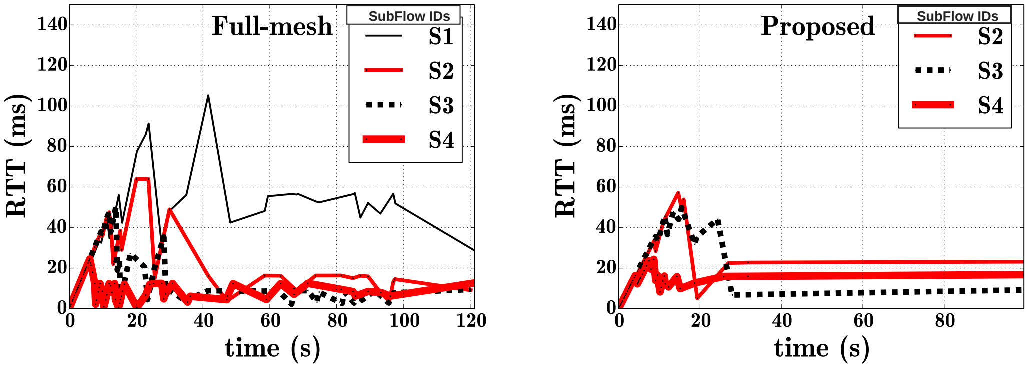


Fig. 3.16: RTT Variation

3.8 Summary

In this chapter, we develop `SDN-MPTCP`, a sub-flow management framework for MPTCP protocol over a SDN controlled LSiN. Our proposed framework reduces out of order segments and HOL blocking in MPTCP. The emulation results show that our proposed sub-flow management heuristic outperforms the existing path manager in MPTCP and very closely approximates a *NP*-hard problem of optimal sub-flow selection in terms of various performance metrics. `SDN-MPTCP` ensures the transport layer performance improvement by taking assistance of SDN. This work shows that, integration of SDN in a LSiN can provide performance benefit along with ease of management concerns. However, deployment of SDN over an existing LSiN is difficult due to increase in deployment cost which we discuss in the next chapter.

Chapter 4

Flipper: Dynamic NIB Placement For Distributed SDN

4.1 Introduction

In the previous chapter, we have seen how Software-Defined Network (SDN) can improve the performance of a transport layer protocol. Additionally, SDN can provide ease of network management. For example, the network administrator of an large scale IoT network (LSiN) service provider wants to dynamically update bandwidth distribution policies based on network usage statistics. The target network is connected with multiple network service providers, and therefore she needs to update configuration at different edge routers and gateways. With traditional network devices, like layer 3 switches, this task is tedious as even a minor configuration inconsistency among edge routers and gateways may lead to severe network underutilization or bandwidth imbalance. Further, the system is also not scalable for such dynamic updates of network configuration policies. Since SDN [15] can help in dynamic network configuration updates by providing programmable control plane, managing networks using SDN becomes less hectic for the administrator. The centralized control plane of SDN converts policies into device configurations

---

Has been published in

[T.3] Subhrendu Chattopadhyay, Niladri Sett, Sukumar Nandi, and Sandip Chakraborty. "Flipper: Fault-Tolerant Distributed Network Management and Control". In: *Fifteenth IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2017 and updates targeted devices in the network with the corresponding configurations. However, deployment of SDN over an existing LSiN suffers from capital expenditure (capex)/operational expenditure (opex) issue, which we describe next.

SDN deployment requires specific hardware that can understand language for SDN, like Open virtual switch (OVS) [180, 181], so that a SDN controller can dynamically update configuration parameters for such hardware. Therefore the important question is: *How much effort and cost does one need to convert an existing network infrastructure to an SDN supported one?* The existing studies in this direction talk about interoperability among SDN supported and non-SDN network devices, such that incremental deployment of SDN supported devices becomes possible [30, 31, 32]. However, concern regarding cost-effectiveness is still there. SDN supported hardware is much costlier than Commercial off-the-shelf (COTS) network devices, and therefore requires huge operational expenditure to replace existing infrastructure by SDN supported infrastructure.

Although it is quite inevitable that the future of network management is SDN, simultaneously we also ask this question: *Can we make our existing network more management friendly, such that dynamic network configuration becomes possible without changing the existing infrastructure much?* This work tries to find out the answer to this question. We show that it is quite possible to use existing COTS routers to work as Policy decision and enforcement point (PDEP), which are known as Network Information Base (NIB). We can turn a COTS router to a NIB by installing a few additional software tools to support Network Function Virtualization (NFV) [44]. With the help of NFV functionalities, a COTS router can dynamically update policy control parameters within its neighborhood [18, 182]. Accordingly, we develop a new network management architecture, which is somewhere in-between the traditional architecture and SDN based architecture, where COTS routers dynamically change their roles from a conventional network router to an NIB and participate in PDEP functionalities. We call this architecture `Flipper`.

`Flipper` has two specific advantages over SDN based network architecture, among others. First, to implement `Flipper`, a network administrator does not need to procure new costly hardware and second, `Flipper` avoids the controller bottleneck problem [30, 183, 44, 184, 185] which is much debated in the SDN research community. `Flipper` is a distributed architecture, where COTS routers execute a distributed self-stabilizing algorithm to decide which nodes can work as an NIB. As the NIBs have limited resources because they are built on top of the existing routers, an NIB can manage, control, and update network policies only among its neighborhood. Therefore, we develop a distributed "*self-stabilizing*" Maximal Indemendent Set (MIS) selection mechanism, which is indeed non-trivial. To maintain consistency in policy decisions across the network, we have developed a fault-tolerant NIB selection mechanism. We analyse the closure, fault-tolerance, and scalability properties of `Flipper`. The performance of `Flipper` is analysed from both simulations through a synthetic network environment and real implementation over an emulation platform using "*`network name-space`*". Our implementation of `Flipper` provides proof-of-concept support of the new architecture while comparing performance with other protocols in terms of flow initiation delay.

4.2 Flipper Architecture

This section gives the details of `Flipper` architecture and its working procedure. `Flipper` uses NFV to convert existing COTS routers to PDEP devices. For this task to convert a COTS router to a PDEP supported device, we use the existing technology called `ONIX` [44] that describes how the NFV modules can be interfaced with existing router operating system to make it work as a PDEP device that can sync up network policies with other PDEP devices, converts it to network configurations and feeds up those configurations to other normal routers in the neighborhood. Although we use `ONIX` technology for this purpose, but deploying it over an existing network is non-trivial, because of the limited processing capacity of the existing COTS routers. As a consequence, such devices introduce large delay and processing overhead if a single `ONIX` node works like a SDN controller. Therefore, in `Flipper`, we introduce a distributed dynamic PDEP selection mechanism, which is self-stabilized and fault-tolerant. The details of this architecture are discussed next.

4.2.1 Proposed `Flipper` Architecture

Our proposed `Flipper` architecture consists of following components which are similar to `ONIX`. Although the components are similar, functionalities and arrangement of the components are different in `Flipper`.

1. OpenFlow supported switch: An OpenFlow supported switch (OFS) is responsible for data forwarding based on forwarding rule set. "*OpenFlow*" [186, 8] is a software component that is installed in the router Operating System (OS) to provide NFV functionalities. However, mere

"*OpenFlow*" support does not make these devices SDN complaint, as specialized hardware (like OVS) is required for this purpose. In our Flipper architecture, we install additional components only at the software level in COTS hardwares.

2. Host: End user devices connected with OFSs that hosts applications and generates data traffic.

3. DHT-NIB: Memory based high update prone eventually-consistent Distributed Hash Table based NIB (DHT-NIB) for storing link level information of switches. DHT-NIB also helps in setting up forwarding rules in switches based on control application. As shown in `ONIX` architecture, an OFS can act as a DHT-NIB with additional functionalities.

4.tran-NIB: Strongly consistent "*tran-NIB*" is used for rarely changed network wide policy management.

A major difference between the existing SDN based architecture and `Flipper` is that the standard SDN components have fixed roles to play. However, in this work we define a `Flipper` device as a service grade router which can dynamically choose a role of either OFS or DHT-NIB. This dynamic change of roles ("*flip*") are possible due to use of NFV in `Flipper` devices. For the sake of readability we refer the "*Flipper*" architecture as `FLIPPER` and "*Flipper*" devices as "*flipper*".

4.2.2 `FLIPPER` Working Principle

To understand the working principle of `FLIPPER`, we take help of Fig. 4.1. The topology consists of a dedicated high performance transactional NIB, hosts (A, B, C, D) and flippers $(R1, R2, ..., R9)$. "*switch-flipper*" if a flipper that acts as an OFS. "*DHT-flipper*" is the flippers that perform DHT-NIB functionalities. Initially, flippers adjust themselves so that a switch-flippers have at least one Distributed Hash Table based flipper (DHT-flipper) in its neighborhood. Upon receiving a flow request from switch-flipper, the distributed control plane consults relevant DHT-flippers based on programmable network rules in tran-NIBs and completes flow table setup procedure in switch-flippers.

4.2.3 Fault-tolerance in `FLIPPER`

The use of NFV for deployment of services provides flexibility towards `FLIPPER`. However, general purpose switches in a service provider network are failure prone. The failure of a DHT-flipper can

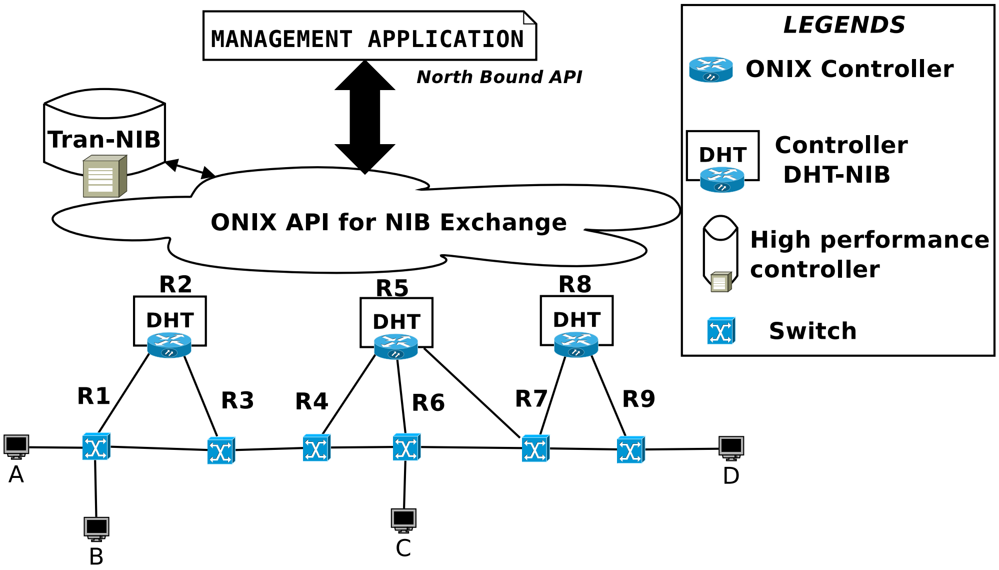


Fig. 4.1: `FLIPPER`: Architecture

significantly affect network performance as it controls all flows in its neighborhood. Therefore, to maintain the robustness of the architecture, `FLIPPER` needs to be fault-tolerant. For example, in Fig. 4.1, let us assume that $R2, R5$ and $R8$ are acting as DHT-flippers. The associated switch-flippers of $R2, R5$ and $R8$ are $\{R1, R3\}, \{R4, R6, R7\}$ and $\{R9\}$, respectively. If $R5$ fails, $R4, R6$ can not work in the absence of DHT-flipper. For maintaining fault tolerance, we propose a distributed flipper readjustment framework. Whenever one or more switch-flippers detect unavailability of DHT-flipper in its (their) neighborhood, it (they) invokes (invoke) flipper readjustment procedure. The re-adjustment procedure provides the newly selected set of DHT-flippers and switch-flippers. After reaching a consensus, each switch-flipper notifies its adjacent DHT-flipper with its state information. A switch-flipper having multiple DHT-flippers in its neighborhood chooses a DHT-flipper randomly. Therefore, they can initiate the distributed flipper readjustment framework. In this case, we use distributed self stabilization technique to make the flipper readjustment fault-tolerant.

4.3 Fault-tolerant Flipper Readjustment

To make readjustment of switch-flippers and DHT-flippers fault-tolerant, we consider use of "*self-stabilization*" [187] which is a popular technique to provide defense against "*transient failures*". A transient failure is defined as irregular and unpredictable brief failure. In this work, we propose a novel flipper readjustment algorithm which expectedly converges with linear time complexity. Our proposed algorithm also satisfies the basic properties of self-stabilization which are as follows.

1. **Convergence:** From any state, the system must reach a legitimate or desired state eventually.
2. **Closure:** In case of no failure, the system is guaranteed to remain in legitimate states.

We consider the network as a graph $G = \{V, E\}$, where V is the set of flippers and E is the set of edges representing physical connections among flippers. Each flipper periodically senses physical medium for detecting link failure. A flipper i maintains label $Label_i \in \{NIB, Swi, Wait\}$ and priority variable $Pri_i \in \{0, 1, \ldots, B\}$, where B denotes the maximum degree of G. Any flipper k with $Label_k = NIB$ signify that, the flipper k is ready to act as a DHT-flipper. Similarly, a flipper l with $Label_l = Swi$ acts as an switch-flipper. We consider flipper with $Label = Wait$ as a flipper with intermediate state whose role is yet to decide. Neighborhood of flipper i is denoted by N_i. Flipper i also maintains $N_i^{NIB} = \{j | j \in N_i \wedge Label_j = NIB\}$ and $N_i^{Wait} = \{j | j \in N_i \wedge Label_j = Wait\}$. We consider the state of i as $(Label_i, Pri_i)$. Each flipper also maintains state of its adjacent neighbor flippers. When a flipper changes its state, it pro-actively notifies its neighbors. Upon detecting a link failure, flipper removes entry about the corresponding neighbor from its table.

We represent our proposed algorithm as a set of guarded actions, where each guarded action is termed as a *rule*. A rule R_j, uses the following representation, $(\mathbf{R_j}) | < \mathbf{G_j} > \longrightarrow < \mathbf{A_j} >$, where $< \mathbf{G_j} >$ represents condition which is required to be satisfied to execute action $< \mathbf{A_j} >$. Upon receiving an update from neighbor, each flipper checks guard statements of the rules. If any one of the guard is found to be true, then the corresponding action is executed.

4.3.1 `flipper` Readjustment in Case of Failure

Following the aforementioned model, the flipper readjustment problem is defined as follows. Given a network graph G, objective of the flipper readjustment problem is to find set of DHT-flippers in such a way that all switch-flippers can have at least one DHT-flipper in their neighborhoods, so that the policy updates can be done with minimal control plane delay and network overhead. The solution approach must find an alternative DHT-flipper dynamically when any flipper or link fails, to incorporate fault-tolerance property. The flipper readjustment mechanism is similar to finding a MIS in flipper connectivity graph. We propose a novel distributed anonymous self-stabilizing MIS (SS-MIS) algorithm to find DHT-flippers dynamically. The reason for using anonymous algorithm is to remove unfairness issue caused by identifier system. Our proposed anonymous SS-MIS protocol has a step complexity [1] of $\mathcal{O}(n)$. Eventhough there exists a linear time self-stabilizing distributed algorithm [188] for solving MIS problem in idntifier system, for anonymous systems the best proposed solution [189] has $\mathcal{O}(n \log n)$ step complexity[2]. In this work, we propose a linear time algorithm for anonymous systems that can significantly reduce the control plane overhead in `FLIPPER`.

4.3.2 SS-MIS Algorithm for flipper Readjustment

The proposed SS-MIS protocol selects switch-flippers and DHT-flippers in terms of assigning $Label = Swi$ and $Label = NIB$ respectively. According to MIS properties, no two DHT-flipper can be adjacent and, each switch-flipper should have at least one DHT-flipper in its adjacency list. The proposed protocol is described in Algorithm 2.

A flipper i which has $Label_i = Wait$ or $Label_i = NIB$, violates the independence property if any of its neighbor is in NIB state. Hence, it must execute (R_2, R_3), and must go to a state having $Label_i = Swi$. If two adjacent flipper have $Label = Wait$, and no other neighbor of them are in $Label = NIB$ state, then both the adjacent flippers will try to enter in a state with $Label = NIB$ state which requires a tie breaking mechanism. Although, the tie breaking can be done using an identifier (ID) of the flipper, in this work ID based tie breaking is not used. ID based tie breaking introduces unfairness problem, because a flipper with higher ID always gets a priority. Therefore, to break this tie, a randomized trial is performed. The proposed random trial is designed in the following way. Each node in $Wait$ state generates a random number in the range $\{0, 1, 2, \ldots, B\}$, and assigns to Pri. A "*Winner*" is decided based on the unique maximum Pri value in a closed neighborhood. If no winner is found in a single experiment, it is repeated until there is a winner. The winner gets the privilege to enter into the NIB state.

---

[1] Execution of an action is called a step. Step complexity of a distributed system is defined as the number of steps executed by the system. Throughout this work, the terms step and move are used invariably.

[2] To the best of our knowledge

Algorithm 2: SS-MIS Protocol

```
Variable: $Label_i = \{NIB, Swi, Wait\}$;
Variable: $Pri_i = \{0, 1, \ldots, B\}$;
def $N^{NIB}$(i: int):
    return $\{j | \forall_{j \in N_i} Label_j = NIB\}$;
def $N^{Wait}$(i: int):
    return $\{j | \forall_{j \in N_i} Label_j = Wait\}$;
def $Max_W$(i: int):
    return $\{j | \forall_{j \in N_i^{Wait}} Max(Pri_j)\}$;
def Trial(i: int):
    return $Pri_i \leftarrow Rand(0, 1, 2, \ldots, B)$;
begin
    // $R_1$
    $(Label_i = Swi) \bigwedge (N^{NIB}(i) = \emptyset) \longrightarrow (Label_i \leftarrow Wait) | Trial(i)$;
    // $R_2$
    $(Label_i = NIB) \bigwedge (N^{NIB}(i) \neq \emptyset) \longrightarrow (Label_i \leftarrow Swi)$;
    // $R_3$
    $(Label_i = Wait) \bigwedge (N^{NIB}(i) \neq \emptyset) \longrightarrow (Label_i \leftarrow Swi)$;
    // $R_{4a}$
    $(Label_i = Wait) \bigwedge (N^{NIB}(i) = \emptyset) \bigwedge (Pri_i = Max_W(i)) \longrightarrow (Label_i \leftarrow Wait) | Trial(i)$;
    // $R_{4b}$
    $(Label_i = Wait) \bigwedge (N^{NIB}(i) = \emptyset) \bigwedge (Pri_i > Max_W(i)) \longrightarrow (Label_i \leftarrow NIB)$;
```

4.4 Properties of Flipper Architecture

In this section we discuss about the properties of proposed flipper architecture. Let the global state of the system be denoted as $\mathcal{S}$; and *legitimate state* is defined as global configuration where no further rule may be applied at any flipper. We claim that the proposed scheme is self-stabilizing. A proof of self-stabilization requires the proof of **Closure property** and **Convergence property**.

4.4.1 `FLIPPER` Supports Closure Property

Theorem 4.1 *If any flipper in the system is in intermediate state then there is at least one rule which can be executed.*

Proof: Assume the state of an intermediate flipper u is $Wait$. Now there can be following scenarios.

Case 1: $\exists v \in N_u : (Label_v = NIB)$. In this case, R_3 is applicable.

Case 2: $\forall v \in N_u : (Label_v = Wait)$ and $(Pri_v < Pri_u)$. In this case flipper u has unique maximum priority. R_{4b} is applicable on flipper u and it acts as DHT-flipper.

Case 3: $\exists v \in N_u : (Label_v = Wait)$ and $(Pri_v = Pri_u)$ where Pri_v and Pri_u are maximum in their neighborhood. In this case flipper u and v must apply rule R_{4a} and retrial for a new priority value.

Case 4: $\exists v \in N_u : (Label_v = Wait)$ and $(Pri_v > Pri_u)$. Also $\exists w \in \mathcal{N}(v) : (Label_w = Wait)$ and $(Pri_w > Pri_v)$. From this statement it can be concluded that $(Pri_w > Pri_u)$. Hence priority of these forms a non-increasing function. Also number of flippers are bounded by N. Hence, at least one flipper has highest priority which can execute rule R_{4b} or R_{4a}. $\square$

Corollary 4.1 ***(Closure property)*** *If the system is in a state where flippers with DHT-flippers form a MIS, it remains in that state forever, provided no further fault occurs.*

Corollary 4.1 also suggests the correctness of the proposed scheme.

4.4.2 `FLIPPER` Converges If a Failure Occurs and It is Scalable

A self-stabilizing system always converges in case of a failure. We analyze the algorithm and prove that, the expected time required to converge is linearly dependent on the number of flipper used.

Theorem 4.2 *Let $P(N, B)$ denotes probability of finding an unique maximum in the closed neighborhood of v where, N denotes the cardinality of the closed neighborhood of any arbitrary flipper v. The probability of one flipper in the closed neighborhood having unique maximum after one trial is as follows.*

$$P(N, B) = \frac{(N \times \sum_{i=1}^{B} i^{(N-1)})}{(B+1)^N}$$

Proof: Let i be the highest priority in a configuration $\mathcal{S}$ after one round, where each round corresponds to the event of generating priority by *"at most one"* flipper in the closed neighborhood of v. To satisfy unique maximum property, i can be assigned to any one of the N flippers and the rest of flippers can have a priority value ranging from 0 to $i-1$. So there are $N \times i^{(N-1)}$ different possibilities. The value of i can vary from 1 to B. The sample space is $(B+1)^N$ as each node in the closed neighborhood of v can take values from 0 to B independently. Hence total probability:

$$P(N,B) = \frac{(N \times \sum_{i=1}^{B} i^{(N-1)})}{(B+1)^N}$$

□

Now consider N flippers in the closed neighborhood of v are executing R_{4a} and R_{4b}. To find expected number of "*round*"s for one of the intermediate flipper to move to DHT-flipper state, we have to find expected number of "*round*"s in which there is only one flipper with unique maximum Pri in the neighborhood.

Theorem 4.3 *If X denote a random variable indicating number of rounds required to find a unique maximum priority in the closed neighborhood of v then $E[X] \leq e$, where e represents Euler-Mascheroni constant (e).*

Proof: For calculating expected number of rounds, we need to determine probability distribution function

$$Pr[X = r] = (1 - P(N,B))^{(r-1)} \times P(N,B)$$

Since this is a geometric distribution, expected number of rounds can be calculated as follows.

$$E[X] = \frac{1}{P(N,B)} = \frac{(B+1)^N}{N \times \sum_{i=1}^{B} i^{N-1}}$$

$$E[X] \leq \frac{(B+1)^{B+1}}{(B+1) \times \int\limits_{0}^{B} i^{B} \, di} = \frac{(B+1)^{B+1}}{B^{B+1}} = \left(1 + \frac{1}{B}\right)^{(B+1)}$$

Note here that the value of N is upper bounded by $(B+1)$. Hence

$$\lim_{B \to \infty} \left(1 + B^{-1}\right)^{(B+1)} = e$$

Therefore $E[X] \leq e$. This result signifies that each flipper needs e moves on average for the transition from $Label = Wait$ state to $Label = NIB$ state. □

Theorem 4.4 Only the following sequence or subsequence of state change is possible for each flipper during execution of the protocol.

(Wait→Swi→ Wait→Swi), *(Wait→Swi→ Wait→NIB),*
(NIB→Swi→ Wait→Swi), *(NIB→Swi→ Wait→NIB)*

Proof: We can see in Algorithm 2 that if a flipper executes R_{4b} then it can not execute any other rule. So no other flipper in its neighborhood can go to $Label = NIB$ state. It can also be shown that if a flipper executes R_{4b} then its neighbors can only execute R_2.

Now from Theorem 4.3 we can say that each node takes expected e moves to go from $Label = Wait$ state to $Label = NIB$ state. Hence the sequences take expected $2 + e$ moves. This is true for each flipper. Therefore, we can conclude $\mathcal{O}(n)$ is the expected number of moves for convergence. □

A new flow initiated during the convergence phase can not be catered by the system due to the unavailability of the NIB-flipper(s). However, we have also shown that, expected convergence time is also within a finite and acceptable bound. Therefore, we can argue that `FLIPPER` is scalable and available most of the time.

4.4.3 `FLIPPER` is Partition Tolerant

A partition tolerant network can function individually and independently, even if it gets partitioned due to link or node failure. As flipper readjustment does not require any bootstrapping, therefore the proposed architecture is partition tolerant. `FLIPPER` requires each switch-flipper to have atleast one DHT-flipper in their neighborhood. Corollary 4.1 ensures this property. Therefore, even the network becomes partitioned due to failure, `FLIPPER` helps them to function individually.

4.5 Analysis of Flipper Performance from Simulation over Synthetic Networks

To evaluate performance of `FLIPPER`, we simulate the proposed method and compared with one standard fault resilient SDN based framework, called `POCO-PLC` [190]. `POCO-PLC` is a distributed SDN platform that uses 20% of controller nodes to provide a "*Pareto optimal*" fault resiliency. The controllers act as NIB also. However, `POCO-PLC` provides an off-line solution of controller

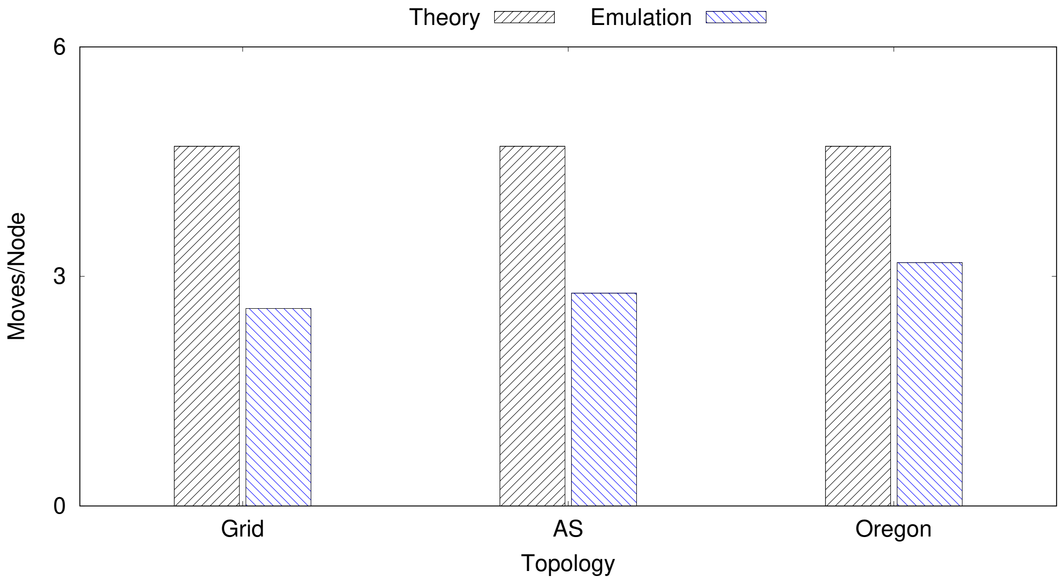


Fig. 4.2: Moves per flipper

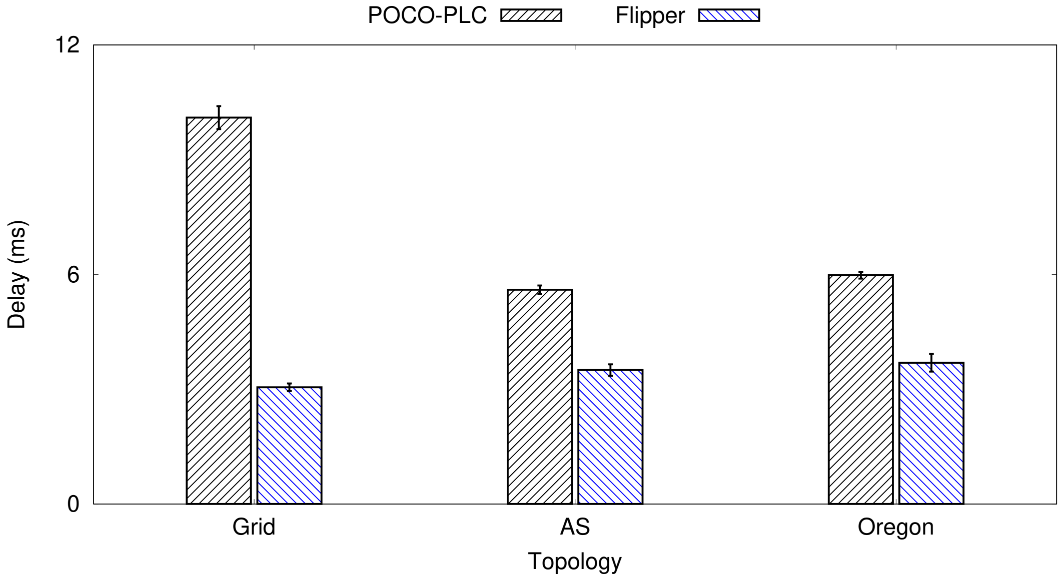


Fig. 4.3: Flow setup delay

placement problem. On the other hand, `POCO-PLC` can handle limited node failure, whereas `FLIPPER` can sustain arbitrary node failures.

4.5.1 Simulation Setup

For simulation we use NS-3.22 [191] network simulator. We use three different topologies. Topology 1 is a synthetic 64×64 regular grid topology. Topology 2 (AS Topology [192]) and Topology 3 (Oregon [193]) are real autonomous system data sets taken from University of Oregon Route Views Project "*BGP*" logs, where each node represents a border router. For simulation purpose, we consider that these border routers are flipper devices. In each case, flippers are connected via 100 Mbps capacity and 2 ms delay "*Ethernet*" links. Each flipper is configured to generate 4 flows/second with 5 Mbps data rate.

4.5.2 Results and Analysis

Fig. 4.2 depicts average number of moves executed by a flipper in case of random number of flipper failures when SS-MIS is used. It shows that simulation results do not exceed theoretical expected bound, which is $2 + e$ (see Theorem 4.4). The number of used DHT-flipper depends not only on the number of nodes but also on the topology itself. We found that, required number of DHT-flippers for the two real dataset does not exceed 30%. This result is optimistic in a sense that, if existing network infrastructure is to be deployed, then $25\% - 30\%$ of total number of flippers are required to act as DHT-flipper for reducing flow set-up delay. Fig. 4.3 presents a comparison between the proposed flipper architecture and the `POCO-PLC` framework in terms of flow setup delay. The `POCO-PLC` framework uses a heuristic Pareto-optimal solution for distributed controller placement. Their work suggests that, delay is Pareto-optimal in case of 20% controller usage in most of the network scenarios. However, Fig. 4.3 shows that, in case of flipper, $5\% - 10\%$ increase in number of controllers can reduce flow setup delay by more than 60% for both of the real networks. Performance improvement in terms of flow setup delay is due to the fact that, each switch-flipper has a DHT-flipper in it's neighborhood.

4.6 Analysis of `FLIPPER` from Emulation over a Testbed

Motivated with simulation results, we emulate our proposed architecture on top of *mininet* [194]. *mininet* creates virtual nodes for emulated environments over a real networking testbed.

Table 4.1: Emulation Topology Properties

Attributes	**Values**	**Attributes**	**Values**
Nodes	50	Edges	136
Avg Degree	5.44	Max Degree	20
Average DHT-flippers	18 ± 1.23	Average flow set-up delay	42.29 ± 7.2ms

4.6.1 Testbed Setup

We have taken a 50 node topology extracted from Oregon dataset [193]. Each node is configured to act a switch-flipper and DHT-flipper with the help of existing OVS [139] and OpenVSwitch database server [195] respectively. The flippers are connected via links of 5 Mbps and 2 ms delay. Link characteristics are configured with Linux "*tc*" utility. Each node generates 4 random Transmission Control Protocol (TCP) flows consuming 5 Mbps of bandwidth each. Rest of the topology characteristics and cumulative results from the emulation are given in Table 4.1. Each flipper periodically checks for states of adjacent neighbors and links within a time period of 20ms. When a DHT-flipper fails, newly appointed DHT-flipper interacts with switch-flippers via "*JSON-rpc*" and gathers flow table as well as link state information. The objective of these experiments is to identify effect of different types of failures on data plane operation. As `POCO-PLC` uses static role assignment, convergence time of the protocol becomes irrelevant. Therefore, we do not compare flipper with POCO-PLC in emulated experiments.

4.6.2 Effect of Node Failure

We select variable number of flippers as candidates for failure. To visualize the effect of mutual separation (in terms of hop counts) between failed flippers, candidate flippers are selected in following two ways. **Experiment 1.a:** The selected flippers are 1-hop away from each other. **Experiment 1.b:** The selected flippers are more than 2 hops distance apart. The chosen nodes are selected carefully, so that there exists at least one path between source and destination of each "*`TCP`*" flow even after the chosen nodes fails. The system can not accept a new flow, until flipper readjustment converges. Therefore, convergence time of the flipper readjustment is significant in case of failure. Convergence time for flipper readjustments are shown in Fig. 4.4. Results suggest that, the effect of multiple flipper failure is dependent on distance of the failed

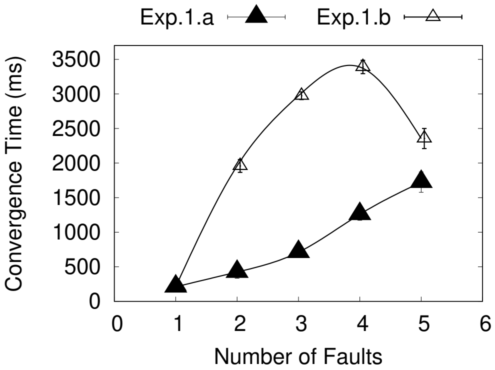


Fig. 4.4: Effect of Flipper Failure: Convergence Time

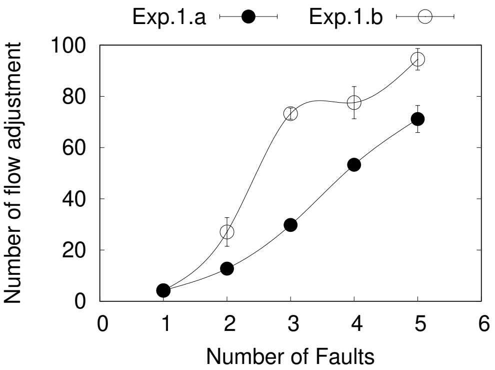


Fig. 4.5: Effect of Flipper Failure: Flow Readjustments

flippers. However, convergence time difference reduces for Experiments 1.a and 1.b at $k = 5$ as diameter of the topology is 5.

Role change of flippers results in path adjustment of flows. Out of generated 200 random flows, Fig. 4.5 shows the number of flows needs to be readjusted due to change in data plane topology. The plot signifies that the number of flow adjustments also depend on the distance between the failed flippers. Higher separation between failed flippers requires large number of role change operations to reach convergence. This results higher number of flow adjustment. The result also shows that, increase in number of flipper failure increases number of flows required to be rerouted.

4.6.3 Effect of Link Failure

To visualize the effect of link failure on data plane operation, we perform similar experiments as mentioned earlier. In this experimental setup, k links are chosen randomly so that there is at least one path between source to destination of each flow. We perform the following two experiments by selecting variable number of k links as following. **Experiment 2.a:** The failed links are 1-hop distance apart and **Experiment 2.b:** The failed links are at least 2-hop distance apart. These selected links are disconnected simultaneously to perform failure experiments. The emulation results shown in Figs. 4.6 and 4.7 reveal that, convergence time and required number of flow adjustments depends on number of failed links and distance between the failed links.

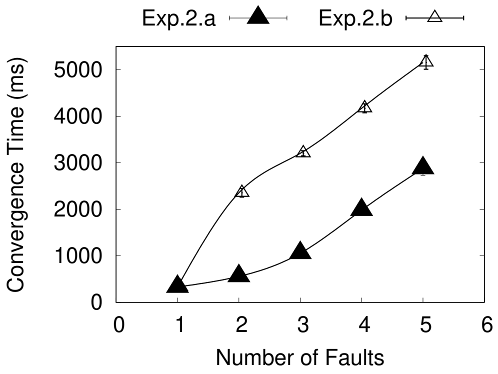


Fig. 4.6: Effect of Link Failure:Convergence Time

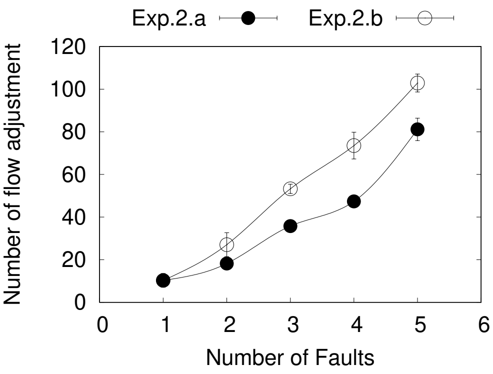


Fig. 4.7: Effect of Link Failure:Flow Readjustments

4.7 Background and Related Works

Traditional "*SNMP*" based for network management systems [99, 196, 197] resulted complex and rigid architectures. [198] shows that, network configurations are highly error prone. Error of configuration happens due to complexity of managing each network devices individually. To reduce network management overhead, SDN came into existence. Some of the popular SDN control plane approaches are, [199, 200, 201, 202, 17]. To ensure scalability, SDN control plane for service provider network needs to be distributed. According to [203], it is not possible to ensure strong consistency, availability and partition tolerance simultaneously in case of distributed control platform. Increase in number of controllers increases scalability and management overhead both [204]. On the other hand, reduction of controllers makes the control plane a bottleneck [205]. Therefore, designing of distributed control platform for service provider network is non-trivial. Although, [44, 18] have proposed distributed control plane, fault-tolerance remains an issue in case of distributed control plane. However, some of the fault resilient distributed control planes are refereed in [206]. Among existing works, `POCO-PLC` [190] proposes a "*`Pareto optimal`*", fault-resilient off-line control plane. Furthermore, designing a fault tolerant SDN network management system is non-trivial due to the fact that selecting a recovery strategy might take longer convergence time. These limitations have motivated us to design a dynamic architecture, which can reduce the flow initiation delay and can provide fault tolerance.

4.8 Commonly Asked Questions

Here we discuss the answer of some questions that may arise while reading this report.

How `FLIPPER` is different from SDN?

Standard SDN platform uses static role assignments at the time of deployment. Static deployment limits performance of SDN in case of topology changing networks. In case of controller failure, SDN might cease to perform. In such cases, `FLIPPER` provides more availability than SDN by utilizing dynamic role assignment of flippers.

Can `FLIPPER` Work in fail-open and fail-close semantic?

Fail-open and fail-close semantics provide partition tolerance in case of fault resilient architecture. Fault-resilience architectures handle specific types of failures. In Section 4.4, we prove that, the proposed `FLIPPER` is fault-tolerant. In a fault-tolerant architecture the effect of failure only affects in terms of delay. Therefore, we argue that `FLIPPER` provides a stronger solution to handle network partitioning problem.

Why our emulation results are not comparable with existing works?

Existing SDN based architectures do not focus on fault tolerance, and most of the cases solutions are off-line and static deployment based. So, they can not handle arbitrary failure. Once the SDN controllers fail, the switches under influence of the controllers can not perform data forwarding until a new controller is configured to work with them. Therefore, the term convergence time becomes irrelevant in that context.

4.9 Summary

In this work, we propose `FLIPPER` which supports SDN like network management and control, while avoiding controller bottleneck problem, and supporting a stronger notion of fault tolerance. Built over the existing `ONIX` architecture, `FLIPPER` supports a scalable notion of dynamic role adaptation based on a distributed self-stabilizing algorithm. Simulation results show benefits of `FLIPPER`, whereas emulation over a real testbed conveys the feasibility of `FLIPPER` implementation over the existing network infrastructure.

the`FLIPPER` improves scalability of the network by exploiting capabilities of LSiN. In our next chapter, we shall see how can LSiN utilize the proposed `FLIPPER` to ensure dynamic network management.

Chapter 5

Aloe: An Elastic Auto-Scaled and Self-stabilized Orchestration Framework for large scale IoT network (LSiN) Applications

5.1 Introduction

In the previous chapter, we have seen how `FLIPPER` can be used to deploy programmable network over an existing LSiN. In this chapter, we explore some of the management challenges of LSiN that can be solved using the `FLIPPER` principle. Since the inception of LSiN, the rapid development and deployment of end-user services have made the architecture difficult to manage. Simultaneously, with the advancement of edge-computing, in-network or In-network

---

Has been published in

[T.4] Subhrendu Chattopadhyay, Soumyajit Chatterjee, Sukumar Nandi, and Sandip Chakraborty. "Aloe: An Elastic Auto-Scaled and Self-stabilized Orchestration Framework for IoT Applications". In: *Thirty Eighth IEEE International Conference on Computer Communications (INFOCOM)*. vol. 38. 2019

[T.5] Subhrendu Chattopadhyay, Soumyajit Chatterjee, Sukumar Nandi, and Sandip Chakraborty. "Aloe: Fault-Tolerant Network Management and Orchestration Framework for IoT Applications". In: *IEEE Transactions on Network and Service Management* (2020). **Accepted** processing (In-network processing), and platform-as-a-service technologies, end-users consider the network as a platform for deployment and execution of myriads of diverse applications dynamically and seamlessly. The use of Software-Defined Network (SDN) has gained momentum over the last decade, where a network manager can monitor, control, and deploy new network services through a central controller. However, management of network services over In-network processing oriented LSiN platform is still challenging even with an SDN based architecture [33].

The primary objectives for supporting In-network processing over an LSiN platform are as follows: (1) The platform must support scalability [35] to cater to "*plug-and-play*" type devices. Furthermore, the system should be agile enough to support rapid deployment of applications without incurring additional overhead [34]. (2) Since LSiN with In-network processing supports micro-service architectures [36], the application services can be divided into multiple micro-services and deployed at different network nodes for reducing application response time with parallel computations. These micro-services may need to communicate with each other. Therefore the flow-setup delay from the in-network nodes needs to be very low to ensure near real-time processing. (3) As the percentage of short-lived flows are high for LSiN based networks [37], reduction in flow-setup delay can significantly improve the performance of the end-user application. (4) Failure rates of LSiN nodes are in-general high [38]. Therefore, the system should support a fault-tolerant or fault-resilient architecture to ensure liveness.

Even though SDN supported In-network processing can solve multiple network management problems; there are certain limitations. First, the SDN controller is a single-point bottleneck. Every flow initiation requires communication between switches and controllers; therefore, performance depends on switch-controller delay. With a single controller bottleneck, the delay between a switch and the controller increases, affecting flow-setup performance. As we mentioned earlier, majority of the flows in an LSiN network are short-lived; the impact of switch-controller delay is more severe on the performance of short-lived flows. To solve this issue, researchers have explored distributed SDN architecture with multiple controllers deployed over the network [207]. However, with distributed SDN architectures, a question arises about how many controllers to deploy and where to deploy them. Static controller deployments may not alleviate this problem, as LSiN networks are mostly dynamic with plug-and-play deployment of devices. Dynamic controller deployment requires hosting the controller software over Commercial off-the-shelf (COTS) devices and designing methodologies for controller coordination, which is a challenging task [208]. The problem is escalated when the objective is to develop a fault-tolerant or fault-resilient architecture in a network where majority of the flows are short-lived.

To alleviate the challenges mentioned above, in this work, an SDN control plane is integrated with the In-network processing infrastructure, such that the control plane can be deployed dynamically over the COTS devices while maintaining a fault-tolerant architecture. We design a distributed, robust, migration-capable, and elastically scalable control plane framework with the help of docker containers [209] and state-of-the-art control plane technologies by exploiting the `FLIPPER` principle. The proposed control plane consists of a set of small controllers, called the micro-controller (μC), which can coordinate with each other and help deploy new applications for In-network processing. The container platform helps in installing these μCs on the COTS devices; a container with an μC can be seamlessly migrated to another target device if the host device fails, yielding a fault-tolerant architecture. In addition to this, deployment mechanism for the μCs ensures elastic auto-scaling of the system; the total number of μCs can grow or shrink based on the number of active devices in the LSiN network. We develop a set of special-purpose programming interfaces to ensure fault-tolerant elastic auto-scaling of the system along with intra-controller coordination. Finally, we design a set of Application Programming Interfacess (APIs) over this platform to ensure language-free independent deployment of applications for In-network processing. Combining all these concepts, we present `Aloe`, a distributed, robust, auto-scalable, platform-independent orchestration framework for edge and In-network processing over LSiN infrastructures.

`Aloe` has multiple advantages for an LSiN framework with In-network processing capabilities. (a) The distributed controller approach ensures that there is no performance bottleneck near the controller. (b) Flow-setup delay is significantly minimized because of the availability of a controller near every device. (c) Fault-tolerant controller orchestration ensures the system's liveness even in the presence of multiple simultaneous devices or network faults. We discuss various trade-offs for `Aloe` deployment and service provisioning performance, based on thorough experimentation and performance analysis under various realistic scenarios. Accordingly, we introduce a resource management module to `Aloe`, which boosts performance under dynamic workload scenarios.

We have implemented a prototype of Aloe using state-of-the-art SDN control plane technologies and deployed the system over an in-house testbed and a 68-node Amazon web service (AWS) platform. The in-house testbed consists of 10 nodes ("*`Raspberry Pi`*" devices) with "*`Raspbian`*" kernel version 8.0. As mentioned, we have utilized docker containers to host the distributed control plane platform. We have tested Aloe with three popular applications for in-network Internet of things (IoT) data processing – (a) A web server (simple "*`python`*" based), (b) a distributed database server ("*Cassandra*"), and (c) a distributed file storage platform ("*Gluster*"). We observe that Aloe can reduce flow-setup delay significantly (more than three times) compared to state-of-the-art distributed control plane technologies while boosting up application performance even in the presence of multiple simultaneous faults.

5.2 Related Work

Traditional single controller architecture is not suitable for LSiN infrastructure, where the network is dynamic and failure prone. One way to address such a problem is to deploy a distributed control plane [185, 210, 152, 211]. Although, some of the previous works [212, 213] have tried to find out placement of distributed controllers in the network to improve scalability of the network, existing distributed control planes are not sufficient for handling LSiN systems that require in-band control. `ONIX` [44] and `ONOS` [18] are two popular distributed control plane architectures. `ONIX` uses a Distributed Hash Table (DHT) data store for storing volatile link state information. On the other hand, `ONOS` uses "*NoSQL*" distributed database and distributed registry to ensure data consistency. Although both of them can scale easily and show a significant amount of fault-resiliency, they require high end distributed computing infrastructure for execution. Deployment of such infrastructure increases cost of LSiN deployment and leads to performance degradation of LSiN services. To tackle high resource requirements, Elasticon [147] uses controller resource pool to enforce load balancing. They also proposed a hand-off protocol for switch controller co-ordination to ensure serializability. However, `Elasticon` is not suitable for failure prone LSiN nodes. Similar problem is also faced by `Kandoo` [124] and [214].

LSiN applications generate short and bursty traffics. To avoid impacts of short flows, `DIFANE` [16] uses special purpose authority switches, which can take localized decisions based on pre-installed wildcard flow entries depending on traffic characteristics and network topology. However, local authority switches creates a problem for global state management of the network. `DevoFlow` [215] tries to solve this problem by proactively deciding wildcarded rules based on global state information. However, the dynamic topology of LSiN platform prevents proactive installation of flow entries. Therefore, although `DIFANE` and `DevoFlow` performs well in case of Data Center Networks (DCN), delivers substandard performance in case of LSiN platforms.

LSiN requires in-band control plane as most of the switches have limited network interfaces. Therefore, disruption in LSiN links impacts severely on multiple LSiN nodes due to disconnection from the control plane. To provide disruption tolerance, `SCL` [216] uses replication of controller applications on strategic places of a network. `SCL` uses a coordination layer inside the switch to provide consistent updates for a single image, lightweight controllers deployed in an in-band fashion. However, use of two-phase commit mechanism for consistency preservation increases higher latency and affects flow setup delay for short flows. Moreover, `SCL` assumes existence of robust channels among switch and controllers, which is not possible in case of low-cost and resource-constrained LSiN platforms. On the other hand, `DIFANE`, `DevoFlow` and `SCL` exploits data plane device capabilities to provide quicker response time. In order to do so, they require special purpose switches which can take decisions locally without requiring controller consultation. Such switch-level modifications which may not be possible for every hardware devices.

To avoid hardware modification, `BLAC` [156] uses a controller scheduling mechanism to dynamically scale the control plane to accommodate need of the system. `BLAC` introduces a scheduling layer to achieve binding less architecture, where all flows from a switch can be dynamically scheduled to one of the many controller instances. Although, `BLAC` reduces switch hand-off issues, increases flow setup time. Therefore, `BLAC` re-introduces performance bottleneck for LSiN short flows.

From the discussion above we can observe that, the existing control plane architectures are not effective in managing LSiN platforms as they pour more focus on consistency of the network state information than availability and partition tolerance of the control plane. However, theoretically it is not possible to achieve all these goals simultaneously [203]. We assume that availability and partition tolerance requires more attention than providing strong consistency for LSiN platforms. Our assumption is grounded upon the fact that, dynamic and failure prone nature of LSiN requires highly available control plane than preserving consistency for volatile short flow information. Our design of `Aloe` is motivated by this observation, and is described in the following section.

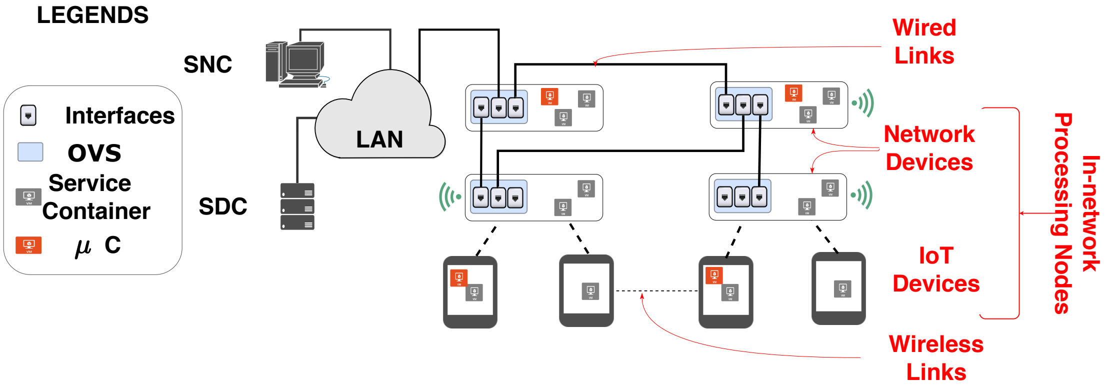


Fig. 5.1: Components of Aloe Infrastructure

5.3 Components of Aloe

The `Aloe` orchestration framework exploits capabilities of In-network processing architecture over an LSiN platform where devices work mostly in a plug-and-play mode. The main components of the architecture are shown in Fig. 5.1. It can be noted here that the proposed architecture does not bring new hardware or software platforms at its base; instead, we utilize available COTS hardware and open-source software suites to design this entire architecture. Our objective is to design an orchestration platform that can be developed with market-available components while integrating innovations in design such that shortcomings of the existing systems can be mitigated. We discuss individual components and their functionalities in this section.

5.3.1 Infrastructure Nodes

Networking equipment and devices are considered as infrastructure nodes. Therefore, nodes are essentially embedded and resource-constraint devices like smart-gateways, smart routers, smart IoT monitoring devices, etc. These devices participate in communication and provide in-network processing platforms for lightweight services by utilizing residual resources. We consider that these nodes are either SDN-supported or can be configured with open-source software platform like Open virtual switch (OVS) to make them SDN capable.

We use containerized platforms like “*`docker`*” [209] to offload services in the LSiN platform for In-network processing. Containerized service deployment helps in supporting service isolation and makes the architecture fail-safe by supporting live migration of containers. Further, containers reduce a programmer's overhead for service delegation and cost of deployment, as the same device can be used for In-network processing of LSiN applications along with execution of custom networking services.

5.3.2 Service Deployment Controller

To identify resource requirement and delegation of services which require In-network processing, we use a centralized Service Deployment Controller (SDC). The SDC periodically monitors resource consumptions of the nodes. Once a new service is ready for deployment in the system, SDC identifies schedules in which services can be executed by the nodes without violating resource demands from individual services. Once the schedule is generated, the SDC is responsible for delegating services based on the schedule. It can be noted that load of an SDC is much less compared to the network management controller. Therefore we maintain a single instance of SDC in our system.

5.3.3 Super Network Controller

Network management in an LSiN platform is non-trivial due to diversified inter-service communication requirements and dynamic nature of the network. `Aloe` uses a two-layer approach. We deploy a high availability Super Network Controller (SNC) [1] at the first layer, which is responsible for storing persistent network information, like routing protocols, quality of service (QoS) requirements, periodicity of statistic collection from nodes, etc. A SNC also manages an Access Control Lists (ACL) to provide necessary security to the infrastructure nodes.

5.3.4 Micro-Controllers (μC

Although super controllers are highly available, an LSiN platform has a time-varying topology due to use of resource constraint devices and devices being plug-and-play most of the times. Therefore, use of a centralized controller cannot achieve fault-tolerance (failure of infrastructure nodes) and partition-tolerance (failure of network links resulting in network partitions). On the other hand, unlike SDC, SNC needs to be consulted by nodes each time a new flow enters the system. This increases the communication overhead and flow initiation delay which also

---

[1] It is possible to use same physical device as SDC and SNC

affects performance of the services deployed in the infrastructure. Therefore, `Aloe` uses multiple light-weight second layer of network controllers named as μC.

μCs are lightweight SDN controllers. A μC stores volatile link layer information of a small group of nodes placed topologically close to it. Thus a μC maintains information consistency by minimizing the delay between the governing μC and the nodes managed by it. The SNC can aggregate these statistics via "***REST***" API queries from the μC. Based on changing QoS of services, network service provisioning can be achieved in the μC via the same "***REST***" API. Based on the configuration of the SNC, a μC collects statistics from individual OVS modules of the nodes. Thus a μC can achieve a fine-tuned network control for infrastructure nodes.

Deployment of μCs in nodes might also create network partitioning issue. To avoid such an undesirable scenario, `Aloe` uses a novel approach where the μCs are encapsulated inside a container and deployed as a service inside the infrastructure nodes itself. Thus `Aloe` supports μC as a Services (μCaaSs) which ensures fault-tolerance of the system. μC containers can be migrated to a target node quite easily with help of live-migration technique of a container when the host node fails. `Aloe` ensures that a set of μCs is always live in the system maintaining requirements for minimized switch-controller delay. On the other hand, a μC container can be customized depending on available capacity of the nodes and resource consumptions by controller applications. It can be noted that this μC architecture is different from existing distributed SDN controller approaches, such as DevoFlow [215] and SCL [216], which require switch-level customization. μCs can run over the existing COTS devices without any requirement for switch-level modifications.

5.4 Design of Aloe Orchestration Framework

This section discusses the `Aloe` orchestration framework by highlighting various functional modules of `Aloe` and their working principles. Finally, we develop a set of APIs for language-independent and robust deployment of applications over the `Aloe` framework. The various functional modules of `Aloe` are shown in Fig. 5.2; the detailed description follows.

5.4.1 `Aloe` Functional Modules

The proposed framework consists of four node-level modules and one SNC-level module. The node-level modules run inside infrastructure nodes and decide topology and service parameters

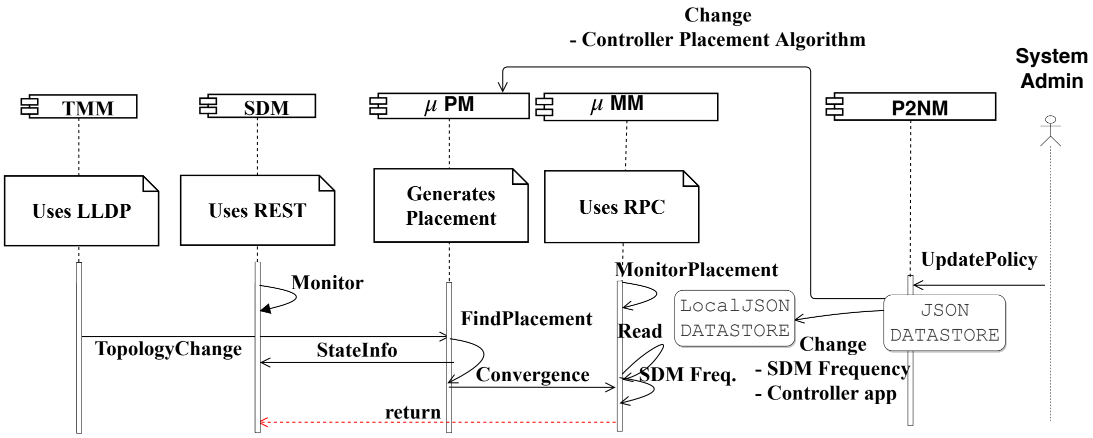


Fig. 5.2: `Aloe` function modules and their interactions

that need to be synchronized across various nodes. These modules collaborate with each other to take distributed decisions in a fault-tolerant way. It can be noted that in `Aloe`, infrastructure nodes are mutable and they can convert themselves as a μC if required. An interesting feature of `Aloe` is that this decision mechanism is executed in a pure distributed way, preserving safety and liveness of the system in presence of faults. The functionalities of various modules are as follows.

a Topology Management Module

We design `Aloe` as a plug-and-play service, where an Aloe-supported LSiN device can be directly deployed in an existing system for flexible auto-scaling support. The Topology Management Module (TMM) initializes the `Aloe` framework on a newly deployed node. Tasks of the TMM are as follows – (i) identify nodes in the neighborhood, and (ii) determine whether an `Aloe` service is running in that node. An `Aloe` service is of two types – (a) μC service, and (b) user application service. To find out active nodes in the neighborhood, TMM uses Link Layer Discovery Protocol (LLDP) which is a standard practice for SDN controllers. We assume that each `Aloe` service deployed in the LSiN cloud uses a unique predefined port address. TMM queries about services in local neighborhood via issuing a "*telnet open port*" requests. Apart from initialization, this module is invoked whenever a node/link failure or μC failure event is detected.

b State Discovery Module

In case of a node or a link failure after initialization through TMM, there is a possibility that infrastructure nodes get disconnected from the μC. To identify such a scenario, `Aloe` maintains various state variables for each node as follows. (i) Controller State (CTLR): This state variable decides whether a node is in a `general` (does not host a μC service), μ`C` (hosts a μC) or `undecided` (an intermediate state between general state and the μC state) state. (ii) Priority (PRIO): This state variable is required only if the node is undecided and denotes priority of the node for becoming a μC. The states associated to nodes are kept and managed by the nodes themselves. However, a node can access a copy of states from its neighbor to decide its state. State Discovery Module (SDM) is responsible for accumulating state information collected from neighbors. SDM uses "***REST***" for this purpose. Once a failure event occurs, TMM invokes the SDM. SDM keeps on executing periodically until the node finds at least one μC in its neighborhood. The periodicity of execution of this module is dependent on link delay. For implementation purpose, we consider periodicity as largest delay observed to fetch data from a neighbor. The above functioning is different than control plane state discovery module which runs inside the μC and keeps track of the network states. In contrast to that, the proposed SDM keeps track of the roles (i.e. acting as μC or not) that a node is playing in its immediate neighborhood. Since this module is inspired by the `FLIPPER`, the role transitions closely matches the state transitions of `FLIPPER` given in Section 4.4

c μC Placement Module

Based on neighbor states collected through SDM, every node independently determines whether it needs to launch a μC service. This is done through the μC Placement Module (μPM). We consider nodes as vertices of a graph where edges determined by connectivity between two nodes, and place μC services to the nodes that form a Maximal Indemendent Set (MIS) on that graph. An MIS based μC placement ensures that there would be a μC at least in one-hop distance from each node, which can take care of configurations and flow-initiations for application services running on that node. As we have claimed earlier and will show in Section 5.6 that the μCs utilized in `Aloe` are significantly light-weight but efficient for performing network and service

Algorithm 3: μPM Controller Placement Algorithm

```
Input: B: Any arbitrary large value greater than maximum degree of the network.
Function Trial():
    // Breaks priority ties
    PRIO ← $\lceil \frac{Rand()}{B} \rceil$;
    return;
Function Neighbor$\mu$C():
    if Another $\mu$C in one-hop neighborhood then
        return true;
    else
        return false;
Function UMPriority:
    // If node has unique maximum priority
    if PRIO of this node > maximum PRIO in neighborhood then
        return true;
    else if PRIO of this node = maximum PRIO in neighborhood then
        return false;
    else
        return None;
Function Main():
    while state change detected do
        if CTLR=general & Neighbor$\mu$C()=false then
            // No $\mu$C in neighborhood
            CTLR← undecided;
            Trial();
            // Initialize priority
        else if CTLR=$\mu$C & Neighbor$\mu$C()=true then
            // Two $\mu$Cs are adjacent
            CTLR← general;
        else if CTLR=undecided & Neighbor$\mu$C()=true then
            // $\mu$C found in neighborhood
            CTLR← general;
        else
            if UMPriority()=None then
                // Executor is not maximum
                continue;
            else if UMPriority()=true then
                // Executor has unique maximum priority, no need for further trial.
                CTLR← $\mu$C;
            else
                // Executor has maximum but not unique priority
                CTLR← undecided;
                // Next round of trial starts
                Trial();
    return;
```

management activities. Therefore the total overhead due to MIS based μC placement is not significant. For identification of a suitable set of μC capable nodes, we develop a distributed randomized MIS algorithm given in Algorithm 3. The novelties of this algorithm are as follows. (1) **Randomized**: The algorithm selects different nodes at different rounds, ensuring that load for μC service hosting is distributed across the network and does not get concentrated on some selected nodes. (2) **Bounded set**: The number of deployed μCs are always bounded based on total number of nodes in the network. (3) **Self-stabilized**: The algorithm is self-stabilized and converges in linear time ensuring fault-tolerance of the system under single or multiple simultaneous faults until complete network partition occurs.[2].

d μC Manager Module

Once a node decides its state through μPM, the μC Manager Module (μMM) initiates μC service on selected nodes and establishes a controller-switch relationship between the μC and nodes with *general* state in one-hop neighborhood. As we mentioned earlier, a μC is initiated as a containerized service over the node designated for hosting a μC by the μPM algorithm. For a node with *general* state, this process may involve changing of controller services from one μC to another μC, which requires reestablishment of the controller-switch relationship. For this purpose, the SDN flow tables need to be migrated from the old μC to the newly associated μC. The flow table migration mechanism is specific to the SDN controller software used, and is discussed in Section 5.5.

e PushToNode Module

Along with fault-tolerance, `Aloe` supports rapid deployment and runtime customization of the system. To implement this feature, we develop PushToNode Module (P2NM). Unlike rest of the modules, P2NM is centralized and/or deployed in the SNC. It provides an interface for monitoring and changing policy level information for the μC at runtime which is useful for system administrators. `Aloe` supported policy level information include (i) ACL, (ii) controller application to be executed in the μC, (iii) routing protocols running in the μC, and (iv) SDM update frequency. Apart from the specified policies, `Aloe` also gives freedom to its user to customize the `Aloe` modules itself. This feature is achieved by developing a set of APIs as discussed next.

5.4.2 Application Programmer's Interfaces

The primary objective of this orchestration framework is to deploy the "*controller as a service*" to the in-network processing infrastructure in form of a μC. There are some significant differences

---

[2] These properties are inherited from the design of `FLIPPER`, and the proofs of these properties are provided in the Section 4.4

between a user application service and a μC service, which makes deployment of the later non-trivial. Unlike many user application services, performance of management is dependent on topological position of the μC services. A location transparent deployment of μC might allocate all μCs in the same node if the node has sufficient resources. Such placement can degrade network performance of the infrastructure. However, our placement algorithm is not an optimal solution. Therefore, during the design of `Aloe`, we consider extendibility of this work. Many of the implemented functionalities of this framework can be reused as API for distributed controller application development. For ease of understanding, we only provide the python sample programs here. However, all APIs can be invoked as "*bash*" shell commands over SNC using P2NM.

a Topology Monitor

Using this API, `Aloe` can detect a topology change event (`TopologyMonitor()`) and take actions accordingly. This API can also be used for general purpose routing application, as given in the following code.

```python
''' Find shortest path between dpidS and dpidD'''
G=TopologyMonitor()
path_dpid_list=FindShortestPath(G,"delay")
```

Listing 5.1: Topology Change Detector

b Distributed State Inspector

We develop this API to observe the state of the nodes (`getNeighborStates()`), which helps in developing new placement algorithms for μPM. This API relies on a remote procedure call (`rpc`).

```python
 ''' Find max priority amongst neighbor'''
 states=getNeighborStates()
 maxPrioUndecided=max([v["Prio"] for v in states.values() if state["CTLR"]=="
    undecided"])
```

Listing 5.2: Distributed State Inspector

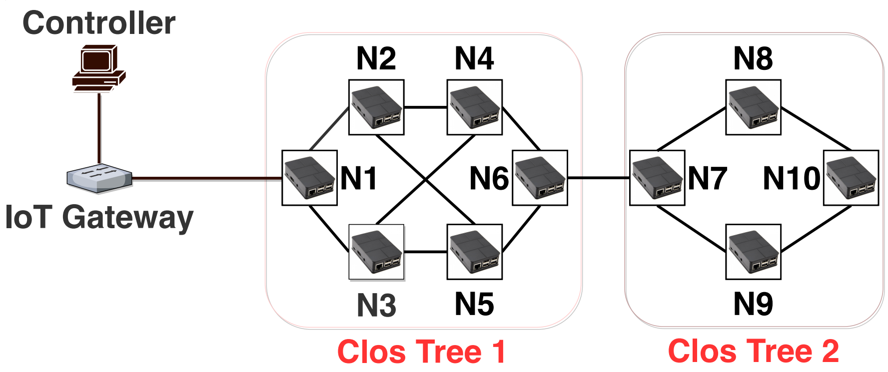


Fig. 5.3: Testbed:Topology

c Find Node Services

The framework requires to identify the deployed services (`getNeighborServices()`) to enforce service level policies. We provide a python API to ease this task. The following example can be used for selective service blocking ACL.

```
'''Service blocking (ACL)'''
services=getNeighborServices()
bport=blocking_port
if(blocking_port in service["dpid"]):
 Execute("ovs-ofctl add-flow match:src=dpid,tcp_port=bport action:drop")
```

Listing 5.3: Find Node Services

Next, we discuss details of the `Aloe` implementation as a general orchestration framework.

5.5 Aloe Implementation

We have implemented `Aloe` as a middleware over Linux kernel with the integration of open-source technologies, like docker containers, various SDN controllers, and REST API (REST) based communication modules. We first discuss the implementation environment that we utilized, followed by a brief description of two different implementation aspects.

Fig. 5.4: Testbed:In Action

5.5.1 Environmental Setup

To analyze performance of `Aloe`, we have deployed an in-house testbed using the topology given in Fig. 5.3 (Fig. 5.4 shows the live-snapshot of the testbed). Our deployed testbed follows clos tree based topology and spans across two different sites to resemble the topology given in [217]. The nodes in the testbed are Raspberry Pi version 3 Model B, which are configured with Raspbian 8.0 operating system with kernel version 4.4.50-v7+. The nodes are connected via multiple 100Mbps USB-to-Ethernet adapter-edges representing physical Ethernet links among the nodes. We use Linux "*`tc`*" to configure each link to use 5Mbps of bandwidth and added 100ms of propagation delay to match real life LSiN deployment specification. Further, to analyze scalability of `Aloe`, we have also deployed `Aloe` in a large-scale 68-node testbed using AWS. For this purpose, we consider a sub-topology from "*`rocketfuel`*" [218] topology which consist 68 nodes. The nodes in the topology are deployed using 18 AWS "*nano*" instances (1 vCPU and 512 MB RAM) and 50 AWS *micro* instances (1 vCPU and 1 GB RAM). The AWS nodes are configured with Ubuntu 16.10 operating system with Debian kernel version 4.4.0. To emulate edges between the nodes, we use the "*`l2tp`*" between the AWS instances. Every infrastructure node, both in the testbed and in the AWS, are configured with "*`OVS`*".

5.5.2 Implementation Aspects

Here we discuss two important implementation aspects of `Aloe` – (i) flow-table consistency preservation during μC migration, and (ii) choice of controller service for μC implementation.

a Migration of μC and consistency preservation

A change in a policy level parameter requires a migration of the flow tables from the old μC instances to new instances of the μC. Similarly, after a μPM execution, there might be a need for change in node-controller association. To implement such functionality, we have implemented a "***`rpc`***" and "***`REST`***" based API (`changeCtlr()`) which can dynamically change a switch's allegiance towards a μC. `changeCtlr()` forces the node to invoke a "***`controller re-association request`***" to the target μC with its previous μC address. After receiving a "***`controller re-association request`***", the target controller invokes migration of flow entries from the previously assigned μC. During the migration procedure, it is important to keep track of the previous state informations. To ensure consistency, `Aloe` preserves snapshots of the μC flow table entries by sending "***`REST`***" queries to the μCs before the migration process starts. To make the migration process lightweight, the container instance is not transferred from one node to another node; instead, the source node container is terminated, and a new container is invoked at the destination node via "***`rpc`***". In case of a network partitioning between the previous μC and the target μC, the target μC obtains the copy of flow table from the requester node itself. In this way, the μMM preserves weak consistency [3] in the system.

b Choice of controller service for μC

Efficiency of `Aloe` is dependent on efficiency of choice of a controller service for the μC. Deployment of a heavy-weight controller can over-consume resources of the nodes; moreover, one μC is only responsible for managing a small set of nodes. Therefore, we target to opt for a light-weight μC for `Aloe`. In order to identify a suitable controller platform for μC, we compare a set of existing SDN controller services like Open Day light (ODL) [219], `ONOS` [18], `ryu` [220] and `Zero` [221] in our in-house testbed in terms of theirs resource utilization. Amongst these controllers, `ONOS` requires high memory consumption ($> 500MB$) which creates an instability in the docker environment. Further, we have observed that approximately 32% times, `ONOS` fails to execute in the testbed nodes due to unavailability of sufficient virtual memory. Therefore, we report performance of the controllers other than `ONOS`. The performance is reported based on three major system parameters – CPU utilization, memory utilization and CPU temperature variation. In Fig. 5.5a, we provide comparison of performance of the competing controllers in

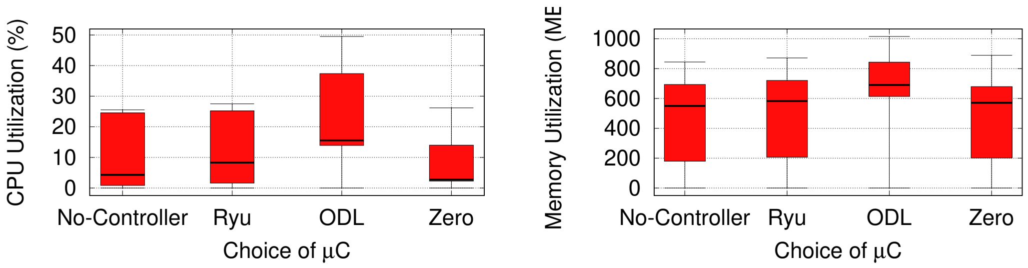
---

[3] `https://en.wikipedia.org/wiki/Weak_consistency`

Table 5.1: Wilcoxon Rank Sum Test (↑indicates μC in top header consumes less resources, ←indicates μC in left header consumes less resources, **X** indicates the choice is undetermined)

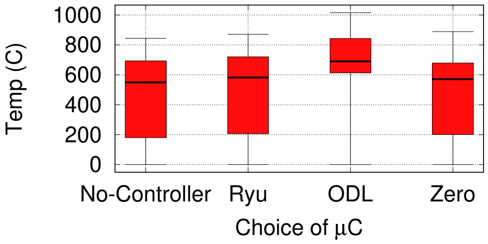
CPU				
μC	**No μC**	**Ryu**	**Zero**	**ODL**
No μC	■	←	←	←
Ryu	< 0.0001	■	↑	←
Zero	< 0.0001	< 0.0001	■	←
ODL	< 0.0001	< 0.0001	< 0.0001	■
Memory				
No μC	■	←	**X**	←
Ryu	< 0.0001	■	**X**	←
Zero	> 0.03	> 0.01	■	←
ODL	< 0.0001	< 0.0001	< 0.0001	■
CPU Temperature				
No μC	■	**X**	←	←
Ryu	> 0.39	■	↑	←
Zero	< 0.0001	< 0.0001	■	↑
ODL	< 0.0001	< 0.0001	< 0.0001	■

terms of CPU utilization. We observe that approximately 30% "`ODL`" μCs use more than 30% of the CPU utilization. In comparison to that, around 40% "`Zero`" μCs use 15% of the CPU utilization. In Fig. 5.5b, we observe that almost 80% "`ODL`" μCs use more than $600MB$ of memory space. All other controllers show lower memory utilization. Fig. 5.5c shows variation in CPU core temperature while executing different types of controller services. The consolidated pair-wise comparison of controllers are provided in Table 5.1 (upper right triangle in blue color). The notation **X** signifies that difference cannot be ascertaind. On the other hand an upper arrow (↑) suggests that μC listed in top header consumes less amount of resources. We use ←to denote higher efficiency of the μC application mentioned in the left header. To determine difference, we perform a statistical hypothesis testing using non-parametric, one-tailed Wilcoxon rank sum test ($\alpha = 0.01$) [222]. Our alternative hypothesis assumes that mean resource consumptions and core temperature is higher than the one in normal case. Left lower triangular part of Ta-



(a) CPU Utilization

(b) Memory Utilization



(c) CPU Temperature Variation

Fig. 5.5: Resource Utilization Comparison of Controller Applications

ble 5.1 (given in green color) signifies the p-values obtained from the rank test. Based on our experimental results we can observe that, "*Zero*" can provide better performance in terms CPU and Memory utilization as "*Zero*" is built upon micro-kernel architecture. Therefore, we use "*Zero*" as our choice of μC in both testbed and AWS.

With this implementation, we evaluate performance of `Aloe`, as discussed in the next section.

5.6 Evaluation

We have tested performance of `Aloe` with three different categories of standard applications which are common and useful for an LSiN based platform – (i) "*HTTP*" service (Python "*SimpleHTTPServer*"): used for bulk data transfer via web clients, (ii) distributed database service ("*Cassandra*"): for data-driven applications, and (iii) distributed file system service ("*Gluster*"): used for file sharing and fault-tolerant file replication over a distributed platform. We further compare performance of `Aloe` with `BLAC` [156], a distributed SDN control platform. To emulate realistic fault models in the system, we have injected faults using Netflix *Chaos Monkey* fault injection tool.

Table 5.2: Wilcoxon Rank Sum Test conclusions with *p*-values over response time of different applications:
X=Inconclusive, ✓=*Aloe* better, ●=In band better

Services **Failure**	**Cassendra**	**HTTP**	**Gluster**
0	**X (>0.11)**	**X (>0.20)**	**●(<0.0001)**
1	**✓(<0.0001)**	**X (>0.04)**	**●(<0.0001)**
2	**✓(<0.0001)**	**✓(<0.0001)**	**X (>0.39)**
3	**✓(<0.0001)**	**✓(<0.0001)**	**✓(<0.01)**

We have taken the measurements under all possible link fault combinations [4] in the testbed and 100 different random fault combinations in AWS.

5.6.1 Application Performance

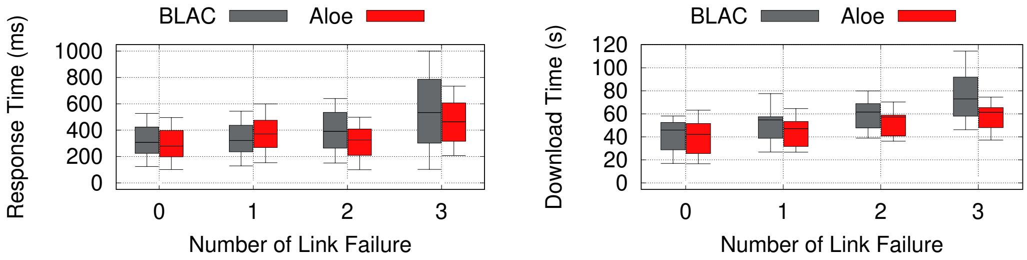
Fig. 5.6b compares download time of a 512MB file hosted using "*`HTTP`*" service under the influence of both `BLAC` and `Aloe` over the in-house testbed. The results are obtained by varying all possible source-destination pairs in the topology. We observe that, even though `Aloe` results in higher download time compared to `BLAC` when there is no failure in the system, performance improves rapidly in presence of link outage. While injecting failure, we observe that approximately 30% connections are timed-out while operating under governance of the BLAC controller. However, `Aloe` reduces such flow termination[5] (< 5% connection time-out for `Aloe`). To compare differences of the nature of the results for each service, we performed a Wilcoxon rank sum test. The *p*-values and conclusion from the test is summarised in Table 5.2.

Fig. 5.6a shows response time of "*`Cassandra`*" search queries. Here, we observe a significant difference in characteristics of the plots due to nature of the service. Unlike "*`HTTP`*", "*`Cassandra`*" utilizes short flows to fetch query results. Therefore, we observe that `Aloe` provides a significant improvement in query response time. However, in case of "*`Gluster`*", `Aloe` performance is marginally poor compared to `BLAC` until there are 3 link failures (Fig. 5.6c).

---

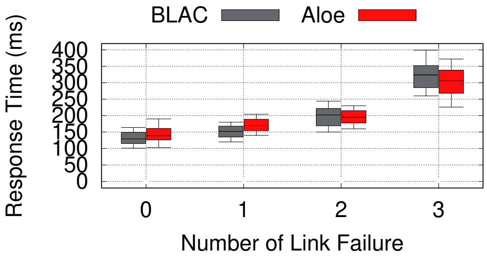
[4] A node failure is equivalent to simultaneous failure of multiple links. Therefore, all possible link failure automatically covers node failure scenarios.

[5] Only in such cases, where the network is partitioned



(a) Cassandra

(b) HTTP Server

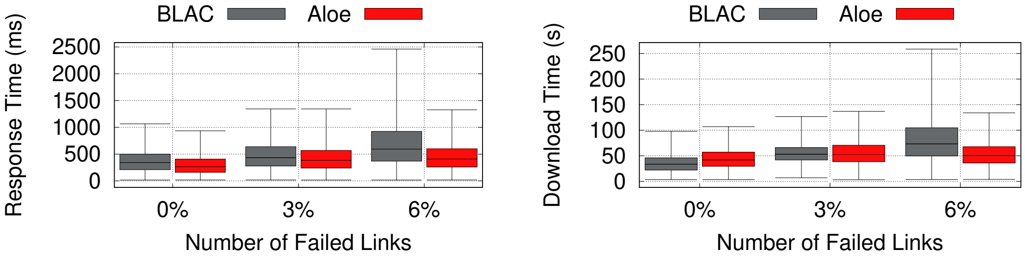


(c) Gluster

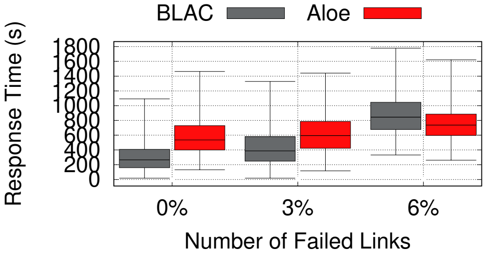
Fig. 5.6: Comparison of response time of services obtained from testbed: Average percentage improvement of `Aloe` – (a) "*HTTP*" server: 4% (0-fail), 11% (2-fails), 21% (3-fails), (b) "*Cassandra*": 9% (0-fail), 26% (2-fails), 37% (3-fails), (c) "*Gluster*": -8% (0-fail), 0.1% (2-fails), 6% (3-fails)

"*Gluster*" flows are short-distant flows, usually within one-hop. Flow-setup delay is almost negligible for a one-hop flow. Therefore the μC deployment overhead of `Aloe` is more when the number of failures is less.

Similar behaviors are observed in the large-scale deployment of `Aloe` in the AWS cloud. In Figs. 5.7b and 5.7c, "*HTTP*" and "*Gluster*" response times show similar characteristics as observed in the testbed. In the case of "*Cassandra*" (Fig. 5.7a), all the cases perform significantly better than `BLAC`. From these observations, we conclude that `Aloe` performs significantly better for services that generate long-distant mice flows (like database synchronization). For a long-distant flow, flow setup delay is high, which gets further affected by link failures. As a consequence, `Aloe` performance is better for failure-prone systems, like LSiN clouds, as the flow-setup delay gets increased with the recovery time due to a failure.



(a) Cassandra (b) HTTP Server



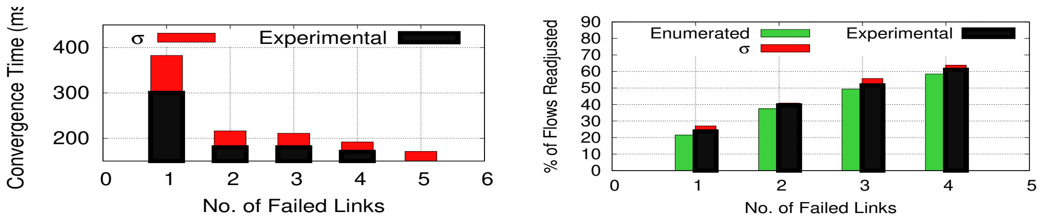
(c) Gluster

Fig. 5.7: Comparison of response time of services obtained from AWS cloud: Average percentage improvement of `Aloe` – (a) "*`HTTP`*" server: -2% (0-fail), 0.1% (2%-fails), 34% (6%-fails), (b) "*`Cassandra`*": 20% (0-fail), 21% (2%-fails), 34% (6%-fails), (c) "*`Gluster`*": -12% (0-fail), -6% (2%-fails), 14% (6%-fails)

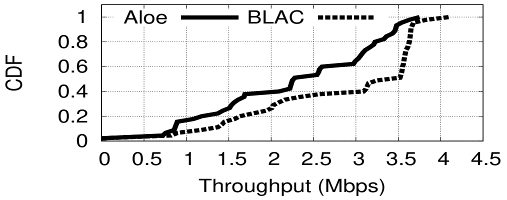
5.6.2 Dissecting `Aloe`

`Aloe` flow-setup time is dependent upon convergence time of μPM and path restoration time. Fig. 5.8a shows distribution of average convergence time of `Aloe` in presence of failure. We have an interesting observation here that as number of simultaneous failures increases, convergence time drops. This can be explained as follows. Let us consider two different faults. If the two faults are at two different sides of the network, then two waves of μPM starts executing simultaneously from two different ends of the network. These two waves get diffused in the network and meet in the middle of the network at convergence. That way, multiple faults create multiple such μPM waves in the network in parallel, and as these individual waves need to deal with a smaller part of the network, they converge quickly.

The convergence phase is followed by path restoration due to a change of controller positions in case of a failure. To identify the performance of path restoration, we measure average number



(a) Avg. `Aloe` Convergence Time (b) Avg. No. Flow Restoration

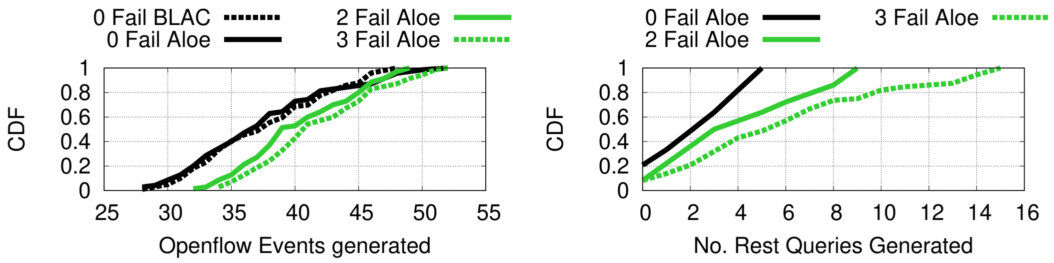


(c) Throughput comparison

Fig. 5.8: Testbed: Effect of failure on `Aloe` performance (σ= standard deviation)

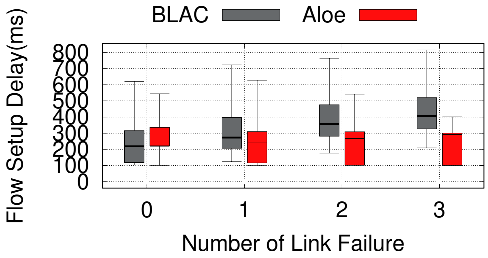
of flow adjustments done by the framework. Number of flow adjustment depends on the topology and number of flows passing through the failed links. Therefore, to compare the result, we provide enumerated number of flow adjustment required for all possible cases of link failures in Fig. 5.8b. We observe that the experimental observations closely match with the results obtained by enumeration. Further, these metrics have a direct impact on flow-setup delay. To understand effect of these factors, we compare flow-setup delay for `BLAC` control plane and `Aloe`. To identify flow-setup delay, we use "*ping*" to transfer a single "*ICMP*" packet. Fig. 5.9c shows that although `Aloe` marginally increases flow setup-delay in absence of a failure, it provides quick flow-setup when multiple faults occur in the network.

We observe that overhead of distributed μC in `Aloe` is responsible for increase in flow-setup delay during no-failure scenario. However, it is difficult to compare exact overhead of the `BLAC` control plane and Aloe due to differences in nature of overhead. We measure overhead with respect to two different factors. Fig. 5.9a shows comparison between `BLAC` and Aloe regarding the number of "*openflow events*" generated over a period of 100s. `Aloe` additionally generates "*REST*" queries to support inter-controller communication, therefore it has more number of openflow events compared to `BLAC`. Fig. 5.9b depicts number of "*REST*" queries generated in `Aloe`. During failure events, `Aloe` μC may need to migrate from one node to another. Fig. 5.10c



(a) Control Plane Overhead

(b) "*REST*" queries generated in `Aloe`

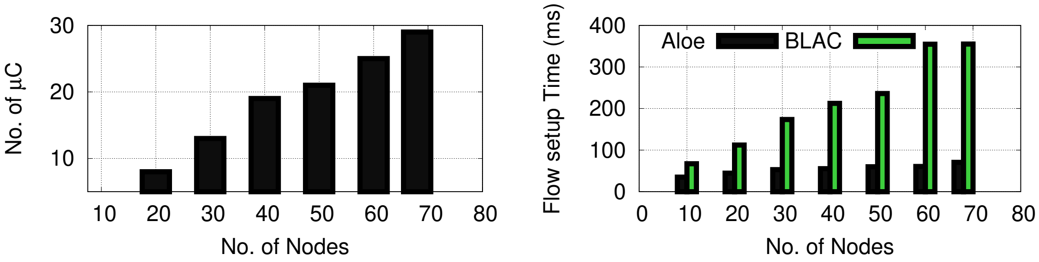


(c) Flow Setup Delay:Measured using "*ping*"

Fig. 5.9: Testbed: Comparison of `Aloe` overhead and Flow setup delay

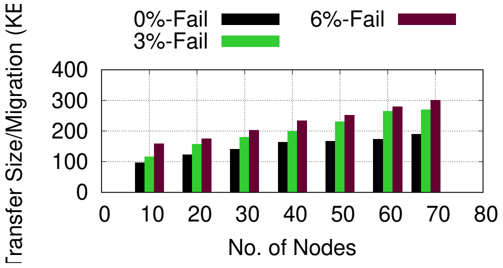
shows data transfer overhead required for migration, which is in the order of a few KB. As number of nodes in the LSiN environment are increased, number of flow table entries are also increased. Therefore, transfer size per migration also increases when the number of nodes are increased. Size of the flow table entries also increases with more number of failures in the network, which introduces some of redundant flow entries ("zombie flows"). However, we observe that, effect of redundant flows has marginal effect when number of nodes in the system are significantly high. Due to these overheads, `Aloe` incurs higher communication overhead than the BLAC control plane. However, due to significant reduction in flow-setup time, `Aloe` ensures better flow throughput than `BLAC`, as shown in Fig. 5.8c.

Although `Aloe` incurs communication overhead, `Aloe` ensures a significant drop in average flow-setup delay. To limit flow-setup delay, `Aloe` provides elastic auto-scaling by increasing the number of μC instances to guarantee that each node can find a μC in its neighborhood. Fig. 5.10a shows the average number of μC instances when the network scales, as obtained from the AWS implementation. Effect of elastic auto-scaling is shown in Fig. 5.10b which indicates that flow-setup delay only increases marginally in comparison to the BLAC controller, which



(a) No. of μC

(b) Avg. Flow Setup Time



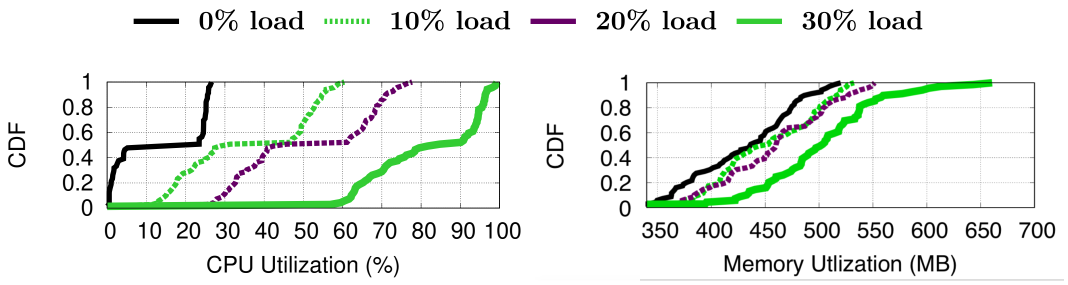
(c) Controller Migration Overhead

Fig. 5.10: Effect of Scaling `Aloe` μC Deployments

incurs a significantly high flow-setup delay as number of nodes in the network increases.

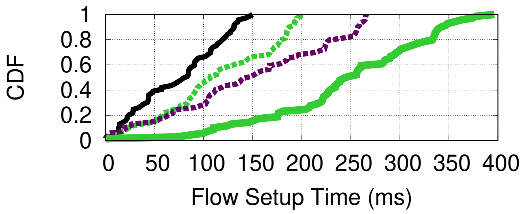
5.7 Aloe Performance Optimization

`Aloe` μCs are deployed over existing network infrastructure (host devices) which may have their own workloads due to application services deployed in them; we call them as application workloads of host devices. We perform a pilot study to check impact of application workload of the host devices on `Aloe` performance. We use the same AWS cloud-based deployment of `Aloe` as discussed earlier. Fig. 5.11 shows impact of application workload on `Aloe` performance. To increase application workload of the host devices, we use `stress-ng` [223] tool. During the experiments, memory and CPU utilization of the host devices have been increased from 0% to 30% of the actual capacity. When application workload of the host devices are increased, the system resources get over-utilized due to thrashing. Therefore, system performance reduces aggressively as the μC receives less CPU time. Additionally more swap events are generated which increases flow setup time, as we observe from Fig. 5.11c. These observations confirms that, `Aloe` μCs require special attention in terms of resource reservation for severely loaded systems.



(a) CPU Utilization

(b) Memory Utilization



(c) Flow Setup Time

Fig. 5.11: Effect of Application Workload of the Host Devices on `Aloe` Performance

We accordingly develop a resource management framework for `Aloe`, which is discussed next.

5.7.1 Effect of Resource Reservation

Reserving resources for μC applications ensures QoS in terms of flow setup time. To optimize performance of the system, resource reservation must match resource demand of the μC. However, resource demands of a μC at a particular time depends on the amount of flows managed by that μC. Therefore, we assume that resource demand of a μC follows a temporal pattern and depends on network state of the LSiN infrastructure. Although over-provisioning resources to the μC improves the QoS, it might affect the primary workload of the host devices; therefore, it can have negative impact on overall application performance. Consequently, we implement a Resource Management Module (RMM) based on "*Monitor-Forecast-Adapt*" strategy which gets executed in each μC to balance μC resource demands and primary workloads of the host devices. Fig. 5.12a shows the components of `Aloe` RMM which consists of three sub-modules. a) Resource Monitor (RM) periodically collects usage statistics of the μC and stores it in a "*`JSON`*" data-store. b) *Usage Estimator* periodically analyzes time-series of resource usage pattern of the μC and predicts probable resource demand for the next time period. c) Resource

Enforcer (RE) is responsible for actually resource reservation for the μC based on the predicted resource demand.

a Prediction of μC Resource Demands

For prediction of resource demands, it is important to identify distributions of resources which depend on the flow arrival pattern. However, in practice, it is difficult to estimate flow arrival distribution for a LSiN platform with heterogeneous applications executing in it. Therefore, we choose a forecasting model based on characteristics of the IoT applications. We focus on two basic characteristics of the IoT applications. (i) IoT applications generate bursty and short lived flows [37]. The bursty and short living nature of LSiN flows reveal that, flow arrival rates per μC during a discrete time interval is cyclic[6]. (ii) These characteristics also suggest that, flow arrival rates follow a non-stationary property[7] . Therefore, we use Autoregressive Integrated Moving Average (ARIMA) [225] model for forecasting of individual resource requirements. ARIMA relies on mean reversion principle of non-stationary data to forecast future strategy based on the time series by employing autoregression.

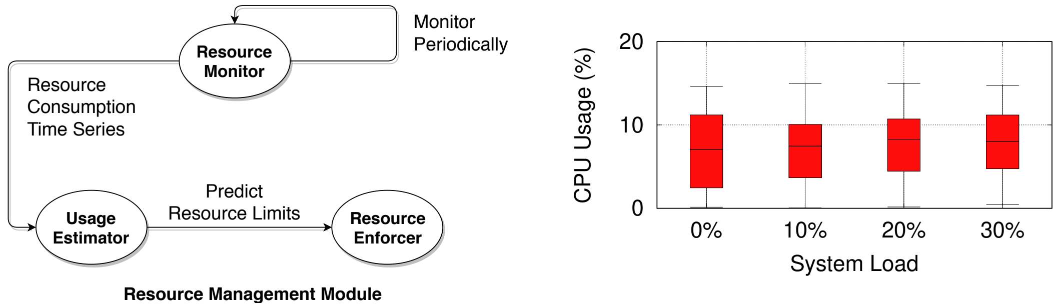
b Performance Improvement with `Aloe` RMM

We have integrated the RMM module with `Aloe` and tested it over the AWS platform as discussed earlier. Like many statistical modeling methods, identification of parameters for ARIMA is a non-trivial challenge. Therefore, we use auto-ARIMA [225] based on our experimental observations to individually forecast the CPU and memory demands according to the resource utilization time series. Fig. 5.12b shows the amount of CPU reserved for the μC in various load scenarios remains almost constant. Fig. 5.12c shows the memory reservation of μC application due to resource reservation module. From results we can observe that, memory reservation amount increases in case of 30% load. The reason behind this observation lies in the OVS to μC mapping technique used in `Aloe` μMM. An OVS chooses a μC based on how quickly the μC responds to its join request. As the system load increases, a lightly loaded μC is more likely to provide a quick response time. Therefore, lightly loaded μCs are likely to get connected with more switches with high number of flows passing through those switches, which in turn increases memory overhead of those μC. Next we check how accurate the RMM prediction model can

---

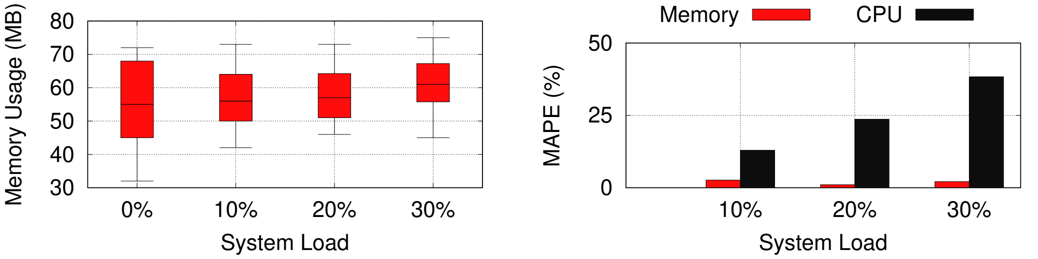
[6] "A cyclic pattern exists when data exhibit rises and falls that are not of fixed period" [224].

[7] "The properties of non-stationary series (viz. mean, variance and co-variance) are functions of time" [224]



(a) Resource Management Module

(b) CPU reservation



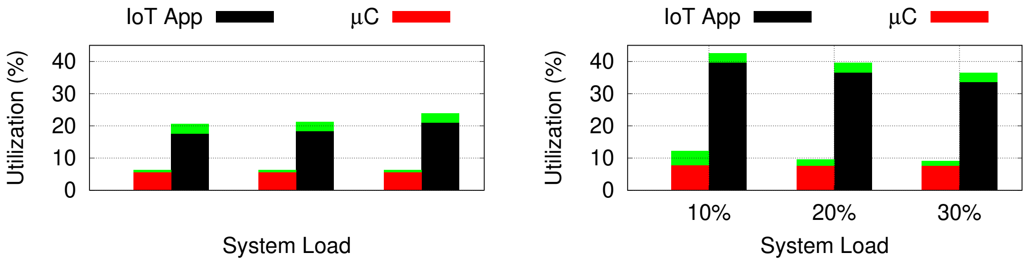
(c) Memory reservation

(d) MAPE Score

Fig. 5.12: Resource Reservation for `Aloe` μCs

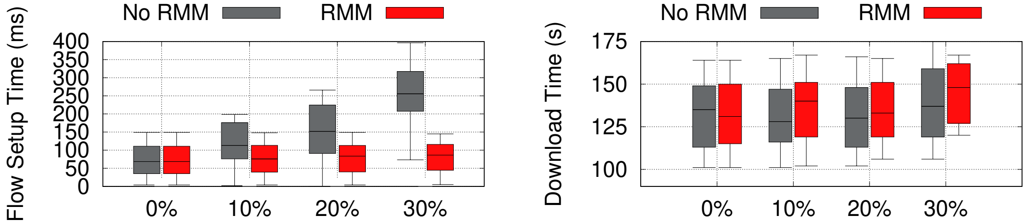
perform based on Mean Average Percentage Error (MAPE). Fig. 5.12d revels that, the proposed RMM provides significantly low MAPE for prediction of memory. We observed frequent fluctuation of CPU during our experiments which the underlying ARIMA can not predict always. Therefore, higher MAPE is observed (Fig. 5.12d) for CPU utilization prediction. Due to this behavior performance of the IoT applications are also influenced. Figs. 5.13a and 5.13b compare resource utilization between the μC and IoT application in terms of CPU and memory. From Fig. 5.13a, we observe that accuracy of RMM does not significantly affect performance of memory utilization by IoT applications. However, reduced accuracy of used ARIMA sometimes over provisions more CPU time to the μCs. As a result the IoT application receives less CPU time than its demand in such cases.

Interestingly, the IoT application (like "*GlusterFS*") shares more host resources than the μC; therefore, a slight resource biasing towards μCs improve their performance significantly, while having marginal impact on performance of the IoT application. We present this observation in Fig. 5.14a, where we compare performance of the RMM in terms of flow setup time with that of no-RMM. The results justifies that the RMM ensures low variations in flow setup time as opposed to the no-RMM case. In fact use of RMM can significantly improve performance of IoT short flows by reducing average flow set-up time by 13% – 120% in various load scenarios. However, due to resource reservation of μC, the application may suffer due to insufficient resources. To understand this effect, we compare performance of the "`GlusterFS`" application before and after implementing the RMM in Fig. 5.14b. We find that, increase in mean download time due to effects of RMM while downloading a $25MB$ file using "`GlusterFS`" varies between 2% – 7% for different load conditions which is considerably small.



(a) Memory Utilization (b) CPU Utilization

Fig. 5.13: Effects of Resource Reservation



(a) Flow setup time (b) GlusterFS file download time

Fig. 5.14: Effects of Resource Reservation for `Aloe` μCs

5.8 Summary

In this work, we present `Aloe`, an orchestration framework, for LSiN which utilizes In-network processing infrastructure for ensuring fault-tolerant network management. `Aloe` uses docker container to support lightweight migration capable in-band controllers. This design choice helps `Aloe` to provide elastic auto-scaling while keeping flow setup time under control. `Aloe` provides controller as a service to exploit in-network processing infrastructure and supports fault and partition tolerance. The performance of `Aloe` has been tested thoroughly and compared with existing controller scheduling framework. The results indicate a significant improvement in response times for distributed LSiN services. In the next chapter, we shall see how the proposed `Aloe` framework can be augmented to cater to various networking services to a diversified set of traffics.

Intentionally Left Blank

Chapter 6

Amalgam: Distributed Network Control With Scalable Service Chaining

6.1 Introduction

In the previous chapter we proposed `Aloe` which provides Software-Defined Network (SDN) controlled network management for dynamic large scale IoT network (LSiN). This chapter we extend the proposed `Aloe`, to provide support for Service Function Chaining (SFC) and traffic steering problem in LSiN. LSiNs serves hundreds of heterogeneous types of sensors and supports millions of devices. Apart from core networking services like topology discovery, path management, quality of service (QoS) management, management of LSiN ecosystems require various network services like Network Address Translation (NAT), firewall, proxy, local Domain Name Server (DNS), etc.; these network services are called Network Functionss (NFs). Depending on network management policies, the application messages require steering through an ordered set of NFs known as "*NF service chain*" [41]. NFs are generally deployed using Virtual

---

Has been published in [T.6] Subhrendu Chattopadhyay, Sukumar Nandi, Sandip Chakraborty, and Abhinandan Prasad. "Amalgam: Distributed Network Control With Scalable Service Chaining". In: *Nineteenth IFIP Networking Conference (IFIP Networking).* 2020

Machiness (VMs) to provide service isolation and reducing capital and operational expenditure by multiplexing the same hardware resources; therefore, they are termed as Virtual Network Function (VNF) [42, 43]. VNFs execution require computation platform to host the VM and execute the NF within the VM.

Researchers have explored various architectures to execute VNFs over a network infrastructure [42, 226]; majority of them rely on SDN [227] to steer flows from one VNF to another. However, for a large-scale network spanning across multiple administrative domains, SDN-based service chaining falls short in several aspects such as-

(a) Lack of scalability: Existing SDN assisted VNF placement and service chaining approaches [228, 229, 227] use a central controller to monitor resource usage statistics of individual devices associated to the platform. Based on resource usage, VNFs is deployed in the actual devices. The use of a central controller for VNF deployment becomes challenging when the network spans across multiple autonomous administrative domains interconnected through different network service providers. To apply SDN assisted VNF placement, administrative privileges across all autonomous domains is required, which is neither scalable nor feasible.

(b) Problem of maintaining state consistency for dynamic service chaining: Existing scalable distributed VNF placement methods [230] and IP based traffic steering proposals [231] are not suitable for dynamic service chaining where VNFs can be added or removed to/from a service chain without terminating the flow. Dynamic service chaining aims to reduce the end host overhead and provides reliability. For example, depending on the flow behaviour perceived by the deep packet inspector (DPI) VNF, a load-balancer VNF may be injected in the service chain without terminating the flow. The SDN control plane needs to keep track of the internal states of all the VNFs to implement the dynamic service chaining over distributed SDN. In a distributed environment, it is challenging to preserve the consistency of the states.

(c) Issues of flow monitoring over multi-administrative platforms: To steer traffic through proper service chains, SDN requires monitoring of flows. SDN flow identification methods using packet header fields are insufficient when there exists a VNF that modifies the packet headers (e.g., NAT, Load balancer, Proxy, etc.). Therefore, existing SDN-based flow monitoring schemes [229, 232] utilize "*vlan/mpls*" tagging. However, in a multi-administrative platform, each autonomous system may use the tags for different purposes that may not be controlled due to the lack of administrative privileges. Therefore, the tag-based monitoring approaches are not suitable for multi-domain platforms.

To avoid the issues of SDN based service chaining, "*session-based service chaining*" [233, 234, 231] has been explored in literature. In session-based service chaining, the VNF placement and flow steering decisions are taken by the end hosts, which reduces the complexity of VNF state management through a centralized authority. However, session-based service chaining can not guarantee QoS; since it can not monitor all flows of the system. Apart from that, the placement of distributed controllers both for VNF management and SDN service steering over a multi-administrative platform is a non-trivial problem.

To avoid these issues, we propose `Amalgam` in this work, which couples distributed placement of VNFs and SDN enabled traffic steering for a multi-administrative platform by exploiting in-network or In-network processing (In-network processing) [46, 47] architecture. In-network processing provides a mini-cloud like platform by utilizing residual capacities of various network devices of the platform. To harness the power of In-network processing, `Amalgam` proposes a novel distributed VNF placement module along with a distributed SDN control plane as a micro-service (μS) which can be deployed over existing network devices by utilizing their residual computing capabilities. Placement of the proposed micro-controllers (μCs) ensures fault-tolerance of the control plane whenever network topology changes. Additionally, the use of SDN ensures fine-grained QoS over multi-domain platforms. Moreover, `Amalgam` is compatible to cater to "*plug-and-play*" nature of the devices without compromising the operation, where, the plug-and-play devices may join and leave the platform dynamically. The coupling of VNF placement and traffic steering in `Amalgam` ensures dynamic service chaining during an on-going session, which is very useful for large-scale platforms. In summary, our major contributions are as follows.

- We develop `Amalgam`, which achieves fine-grained network control over multiple administrative domains along with distributed management of service chains. `Amalgam` exploits μS architecture of the In-network processing platform to deploy the network and service chain management modules to attain distributed control over the multi-domain platform.
- We develop a distributed heuristic to identify the placement of VNFs for a large-scale networked system spanning multiple administrative domains. The proposed greedy heuristic can deploy the VNFs very quickly without gathering resource statistics from all the devices. Thus it can provide significant performance improvement in terms of flow initiation

delay.

- To support the plug and play nature of the devices, `Amalgam` is developed to provide "*`zero touch deployment`*". Zero-touch deployment ensures that whenever a new device enters the eco-system, it requires minimal attention from the system administrator.

For performance evaluation, we develop an emulation framework *MiniDockNet* for VNF deployment using "*`docker`*" [48] over In-network processing, as the existing network name-space oriented mininet [49] emulator is not sufficient for In-network processing. We compare performance of `Amalgam` with an exiting service function chaining framework `Dysco` [234]. Our emulation over a realistic large-scale system, which consists of 70 devices and 6 different service chain scenarios, reveals that `Amalgam` can ensure fine-grained QoS without significant increase of resource utilization of the devices. Since Dysco does not specify any VNF placement mechanism, we compare our results with one of the state-of-the-art distributed VNF placement framework `WGT` [230] on top of Dysco framework. Our experiments show that `Amalgam` is capable of a significant reduction in the flow initiation delay. Therefore, `Amalgam` provides a better end-to-end delay than it's predecessors for short-duration flows.

6.2 Related Work

In the literature, most of the SDN enable service chain management primitives [235, 232, 236, 237] rely on the logically centralized view of the SDN control plane. `OpenNF` [236] and Split-Merge [237] keeps track of the VNF states to ensure fault-tolerant migration. However, middlebox developers must modify, or at least annotate, their code to perform custom state allocation to use these two platforms. The same issue is also found in `S6` [238], which relies on DHT based shared objects for NF state management. In comparison these existing works, `Amalgam` does not require any custom development since it uses containerization and provides a decentralized architecture.

`Network service header` [233] is an example of session based methods that provide an encapsulation mechanism to forward data packets from one middlebox to another. `Dysco` [234] proposed a distributed architecture for managing service chaining. `Dysco` primarily addresses two challenges in service function chaining; (a) Scalability: which is addressed by implementing distributed management of traffic steering, and (b) Multiple administrative domain issues which are handled by intercepting TCP sessions in the hosts. However, to achieve this, `Dysco` requires it's agents to be installed in the hosts and all intermediate nodes. Installation of `Dysco` agents in all devices is difficult to achieve in case of in-network processing platforms due to the plug and play nature of its constituent devices. On the other hand, most of the session based frameworks use encapsulation for steering, monitoring of the flow characteristics become almost impossible. This is a common problem with nearly all the session based proposals.

`Kariz` [239] has proven that optimal deployment of VNFs under resource and service level agreement constraints is $\mathcal{NP}$-hard in the presence of centralized Service Chain Manager (SCM). Authors in [240, 241, 242] have proposed online proactive heuristics for the VNF deployment problem targeted towards various IoT scenarios. All these previous studies work under a common presumption that the IoT platform is within a single administrative domain and managed by a single controller. However, in reality, LSiN In-network processing infrastructure may span across multiple administrative domains, and a single controller architecture creates performance bottleneck. `WGT` [230] presents an iterative VNF placement heuristic for the multidomain network. `WGT` uses a hierarchical aggregation controller to construct an abstracted view of the network by obtaining feedback from each domain. Therefore, this hierarchical approach requires a larger initiation time. Since LSiN generates a huge amount of short-lived flows [4], WGT can reduce the overall application performance.

6.3 Architecture

The proposed `Amalgam`[1] is constructed on top of `Aloe` (Chapter 5) framework. `Aloe` provides an orchestration framework for a fault-tolerant self-stabilizing distributed control plane on top of the in-network processing platform using μCs instead of the standard SDN controller. Hence, the proposed `Amalgam` is also fault-tolerant and self-stabilizing. Each device of `Amalgam` supports the following modes of operations:

- **Host (default) mode:** A device is in this mode if it executes at least one client or server application.
- **Forwarding mode:** If the device has multiple active network interfaces, then the forwarding mode is activated. For ease of reference, we describe a device in forwarding mode

---

[1] `https://github.com/subhrendu1987/NFV_MiniDockNet`

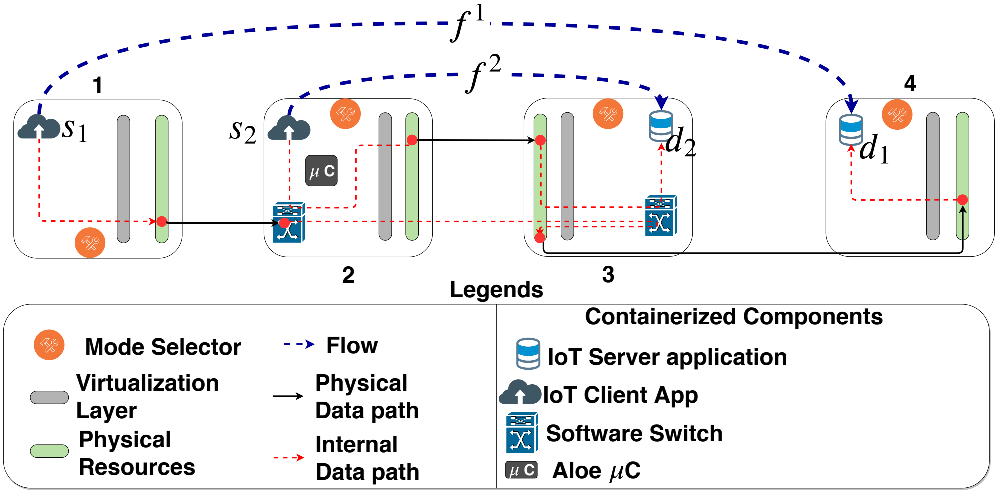


Fig. 6.1: In-network Processing Architecture

as "forwarders". Forwarders have containerized software switch.

- **μC mode:** During execution of `Aloe` a device can be selected as μC. To handle such a scenario, `Amalgam` supports μC mode of operation.

At any point in time, a device is in "*atleast*" any one of the above modes. Based on the current network state, device can select their mode of operations dynamically. Each device has a mode selector module that periodically checks neighborhood of the device and activates and/or deactivates the μC and/or forwarding mode.

In the example given in Fig. 6.1, four devices $(1, \ldots, 4)$ are connected within a linear fashion. There are two flows in the system (f^1 and f^2) such that 1 and 2 host client applications s_1 and s_2 respectively. Traffic generated from s_1 and s_2 are served by server applications d_1 and d_2 hosted on d_4 placed in 3 and 4 respectively. In this scenario, device 1 and 4 work in a host mode. On the other hand, 2 and 4 have multiple active physical interfaces; so, they work in a forwarding mode. Between these two forwarders, we assume that 2 is assigned the role of μC by `Aloe`.

To facilitate the different modes of operation of each device, `Amalgam` can be described as a composition of several components. The working of the components is shown in Fig. 6.2.

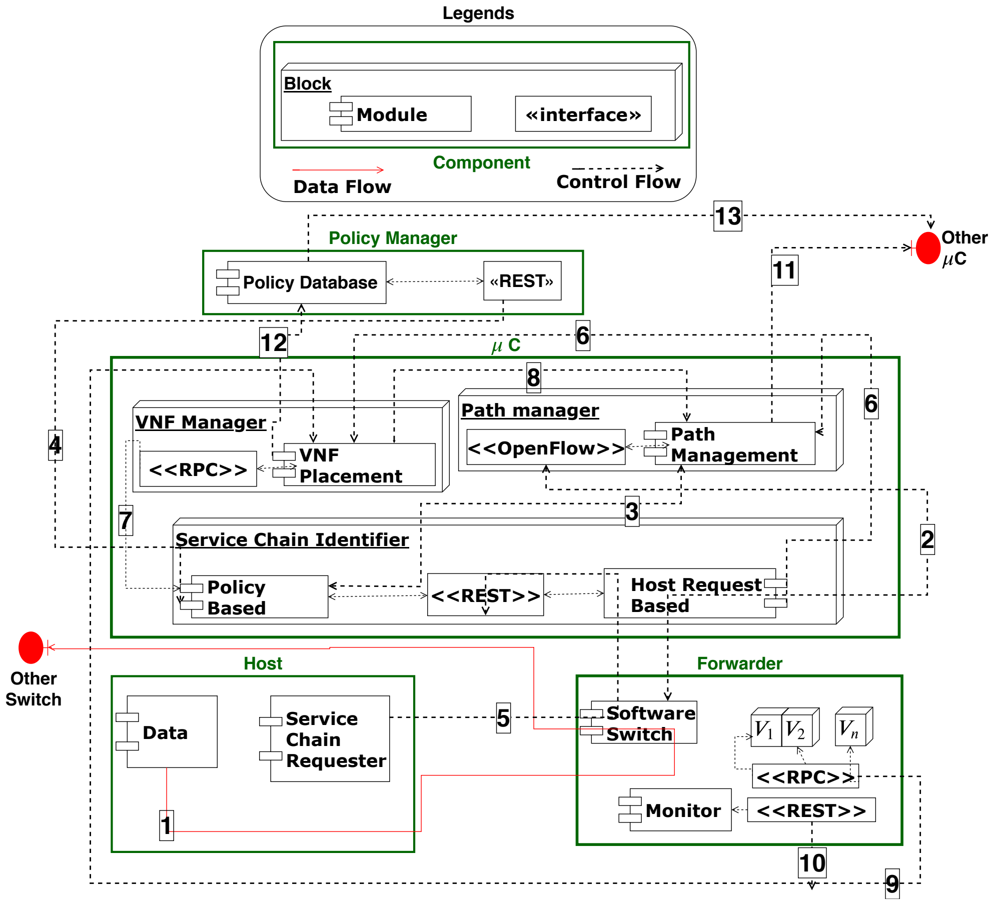


Fig. 6.2: Component diagram of `Amalgam`

6.3.1 Host Component

The host component is composed of two modules namely *data flow* and *service chain requester module*. For the sake of abstraction, we refer to the Internet of things (IoT) client and server applications as part of the data flow module that is responsible for generation/consumption of LSiN traffic. All packets generated by data flow modules are directly forwarded to the forwarder module, which in turn forward towards the destination host device through service chains.

Consider a scenario of higher/lower server load during an data exchange sessions. Normally additional load balancers are dynamically added or removed from the service chains. In such a scenario, a host also get affected. Hence, we propose service requester module, a "*`REST`*" based interface to communicate with the μC to dynamically adjust the hosts with modified service chain.

6.3.2 Policy Manager Component

For any ongoing data transfer session, generated traffic from a host component needs to be forwarded through specific sequences of VNFs (service chain). Since sequence VNFs depend on the flow characteristics such as application, source, and destination. We utilize a separate policy manager component to provide the service chain related policy information for an individual flow. In this context, we adopt the definition of flow as given in the SDN literature [15]. We also consider that each flow can be identified by a suitable set of match fields using "*`OpenFlow`*". Policy for each flow contains two parts; (i) the flow identifier (i.e., "*`OpenFlow`*" match field) and (ii) ordered list of the types of VNF (service chain) through which the flow should be steered. The policy manager keeps the list of service chain policies in a distributed database. A system administrator can update policies via the "*`REST`*" interface, that allows dynamic adjustment of policies during execution. The μC components read, process, and translate these policies and modifies flow table entries of the forwarder component as recommended by the policy.

6.3.3 Forwarder Component

As mentioned in Section 6.3.1, data generated by the host components are forwarded to the forwarder component. This component can be either in the same device where the host component is or in an adjacent forwarder device. Forwarder provides two basic services; a) Data forwarding via software switch module and b) VNF hosting by utilizing residual capacity of the device executing this component. The software switch module is connected to μC adjacent to it via standard `OpenFlow` interfaces and acts as a SDN capable switch.

Forwarder components use containerization tools (e.g., Docker [48], Kubernates [243], etc.) to host the VNFs and utilize residual resources of forwarder devices. To manage the containers, `Amalgam` forwarder component provides "*`RPC`*" interfaces, which can be accessed by the μC. Successful placement of VNFs in the forwarder depends on residual resource of the forwarder. To keep track of residual capacities of the forwarder, we implement a separate monitor module that can provide resource utilization of the device to the μC associated with it via "*`REST`*".

6.3.4 μC Component

For a large scale and multiple administrative domains based LSiN eco-system has multiple μC devices that take care of the forwarders adjacent to them. Each μC has information about its neighbor μCs. `Amalgam` introduces Service Chain Identifier (SCI) and VNF manager module (VMM) module along with the Path Manager Module (PMM) module in each μC as shown in Fig. 6.2. The task of the PMM is to find path between a source device under its influence and a remote destination device. To find path, each μC exchanges the "*constituent list*" of devices periodically. Other two modules can obtain path related information by querying PMM. The detailed working principles of rest of the modules are as follows.

a Service Chain Identifier

At the startup phase of the μC, SCI caches policy in a local cache. The local cache is updated whenever policy manager database is updated. SCI module is consulted when an "*`OpenFlow`*" "*`packet in`*" event is initiated at the μC. From list of VNFs in the service chain, SCI chooses first VNF, and it's execution status in the local domain. If VNF is executing inside a forwarder connected to μC, SCI consults PMM to establish data flow path by installing flow table entries via standard "*`OpenFlow`*" protocol. Otherwise, it sends a search query to the other μCs to identify target VNF address. If address of the VNF is not found, then SCI consults VMM(Section b) to start execution of the VNF. This procedure is iterated for all the VNFs in the service chain.

b VNF Manager

The VMM works in a distributed fashion and communicates with the neighbor μCs. VMM tries to answers the following two questions; (a) should the VNF be placed in any of the forwarders associated with the μC? (b) which forwarder should take care of the VNF?. The detailed protocol to find an answer to these questions is described in Section 6.4.2. Additionally, VMM also takes care of dynamic addition or removal of the VNFs to an ongoing flow.

6.3.5 Interaction of `Amalgam` Components

The interactions between these components are represented in Fig. 6.2 using labeled edges (referred using boxed numbers). Let us consider the following scenario where the host data flow module generates a flow that needs to be forwarded through VNF-1. The flow path is shown using ($\boxed{1}$). As soon as the flow enters, software switch module consults μC PMM via “*`OpenFlow`*” interface ($\boxed{2}$) since the software switch module does not have any pre-loaded action for this flow. The PMM identifies service chain policy for the flow by querying SCI ($\boxed{3}$). SCI generally pre-loads the policy during bootstrapping. However, in the event of an absence of a suitable policy, SCI queries the policy manager through the “*`REST`*” interface ($\boxed{4}$) to obtain policy information. Once the policy information is obtained, PMM consults VMM to obtain exact location of the VNFs required for the flow ($\boxed{8}$). The VMM identifies location of the VNFs by consulting with rest of the μCs ($\boxed{12}$) if the service chain has already been deployed. For a first time flow, service chain may not exist. Therefore, VMM takes responsibility of deploying the service chain. To deploy individual VNF, VMM communicates to the forwarders to obtain their resource allocation via “*`REST`*” ($\boxed{10}$). Once suitable forwarder is found, VMM deploys the VNFs by using “*`RPC`*” ($\boxed{9}$). After deployment of the service chain, PMM calculates the flow table entries to send the flow through sequence of VNFs and pro-actively installs them to the forwarders, which are being controlled by it. During the session, a host may decide to add/remove one/some of the VNFs, as mentioned in Section 6.3.1. To cater to such scenarios, the service chain requester module in the host component can invoke a “*`REST`*” request to the SCI ($\boxed{5}$). Upon receiving such a request, SCI notifies VMM and PMM ($\boxed{6}$) and VMM and PMM act accordingly. The detailed implementation and design choices are described in the next section.

6.4 Implementation Details and Design Choices

`Amalgam` is targeted for a highly dynamic In-network processing platform. The scalability issues and dynamic behavior of the platform is responsible for the challenges we faced during implementation. In this section, we describe the implementation challenges and the proposed solutions to overcome the issues.

6.4.1 Plug-and-Play Capability

A typical LSiN platform is composed of plug-and-play devices where “*`zero touch deployment`*” [244] is highly desired. Whenever a new device enters the eco-system, it requires to be configured. To avoid individually configuring the devices, we design each component of `Amalgam` (except the host component) as Docker containers. Once a device enters the eco-system, it assumes the

Algorithm 4: Distributed Placement of VNF

```
Function GreedyPlace(Path: $P^a$, Service Chain: $C^j$, $\mu$C: $l$):
    Find ordered set of unplaced VNFs from $C^j$;
    $I \leftarrow \{i : i \in P^a, \varphi_i = l\}$;
    Place as many VNFs as possible among $I$;
    return number of VNFs placed;
Function Main(Flow: $f^j$,$\mu$C: $l$):
    // Find VNF placement profile for $f^j$ in $\varphi_i$
    Find set of paths ($P$) from $s_j$ to $d_j$ by querying "Path Management" module of $l$;
    $\underset{P^a \in P}{maximize}$ GreedyPlace($P^a$, $C^j$, $l$);
    if $\exists c_{j,k} \cdots c_{j,j_{max}}$ not placed then
        // All devices under $l$
        Obtain the list of adjacent $\mu$C of $l$ and store it in $N\mu$foreach $l' \in N\mu$ do
            Main($f^j$,$l'$);
    return;
```

host mode of operation. Since the host mode does not require anything more than the IoT applications (clients and servers), they can work smoothly. Whenever the device wishes to change its mode, it can pull container image of the `Amalgam` component from the nearest forwarder.

6.4.2 Distributed VNF Placement

In a short-lived flow heavy system, minimization of the flow initiation delay is critical. The flow initiation delay consists of following components namely (a) Controller consultation delay (b) SFC deployment delay, and (c) path setup delay. The proposed VNF placement reduces the SFC deployment delay. A SFC for a particular flow is composed of multiple VNFs, which requires resource consumption. Each device of a IoT in-network processing platform has residual resources that can be used for deployment of these VNFs's. The proposed VNF placement identifies the set of devices where the VNFs of the SFCs can be placed for a given network and flow profile while satisfying the capacity constraints of the devices. Maintaining capacity constraints in a multi-domain system is non-trivial since the residual capacity of a device residing in a different administrative domain is difficult to collect. Therefore, we propose the greedy heuristic as given in the Algorithm 4.

Each μC in the end-to-end path (P) executes the proposed heuristic for each flow (f^j represents jth flow) from source (s_j) to destination (d_j). We denote SFC of f^j with C^j. Certain

μC with ID l maintains the topology information as the list of devices (D_l) and list of links (E_l) where each link $e_{i,i'} \in E_l$ represents the physical connection between two devices (i and i'). For the sake of simplicity, we denote the μC associated with $i$as φ_i. The proposed heuristic identifies a path P^a between s_j to d_j from the set of P such that, most of the VNFs of C^j are placed near s_j in a distributed fashion. This way, one μC does not need the resource utilization of devices from other administrative domains. Once the flow is established, the resource utilization of devices in the path (`info`) is piggybacked with the data packets. The VNF manager can re-solve the Algorithm 4 and find a new allocation of VNF with updated utilization.

6.4.3 Migrations of the VNFs

A VNF may need to be relocated in the following circumstances.

a Sub-optimal VNF placement

The initial path for a service chain is sub-optimal and during the optimization period some of the VNFs requires migration from one to another. The decision of moving an arbitrary VNF $c_{j,k}$ from i to i' is taken by $\varphi(i)$. In order to ease migration, VNFs are deployed using containers which allows save the state of $c_{j,k}$ via standard APIs. Once the decision is made, $\varphi(i)$ notifies $\varphi(i')$ to copy the snapshot of $c_{j,k}$ to i' via implemented "***`REST`***" interface. After restoration of the snapshot, the path manager module of $\varphi(i)$ and $\varphi(i')$ are consulted to reconfigure the flow table entries of the intermediate forwarders between i to i'.

b Addition/removal of devices

The dynamic nature of the LSiN eco-system allows a device to move in or out of the platform. The addition of VNF introduces a new opportunity to optimize VNF placement further. On the other hand, removal of a device requires migration of VNFs running in the device to another device. Let i is either going to join or exits from the system. This event results in a topology change event in $\varphi(i)$. Based on the topology change event type, $\varphi(i)$ decides the services that need to be migrated to/from i.

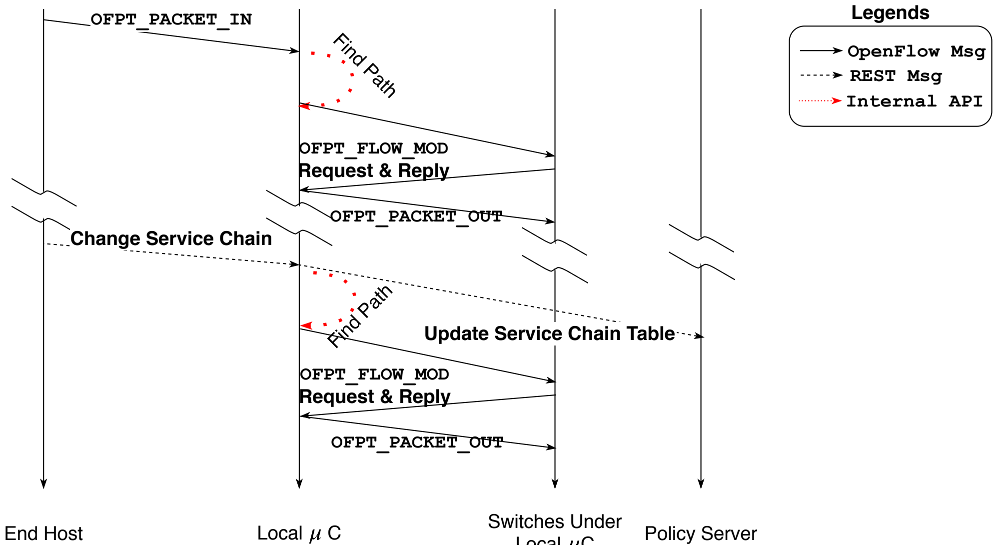


Fig. 6.3: Service Chain Management

6.4.4 Dynamic management of service chains

`Amalgam` provides a dynamic change of service chaining. This feature is required for the following scenario. Let's assume, flow f^j passes through a Deep Packet Inspector (DPI) VNF $c_{j,k}$. Based on the signature of f^j, $c_{j,k}$ conditionally decides if the flow needs to be steered through a firewall VNF $c_{j,k+1}$. To implement this, `Amalgam` allows the VNFs to interact with the local μC via "*`REST`*" interface as given in Fig. 6.3. The local μC can deploy the $c_{j,k+1}$ if it is not available and sends the "*`OFPT_FLOW_MOD`*" events to the forwarder component in order to enable the flow steering.

6.4.5 Flow Tags for Monitoring

Once the VNFs are placed, the path management module of μCs set-up flow table entries of the participating forwarders via `OpenFlow` protocol. One issue regarding path management through service chains is to identify an end-to-end flow that arises in presence of the "5-tuple" changing VNFs (e.g., Load balancer, web proxy cache, NATs, etc.). Since such VNFs may alter the packets in unpredictable ways, fine-grained management and monitoring of the flows passing through becomes difficult. To avoid this issue, `Amalgam` uses packet tagging. Let us consider a scenario where f^j requires $c_{j,k}$, which is a flow identifier modifying VNF hosted in i. In this case, at the time of flow table entry installation $\varphi(i)$ attaches a "*VLAN*" tag entry to the flow. Using this "*VLAN*" tag i can identify f^j even if the original flow identifier is modified by $c_{j,k}$. The μC also maintains a table for flows like f^a, which keeps track of the original id of the flow and the modified ids (alias).

6.4.6 Providing QoS

`Amalgam` is developed on top of the SDN decentralized control plane, which enables us to ensure flow specific QoS guarantees. On the other hand, since the VNF deployment is done using containers, using "*cgroups*"[2] can ensure the VNF specific QoS like reservation of CPU, Memory, etc. The policy server module contains the "*cgroups*" parameters for each VNFs of a service chain, which is used to ensure VNF specific QoS.

6.5 Prototype and Experimental Results

"*Mininet*" [49] a popular emulation framework used by both academia and industry mainly for prototyping SDN applications. Internally, "*Mininet*" uses network name-spaces for emulation of nodes, and all the switches are emulated using Open virtual switch (OVS) bridges. However, "*Mininet*" is not suitable for testing In-network processing framework and VNFs for following reasons:

- In-network processing architecture requires resource isolation for individual nodes, which is non-trivial to achieve in "*Mininet*" OVS-bridges.
- Each VNF may require a collection of dependent software to execute actual network function. However, "*Mininet*" hosts and "*network name-spaces*" can not guarantee isolation at software granularity.

Therefore, we develop "*MiniDockNet*" over existing Application Programming Interfacess (APIs) of the "*Mininet*" framework. The salient feature of "*MiniDockNet*" is the usage of actual docker containers to emulate individual devices instead of network namespaces to address above issues. To implement "*MiniDockNet*" we use "docker-py"[245] and original "*Mininet*" sources. "*MiniDockNet*" supports the following types of devices; (a) hosts, (b) μC, (c) Switches, and (d) VNFs. Here hosts are standard docker containers equipped with IoT client-server applications.

---

[2] `http://man7.org/linux/man-pages/man7/cgroups.7.html`

Each μCs can act as SDN controller. Switches have OVS installed in them. VNFs mimic real-life VNFs and execute inside switch dockers using the "*Docker-in-Docker*" configuration to implement In-network processing of VNFs. Therefore, the VNFs can migrate from one switch to another, depending on the VNF placement decisions. To implement migration, we use standard live container migration using "*CRIU*"[3]. For the emulation of the links between any two nodes, we use Layer Two Tunneling Protocol (l2tp)[4]

Name	Service chain
C^1	(N)
C^2	(L)
C^3	(W)
C^4	(N→L)
C^5	(L→W))
C^6	(N→L→W)

Table 6.1: List of Service Chains

Resource	Distinguishable
CPU	Y (<0.05)
Memory	N (0.42)
Bandwidth	Y (<0.01)
Delay	Y (<0.01)

Table 6.2: Wilcoxson test for QoS provisioning

6.5.1 Experimental Setup

For experimental purpose, we use "***rocketfuel***" [218] topology[5] with 68 nodes. Each link is configured to emulate $3ms$ of delay and $10Mbps$ of bandwidth using linux "***tc***" utility. We use "***iperf***" to generate long flows; for shorter flows we use "***ping***". The clients and server applications are hosted on *diameter* of the topology. For background traffic we use python based "***HTTP***" client and server.

We use "***cassendra***"[6] to implement the policy server module. Rest of the `Amalgam` modules targeted for μC are implemented on top of "***Ryu***"[7], a python based SDN controller framework. For experiments, we use 3 different VNFs (NAT (N), Load Balancer (B)[8] and Web Proxy(W)[9])

---

[3] `https://criu.org/Live_migration`

[4] `https://en.wikipedia.org/wiki/Layer_2_Tunneling_Protocol`

[5] `https://raw.githubusercontent.com/subhrendu1987/NFV_MiniDockNet/master/topology/rocket_fuel_68.graphml`

[6] `https://gitbox.apache.org/repos/asf?p=cassandra.git`

[7] `https://ryu.readthedocs.io/en/latest/`

[8] `https://hub.docker.com/_/haproxy/`

[9] `https://hub.docker.com/r/sameersbn/squid/`

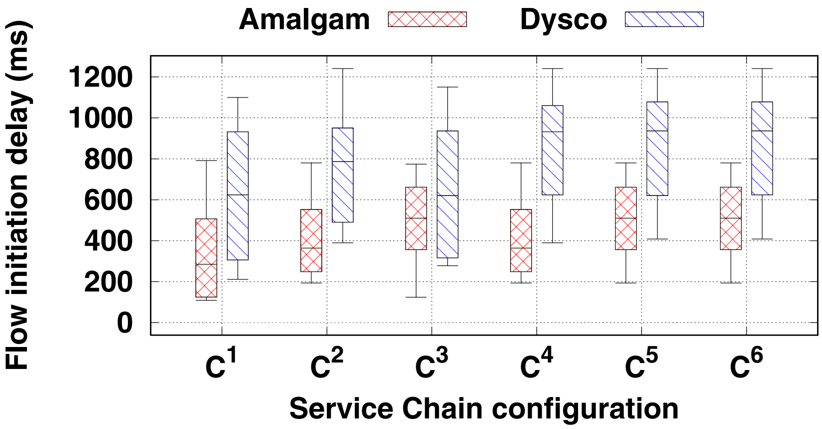


Fig. 6.4: Flow Initialization Delay

to create 6 different combination of service chain as given in Table 6.1. In order to ensure the confidence on the results, each experiments are performed atleast 30 times.

6.5.2 Results

We compare the performance of `Amalgam` with `P4` based distributed session-oriented service function chaining framework called `Dysco` [234]. Since, Dysco ensures session related performance and does not provide any VNF placement strategy, performance evaluation of the proposed distributed VNF placement algorithm is done with another existing work `WGT` [230] which proposes a distributed heuristic for VNF placement for the multi-domain network.

a Session Related Performances

Figure 6.4 shows comparison between `Dysco` and `Amalgam` in terms of flow initialization delay. We found that `Amalgam` is capable of quicker flow initialization than `Dysco`. This reduction in flow initialization delay comes from the parallel deployment of VNFs as opposed to the hop by hop deployment of VNFs in `Dysco`. The advantage of flow initialization delay becomes much evident in case of longer service chains like C^6 than smaller service chain like C^1. Since `Amalgam` uses containers to deploy the VNFs as opposed to the `P4` applications used in `Dysco`, deployment of VNFs using `Amalgam` incurs greater latency, as shown in Fig. 6.5. The increase in VNF deployment time for `Amalgam` depends on VNF container size. Therefore, deployment latency is higher for C^6 in compared to C^1. However, in a large scale network, VNF deployment

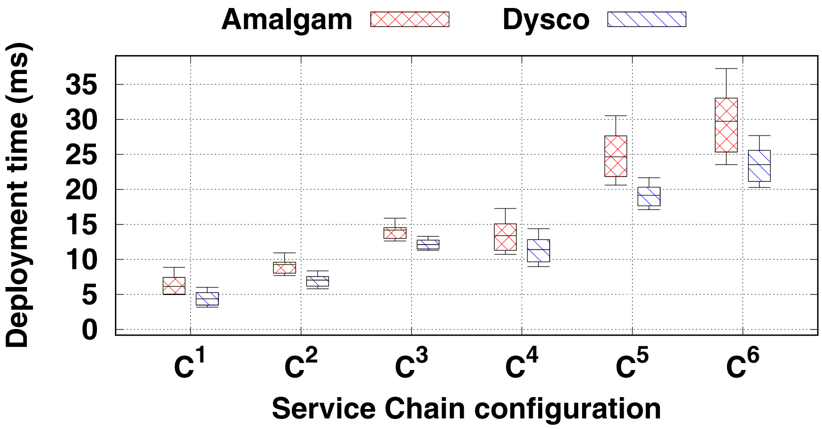


Fig. 6.5: Latency of Deployment

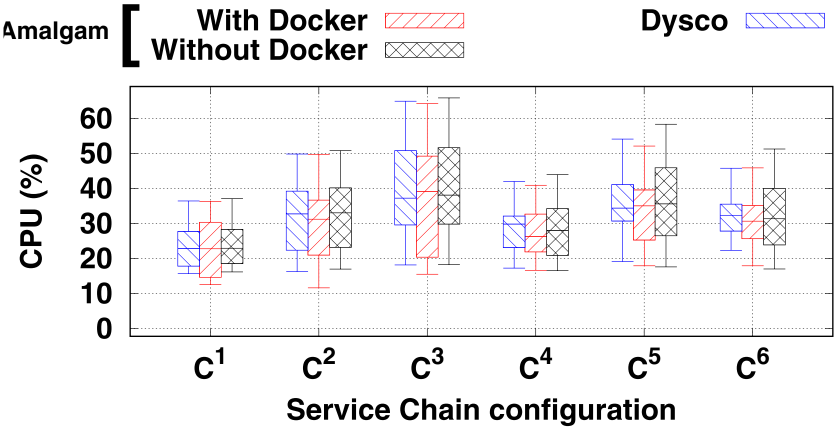


Fig. 6.6: CPU Utilization

events are far rare than a flow generation event. On the other hand, the use of containers provide greater flexibility as creation of new "*middlebox*" application using container requires less programming overhead than creation of a new `P4` application. As a result, state management during migration of VNFs from one node to another becomes easy when they are running inside a container as compared to the `P4` applications of `Dysco`. However, these management benefits of containers come at the cost of resource utilization.

The placement of VNFs requires resource occupancy in the deployed devices, which is an important aspect of resource constraint LSiN devices. In Fig. 6.6, we compare the performance of `Amalgam` with `Dysco` in terms of CPU utilization of devices due to the placement of VNFs.

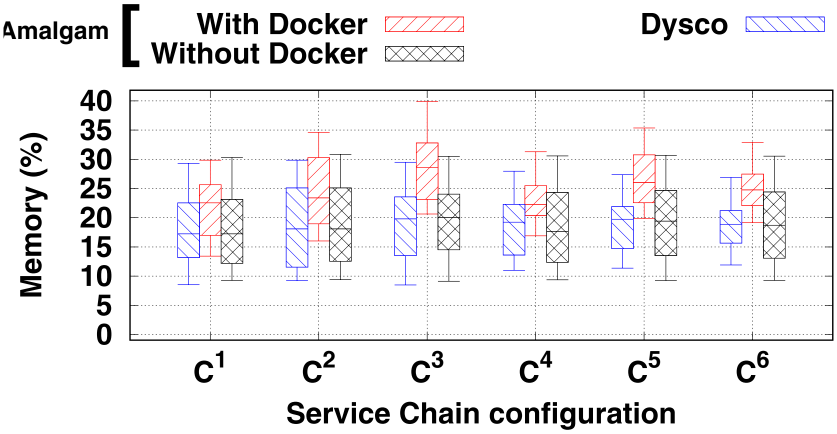


Fig. 6.7: Memory Utilization

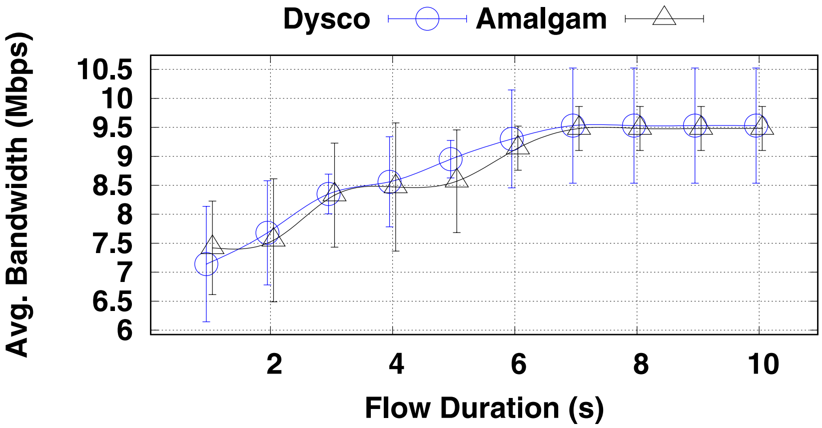


Fig. 6.8: Average Throughput: $P > 0.05$

In order to normalize additional resource consumption of `Amalgam` due to the use of containers, we also compare resource utilization of `Amalgam` without using docker. Similarly, we provide a comparison of memory utilization for `Amalgam` and `Dysco` in Fig. 6.7. Based on these two experiments, we observe that `Dysco` incurs less utilization of resources than the proposed `Amalgam` with the container. However, based on the "*Wilcoxon Rank Sum test*" [222] we find that, difference of resource utilization of `Amalgam` without Docker and `Dysco` is statistically insignificant (i.e. $p-value > 0.05$) for C^4, C^5 and C^6. Fig. 6.8 shows comparison of throughput between `Amalgam` and `Dysco`. The Wilcoxson rank sum test reveals that throughput between `Amalgam` and `Dysco` are statistically indistinguishable (Here our alternate hypothesis H_a is `Amalgam` provides less

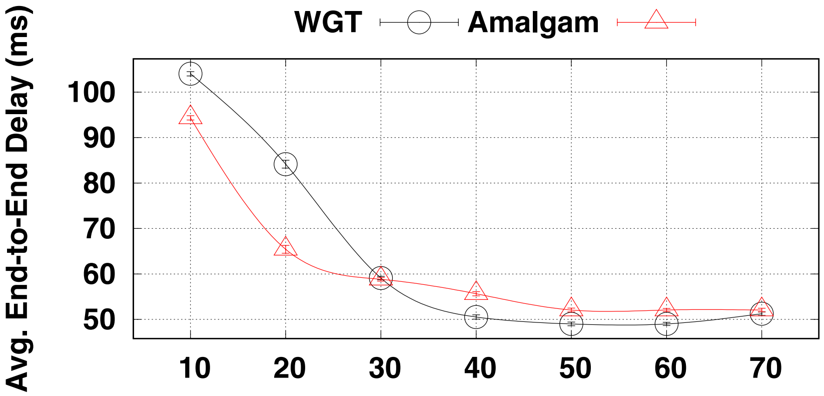


Fig. 6.9: Average End-To-End Delay for C^4

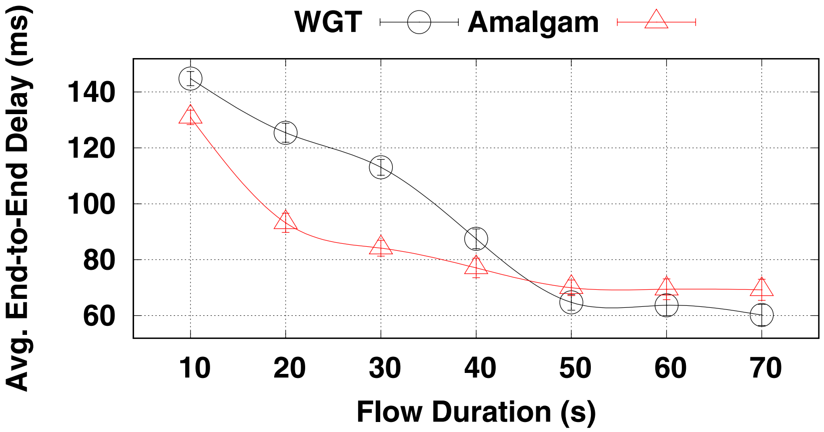


Fig. 6.10: Average End-To-End Delay for C^5

throughput than `Dysco`).

b Performance of Distributed VNF Placement

To measure performance of distributed VNF placement heuristic used in `Amalgam`, as mentioned earlier, we deploy `WGT` [230] on top of `Dysco`. However, it is difficult to deploy a centralized controller for a large scale multi-domain system. Therefore, we place `WGT` in μC nearest to the source device. We measure and compare effect of delay for all the service chains when flow duration increases. Based on experimental results, we found that effect of delay for single VNF

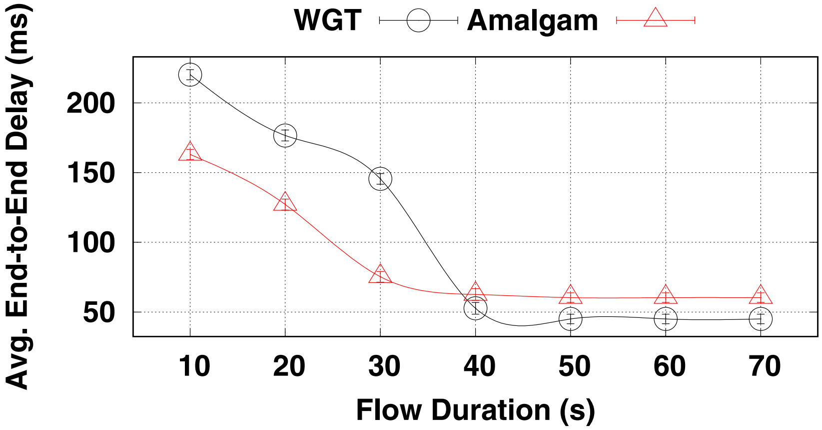


Fig. 6.11: Average End-To-End Delay for C^6

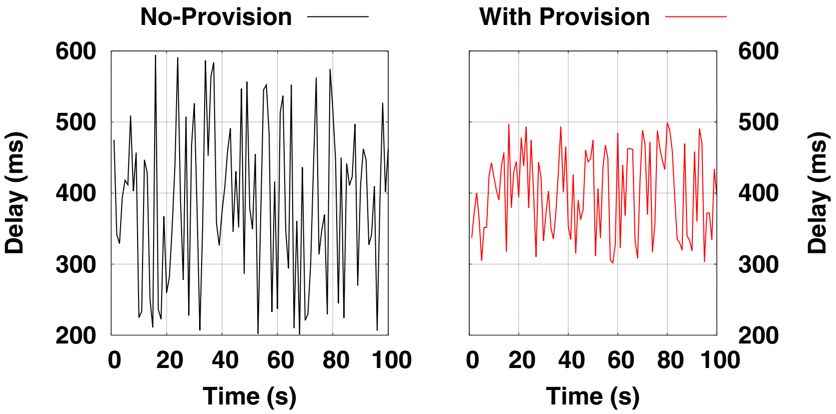


Fig. 6.12: Effect of QoS

does not change since "*Amalgam*" and `WGT` provides the same results for VNF placement. Hence, we omit the plots for C^1, C^2, C^3. For multiple VNF oriented service chains like C^4, C^5 and C^6, we provide average end-to-end delay in Figs. 6.9 to 6.11 respectively.

Based on the results, we can observe that `Amalgam` can perform significantly well for shorter flows as the iterative `WGT` requires a significant amount of feedback rounds to find the proper placements of VNFs.

6.5.3 QoS Provisioning

`Amalgam` is capable of showing QoS provisioning by reserving resources limiting CPU, memory, bandwidth, and link delay. We perform two experiments for each resource type, one with no provisioning and another with resource reservation limit set as the mean value found in the previous experiment. Based on the two results, we tried to identify if resource utilization is statistically significant based on Wilcoxson rank-sum test. We report the results in Table 6.2 along with the P-value. We found that except memory utilization rest of the resource reservation works significantly well. We also find that resource reservation can reduce the jitter of the flow, as given in Fig. 6.12.

6.6 Summary

In this work, we present "*Amalgam*", which integrates distributed SDN orchestration framework with a distributed service chain management framework. The proposed "*Amalgam*" is suitable for large scale multi-domain IoT in-networking platforms. We also provide a distributed heuristics for the placement of constituent VNFs of service chains. The lack of an existing emulation platform for container oriented VNF service chain has motivated us to develop "MiniDockNet". Using this emulation platform, we found that "*Amalgam*" incurs a lesser flow initialization delay than that of a very recent distributed service chain management framework (Dysco). We also show that "*Amalgam*" is capable of ensuring less end-to-end delay for short flows.

Intentionally Left Blank

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have investigated scalability and network management issues of large scale Internet of things (IoT) network (large scale IoT network (LSiN)) by utilizing Software-Defined Network (SDN). Apart from standard IoT features LSiN supports in-network or In-network processing (In-network processing) and spans across multiple administrative domains. Scale of the network and use of resource constrained Commercial off-the-shelf (COTS) devices makes the network unstable and prone to failures. The proposed techniques and architectures in this thesis are designed to provide failure tolerance. During design of the architectures we have also considered traffic characteristics of LSiN to improve the end-to-end network performance. The major contributions of this thesis can be summarized as follows.

The use of SDN for network management has been well studied in the literature. We observed that the network management decisions like route discovery have a strong correlation with the transport layer decision like path management. This correlation increases in the presence of a multi-path transport protocol. In Chapter 3, we developed `SDN-MPTCP`, which helps the transport layer decision making of Multipath TCP (MPTCP), a popular multi-path transport layer protocol, by exploiting SDN. We observed that the proposed `SDN-MPTCP` can reduce the Head of Line (HOL) blocking problem of MPTCP sub-flow selection.

Even though, SDN can improve performance of LSiN, deployment of SDN over an existing LSiN is challenging. The capital expenditure (capex)/operational expenditure (opex) introduced at the time of deployment of SDN over a LSiN environment restricts the system managers to adopt SDN for network management. To overcome this issue, we propose `FLIPPER` in Chapter 4. `FLIPPER` lies some where between the traditional network architecture and SDN and provides SDN like control of the COTS devices of the LSiN. Additionally `FLIPPER` can provide fault-tolerance by dynamically assigning roles using Network Function Virtualization (NFV). `FLIPPER` can reduce the flow initiation delay by dynamically deploying Network Information Base (NIB).

One of the major challenges in SDN based management of LSiN lies in its dynamic nature, where the devices can enter/leave the eco-system without notification. In Chapter 5 we extended the `FLIPPER` to provide plug-and-play support to tackle the dynamic behaviour of LSiN components by developing `Aloe`. `Aloe` provides zero-touch deployment with self-healing properties through self-stabilization. `Aloe` proposes servicification of control plane over the In-network processing platform provided by LSiN. We also proposed a light-weight controller independent control plane framework to enhance the capabilities of `Aloe`.

Apart from core services like routing, quality of service (QoS) etc., management of a large scale network like LSiN requires multiple auxiliary Virtual Network Function (VNF) services like Network Address Translation (NAT), proxy etc. Where the core functionalities of SDN are supported in `Aloe`, the existence of VNF requires more attention. In Chapter 6 we propose `Amalgam` which proposes the integration of middle-box application into `Aloe`. Like `Aloe`, `Amalgam` provides plug-and-play support for LSiN which is spanned over multiple administrative domain. Additionally, `Amalgam` proposes a novel VNF deployment and traffic steering framework to support Service Function Chaining (SFC) over LSiN. For experimental purpose, we developed a new emulation tool `MiniDockNet` which overcomes the issue of In-network processing emulation using existing "*Mininet*" emulation framework. Based on the experimental results we found that, `Amalgam` performs well for short flows.

7.2 Future Direction

The management capabilities and performance of LSiN can further be improved by providing support for advanced features to the proposed orchestration frameworks and architectures which are kept as the future direction of this thesis. Some of the features are discussed as follows.

7.2.1 Enhancement of Amalgam

In this thesis, the proposed `Amalgam` uses a distributed greedy heuristics for the placement of VNF. The proposed VNF placement strategy provides initial benefits to the short-duration flows. However, there is room for improvement here. A pro-active deployment of VNFs can improve the performance for short as well as long-duration flows. On the other hand, pro-active placement of VNFs requires the prediction of upcoming traffic requirements and load distribution of the devices. We have some initial experimental results [O.1] to believe that an online reinforcement learning mechanism can help in this direction, which we kept as a future work of this research.

7.2.2 Dynamic Telemetric Application Deployment

The proposed `Aloe` orchestration framework relies on the dynamic deployment of control plane applications based on the necessity (in our case failure events). However, this feature can be customized for multiple other event monitoring purposes. For example, a flow monitor can be auto-inserted into a particular region of the LSiN to estimate security lapses, traffic pattern analysis, identification of a heavy hitter flow etc. based on a system administrator defined policy. We kept this customizability as a future work for `Aloe`.

References

[1] Boubakr Nour, Kashif Sharif, Fan Li, Sujit Biswas, Hassine Moungla, Mohsen Guizani, and Yu Wang. "A survey of Internet of Things communication using ICN: A use case perspective". In: *Elsevier Computer Communications* 142-143 (2019), pp. 95–123.

[2] IoT News. *Sigfox Canada launches world's largest IoT network with coast-to-coast availability*. `https://www.iottechnews.com/news/2019/apr/09/sigfox-canada-launches-worlds-largest-iot-network-availability/`. [Online; accessed 10 May, 2020]. 2019.

[3] Cisco. *VNI Forecast Highlights*. `http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html`. [Online; accessed 10 May, 2020].

[4] M. Zubair Shafiq, Lusheng Ji, Alex X. Liu, Jeffrey Pang, and Jia Wang. "Large-Scale Measurement and Characterization of Cellular Machine-to-Machine Traffic". In: *IEEE/ACM Transactions on Networking* 21.6 (2013), pp. 1960–1973.

[5] James McCauley, Yotam Harchol, Aurojit Panda, Barath Raghavan, and Scott Shenker. "Enabling a Permanent Revolution in Internet Architecture". In: *Proc. of International Confrence on SIGCOMM*. Beijing, China: ACM, 2019, pp. 1–14.

[6] Randy Presuhn, Jeffrey Case, Keith McCloghrie, Marshall T. Rose, and Steven Waldbusser. *Version 2 of the protocol operations for the simple network management protocol (SNMP)*. Tech. rep. IETF, RFC3416, 2002.

[7] Rob Frye, David B. Levi, Bert Wijnen, and Shawn A. Routhier. *Coexistence between Version 1, Version 2, and Version 3 of the Internet-standard Network Management Framework*. Tech. rep. IETF, RFC3584, 2003.

[8] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. “OpenFlow: enabling innovation in campus networks”. In: *Proc. International Conference of SIGCOMM.* ACM, 2008, pp. 69–74.

[9] Nick Feamster, Jennifer Rexford, and Ellen Zegura. “The road to SDN: an intellectual history of programmable Networks”. In: *Proc. International Conference of SIGCOMM.* ACM, 2014, pp. 87–98.

[10] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. “A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks”. In: *IEEE Communications Surveys Tutorials* 16.3 (2014), pp. 1617–1634.

[11] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. “Design and implementation of a routing control platform”. In: *Proc. of International Symposium on NSDI.* USENIX, 2005, pp. 15–28.

[12] David Clark. “The Design Philosophy of the DARPA Internet Protocols”. In: *SIGCOMM Computer Communication Review* 18.4 (1988), pp. 106–114.

[13] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. “Fabric: a retrospective on evolving SDN”. In: *Proc. of International Workshop on HotSDN.* ACM, 2012, pp. 85–90.

[14] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. “Resonance: Dynamic access control for enterprise networks”. In: *Proc. of International Workshop on REN.* ACM, 2009, pp. 11–18.

[15] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. “Software-defined networking: A comprehensive survey”. In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76.

[16] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. “Scalable flow-based networking with DIFANE”. In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 351–362.

[17] Ze Yu, Min Li, Xin Yang, and Xiaolin Li. “Palantir: Reseizing network proximity in large-scale distributed computing frameworks using sdn”. In: *Proc. of International Conference of CLOUD*. IEEE, 2014, pp. 440–447.

[18] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O’Connor, Pavlin Radoslavov, William Snow, et al. “ONOS: towards an open, distributed SDN OS”. In: *Proc. of International Workshop on HotSDN*. ACM, 2014, pp. 1–6.

[19] Mehdia Ajana El Khaddar and Mohammed Boulmalf. *Smartphone: the ultimate IoT and IoE device*. BoD–Books on Demand, 2017, p. 137.

[20] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z. Morley Mao, and Subhabrata Sen. “An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design”. In: *Proc. of International Conference on MobiCom*. ACM, 2016, pp. 189–201.

[21] Tongguang Zhang, Shuai Zhao, Bo Cheng, Bingfei Ren, and Junliang Chen. “Research, implementation, and improvement of MPTCP on mobile smart devices”. In: *Taylor & Francis International Journal of Computers and Applications* 41.6 (2019), pp. 407–417.

[22] Bo Han, Feng Qian, Shuai Hao, and Lusheng Ji. “An Anatomy of Mobile Web Performance over Multipath TCP”. In: *Proc. of International Conference on CoNext*. ACM, 2015, 5:1–5:7.

[23] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. “WiFi, LTE, or Both?: Measuring Multi-Homed Wireless Internet Performance”. In: *Proc. of International Conference on IMC*. ACM, 2014, pp. 181–194.

[24] Nitinder Mohan, Tanya Shreedhar, Aleksandr Zavodovski, Jussi Kangasharju, and Sanjit K. Kaul. “Is two greater than one?: Analyzing Multipath TCP over Dual-LTE in the Wild”. In: *ArXiv* abs/1909.02601 (2019).

[25] Yuanlong Cao, Qinghua Liu, Yi Zuo, Fenfen Ke, Hao Wang, and Minghe Huang. “Receiver-centric Buffer Blocking-aware Multipath Data Distribution in MPTCP-based Heterogeneous Wireless Networks”. In: *KSII Transaction on Internet & Information Systems* 10.10 (2016).

[26] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H Low. “Multipath TCP: Analysis, design, and implementation”. In: *IEEE/ACM Transaction on Networking (ToN)* 24.1 (2016), pp. 596–609.

[27] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. “MPTCP is not pareto-optimal: performance issues and a possible solution”. In: *IEEE/ACM Transaction on Networking* 21.5 (2013), pp. 1651–1665.

[28] Bong-Hwan Oh and Jaiyong Lee. “Constraint-based proactive scheduling for MPTCP in wireless networks”. In: *Computer Networks* 91 (2015), pp. 548–563.

[29] Morteza Kheirkhah, Ian Wakeman, and George Parisis. “MMPTCP: A Multipath Transport Protocol for Data Centers”. In: *Proc. of International Conference of INFOCOM*. IEEE, 2016, pp. 1–9.

[30] Sakir Sezer, Sandra Scott-Hayward, Pushpinder Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Marc Miller, and Navneet Rao. “Are we ready for SDN? Implementation challenges for software-defined networks”. In: *IEEE Communications Magazine* 51.7 (2013), pp. 36–43.

[31] Dan Levin, Marco Canini, Stefan Schmid, and Anja Feldmann. “Incremental SDN deployment in enterprise networks”. In: *ACM SIGCOMM Computer Communication Review* 43 (2013), pp. 473–474.

[32] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann. “Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks”. In: *Proc. of International Conference on ATC*. USENIX, 2014, pp. 333–345.

[33] Ahmet Cihat Baktir, Atay Ozgovde, and Cem Ersoy. “How can edge computing benefit from software-defined networking: a survey, use cases, and future directions”. In: *IEEE Communication Surveys & Tutorials* (2017), pp. 2359–2391.

[34] Luis M Vaquero and Luis Rodero-Merino. “Finding your way in the fog: Towards a comprehensive definition of fog computing”. In: *SIGCOMM Computer Communication Review* (2014), pp. 27–32.

[35] Mung Chiang, Sangtae Ha, I Chih-Lin, Fulvio Risso, and Tao Zhang. “Clarifying fog computing and networking: 10 questions and answers”. In: *IEEE Communication Magazine* (2017), pp. 18–20.

[36] Mennan Selimi, Llorenç Cerdà-Alabern, Marc Sánchez-Artigas, Felix Freitag, and Luís Veiga. "Practical service placement approach for microservices architecture". In: *Proc. of International Conference of CCGRID*. IEEE, 2017, pp. 401–410.

[37] M. Zubair Shafiq, Lusheng Ji, Alex X. Liu, Jeffrey Pang, and Jia Wang. "Large-Scale Measurement and Characterization of Cellular Machine-to-Machine Traffic". In: *IEEE/ACM Transaction on Networking* (2013), pp. 1960–1973.

[38] Omprakash Kaiwartya, Abdul Hanan Abdullah, Yue Cao, Jaime Lloret, Sushil Kumar, Rajiv Ratn Shah, Mukesh Prasad, and Shiv Prakash. "Virtualization in wireless sensor networks: fault tolerant embedding for internet of things". In: *IEEE Internet of Things Journal* (2018), pp. 571–580.

[39] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks". In: *Proc. of International Conference of INFOCOM*. IEEE, 2015, pp. 513–521.

[40] Fengxiao Tang, Zubair Md Fadlullah, Bomin Mao, and Nei Kato. "An Intelligent Traffic Load Prediction Based Adaptive Channel Assignment Algorithm in SDN-IoT: A Deep Learning Approach". In: *IEEE Internet of Things Journal* (2018).

[41] Walter Haeffner, Jeffrey Napper, Martin Stiemerling, Diego R. Lopez, and Jim Uttaro. *Service function chaining use cases in mobile networks*. Tech. rep. IETF-Draft, 2020.

[42] Aurojit Panda. "A New Approach to Network Function Virtualization". PhD thesis. EECS Department, University of California, Berkeley, 2017.

[43] Fabien Duchene, David Lebrun, and Olivier Bonaventure. "SRv6Pipes: enabling in-network bytestream functions". In: *Proc. of International Conference of Networking*. IFIP, 2018, pp. 1–9.

[44] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. "Onix: A Distributed Control Platform for Large-scale Production Networks". In: *Proc. of International Conference on OSDI*. USENIX, 2010, pp. 1–6.

[45] *Servicification Definition*. `https://en.wiktionary.org/wiki/servicification`. [Online; accessed 10 May, 2020].

[46] Moses Charikar, Yonatan Naamad, Jennifer Rexford, and X. Kelvin Zou. "Multi-commodity Flow with In-Network Processing". In: *Algorithmic Aspects of Cloud Computing.* Springer, 2019, pp. 73–101.

[47] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. "dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces". In: *Proc. of International Symposium of NSDI.* USENIX, 2019, pp. 207–220.

[48] Docker Inc. *What is Docker?* `https://www.docker.com/what-docker`. [Online; accessed 10 May, 2020]. 2018.

[49] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proc. of International Workshop of Hotnets.* ACM, 2010, 19:1–19:6.

[50] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys Tutorials* 17.4 (2015), pp. 2347–2376.

[51] R. M. Gomathi, G. H. S. Krishna, E. Brumancia, and Mistica Dhas Y. "A Survey on IoT Technologies, Evolution and Architecture". In: *Proc. of International Conference of ICCCSP.* IEEE, 2018, pp. 1–5.

[52] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A survey". In: *Elsevier Computer Networks* 54.15 (2010), pp. 2787–2805.

[53] IEEE. *Towards a definition of the Internet of Things(IoT).* `https://iot.ieee.org/definition.html`. [Online; accessed 10 May, 2020]. 2019.

[54] Rakesh Roshan, Anukrati Sharma, and Om Prakash Rishi. "IoT Platform for Smart City: A Global Survey". In: *Emerging Trends in Expert Applications and Security.* Springer Singapore, 2019, pp. 197–202.

[55] Hamidreza Arasteh, Vahid Hosseinnezhad, Vincenzo Loia, Aurelio Tommasetti, Orlando Troisi, Miadreza Shafie-khah, and Pierluigi Siano. "Iot-based smart cities: A survey". In: *Proc. of International Conference of EEEIC.* IEEE, 2016, pp. 1–6.

[56] Abdulla AlHammadi, Aisha AlZaabi, Bashayer AlMarzooqi, Suhail AlNeyadi, Zayed Al-Hashmi, and Maad Shatnawi. “Survey of IoT-Based Smart Home Approaches”. In: *Proc. of International Conference of ASET*. IEEE, 2019, pp. 1–6.

[57] Jonas Wahlfrid and Simon Irengård Gullstrand. “Forming Emergent Configurations in Smart Office IoT Systems”. MA thesis. Malmö högskola/Teknik och samhälle, 2017.

[58] Sheng-hai An, Byung-Hyug Lee, and Dong-Ryeol Shin. “A Survey of Intelligent Transportation Systems”. In: *Proc. of International Conference of CICSyN*. IEEE, 2011, pp. 332–337.

[59] *Cisco Visual Networking Index: Forecast and Methodology, 20132018*. `https://www.cisco.com/c/dam/m/en_in/innovation/enterprise/assets/mobile-white-paper-c11-520862.pdf`. [Online; accessed 10 May, 2020]. 2014.

[60] Sana Habib, Junaid Qadir, Anwaar Ali, Durdana Habib, Ming Li, and Arjuna Sathiaseelan. “The Past, Present, and Future of Transport-Layer Multipath”. In: *Academic Press Journal of Network and Computer Application* 75.C (2016), 236–258.

[61] Michele Polese, Federico Chiariotti, Elia Bonetto, Filippo Rigotto, Andrea Zanella, and Michele Zorzi. “A Survey on Recent Advances in Transport Layer Protocols”. In: *IEEE Communications Surveys & Tutorials* 21.4 (2019), pp. 3584–3608.

[62] Randall Stewart, Michael Txen, and Maksim Proshin. *Stream Control Transmission Protocol*. Tech. rep. IETF, RFC4960, 2007.

[63] Noel Farrugia, Victor Buttigieg, and Johann A. Briffa. “Multi-Stream TCP: Leveraging the Performance of a Per-Packet Multipath Routing Algorithm When Using TCP and SDN”. In: *Proc. of International Conference of LCN*. IEEE, 2019, pp. 218–221.

[64] Anwar Walid, Qiuyu Peng, Steven H. Low, and Jaehyun Hwang. *Balanced Linked Adaptation Congestion Control Algorithm for MPTCP*. Tech. rep. IETF-Draft, 2016.

[65] Eckard Bogenfeld, Andreas Kassler, Markus Amenda, Veselin Rakocevic, and Anna Brunstrom. *A multipath framework for UDP traffic over heterogeneous access networks*. Tech. rep. IETF-Draft, 2020.

[66] Yannis Thomas, George Xylomenos, and George C. Polyzos. “Multipath congestion control with network assistance”. In: *Elsevier Computer Communications* 153 (2020), pp. 264–278.

[67] Nitinder Mohan, Tanya Shreedhar, Aleksandr Zavodavoski, Otto Waltari, Juss Kangasharju, and Sanjit K. Kaul. "Poster: Redesigning MPTCP for Edge Clouds". In: *Proc. of International Conference of MobiCom.* ACM, 2018, 675–677.

[68] multipath tcp.org. *Apple uses Multipath TCP.* `http://blog.multipath-tcp.org/blog/html/2018/12/15/apple_and_multipath_tcp.html`. [Online; accessed 10 May, 2020]. 2018.

[69] Quentin De Coninck and Olivier Bonaventure. "MultipathTester: Comparing MPTCP and MPQUIC in Mobile Environments". In: *Proc. of International Conference of TMA.* IFIP, 2019, pp. 221–226.

[70] Martin Becke, Hakim Adhari, Erwin P. Rathgeb, Fu Fa, Xiong Yang, and Xing Zhou. "Comparison of Multipath TCP and CMT-SCTP based on intercontinental measurements". In: *Proc. of International Conference on GLOBECOM.* IEEE, 2013, pp. 1360–1366.

[71] Quentin De Coninck and Olivier Bonaventure. "Multipath QUIC: Design and Evaluation". In: *Proc. of International Conference on CoNext.* ACM, 2017, 160–166.

[72] Alan Ford, Costin Raiciu, Mark Handley, Sebastien Barre, and J. Iyengar. *Architectural guidelines for multipath TCP development.* Tech. rep. IETF, RFC6824, 2011.

[73] *MultiPath TCP - Linux Kernel implementation.* `https://multipath-tcp.org`. [Online; accessed 10 May, 2020].

[74] Changqiao Xu, Jia Zhao, and Gabriel-Miro Muntean. "Congestion Control Design for Multipath Transport Protocols: A Survey". In: *IEEE Communication Surveys & Tutorials* 18.4 (2016), pp. 2948–2969.

[75] Costin Raiciu, Mark Handley, and Damon Wischik. *Coupled congestion control for multipath transport protocols.* Tech. rep. IETF, RFC6356, 2011.

[76] Dizhi Zhou, Wei Song, and Minghui Shi. "Goodput improvement for multipath TCP by congestion window adaptation in multi-radio devices". In: *Proc. of International Conference on CCNC.* IEEE, 2013, pp. 508–514.

[77] Shih-Hao Ou, Chih-Wei Huang, Tzu-Kuan Lee, and Chih-Yang Huang. "Out-of-order transmission enabled congestion and scheduling control for multipath TCP". In: *Proc. of International Conference on IWCMC.* IEEE, 2016, pp. 1069–1073.

[78] Yingwen Chen, Hong Va Leong, Ming Xu, Jiannong Cao, K. C. C. Chan, and A. T. S. Chan. "In-Network Data Processing for Wireless Sensor Networks". In: *Proc. of International Conference on MDM.* IEEE, 2006, pp. 26–26.

[79] Donghao Zhou, Zheng Yan, Yulong Fu, and Zhen Yao. "A survey on network data collection". In: *Elsevier Journal of Network and Computer Applications* 116 (2018), pp. 9–23.

[80] Julien Gedeon. "Edge Computing via Dynamic In-Network Processing". In: *Proc. of International Conference on NetSys.* IEEE, 2017, pp. 1–2.

[81] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. "Jumpgate: In-Network Processing as a Service for Data Analytics". In: *Proc. of International Workshop on HotCloud.* USENIX, 2019, pp. 1–11.

[82] Radu Stoenescu, Vladimir Olteanu, Matei Popovici, Mohamed Ahmed, Joao Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, and Costin Raiciu. "In-Net: In-network Processing for the Masses". In: *Proc. of International Conference on EUROSYS.* ACM, 2015, 23:1–23:15.

[83] Marcel Blöcher, Tobias Ziegler, Carsten Binnig, and Patrick Eugster. "Boosting Scalable Data Analytics with Modern Programmable Networks". In: *Proc. of International Workshop on DAMON.* ACM, 2018, 1:1–1:3.

[84] Jiarong Xing, Wenqing Wu, and Ang Chen. "Architecting Programmable Data Plane Defenses into the Network with FastFlex". In: *Proc. of International Workshop of Hotnets.* ACM, 2019, pp. 161–169.

[85] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. "All one needs to know about fog computing and related edge computing paradigms: A complete survey". In: *Elsevier Journal of Systems Architecture* 98 (2019), pp. 289–330.

[86] ETSI. *Mobile Edge Computing.* `https://www.etsi.org/images/files/ETSITechnologyLeaflets/MobileEdgeComputing.pdf`. [Online; accessed 10 May, 2020]. 2019.

[87] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. "Edge computing: A survey". In: *Elsevier Future Generation Computer Systems* 97 (2019), pp. 219–235.

[88] Ranesh Kumar Naha, Saurabh Garg, Dimitrios Georgakopoulos, Prem Prakash Jayaraman, Longxiang Gao, Yong Xiang, and Rajiv Ranjan. “Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions”. In: *IEEE Access* 6 (2018), pp. 47980–48009.

[89] IEEE. “IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing”. In: *IEEE Std 1934-2018* (2018), pp. 1–176.

[90] Nuyun Zhang, Hongda Li, Hongxin Hu, and Younghee Park. “Towards Effective Virtualization of Intrusion Detection Systems”. In: *Proc. of International Workshop on SDN-NFVSec.* ACM, 2017, pp. 47–50.

[91] ETSI. *ETSI-Open Source MANO (OSM).* `https://www.etsi.org/technologies/open-source-mano`. [Online; accessed 10 May, 2020]. 2020.

[92] Sai Qian Zhang, Ali Tizghadam, Byungchul Park, Hadi Bannazadeh, and Alberto Leon-Garcia. “Joint NFV placement and routing for multicast service on SDN”. In: *Proc. of International Conference on NOMS.* IFIP/IEEE, 2016, pp. 333–341.

[93] Jianing Pei, Peilin Hong, Kaiping Xue, and Defang Li. “Resource Aware Routing for Service Function Chains in SDN and NFV-Enabled Network”. In: *IEEE Transactions on Services Computing* (2018), pp. 1–1.

[94] Jianing Pei, Peilin Hong, Miao Pan, Jiangqing Liu, and Jingsong Zhou. “Optimal VNF Placement via Deep Reinforcement Learning in SDN/NFV-Enabled Networks”. In: *IEEE Journal on Selected Areas in Communications* 38.2 (2020), pp. 263–278.

[95] “Chapter 3 - Genesis of SDN”. In: *Software Defined Networks (Second Edition).* Ed. by Paul Gransson, Chuck Black, and Timothy Culver. Second Edition. Boston: Morgan Kaufmann, 2017, pp. 39–60.

[96] Andrew T Campbell, Herman G De Meer, Michael E Kounavis, Kazuho Miki, John B Vicente, and Daniel Villela. “A survey of programmable networks”. In: *ACM SIGCOMM Computer Communication Review* 29.2 (1999), pp. 7–23.

[97] Paul Gransson, Chuck Black, and Timothy Culver. “Chapter 10 - Network Functions Virtualization”. In: *Software Defined Networks (Second Edition).* Ed. by Paul Gransson, Chuck Black, and Timothy Culver. Second Edition. Boston: Morgan Kaufmann, 2017, pp. 241–252.

[98] Jaehyun Nam, Hyeonseong Jo, Yeonkeun Kim, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. "Operator-Defined Reconfigurable Network OS for Software-Defined Networks". In: *IEEE/ACM Transactions on Networking* 27.3 (2019), pp. 1206–1219.

[99] Theophilus Benson, Aditya Akella, and David A Maltz. "Unraveling the Complexity of Network Management". In: *Proc. of International Symposium on NSDI.* USENIX, 2009, pp. 335–348.

[100] Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Raghavan Raghavan, and James Wilcox. "Intelligent Design Enables Architectural Evolution". In: *Proc. of International Workshop of Hotnets.* ACM, 2011, pp. 3.1–3.6.

[101] Hyojoon Kim and Nick Feamster. "Improving network management with software defined networking". In: *IEEE Communications Magazine* 51 (Jan. 2013), pp. 114–119.

[102] Nick Feamster and Hari Balakrishnan. "Detecting BGP Configuration Faults with Static Analysis". In: *Proc. of International Symposium on NSDI.* USENIX, 2005, 43–56.

[103] Kevin Butler, Toni Farley, Patrick McDaniel, and Jennifer Rexford. "A Survey of BGP Security Issues and Solutions". In: *Proceedings of the IEEE* 98 (Feb. 2010), pp. 100–122.

[104] Evangelos Haleplidis, Kostas Pentikousis, Spyros Denazis, Jamal Hadi Salim, David Meyer, and Odysseas Koufopavlou. *Software-Defined Networking (SDN): Layers and Architecture Terminology.* Tech. rep. IETF, RFC7426, 2015.

[105] Joel M. Halpern and Jamal Hadi Salim. *Forwarding and Control Element Separation (ForCES) Forwarding Element Model.* Tech. rep. IETF, RFC5812, 2010.

[106] ONF. *OpenFlow V 1.3.* `http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html`. [Online; accessed 10 May, 2020].

[107] Martin Bjrklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF).* Tech. rep. IETF, RFC6020, 2010.

[108] Randy Presuhn, Jeffrey Case, Keith McCloghrie, Marshall T. Rose, and Steven Waldbusser. *Management Information Base (MIB) for the Simple Network Management Protocol (SNMP).* Tech. rep. IETF, RFC3418, 2002.

[109] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. “P4: Programming Protocol-Independent Packet Processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), 8795.

[110] David C. Plummer. *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware.* Tech. rep. IETF, RFC826, 1982.

[111] Suresh Krishnan, Siva Veerepalli, Eric Njedjou, Alper E. Yegin, and Nicolas Montavont. *Link-Layer Event Notifications for Detecting Network Attachments.* Tech. rep. IETF, RFC4957, 2007.

[112] *openflow-spec-v1.1.0.* `http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf`. [Online; accessed 10 May, 2020].

[113] Wikipedia. *OpenFlow Management and Configuration Protocol.* `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf`. [Online; accessed 10 May, 2020]. 2019.

[114] Marcelo R. Nascimento, Christian E. Rothenberg, Marcos R. Salvador, Carlos N. A. Corra, Sidney C. de Lucena, and Maurcio F. Magalhes. “Virtual Routers as a Service: The RouteFlow Approach Leveraging Software-Defined Networks”. In: *Proc. of International Conference on FIT.* ACM, 2011, pp. 34–37.

[115] T. V. Lakshman, Thyaga Nandagopal, Ramachandran Ramjee, K. Sabnani, and T. Woo. “The SoftRouter Architecture”. In: *Proc. of International Workshop on HotNets.* ACM, 2004, pp. 1–6.

[116] Ben Pfaff and Bruce Davie. *The Open vSwitch Database Management Protocol.* Tech. rep. IETF, RFC7047, 2013.

[117] Rob Enns, Martin Bjrklund, Andy Bierman, and Jrgen Schnwlder. *Network Configuration Protocol (NETCONF).* Tech. rep. IETF, RFC6241, 2011.

[118] David Harrington, Bert Wijnen, and Randy Presuhn. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks.* Tech. rep. IETF, RFC3411, 2002.

[119] Joel M. Halpern, Robert HAAS, Avri Doria, Ligang Dong, Weiming Wang, Hormuzd M. Khosravi, Jamal Hadi Salim, and Ram Gopal. *Forwarding and Control Element Separation (ForCES) Protocol Specification.* Tech. rep. IETF, RFC5810, 2010.

[120] Robert Thurlow. *RPC: Remote Procedure Call Protocol Specification Version 2.* Tech. rep. IETF, RFC5531, 2009.

[121] Wikipedia. *Representational state transfer.* `https://en.wikipedia.org/wiki/Representational_state_transfer`. [Online; accessed 10 May, 2020]. 2019.

[122] Wikipedia. *Common Object Request Broker Architecture.* `https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture`. [Online; accessed 10 May, 2020]. 2019.

[123] Felix Gessert an Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. “NoSQL database systems: a survey and decision guidance”. In: *Springer Computer Science - R&D* 32.3-4 (2017), pp. 353–365.

[124] Soheil Hassas Yeganeh and Yashar Ganjali. “Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications”. In: *Proc. of International Workshop on HotSDN.* ACM, 2012, pp. 19–24.

[125] Marco Canini, Iosif Salem, Liron Schiff, Elad M. Schiller, and Stefan Schmid. “A Self-Organizing Distributed and In-Band SDN Control Plane”. In: *Proc. of International Conference on ICDCS.* IEEE, 2017, pp. 2656–2657.

[126] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. “In-Band Synchronization for Distributed SDN Control Planes”. In: *ACM SIGCOMM Computer Communication Review* 46.1 (2016), 3743.

[127] David Ke Hong, Yadi Ma, Sujata Banerjee, and Z. Morley Mao. “Incremental Deployment of SDN in Hybrid Enterprise and ISP Networks”. In: *Proc. of International Symposium on SOSR.* ACM, 2016, pp. 1–7.

[128] Rashid Amin, Martin Reisslein, and Nadir Shah. “Hybrid SDN Networks: A Survey of Existing Approaches”. In: *IEEE Communications Surveys & Tutorials* 20.4 (2018), pp. 3259–3306.

[129] Arne Ludwig, Matthias Rost, Damien Foucard, and Stefan Schmid. “Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies”. In: *Proc. of International Workshop on HotNets*. ACM, 2014, 1–7.

[130] Microchip Technology Inc. *KSZ8795CLX*. `http://ww1.microchip.com/downloads/en/DeviceDoc/00002112B.pdf`. [Online; accessed 10 May, 2020]. 2016.

[131] Pica8.org. *Pica 8 OpenFlow Supported Switch*. `https://www.pica8.com/`. [Online; accessed 10 May, 2020]. 2016.

[132] Israr Iqbal Awan, Nadir Shah, Muhammad Imran, Muhammad Shoaib, and Nasir Saeed. “An improved mechanism for flow rule installation in-band SDN”. In: *Elsevier Journal of Systems Architecture* 96 (2019), pp. 1–19.

[133] Kun Qiu, Jing Yuan, Jin Zhao, Xin Wang, Stefano Secci, and Xiaoming Fu. “FastRule: Efficient Flow Entry Updates for TCAM-Based OpenFlow Switches”. In: *IEEE Journal on Selected Areas in Communications* 37.3 (2019), pp. 484–498.

[134] Hongli Xu, Jianchun Liu, Chen Qian, He Huang, and Chunming Qiao. “Reducing controller response time with hybrid routing in software defined networks”. In: *Elsevier Computer Networks* 164 (2019), p. 106891.

[135] Yuki Goto, Bryan Ng, Winston K.G. Seah, and Yutaka Takahashi. “Queueing analysis of software defined network with realistic OpenFlowbased switch model”. In: *Elsevier Computer Networks* 164 (2019), pp. 106–892.

[136] Reza Rahimi, M. Veeraraghavan, Y. Nakajima, H. Takahashi, Y. Nakajima, S. Okamoto, and N. Yamanaka. “A high-performance OpenFlow software switch”. In: *Proc. of International Conference on HPSR*. IEEE, 2016, pp. 93–99.

[137] Intel. *Home - DPDK*. `https://www.dpdk.org/`. [Online; accessed 10 May, 2020]. 2019.

[138] Eder Leo Fernandes, Elisa Rojas, Joaquin Alvarez-Horcajo, Zoltn Lajos Kis, Davide Sanvito, Nicola Bonelli, Carmelo Cascone, and Christian E. Rothenberg. “The road to BOFUSS: The basic OpenFlow userspace software switch”. In: *Elsevier Journal of Network and Computer Applications* 165 (2020), pp. 102–685.

[139] OVS. *Open vSwitch*. `http://openvswitch.org/`. [Online; accessed 10 May, 2020].

[140] Seppo Htnen, Petri Savolainen, Ashwin Rao, Hannu Flinck, and Sasu Tarkoma. “SWIFT: Bringing SDN-Based Flow Management to Commodity Wi-Fi Access Points”. In: *Proc. of International Conference on Networking.* IFIP, 2018, pp. 1–9.

[141] Seppo Htnen, Md Tanvir Ishtaique ul Huque, Ashwin Rao, Guillaume Jourjon, Vincent Gramoli, and Sasu Tarkoma. “An SDN Perspective on Multi-Connectivity and Seamless Flow Migration”. In: *IEEE Networking Letters* 2.1 (2020), pp. 19–22.

[142] Stefano Vissicchio, Laurent Vanbever, and Jennifer Rexford. “Sweet Little Lies: Fake Topologies for Flexible Routing”. In: *Proc. of International Workshop on HotNets.* ACM, 2014, 1–7.

[143] Marcel Caria, Admela Jukan, and Marco Hoffmann. “SDN Partitioning: A Centralized Control Plane for Distributed Routing Protocols”. In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 381–393.

[144] Franois Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier Bonaventure. “SCMon: Leveraging segment routing to improve network monitoring”. In: *Proc. of International Conference on INFOCOM.* IEEE, 2016, pp. 1–9.

[145] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. “Ravana: Controller Fault-Tolerance in Software-Defined Networking”. In: *Proc. of International Symposium on SOSR.* ACM, 2015, pp. 1–12.

[146] Amin Tootoonchian and Yashar Ganjali. “HyperFlow: A Distributed Control Plane for OpenFlow”. In: *Proc. of International Workshop on REN.* ACM, 2010, pp. 1–3.

[147] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Rao Kompella. “ElastiCon: an elastic distributed sdn controller”. In: *Proc. of International Confrence on ANCS.* ACM, 2014, pp. 17–28.

[148] Mohammad M. Tajiki, Mohammad Shojafar, Behzad Akbari, Stefano Salsano, Mauro Conti, and Mukesh Singhal. “Joint failure recovery, fault prevention, and energy-efficient resource management for real-time SFC in fog-supported SDN”. In: *Elsevier Computer Networks* 162 (2019), pp. 106–850.

[149] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. “B4: Experience with a globally-deployed software defined WAN”. In: *Proc. of International Conference on SIGCOMM*. ACM, 2013, pp. 3–14.

[150] Rinku Shah, Mythili Vutukuru, and Purushottam Kulkarni. “Cuttlefish: Hierarchical SDN controllers with adaptive offload”. In: *Proc. of International Conference on ICNP*. IEEE, 2018, pp. 198–208.

[151] Mohammed A. Togou, Djabir A. Chekired, Lyes Khoukhi, and Gabriel-Miro Muntean. “A Hierarchical Distributed Control Plane for Path Computation Scalability in Large Scale Software-Defined Networks”. In: *IEEE Transactions on Network and Service Management* 16.3 (2019), pp. 1019–1031.

[152] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. “Distributed SDN control: Survey, taxonomy, and challenges”. In: *IEEE Communications Surveys & Tutorials* 20.1 (2018), pp. 333–354.

[153] Woojin Shim and Keecheon Kim. “A Study on Communication Optimization in Multi-SDN Controller”. In: *Proc. of International Conference on ICOIN*. IEEE, 2019, pp. 461–464.

[154] Wenchao Xia, Jun Zhang, Tony Q. S. Quek, Shi Jin, and Hongbo Zhu. “Mobile Edge Cloud-Based Industrial Internet of Things: Improving Edge Intelligence With Hierarchical SDN Controllers”. In: *IEEE Vehicular Technology Magazine* 15.1 (2020), pp. 36–45.

[155] Mike Ojo, Davide Adami, and Stefano Giordano. “A SDN-IoT Architecture with NFV Implementation”. In: *Proc. of International Workshop on Globecom*. IEEE, 2016, pp. 1–6.

[156] Victoria Huang, Qiang Fu, Gang Chen, Elliott Wen, and Jonathan Hart. “BLAC: A Bindingless Architecture for Distributed SDN Controllers”. In: *Proc. of International Conference on LCN*. IEEE, 2017, pp. 146–154.

[157] Ivan Farris, Tarik Taleb, Yacine Khettab, and Jaeseung Song. “A Survey on Emerging SDN and NFV Security Mechanisms for IoT Systems”. In: *IEEE Communications Surveys Tutorials* 21.1 (2019), pp. 812–837.

[158] Robert Krosche, Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. “I DPID It My Way! A Covert Timing Channel in Software-Defined Networks”. In: *Proc. of International Conference on Networking.* IFIP, 2018, pp. 217–225.

[159] Mauro Conti, Pallavi Kaliyar, and Chhagan Lal. “CENSOR: Cloud-enabled secure IoT architecture over SDN paradigm”. In: *Wiley Concurrency and Computation: Practice and Experience* 31.8 (2019), e4978.

[160] Intidhar Bedhief, Meriem Kassar, Taoufik Aguili, Luca Foschini, and Paolo Bellavista. “Self-Adaptive Management of SDN Distributed Controllers for Highly Dynamic IoT Networks”. In: *Proc. of International Conference on IWCMC.* IEEE, 2019, pp. 2098–2104.

[161] Nakjung Nguyen Binh ; Choi, Marina Thottan, and Jacobus Van der Merwe. “SIMECA: SDN-based IoT Mobile Edge Cloud Architecture”. In: *Proc. of International Conference on IM.* IFIP/IEEE, 2017, pp. 503–509.

[162] Adel N. Toosi, Jungmin Son, Qinghua Chi, and Rajkumar Buyya. “ElasticSFC: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds”. In: *Journal of Systems and Software* 152 (2019), pp. 108–119.

[163] Mari Otokura, Kenji Leibnitz, Yuki Koizumi, Daichi Kominami, Tetsuya Shimokawa, and Masayuki Murata. “Evolvable Virtual Network Function Placement Method: Mechanism and Performance Evaluation”. In: *IEEE Transactions on Network and Service Management* 16.1 (2019), pp. 27–40.

[164] Bo Yi, Xingwei Wang, Min Huang, and Anwei Dong. “A multi-criteria decision approach for minimizing the influence of VNF migration”. In: *Elsevier Computer Networks* 159 (2019), pp. 51–62.

[165] Sisi Xiong, Qing Cao, and Weisheng Si. “Adaptive Path Tracing with Programmable Bloom Filters in Software-Defined Networks”. In: *Proc. of International Conference on INFOCOM.* IEEE, 2019, pp. 496–504.

[166] K. Tolga Bagci and A. Murat Tekalp. “SDN-enabled distributed open exchange: Dynamic QoS-path optimization in multi-operator services”. In: *Elsevier Computer Networks* 162 (2019), p. 106845.

[167] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. “Design, Implementation and Evaluation of Congestion Control for Multipath TCP”. In: *Proc. of International Symposium on NSDI*. USENIX, 2011, pp. 99–112.

[168] Ming Li, Andrey Lukyanenko, Sasu Tarkoma, Yong Cui, and Antti Yl-Jski. “Tolerating path heterogeneity in multipath TCP with bounded receive buffers”. In: *Computer Networks* 64 (2014), pp. 1–14.

[169] Kaiping Xue, Jiangping Han, Hong Zhang, Ke Chen, and Peilin Hong. “Migrating unfairness among subflows in MPTCP with network coding for wired–wireless networks”. In: *IEEE Transactions on Vehicular Technology* 66.1 (2017), pp. 798–809.

[170] Yihua Ethan Guo, Ashkan Nikravesh, Z. Morley Mao, Feng Qian, and Subhabrata Sen. “Accelerating Multipath Transport Through Balanced Subflow Completion”. In: *Proc. of International Confrence on MobiCom*. ACM, 2017, pp. 141–153.

[171] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. “ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths”. In: *Proc. of International Confrence on CoNEXT*. ACM, 2017, pp. 147–159.

[172] Jinhwan Kim, Bong-Hwan Oh, and Jaiyong Lee. “Receive Buffer based Path Management for MPTCP in heterogeneous networks”. In: *Proc. of International Conference on IM*. IFIP/IEEE, 2017, pp. 648–651.

[173] Yang Zhang, Hesham Mekky, Zhi-Li Zhang, Fang Hao, Sarit Mukherjee, and TV Lakshman. “SAMPO: Online subflow association for multipath TCP with partial flow records”. In: *Proc. of International Conference on INFOCOM*. IEEE, 2016, pp. 1–9.

[174] Pierrick Seite and Marc Blanchet. *Multiple Interfaces and Provisioning Domains Problem Statement*. Tech. rep. IETF, RFC6418, 2011.

[175] Felicián Németh, Balázs Sonkoly, Levente Csikor, and András Gulyás. “A large-scale multipath playground for experimenters and early adopters”. In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 481–482.

[176] Balázs Sonkoly, Felicián Németh, Levente Csikor, László Gulyás, and András Gulyás. “SDN based testbeds for evaluating and promoting multipath tcp”. In: *Proc. of International Conference on ICC*. IFIP/IEEE, 2014, pp. 3044–3050.

[177] Olivier Mehani, Ralph Holz, Simone Ferlin, and Roksana Boreli. "An early look at multipath TCP deployment in the wild". In: *Proc. of International Workshop on HotPlanet*. ACM, 2015, pp. 7–12.

[178] Fenghua Wang, Dongliang Xie, Jingyu Wang, Peng Zhang, and Yan Shi. "Paths selection-based resequencing queue length in concurrent multipath transfer". In: *Wiley International Journal of Communication Systems* 28.11 (2015), pp. 1805–1827.

[179] Harvey M. Salkin and Cornelis A. De Kluyver. "The knapsack problem: A survey". In: *Naval Research Logistics Quarterly* 22.1 (1975), pp. 127–144.

[180] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. "SIMPLE-fying middlebox policy enforcement using SDN". In: *ACM SIGCOMM computer communication review* 43.4 (2013), pp. 27–38.

[181] Recep Ozdag. "Intel® Ethernet Switch FM6000 Series-Software Defined Networking". In: *See goo.gl/AnvOvX* (2012), p. 5.

[182] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. "Frenetic: A network programming language". In: 46 (2011), pp. 279–291.

[183] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. "Towards an elastic distributed SDN controller". In: *ACM SIGCOMM Computer Communication Review* 43 (2013), pp. 7–12.

[184] Guang Yao, Jun Bi, Yuliang Li, and Luyi Guo. "On the capacitated controller placement problem in software defined networks". In: *IEEE Communications Letters* 18.8 (2014), pp. 1339–1342.

[185] Kévin Phemius, Mathieu Bouet, and Jérémie Leguay. "Disco: Distributed multi-domain SDN controllers". In: *Proc. of International Conference on NOMS*. IFIP/IEEE, 2014, pp. 1–4.

[186] Fei Hu, Qi Hao, and Ke Bao. "A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation". In: *IEEE Communications Surveys Tutorials* 16.4 (2014), pp. 2181–2206.

[187] Edsger W Dijkstra. "Self-stabilizing systems in spite of distributed control". In: *Communications of the ACM* 17.11 (1974), pp. 643–644.

[188] Volker Turau. “Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler”. In: *Elsevier Information Processing Letters* 103.3 (2007), pp. 88–93.

[189] Maria Gradinariu, Sébastien Tixeuil, et al. “Self-stabilizing vertex coloring of arbitrary graphs”. In: *Proc. of International Conference on OPODIS*. Springer, 2000, pp. 55–70.

[190] David Hock, Matthias Hartmann, Steffen Gebert, Thomas Zinner, and Phuoc Tran-Gia. “POCO-PLC: Enabling Dynamic Pareto-Optimal Resilient Controller Placement in SDN Networks”. In: *Proc. of International Conference on INFOCOM*. IEEE, 2014, pp. 115–116.

[191] *Ns-3.22 - Nsnam*. `https://www.nsnam.org/wiki/Ns-3.22`. [Online; accessed 10 May, 2020].

[192] *SNAP Autonomous systems AS-733 data set*. `http://snap.stanford.edu/data/as.html`. [Online; accessed 10 May, 2020].

[193] *SNAP Autonomous systems - Oregon-1 data set*. `http://snap.stanford.edu/data/oregon1.html`. [Online; accessed 10 May, 2020].

[194] Bob Lantz, Brandon Heller, and Nick McKeown. “A Network in a Laptop: Rapid Prototyping for Software-defined Networks”. In: *Proc. of International Workshop on Hotnets*. ACM, 2010, 19:1–19:6.

[195] *Open vSwitch database server*. `http://openvswitch.org/support/dist-docs/ovsdb-server.1.txt`. [Online; accessed 10 May, 2020].

[196] Hitesh Ballani and Paul Francis. “CONMan: a step towards network manageability”. In: *ACM SIGCOMM Computer Communication Review* 37 (2007), pp. 205–216.

[197] Xu Chen, Z. Morley Mao, and Jacobus van der Merwe. “Towards Automated Network Management: Network Operations Using Dynamic Views”. In: *Proc. of International Workshop on INM*. ACM, 2007, pp. 242–247.

[198] Andy Bierman, Martin Bjorklund, Kent Watsen, and Rex Fernando. “RESTCONF protocol”. In: *IETF draft, work in progress* (2014).

[199] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. “SANE: A Protection Architecture for Enterprise Networks”. In: *Proc. of International Symposium on Security*. USENIX, 2006, pp. 1–15.

[200] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. “Ethane: Taking control of the enterprise”. In: *ACM SIGCOMM Computer Communication Review* 37 (2007), pp. 1–12.

[201] Maulik Desai and Thyagarajan Nandagopal. “Coping with Link Failures in Centralized Control Plane Architectures”. In: *Proc. of International Conference on COMSNET.* IEEE, 2010, pp. 79–88.

[202] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. “PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric”. In: *ACM SIGCOMM Computer Communication Review* 39.4 (2009), pp. 39–50.

[203] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. “CAP for Networks”. In: *Proc. of International Workshop of HotSDN.* ACM, 2013, pp. 91–96.

[204] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. “Logically centralized?: state distribution trade-offs in software defined networks”. In: *Proc. of International Workshop on HotSDN.* ACM, 2012, pp. 1–6.

[205] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali. “On scalability of software-defined networking”. In: *IEEE Communications Magazine,* 51.2 (2013), pp. 136–141.

[206] Anderson Santos da Silva, Paul Smith, Andreas Mauthe, and Alberto Schaeffer-Filho. “Resilience support in software-defined networking: A survey”. In: *Elsevier Computer Networks* 92.1 (2015), pp. 189–207.

[207] Di Wu, Dmitri I. Arkhipov, Eskindir Asmare, Zhijing Qin, and Julie A. McCann. “UbiFlow: Mobility management in urban-scale software defined IoT”. In: *Proc. of International Conference on INFOCOM.* IEEE, 2015, pp. 208–216.

[208] Md Tanvir ul Huque, Weisheng Si, Guillaume Jourjon, and Vincent Gramoli. “Large-scale dynamic controller placement”. In: *IEEE Transactions on Network and Service Management* (2017), pp. 63–76.

[209] Lele Ma, Shanhe Yi, and Qun Li. “Efficient service handoff across edge servers via docker container migration”. In: *Proc. of International Confrence on SEC.* ACM, 2017, pp. 1–11.

[210] Mateus AS Santos, Bruno AA Nunes, Katia Obraczka, Thierry Turletti, Bruno T De Oliveira, and Cintia B Margi. "Decentralizing SDN's control plane". In: *Proc. of International Conference on LCN.* IEEE, 2014, pp. 402–405.

[211] Amin Tootoonchian and Yashar Ganjali. "HyperFlow: A Distributed Control Plane for OpenFlow". In: *Proc. of International Workshop on REN.* ACM, 2010, pp. 1–3.

[212] Qiaofeng Qin, Konstantinos Poularakis, George Iosifidis, and Leandros Tassiulas. "SDN Controller Placement at the Edge: Optimizing Delay and Overheads". In: *Proc. of International Conference on INFOCOM.* 2018, pp. 684–692.

[213] Marco Canini, Iosif Salem, Liron Schiff, Elad Michael Schiller, and Stefan Schmid. "Renaissance: A Self-Stabilizing Distributed SDN Control Plane". In: *Proc. of International Conference on ICDCS.* IEEE, 2018, pp. 233–243.

[214] Burak Grkemli, Sinan Tatlcolu, A. Murat Tekalp, Seyhan Civanlar, and Erhan Lokman. "Dynamic Control Plane for SDN at Scale". In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2688–2701.

[215] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. "DevoFlow: Scaling flow management for high-performance networks". In: *ACM SIGCOMM Computer Communication Review* 41 (2011), pp. 254–265.

[216] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. "SCL: simplifying distributed SDN control planes". In: *Proc. of International Symposium on NSDI.* USENIX, 2017, pp. 329–345.

[217] Cisco. *Cisco ACI Multi-POD and Multi-Site Demystified.* `https://netdesignarena.com/index.php/2018/07/01/cisco-aci-multi-pod-and-multi-site-demystified/`. [Online; accessed 10 May, 2020]. 2019.

[218] *UW CSE | Systems Research | Rocketfuel.* `https://research.cs.washington.edu/networking/rocketfuel`. [Online; accessed 10 May, 2020]. 2005.

[219] *OpenDaylight.* `http://www.opendaylight.org/`. [Online; accessed 10 May, 2020].

[220] ryu. *Ryu 4.30 documentation.* `https://ryu.readthedocs.io/en/latest/`. [Online; accessed 10 May, 2020]. 2019.

[221] Thomas Kohler, Frank Drr, and Kurt Rothermel. “ZeroSDN: A Highly Flexible and Modular Architecture for Full-Range Distribution of Event-Based Network Control”. In: *IEEE Transactions on Network and Service Management* 15.4 (2018), pp. 1207–1221.

[222] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* (1945), pp. 80–83.

[223] stress ng. *stress-ng - a tool to load and stress a computer system.* `http://manpages.ubuntu.com/manpages/xenial/man1/stress-ng.1.html`. [Online; accessed 10 May, 2020]. 2018.

[224] Spyros Makridakis, Steven C Wheelwright, and Rob J Hyndman. *Forecasting methods and applications.* John wiley & sons, 2008.

[225] Jan G De Gooijer and Rob J Hyndman. “25 years of time series forecasting”. In: *Elsevier International Journal of Forecasting* 3.22 (2006), pp. 443–473.

[226] Deval Bhamare, Raj Jain, Mohammed Samaka, and Aiman Erbad. “A Survey on Service Function Chaining”. In: *Elsevier Journal of Network and Computer Applications* 75.C (2016), pp. 138–155.

[227] Ali Mohammadkhan, Sheida Ghapani, Guyue Liu, Wei Zhang, K. K. Ramakrishnan, and Timothy Wood. “Virtual function placement and traffic steering in flexible and dynamic software defined networks”. In: *Proc. of International Workshop on LAN/MAN.* IEEE, 2015, pp. 1–6.

[228] Zafar Ayyub Qazi, Phani Krishna Penumarthi, Vyas Sekar, Vijay Gopalakrishnan, Kaustubh Joshi, and Samir R. Das. “KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core”. In: *Proc. of International Symposiam on SOSR.* ACM, 2016, pp. 1–2.

[229] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. “Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags”. In: *Proc. of International Symposium on NSDI.* USENIX, 2014, pp. 543–546.

[230] Gang Sun, Yayu Li, Dan Liao, and Victor Chang. “Service Function Chain Orchestration Across Multiple Domains: A Full Mesh Aggregation Approach”. In: *IEEE Transactions on Network and Service Management* 15.3 (2018), pp. 1175–1191.

[231] Mathieu Wion Adrien ; Bouet, Luigi Iannone, and Vania Conan. “Let there be Chaining: How to Augment your IGP to Chain your Services”. In: *Proc. of International Conference on Networking.* IFIP, 2019, pp. 1–9.

[232] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. “Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud”. In: *arXiv CoRR* abs/1305.0209 (2013).

[233] Paul Quinn, Elzur Uri, and Pignataro Carlos. *Network Service Header (NSH).* Tech. rep. IETF, RFC8300, 2019.

[234] Pamela Zave, Ronaldo A. Ferreira, Xuan Kelvin Zou, Masaharu Morimoto, and Jennifer Rexford. “Dynamic Service Chaining with Dysco”. In: *Proc. of International Confrence on SIGCOMM.* ACM, 2017, pp. 57–70.

[235] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. “Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization”. In: *Proc. of International Symposium on NSDI.* USENIX, 2018, pp. 373–387.

[236] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. “OpenNF: Enabling innovation in network function control”. In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 163–174.

[237] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. “Split/Merge: System Support for Elastic Execution in Virtual Middleboxes”. In: *Proc. of International Symposium on NSDI.* USENIX, 2013, pp. 227–240.

[238] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. “Elastic Scaling of Stateful Network Functions”. In: *Proc. of International Symposium on NSDI.* USENIX, 2018, pp. 299–312.

[239] Milad Ghaznavi, Nashid Shahriar, Shahin Kamali, Reaz Ahmed, and Raouf Boutaba. “Distributed Service Function Chaining”. In: *IEEE Journal on Selected Areas in Communications* 35.11 (2017), pp. 2479–2489.

[240] Junxiao Wang, Heng Qi, Keqiu Li, and Xiaobo Zhou. “PRSFC-IoT: A Performance and Resource Aware Orchestration System of Service Function Chaining for Internet of Things”. In: *IEEE Internet of Things Journal* 5.3 (2018), pp. 1400–1410.

[241] Binxu Yang, Wei Koong Chai, Zichuan Xu, V. Katsaros Konstantinos, and George Pavlou. “Cost-Efficient NFV-Enabled Mobile Edge-Cloud for Low Latency Mobile Applications”. In: *IEEE Transactions on Network and Service Management* 15.1 (2018), pp. 475–488.

[242] Zichuan Xu, Weifa Liang, Mike Jia, Meitian Huang, and Guoqiang Mao. “Task Offloading with Network Function Requirements in a Mobile Edge-Cloud Network”. In: *IEEE Transactions on Mobile Computing* 18.11 (2019), pp. 2672–2685.

[243] Kubernates. *Production-Grade Container Orchestration -Kubernates.* `https://kubernetes.io/`. [Online; accessed 10 May, 2020]. 2019.

[244] ETSI. *ETSI - ZSM - Zero touch network and service management.* `https://www.etsi.org/technologies/zero-touch-network-service-management`. [Online; accessed 10 May, 2020]. 2019.

[245] Docker Inc. *Docker SDK for Python.* `https://docker-py.readthedocs.io/en/stable/`. [Online; accessed 10 May, 2020]. 2019.

Intentionally Left Blank

Publications Related to Thesis

[T.1] Subhrendu Chattopadhyay, Sukumar Nandi, Samar Shailendra, and Sandip Chakraborty. "Primary Path Effect in Multi-Path TCP: How Serious Is It for Deployment Consideration?" In: *Eigthteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*. 2017, p. 36.

[T.2] Subhrendu Chattopadhyay, Samar Shailendra, Sukumar Nandi, and Sandip Chakraborty. "Improving MPTCP Performance by Enabling Sub-Flow Selection over a SDN Supported Network". In: *Fourteenth International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2018.

[T.3] Subhrendu Chattopadhyay, Niladri Sett, Sukumar Nandi, and Sandip Chakraborty. "Flipper: Fault-Tolerant Distributed Network Management and Control". In: *Fifteenth IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2017.

[T.4] Subhrendu Chattopadhyay, Soumyajit Chatterjee, Sukumar Nandi, and Sandip Chakraborty. "Aloe: An Elastic Auto-Scaled and Self-stabilized Orchestration Framework for IoT Applications". In: *Thirty Eighth IEEE International Conference on Computer Communications (INFOCOM)*. Vol. 38. 2019.

[T.5] Subhrendu Chattopadhyay, Soumyajit Chatterjee, Sukumar Nandi, and Sandip Chakraborty. "Aloe: Fault-Tolerant Network Management and Orchestration Framework for IoT Applications". In: *IEEE Transactions on Network and Service Management* (2020). **Accepted**.

[T.6] Subhrendu Chattopadhyay, Sukumar Nandi, Sandip Chakraborty, and Abhinandan Prasad. "Amalgam: Distributed Network Control With Scalable Service Chaining". In: *Nineteenth IFIP Networking Conference (IFIP Networking)*. 2020.

Brief Biography of the Author

Subhrendu Chattopadhyay born in West Bengal, India. After completion of school education, he has completed the B.Tech from *West Bengal University of Technology* (Now MAKAUT) in the area of *Computer Science and Engineering* in 2010. He has completed M.Tech from the *Department of Computer Science and Engineering*, *Indian Institute of Technology Guwahati*, India in 2014, where he has continued working towards the PhD degree under the supervision of **Prof. Sukumar Nandi**. He has received *TCS Research Scholar Fellowship* from Tata Consultancy Services, India for pursuing his PhD program. He has received best paper awards from *IEEE ANTS 2013*, *IEEE COMSNETS 2016* and best in session award from *IEEE INFOCOM 2019.* His research interests include network architecture and management, wireless networks, distributed algorithms. He is interested to develop a cognitive network management architecture.

Other Publications of the Author

[O.1] Shubha Brata Nath, Subhrendu Chattopadhyay, Raja Karmakar, Sourav Kanti Addya, Sandip Chakraborty, and Soumya K. Ghosh. "PTC: Pick-Test-Choose to Place Containerized Micro-Services in IoT". In: *2019 IEEE Global Communications Conference (GLOBECOM)*. 2019, pp. 1–6.

[O.2] Sandip Chakraborty, Sukumar Nandi, and Subhrendu Chattopadhyay. "Surpassing Flow Fairness in a Mesh Network: How to Ensure Equity among End Users?" In: *Seventh IEEE International Conference on Advanced Networks and Telecommunication Systems (ANTS 2013)*. 2013.

[O.3] Sandip Chakraborty, Subhrendu Chattopadhyay, Suchetana Chakraborty, and Sukumar Nandi. "Defending Concealedness in IEEE 802.11n". In: *Sixth IEEE International Conference on COMmunication System & NETworks (COMSNET 2014)*. 2014, pp. 1–8.

[O.4] Sushanta Karmakar and Subhrendu Chattopadhyay. "A Trigger Counting Mechanism for Ring Topology". In: *Thirty Seventh Australasian Computer Science Conference-Volume (ACSC 2014)*. 2014, pp. 81–87.

[O.5] Subhrendu Chattopadhyay, Sandip Chakraborty, and Sukumar Nandi. "Leveraging the Trade-off Between Spatial Reuse and Channel Contention in Wireless Mesh Networks". In: *Eighth International Conference on COMmunication System & NETworks (COMSNET 2016)*. Vol. 8. 2016.

[O.6] Sandip Chakraborty and Subhrendu Chattopadhyay. “ES2: Managing Link Level Parameters for Elevating Data Rate and Stability in High Throughput WLAN”. In: *Eighth International Conference on COMmunication System & NETworks (COMSNET 2016)*. Vol. 8. 2016.

[O.7] Niladri Sett, Subhrendu Chattopadhyay, Sanasam Ranbir Singh, and Sukumar Nandi. “A time aware method for predicting dull nodes and links in evolving networks for data cleaning”. In: *Fourteenth IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. 2016, pp. 304–310.

[O.8] Sandip Chakraborty, Sukumar Nandi, and Subhrendu Chattopadhyay. “Alleviating hidden and exposed nodes in high-throughput wireless mesh networks”. In: *IEEE Transactions on Wireless communications* 15.2 (2016), pp. 928–937.

[O.9] Pranav Kumar Singh, Subhrendu Chattopadhyay, Pradeepkumar Gajendra Bhale, and Sukumar Nandi. “Fast and Secure Handoffs for V2I communication in Smart City Wi-Fi Deployement”. In: *Fourteenth International Conference on Distributed Computing and Internet Technology (ICDCIT)*. 2017.

Index

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India