Enhancing Networks via Virtualized Network Functions

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Yang Zhang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Professor Zhi-Li Zhang

Spring, 2019

Acknowledgements

This dissertation is accomplished with tremendous help from many amazing people.

First of all, I would like to acknowledge my advisor, Professor Zhi-Li Zhang, for his continuous support, criticism, encouragement, and guidance on conducting research and achieving personal long-term career goals throughout my stay in graduate school. He taught me, among many things, how to transform my stilted speaking and writing into engaging storylines, and how to identify the simplicity at the core of complex technical concepts. His knowledge, wisdom, dedication, and strive for excellence amazed me, and I have been constantly inspired by his never-ending passion and commitment towards research.

I am also indebted to a variety of brilliant researchers outside of university over the years. I want to thank my collaborators: Ruben Torres, Fang Hao, Sarit Mukherjee, T V Lakshman, Bo Han, Bilal Anwer, Joshua Reich, Aman Shaikh, Vijay Gopalakrishnan, Jean Tourrilhes, and Puneet Sharma. I am grateful to all of them for their time, help, and the wisdom they have shared with me. Their valuable feedback helped me turn my research efforts into reality.

I cherish the awesome time that I spent with my lab mates in discussing research, daily life, and our future. In particular, Hesham Mekky, Eman Ramadan, Zhongliu Zhuo, Arvind Narayanan, and Zehua Guo. I have learned a lot through the discussions over the years.

I am thankful to all my fellow graduate students working with Professors David Du, Tian He, and Feng Qian for their valuable feedback on my research during our group meetings and for maintaining a friendly and intellectually creative environment.

I am grateful to Professors Abhishek Chandra, Yousef Saad, Georgios Giannakis, and David Du, who were very kind to serve as my committee members. I would like to especially acknowledge Professor Chandra here, who served as a committee member of my PhD qualification, thesis proposal, defense, and wrote recommendation letter for my fellowship application.

Finally, I thank my family and friends for support during this arduous but fruitful journey.

Dedication

To those who held me up over the years

Abstract

In an era of ubiquitous connectivity, various new applications, network protocols, and online services (e.g., cloud services, distributed machine learning, cryptocurrency) have been constantly creating, underpinning many of our daily activities. Emerging demands for networks have led to growing traffic volume and complexity of modern networks, which heavily rely on a wide spectrum of specialized network functions (e.g., Firewall, Load Balancer) for performance, security, etc. Although (virtual) network functions (VNFs) are widely deployed in networks, they are instantiated in an *uncoordinated* manner failing to meet growing demands of evolving networks.

In this dissertation, we argue that networks equipped with VNFs can be designed in a fashion similar to how computer software is today programmed. By following the blueprint of *joint* design over VNFs, networks can be made more effective and efficient. We begin by presenting Durga, a system fusing wide area network (WAN) virtualization on gateway with local area network (LAN) virtualization technology. It seamlessly aggregates multiple WAN links into a (virtual) big pipe for better utilizing WAN links and also provides fast fail-over thus minimizing application performance degradation under WAN link failures. Without the support from LAN virtualization technology, existing solutions fail to provide high reliability and performance required by today's enterprise applications. We then study a newly standardized protocol, Multipath TCP (MPTCP), adopted in Durga, showing the challenge of associating MPTCP subflows in network for the purpose of boosting throughput and enhancing security. Instead of designing a customized solution in every VNF to conquer this common challenge (making VNFs aware of MPTCP), we implement an online service named SAMPO to be readily integrated into VNFs. Following the same principle, we make an attempt to take consensus as a service in software-defined networks. We illustrate new network failure scenarios that are not explicitly handled by existing consensus algorithms such as Raft, thereby severely affecting their correct or efficient operations. Finally, we re-consider VNFs deployed in a network from the perspective of network administrators. A global view of deployed VNFs brings new opportunities for performance optimization over the network, and thus we explore parallelism in service function chains composing a sequence of VNFs that are typically traversed in-order by data flows.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The classical description of the Internet architecture is based on the hourglass model [3]. In this architecture, the only function in the neck of the hourglass are IP routers which determine routes and forward packets. Today, networks have been growing to meet new and sophisticated demands. For example, demands for securing networks – whether backbone networks of Internet service providers, campus/enterprise networks, data center networks, or even satellite networks – have been growing rapidly; carrier network keeps track of bandwidth consumption to bill users for usage; real-time streaming service designs new protocols for pursuing extremely low latency; IP depletion problem has been discussed for decades. There are many other requirements – load balancing, data compressing and caching, proxing, to name a few – need to be satisfied, and thus today's networks support far beyond merely forwarding packets.

To satisfy these demands for networks, virtual network functions (VNFs) are inserted into networks to perform specialized functions. VNFs run in one or more virtual machines on top of the hardware networking infrastructure, and provide functions such as transforming, inspecting, filtering, or otherwise manipulating traffic for purposes other than packet forwarding. Example of VNFs include the following.

- *Load balancers*. These functions distribute network or application traffic across a number of servers for the purpose of balancing traffic load required to be processed on each server.
- *Intrusion Detection/Prevention Systems (IDS/IPS)*. These functions inspect the entire network for suspicious traffic by analyzing packet header and payload; upon detection of malicious behaviors, the functions alert administrator and may block the traffic connection.
- *Proxies*. Depending on configurations, these functions act as intermediaries between endpoint

clients and servers to provide content filtering, censorship bypassing, caching, etc.

While VNFs are widely deployed to bring various benefits such as regulated control and performance enhancement, they are typically developed and deployed in an isolated way, without interacting with each other. VNFs are programmed as gigantic black-boxes and instantiated in an uncoordinated manner, failing to meet growing demands of evolving networks.

Thesis: *By following the blueprint of joint design over VNFs, networks can be made more effective and efficient.*

In this thesis, we advocate for a joint design in how VNFs are implemented and operated. Instead of designing and deploying VNFs separately – where each VNF has no interaction with others or hosts – we argue that VNF development should break barriers among isolated VNFs. Rather, VNFs should provide necessary interfaces to expose certain information and to be integrated into networks, allowing VNF developers or network administrators to fuse them together. As we will show, the jointly designing blueprint is feasible for VNF development and deployment in networks, and brings better network performance, security, and manageability.

1.1 Contributions and Organization

This dissertation explores network enhancement via virtualized network functions. The outline of this dissertation, along with the primary contributions of this dissertation are as follows:

In Chapter 2, we provide a review for required background knowledge including software-defined wide area network, multi-path TCP (MPTCP) awareness, consensus in software defined network, service function chaining, and parallelizable VNFs.

In Chapter 3, we fuse LAN Virtualization with SD-WAN Virtualization. We advance WAN-aware MPTCP (*WaMPTCP*) which fuses *LAN virtualization* with *WAN virtualization* for fine-grained load balancing and fast failover across WAN links. We present the detailed implementation of HydraNF, a comprehensive SD-WAN virtualization framework. We propose two new metrics to better capture the performance of tunnel handoff under link failures; through extensive evaluation in both emulated testbeds and real-world deployment, we show the superior performance of HydraNF over existing SD-WAN solutions.

In Chapter 4, we take a first step towards making the network devices MPTCP-aware by

investigating how to associate subflows that belong to the same MPTCP session. We present an online subflow association mechanism for MPTCP with partial flow record. The mechanism adopts a data sequence number (DSN)-based algorithm that can associate subflows based on analysis of DSN values of each subflow, their range, and overlapping pattern.

In Chapter 5, we make an attempt to take consensus as a service in software defined network. The reliance of network OS on consensus protocols to maintain *consistent* network state introduces an intricate inter-dependency between the network OS and the network under its control, thereby creating new kinds of fault scenarios or instabilities. Thus, we use Raft to illustrate the problems that this inter-dependency may introduce in the design of distributed SDN controllers and discuss possible solutions to circumvent these issues. We then propose a network-assisted Raft consensus algorithm that takes advantage of programmable network and offloads certain Raft [4] functionality to P4 [5] switches.

In Chapter 6, we parallelize VNFs for accelerating service function chains. We present HydraNF, a novel framework which intends to reduce SFC latency by parallelizing VNF processing. HydraNF statically analyzes SFCs to construct an abstract dataplane model that contains necessary information for SFC optimization. Based on the model, HydraNF employs novel algorithms to parallelize SFCs running entirely on a single server or across multiple servers, while correctly preserving sequential processing semantics. We conduct extensive evaluation of HydraNF prototype, using real network configurations and SFCs that contain both open-source and production NFs.

Chapter 7 presents concluding remarks, lessons learned, and thoughts for the future.

1.2 Bibliographic Note

The work presented in this dissertation has been published in several conferences and journals in networking field. The related publications are as follows:

- Yang Zhang, Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, T V Lakshman. Network-Assisted Raft Consensus Algorithm. In IEEE International Conference on Computer Communications (INFOCOM), 2016.
- Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, Zhi-Li Zhang. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In ACM SIGCOMM Symposium on SDN Research (SOSR), 2017.

- Yang Zhang, Eman Ramadan, Hesham Z Mekky, Zhi-Li Zhang. When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network. In ACM SIGCOMM 1st Asia-Pacific Workshop on Networking (APNET), 2017.
- Yang Zhang, Bo Han, Zhi-Li Zhang, Vijay Gopalakrishnan. Network-Assisted Raft Consensus Algorithm. In Proceedings of the SIGCOMM Posters and Demos (SIGCOMM Posters and Demos), 2017.
- Yang Zhang, Hesham Mekky, Zhi-Li Zhang, Ruben Torres, Sung-Ju Lee, Alok Tongaonkar, Marco Mellia. Detecting Malicious Activities with User-Agent Based Profiles. In International Journal of Network Management (IJNM), 2015

Chapter 2

Background and Motivation

2.1 Software Defined Wide Area Network

Many modern enterprises are geographically dispersed across multiple sites over a wide area network (WAN). Typically, branch office site networks are connected to a central office core network as well as a core data center (private cloud) via "dedicated" WAN links provisioned by one or more service providers. For security and privacy, WAN gateways at each site route enterprise traffic over VPN tunnels connecting edge networks with core/cloud networks. With increasing IT cloudification and IoT deployment, traffic loads are bumping up against the WAN link capacity [6]. Novel WAN *virtualization* solutions using software-defined wide area networking (SD-WAN) mechanisms are being developed and deployed for managing traffic traversing these WAN tunnels SD-WAN allows multiple WAN links (tunnels) to be logically combined into a virtual "big pipe" with higher capacity: it dynamically distributes traffic across heterogeneous WAN links for load balancing; when WAN link failures are detected, it employs *vertical tunnel handoff* to re-distribute traffic from failed links to other available links [7].

In Chapter 3, we argue that simple *tunnel handoff* employed by existing SD-WAN solutions fail to provide high reliability required by today's enterprise applications. We propose two new performance metrics to better capture the effect of tunnel handoff on standard (single-path) TCP connections under WAN network failures. Our experimental results show that WAN network failures significantly degrade application performance. We postulate that mitigating impacts of WAN failures on application performance requires *joint* architectural innovation at both SD-WAN gateways and connected end systems.

2.2 MPTCP Awareness

MPTCP is designed to boost data transmission throughput by taking advantage of multiple available paths in network. It is a major extension to TCP and has been standardized by the Internet Engineering Task Force (IETF) [8]. MPTCP allows a pair of hosts to use multiple interfaces for transmission of the upper layer data stream and has been becoming a promising technique. MPTCP can not only increase data throughput, but also seamlessly perform vertical handover between multiple paths, which makes the data transmission more robust against link failures [9]. Moreover, these features are obtained without requiring any modification at the application level. By large, MPTCP can be deployed in today's Internet without much impact on the proper functioning of the existing network devices [10].

The reason why MPTCP can achieve such benefits is because MPTCP directly relies on TCP as its subflow protocol. Once MPTCP subflows have been established, upper layer data can be scheduled to traverse over any of the established MPTCP subflow sessions. In order to coordinate across multiple paths, MPTCP adopts two levels of sequence numbers: a data sequence number (DSN) at MPTCP session level and a regular sequence number at MPTCP subflow session level. As in regular TCP, the subflow sequence number guarantees that the data sent over each subflow can be reliably received and assembled at the subflow receiver buffer. DSN is shared across multiple subflows and designed to guarantee reliable data delivery at MPTCP session level, i.e., to ensure the entire data stream can be assembled back in sequence.

Although MPTCP is designed to be compatible with most network devices, MPTCP can not be necessarily understood by these network devices. To the best of our knowledge, few network devices are designed with explicit consideration of MPTCP, and little work has been done to investigate how to better support this new protocol in the network. For example, MPTCP is designed to be no more aggressive than a regular TCP on a shared bottleneck link [11]. This implies that multiple subflows of a MPTCP session can only bring throughput improvement if the subflows do not share the same bottleneck link. However, no mechanism in either the end host or the network to allow MPTCP subflows to avoid common links currently exists. On the other hand, if routers could spread the MPTCP subflows onto disjoint paths, the overall data goodput could be greatly improved [12].

Furthermore, making network devices MPTCP-aware may improve the functionality of certain network services. Take application identification service or intrusion detection service as an

example. If an application/malware signature spans across multiple MPTCP subflows, the accuracy of the identification outcome may be improved by assembling subflows together. Likewise, when a subflow that carries the signature is identified/blocked, all other subflows belonging to the same MPTCP session can be identified/blocked too.

2.3 Consensus in Software-Defined Network

Software-defined networking (SDN) simplifies network devices by moving control plane functions to a logically centralized control plane; therefore data plane devices become simple programmable forwarding elements. SDN controllers use OpenFlow APIs [13] to set up forwarding rules and collect statistics at the data plane, which enables controller software and data plane hardware to evolve independently. Under SDN, physical connectivity between two end points do not guarantee they can communicate with each other – the underlying (logical) communication graph depends on the network policies reflected by the flow entries installed by the controller. For scalability and reliability, the logically centralized control plane ("network OS") is often realized via multiple SDN controllers (see Figure 5.1), forming a distributed system. Open Network Operating System (ONOS) [14] and OpenDayLight (ODL) [15] are two such Network OS examples supporting multiple SDN controllers for high availability.

In distributed network OS such as ONOS and ODL, the replicated controllers rely on conventional distributed system mechanisms such as consensus protocols for state replication and consistency. Paxos [16, 17] is a widely used distributed consensus protocol in production software [18–21] to ensure liveness and safety. Unfortunately, Paxos is very difficult to understand and implement in practical systems [4]. Raft [4] attempts to address these complexities by decomposing the consensus problem into relatively independent sub-problems: leader election, log replication, and safety. It implements a more "easy-to-understand" consensus protocol that manages a replicated log to provide a building block for building practical distributed systems. Both ONOS and ODL use certain implementations of Raft to ensure consistency among replicated network states. For example, ONOS maintains a global network view to SDN control programs that is logically centralized, but physically distributed among multiple controllers. It employs Raft to manage the switch-to-controller mastership and to provide distributed primitives to control programs such as ConsistentMap, which guarantees strong consistency for a key-value store.

The reliance of distributed network OS on consensus protocols to maintain *consistent* network state introduces an intricate *inter-dependency* between the network OS (as a distributed system) and the network it attempts to control. This inter-dependency may create new kinds of fault scenarios or instabilities that have neither been addressed in distributed systems nor in networking. In particular, it may severely affect the correct or efficient operations of consensus protocols, as will be expounded in Chapter 5. The key issue lies in the fact that the design of fault-tolerant distributed system mechanisms such as consensus algorithms typically focuses on server failures alone, while assuming the underlying network will handle connectivity issues on its own. For example, the design of Paxos or Raft assumes that the network may arbitrarily delay or drop messages; however, *as long as the network is not partitioned*, messages from one end point will *eventually* be delivered to another end point. Such assumptions about the network hold true in classical IP networks, where distributed routing algorithms running on routers cooperate with each other to establish new paths after failures. SDN now creates *cyclic dependencies* among *control network connectivity*, *consensus protocols*, and *control logic managing the network*, where the control logic managing the network is built on top of a distributed system (*e.g.,* ONOS) which relies on consensus protocols for consistency and control network connectivity for communication, whereas the network data plane (and control network) hinges on this distributed system to set up rules to control and enforce "who can talk to whom" among networking elements. Consequently, new failure scenarios can arise in SDN.

Besides new failure scenarios, consensus mechanisms typically involve expensive operations, especially for strong consistency guarantee, which may require multiple rounds of communications. Even without failure, consensus latency is bound to round-trip time between servers running consensus algorithms. Offloading application-level implementation of consensus algorithms to network may be a promising solution to improve the performance of consensus algorithms. Several recent projects investigate offloading consensus algorithms to either switches [22] or FPGA devices [23]. NetPaxos [22] proposes to implement the Paxos consensus algorithm in network by leveraging programmable switches. Besides the Paxos roles implemented on servers, NetPaxos requires one switch serving as a Paxos coordinator and several others as Paxos acceptors. NetPaxos can be implemented using P4 [5], a domain specific language that allows the programming of packet forwarding planes. However, Paxos consensus algorithm is very difficult to understand and implement due to its notoriously opaque explanation and lack of details for building practical systems [4]. Thus, offloading such a complex

consensus algorithm to network is error-prone. István *et al.* [23] takes the efforts of implementing the entire ZAB consensus algorithm [24] on FPGA devices using a low-level language which is difficult to program. Moreover, this hardware-based solution may not be scalable as it requires the storage of potentially large amounts of consensus states, logic, and even the application data.

2.4 Service Function Chaining and Parallelizable VNFs

A Service Function Chain (SFC) defines a sequence of VNFs, such as firewalls and load balancers (LBs), and stitches them together [25]. SFC has been a key enabler for network operators to offer diverse services and an important application of Software Defined Networking (SDN) [26–28]. Recently, operators can create, update, remove, or scale out/in network functions (NFs) *on demand* [29–31], construct a sequence of NFs to form a SFC [25], and steer traffic through it to meet service requirements [32–34]. However, virtualization and "softwarization" of NFs pose many new challenges [35]. In particular, traffic traversing virtualized NFs suffers from reduced throughput and increased latency, compared to physical NFs [32–34, 36]. The flexibility offered by SDN will enable more complex network services to be deployed, which will likely lead to longer SFC. As the length of an SFC (*i.e.,* number of NFs) increases, so does its overhead.

Exploiting parallelism to reduce packet processing latency and increase the overall system throughput is a classical approach that is widely used in networked systems. For example, most of today's web-based cloud computing applications take advantage of the stateless HTTP protocols for parallel HTTP transaction processing. Data analytics frameworks such as MapReduce and Spark utilize task level parallelism to speed up massive compute jobs using multiple servers. In terms of NFV, NF-level parallelism is first explored in ParaBox [1] and later in NFP [2] by exploiting order independence of certain NFs for parallel packet processing within an SFC. Both efforts focus on parallelizing packet processing for SFCs on a *single* (multi-core) server. Real-world NFs, on the other hand, will likely be operating in edge clouds or data centers with clusters of servers [37]. How to effectively utilize multiple servers to reduce per-packet processing latency and increase the overall system throughput is the main problem we explore.

Chapter 3

Fusing LAN Virtualization with WAN Virtualization

3.1 Main Results

We develop a novel *WAN-awareness* mechanism that enables end systems with MPTCP support (even with only one network interface) to generate multiple MPTCP subflows using *virtual subnet* addresses. This mechanism is enhanced at SD-WAN gateways to load-balance subflows across WAN links and dynamically reroute them away from failed WAN links in a scalable manner.

Building on top of *WAN-aware MPTCP* (WaMPTCP), we present *Durga*, a novel scalable SD-WAN virtualization framework which not only can aggregate multiple (heterogeneous) WAN links into a (virtual) "big pipe", but is also capable of providing fast failover with minimal application performance degradation. *Durga* is designed to handle diverse enterprise traffic. In addition to WaMPTCP for support of performance critical applications running on hosts with MPTCP kernel modules, it also incorporates MPTCP proxies for legacy TCP connections running on hosts with no MPTCP support as well as the default tunnel handoff mechanism for non-TCP traffic. In summary, we make the following contributions:

- We advance *WaMPTCP* (Section 3.2) which fuses *LAN virtualization* with *WAN virtualization* for fine-grained load balancing and fast failover across WAN links;
- We present the detailed implementation of Durga, a comprehensive SD-WAN virtualization framework (Section 3.3);

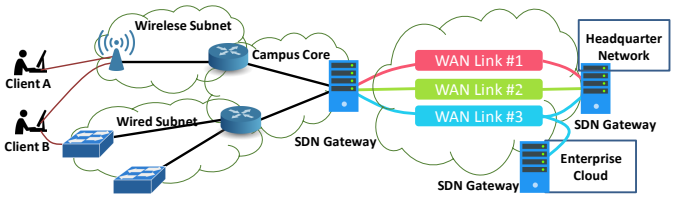


Figure 3.1: Motivating Scenarios

- We propose two new metrics (Section 3.4) to better capture the performance of tunnel handoff under link failures; through extensive evaluation (Section 3.5) in both emulated testbeds and real-world deployment, we show the superior performance of Durga over existing SD-WAN solutions.

3.2 Mitigating the Impact of WAN Link Failures

Let us consider Figure 3.1 depicting a simple enterprise branch site network consisting of two subnets (one wired and one wireless) to a campus core network with an SD-WAN gateway connecting to a central office site or a corporate private cloud via three separate WAN links (each, say, to a different WAN provider). Client A has only one network interface connected to WiFi subnet, whereas client B has two network interfaces, one connected to WiFi subset and the other to wired subnet. Neither clients have awareness of multiple WAN links available at the WAN gateway. Hence traditional SD-WAN systems use gateways for performing vertical handoffs in event of WAN link failures or policy change. The key question this project aims to answer is: what novel mechanisms can we develop to mitigate the impact of WAN link failures on application performance by taking advantage of software-define control of WAN links? We believe that a comprehensive solution with modifications to end-systems and SD-WAN gateways (connecting to multiple WAN links) is required.

3.2.1 End system MPTCP to the Rescue?

Given an end host with multiple interfaces, MPTCP [38] enables a TCP session to exploit these multiple parallel network paths to achieve higher throughput. MPTCP manages multiple subflows (each operating as a separate TCP connection with its own congestion control), and dynamically assigns data to each TCP subflow based on available capacity, thus load-balancing among multiple paths. If one path fails (or is highly congested), MPTCP automatically routes

data away from the failed (or highly congested) path, thereby alleviating the impact of network failures/congestion on application performance. The benefits of using MPTCP for load balancing, wireless handoff and coping with network failures have been widely studied [39–43]. Furthermore, switching from standard TCP to MPTCP is application-transparent – it requires no modification to applications. Hence it is natural to think about employing MPTCP at end systems (for those with OS support for MPTCP) for mitigating the impact of WAN link failures on application performance.

However, is employing MPTCP at end systems alone sufficient? Again considering the network in Figure 3.1, we assume that the OSes on both machines support MPTCP. Client A can not avail itself of the three available WAN links using MPTCP because it is only aware of one directly connected network interface. Thus, traffic from an application running on client A can only traverse one of the three available WAN links, failure of which will affect its performance. In the case of client B, it can employ MPTCP to generate two subflows, one over the WiFi subnet and the other over the wired subnet. However, the two subflows will converge at the SD-WAN gateway, and both may be routed along the same WAN link to their common destination. If this WAN link fails or is highly congested, MPTCP running on client B does not soften the impact of this WAN link failure/congestion. In general, we note that there can be a mismatch between the number of physical interfaces an end system has and the number of available WAN links at an SD-WAN gateway. Making end systems aware of the availability of WAN links at the gateway, enabling them to generate appropriate number of MPTCP subflows, and routing these subflows across different WAN links in a *scalable* manner at the SD-WAN gateway[1] is a challenging issue.

The above scenarios raise the following questions: i) *Is it possible to enable an MPTCP-compatible end system with only one network interface to exploit multiple WAN links via MPTCP?* ii) *Given an end system running MPTCP with multiple subflows, is there a scalable way to inform and ensure that the SD-WAN gateway always routes subflows of an MPTCP session across different WAN links?* For question i), an astute reader may suggest that one can use the **ndiffports** path manager option [38] in MPTCP to create multiple subflows across the same pair of IP addresses. However, all these MPTCP subflows will likely traverse the same WAN link, thus still suffering from the same issues as client B in the above scenario. An alternative approach

[1]The SD-WAN gateway could assign flows randomly to WAN links, e.g., using hashing, but this does not guarantee MPTCP subflows from one application will always traverse different WAN links.

is to deploy MPTCP proxies co-located at (or implemented as a module within) SD-WAN gateways [44–46]. This requires a pair of MPTCP proxies, e.g., one at the branch site and the other at the central office/private cloud site: a TCP connection from a client machine to a remote server is split by the MPTCP proxy at the local SD-WAN gateway to generate multiple MPTCP subflows, and the gateway routes them across different WAN links to the remote SD-WAN gateway on the other side of the WAN; these MPTCP subflows are then merged by the remote MPTCP proxy before routed to the server. Clearly, this approach incurs significant overheads, requiring MPTCP/SD-WAN gateways to keep track of every TCP connection over the enterprise network to the WAN. Morever, the congestion control operated by the TCP connection of an application and the MPTCP congestion control operated by MPTCP proxies can interact negatively, creating performance penalties for high-speed WAN links (Section 3.5).

3.2.2 WAN-aware MPTCP

We advance a novel scalable solution, dubbed *WAN-aware MPTCP (WaMPTCP)*. This solution combines a low-overhead mechanism at the end system side to create multiple *virtual* network interfaces driven by"WAN-awareness" (*even though the end system may only have one physical interface*), and a *scalable* mechanism at the SD-WAN gateway side that routes different MPTCP flows from the same application across different WAN links, detects WAN tunnel failures, and adaptively reroutes MPTCP flows from failed links to other available links. This is schematically depicted in Figure 3.2.

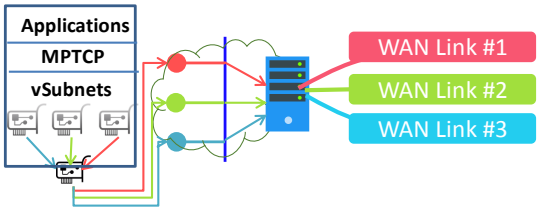


Figure 3.2: Virtualizing NIC for MPTCP to generate multiple subflows and for gateway to forward traffic in a stateless manner

The virtual network interfaces are *WAN-aware*, as each corresponds to one WAN link, each assigned from a (virtual) subnet address mapped to a WAN link at the SD-WAN gateway (and re-mapped to another available link if the current WAN link fails). Hence the SD-WAN

gateway can route the MPTCP subflows to the corresponding WAN link via a simple source-destination addresss-based flow lookup. The details regarding how the virtual subnet addresses are created and advertised to end systems and how WaMPTCP is implemented are discussed in Section 3.3.3. While WaMPTCP is developed as a key component of Durga, we remark that WaMPTCP is a general solution, and can be applied in other contexts as well.

3.3 Durga Overview and Mechanisms

Durga is designed specifically to address the requirements of SD-WAN for connecting branch office enterprise networks to a central office and/or private cloud over the Internet with multiple WAN links provisioned by several WAN service providers. The goals of Durga are multi-fold:

- Aggregate capacity of WAN links (as a single "big pipe") to deliver higher bandwidth than existing SD-WAN solutions;
- Support fast failover in the event of WAN network failure, and mitigate the impact of such failures on applications;
- Support diverse types of traffic (TCP, UDP, MPTCP);
- Maintain fewer states at SD-WAN gateways for scalability and reliability, and minimize end system software changes.

Durga framework implements multiple WAN virtualization mechanisms to support different types of clients and traffic. The most innovative component is WaMPTCP which supports application connections from MPTCP-compatible end systems where MPTCP subflows from each connection are routed across multiple WAN links for enhanced throughput and fast failover. For legacy end systems with no OS support for MPTCP, an MPTCP proxy implemented as part of Durga is employed to provide the same support [2]. In addition, UDP traffic from end systems (as well as TCP flows from non-performance critical applications) are handled using the default vertical tunnel handoff mechanism.

3.3.1 Tunnel Handoff

The Tunnel Handoff mechanism is a baseline of our framework and is used for both UDP and TCP traffic when a MPTCP proxy is not available. It depends on the tunnels between SDN

[2] As the number of legacy end systems is relatively small and continues to dwindle, the overhead incurred by MPTCP proxy becomes a less concern.

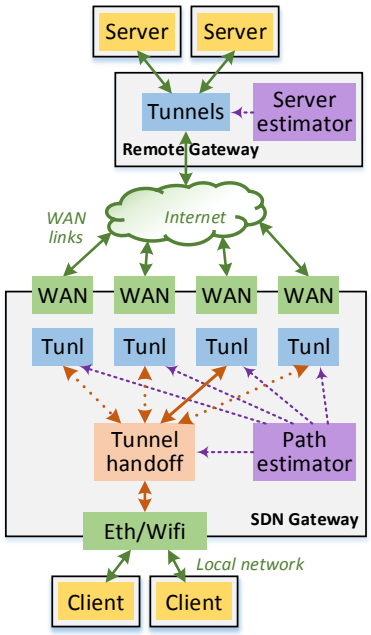


Figure 3.3: Tunnel Handoff

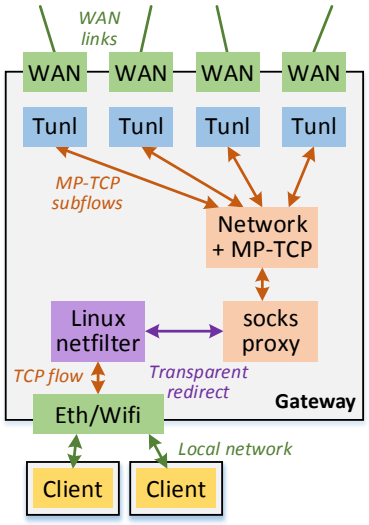


Figure 3.4: MPTCP Proxy

gateway and remote gateway to reroute IP traffic without having to change the external routing of the gateway on the Internet, which would take too much time. The tunnels provide a simple overlay, enabling both gateways to reroute traffic by only changing local rules. Our implementation (Figure 3.3) uses OpenvSwitch [47], VxLAN tunnels [48], and simple OpenFlow rules [49]. More secure tunnels can be implemented without changing the architecture.

Durga maps the traffic to the tunnels using one of two schemes. The first one is a strict priority scheme, where all the traffic is sent to the preferred link and fallbacks to a backup link as needed (OpenFlow failover group). The second one is a static load balancing scheme, where the TCP flows are distributed to the various WAN links using a hash based on the IP 5 tuples and static allocation for each WAN link (OpenFlow select group). We are looking into dynamic allocation of traffic to the WAN links. The mapping is always flow aware, to avoid splitting a flow across two tunnels.

Tunnels are monitored using the Bidirectional Forwarding Detection (BFD) protocol [50], which uses a simple periodic request/reply handshake between gateways. When BFD detects a failed path, the hash is automatically reconfigured to distribute the TCP flows on the remaining

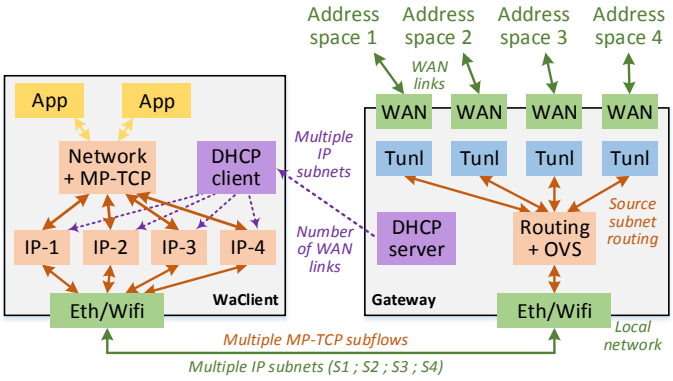


Figure 3.5: WaMPTCP

WAN links, using an OpenFlow group with fast failover property.

3.3.2 MPTCP Proxy

One way to address WAN link unawareness at end system is to originate all MPTCP flows at the gateway. The MPTCP stack on the gateway is directly mapped to WAN interfaces and can generate the right number of subflows. Some SD-WAN solutions exploit this by using a MPTCP proxy on the gateway [44–46], which converts a plain TCP connection into a MPTCP connection using all WAN links.

The plain TCP connection originating from end system is intercepted by the SDN gateway and directed to the MPTCP proxy. The MPTCP proxy terminates the TCP connection, opens a MPTCP connection on behalf of this connection, and joins those two connections (Figure 3.4). The MPTCP proxy can directly generate one MPTCP subflow for each WAN link of the SDN gateway. On the remote end, another MPTCP proxy may terminate the MPTCP connection and open a TCP connection to the intended destination.

We build our first MPTCP proxy (Figure 3.4) based on the OpenSource HPSockd proxy [51], and due to limited performance, we build a second MPTCP proxy based on the Open-Source Dante proxy [52]. We selected both of those proxies for their enterprise features including flexible access control rules and logging.

For each proxy, we added support for transparent proxying feature in addition to the original Socks protocol. The client must be modified to use the Socks protocol to initiate connections. With transparent proxying, TCP connections are redirected to the proxy without client

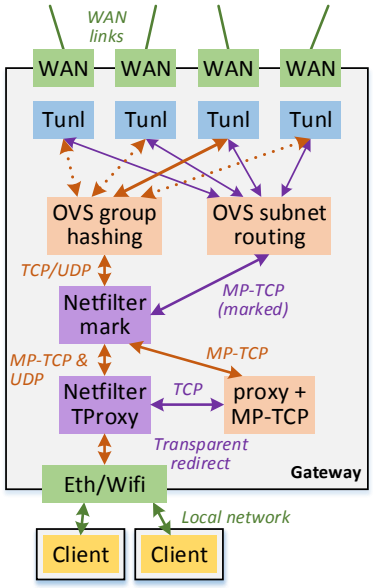


Figure 3.6: Durga Integration

intervention, which enables the use of unmodified applications on the client. We implemented transparent proxying using two netfilter mechanisms of the Linux kernel, NAT redirection [53], and TProxy [54]. Both mechanisms offer similar performance in our tests, but TProxy should be more scalable because it does not rewrite packet header.

MPTCP proxies offer the benefits of MPTCP : path aggregation, traffic re-balancing, and resilience to outages. However, using indirect TCP proxy has well known issues. Splitting the TCP connection at the proxy breaks the assumed end-to-end semantics of TCP [55]. The scalability of such proxy is problematic, as it needs to keep track of every TCP and MPTCP connection. Moreover, with these proxy implementations, we observed performance penalties for high speed WAN links (Section 3.5.1).

3.3.3 WAN aware MPTCP

Due to limitations of MPTCP proxies, we invented WaMPTCP, a new mechanism to address the lack of WAN link awareness at end system. The goal is to make MPTCP take advantage of multiple WAN links.

We want end system to initiate the same number of MPTCP flows as the number of available WAN links at a SDN gateway (or multiple of that number if the server is multihomed and fullmesh is used [38]), despite the end system having a single interface connected to the gateway. Further, the SDN gateway must map those subflows to the WAN links efficiently and in a scalable manner. The clients must be informed of the number of WAN links and generate subflows that makes mapping at the gateway easy.

WaMPTCP is composed of three parts, a specific IP subnet allocation for the local network, a modified IP provisioning at the client, and finally a simple MPTCP subflow routing on the gateway.

IP subnet allocation: For each network segment of the local network, instead of allocating a single IP subnet, we allocate one subnet associated with each WAN link (gateway side of Figure 3.5). To simplify routing, an IP address space is associated with each WAN link, and then one subnet is created in each IP address space for each local network segment served by the gateway, based on the networking topology inside a campus network.

Direct routing adds additional constraints on the IP address space. Direct routing enables MPTCP to be routed directly to the Internet without any tunneling, instead of being restricted to the tunnels (which is the case for Tunnel Handoff). If using NAT, those subnets must reside in the private IP address space. Otherwise, each subnet associated with a WAN link must be within the IP address space routed on the Internet to that WAN link. This guarantees that the return path of any MPTCP subflow is valid and congruent with the forward path, a requirement for direct routing.

IP provisioning: We modify IP provisioning to inform the client of multiple IP subnets available on the network segment where it is connected and assign it one IP address associated with each WAN link (client side of Figure 3.5).

For IPv4, we modify the DHCPv4 protocol [56], we insert in the DHCP response a DHCP option informing the clients about the various subnets available. DHCP is a flexible protocol and allows custom DHCP options, the WaMPTCP option is a simple string listing the various subnets available on the network segment. DHCPv4 client is modified to request an IP address in each subnet of the list and configure those IP addresses on the network interface using IP aliases. We configured the ISC DHCP server and modified the ISC DHCP client to implement this option [57]. If the number of WAN links changes (not frequent in enterprise networks), DHCP TTL could be used to refresh the number of subnets on hosts.

IPv6 is simpler and does not require client modifications, because multiple IP addresses per interface is part of the IPv6 standard. DHCPv6 server is configured to assign multiple IP addresses to the client, one in each subnet, and most DHCPv6 clients automatically configure those IP addresses on the network interface. ISC DHCP server does not support multiple subnets per segment, and therefore we use the dhcpy6d server with the appropriate configuration [58]. IPv6 may alternatively use Stateless Address Autoconfiguration (SLAAC) [59]. We did not implement SLAAC, but the gateway would need to broadcast multiple Router Advertisement messages, one for each subnet.

Gateway routing: Routing on the SDN gateway forwards any packet originated in a subnet to the associated WAN link. This uses simple routing based on source subnet. A few static rules can match each source subnet, and this is supported by most OSes. Packets coming from the WAN links are routed based on destination subnet (classical routing). Our routing technique is implemented to support VxLAN tunneling to a remote gateway and direct routing to the Internet without tunneling.

Here is an example on how WaMPTCP works. Assigning multiple IP addresses to the interface of a client creates multiple virtual interfaces in the network stack, one per WAN link, the MPTCP stack by default uses all the virtual interfaces, and therefore it generates one MPTCP subflow using each source IP address. Assuming there are two WAN links available, A and B, each link would have an IP address space associated with it, IPSA and IPSB. If direct routing is needed, the gateway routes IPSA on link A and IPSB on link B. **IP subnet allocation** assigns two subnets, IPSA.s and IPSB.s, to each network segment. If client c joins the local network, **IP provisioning** allocates two addresses to its network interface, IPSA.s.c and IPSB.s.c. Each MPTCP session generates two subflows, with respective source addresses IPSA.s.c and IPSB.s.c. **Gateway routing** directs packets with source address IPSA.s.c to link A and packets with source address IPSB.s.c to link B.

3.3.4 MPTCP Recovery Optimization

During a path failure, a MPTCP subflow using the path is affected: packets don't reach the receiver, and the TCP congestion algorithm triggers retransmissions and eventually timeouts, causing the subflow to get stalled. When a subflow is stalled, it does not send packets to probe the path and can suffer an outage much longer than the actual L3 outage (Section 3.5.2). In

certain circumstance, the subflow cannot resume at all, due to the TCP exponential backoff algorithm. We propose MPTCP Recovery Optimization (RO) to address stalled MPTCP subflows on failed paths, and it can be optionally applied to both MPTCP proxy and WaMPTCP.

One obvious solution would be to crank up the retransmission frequency of all the MPTCP subflows, so that the subflows do not wait such a long time before probing the path again. However, this has a number of downsides. First, this is not a standard configuration of MPTCP stack, so we would need a new mechanism for the gateway to communicate with end systems to use more aggressive retransmissions. Second, if subflows probe failed path too often, probing packets consume extra resources on the local network and on the gateway, possibly reducing the performance of subflows using healthy paths. Third, probing packets carry MPTCP session data, and this data must first be sent on the failed subflow, wait for the failure timer, before being resent on the healthy subflow. This is similar to what is measured by the T-latency metric (Section 3.4), and this can add significant latency to the data (around 300ms - Section 3.5.3).

Our solution is to combine strength of MPTCP and tunnel handoff and to allow the gateway to probe efficiently for failed subflows. When tunnel handoff detects that an path is down, it reroutes MPTCP subflows on that path to an alternate healthy path, similar to what it does for non-MPTCP traffic (Section 3.3.1). When those MPTCP subflow are on a healthy path, they can resume progress and it prevents them to stall. At this point, the subflow interfere with the subflow that was already on that path, but the effect is negligible (Section 3.5.2). Once tunnel handoff detects that the path has recovered, it reroutes all the affected TCP subflows back to their original path. This MPTCP recovery optimization is able to improve the recovery time of MPTCP sessions.

3.3.5 Mechanisms Integration

An additional challenge was integrating all the mechanisms together on the same gateway and making sure connections are routed to the proper mechanism. The integration is based on Netfilter [60] and OpenvSwitch [47] which are part of the Linux kernel (Figure 3.6). Filtering packet for the proxy is optional and done first using the TProxy mechanism [54]. Then, Netfilter mark the MPTCP packet, because OpenvSwitch does not have this field in its classifier. In OVS, there is two set of OpenFlow rules predicated on this mark, the rules for TCP and UDP use an OpenFlow failover or select group, whereas the rules for MPTCP match on the source subnet.

3.4 Vertical Handoff Performance Metrics

Tunnel Handoff and MPTCP use two completely different way to implement handoff, therefore we propose two new metrics for measuring the impact of various SD-WAN management techniques on application and network session performance. These metrics can not only capture the SD-WAN performance but also motivate and drive the design of Durga.

Prior studies have defined L3 metrics for measuring handoff performance under various network settings, including SDN (see, e.g., [61–64]). In addition, rerouting events adversely affect TCP performance [65], so several TCP optimization schemes have also been proposed, e.g., [64, 66, 67]. Most prior studies examine the impact of link failures on TCP performance by observing congestion window reduction, instant TCP throughput degradation, or packet losses.These metrics are too specific and are not directly related to the application experience. For this reason, we propose two new metrics to evaluate the impact of vertical tunnel handoff on TCP connections. For these new metrics to be useful, we pose the following requirements: 1) be a direct measurement of the effect of handoff; 2) applicable to both reactive and proactive handoff, due to path outages or policy changes; 3) relate directly to the application performance.

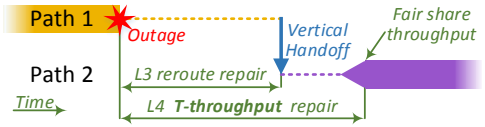


Figure 3.7: L4 T-throughput repair

The first metric is *T-throughput*: time for throughput repair (Figure 3.7). It measures the time elapsed from the start of the outage or policy change, i.e. as soon as the link becomes undesirable, to when the throughput of an affected TCP connection is fully restored. As the conditions on the old link and the new link may be different, the expected throughput on the new link is likely different from the old link. Therefore, we consider that the TCP throughput has been fully restored when it has obtained its TCP-fair share of the (new) link bandwidth. In other words when the TCP slow start phase has ended and the throughput becomes stable. This metric provides a better measure of the effect of handoff on an application that is *bandwidth-bound* (e.g., a file download), as it quantifies how long an application suffers performance degradation before it fully recovers. This metric is a function of how efficient a TCP congestion control algorithm operates after the handoff.

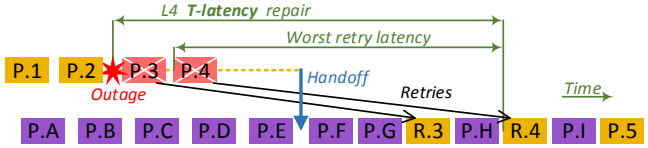


Figure 3.8: L4 T-latency repair

The second metric is *T-latency*: time for latency repair (Figure 3.8). It measures the time elapsed from the start of the outage or policy change, to when all packets that were lost during the handoff are retransmitted by a TCP connection. This metric provides a better measure of the effect of handoff on an application that is *latency-bound*, e.g., interactive gaming over TCP. This metric is a function how efficient a TCP retransmission mechanism operates after the handoff.

A key benefit of these two metrics lies in that they allow us to evaluate how TCP configurations and network conditions affect the network failure recovery time. In particular, they enable us to design better failover and recovery mechanisms under MPTCP and evaluate them fairly against existing (single-path) TCP mechanisms.

3.5 Evaluation of Durga

We evaluate Durga in both emulated testbeds and real-world deployment, allowing us to evaluate performance across a wide range of workloads and network conditions.

We compare the following mechanisms implemented in the experiments:

- **Tunnel Handoff** is a baseline of Durga framework (Section 3.3.1). It uses Open vSwitch version 2.4.2 [47] to reroute flows across VxLAN tunnels. Unless specified, the traffic is plain TCP, and some tests do use plain MPTCP without WAN awareness;
- **MPTCP Proxy** is a transparent proxy solution (Section 3.3.2). By default, MPTCP Proxy in our experiments is our proxy implementation based on Dante Socks proxy version 1.4.1 [52]. For performance comparison, we also use our modified version of HPsockd v0.17 [51];
- **MPTCP Tunnel [68]** is a overlay solution which tunnels MPTCP over TCP. All TCP flows are encapsulated in a single MPTCP flow between gateways. We use the implementation available online [69];
- **WaMPTCP** implements WAN aware MPTCP in end systems and gateways (Section 3.3.3). End systems are provided with one IP address per WAN link, and generate appropriate number of MPTCP subflows. *Fullmesh*, the default MPTCP path manager, is used, and gateways

route MPTCP subflows based on source subnets;

- **WaMPTCP+RO** implements WAN aware MPTCP with MPTCP recovery optimization (Section 3.3.4). When the gateway detects an path failure, it reroutes infected MPTCP subflows away from the failed path, and reroutes back when the path is recovered.

Except for the bandwidth aggregation experiments in Section 3.5.1, controlled testbed setup is described as follows. The testbed uses six Ubuntu 16.04 servers with MPTCP 0.93 implementation. The setup is similar to Figure 3.3: two gateways, two clients, and two servers. The gateways have two pairs of Ethernet interfaces, and they are configured at 100Mb/s to emulate a wide area network. The remote gateway terminates VxLAN tunnels originated at the SDN gateway and routes to the servers. One legacy client and one WaMPTCP client connect to a local SDN gateway at 1Gb/s. Through local and remote gateways, clients can access servers which are connected to the remote gateway at 1Gb/s. One pair of client and server is used for generating background traffic as needed.

3.5.1 Bandwidth Aggregation Evaluation

Table 3.1 shows the overall throughput of the 4 mechanisms with a single IPv4 connection, a single IPv6, or 10 IPv4 connections. These tests use 5 WAN links configured at 1 Gb/s between the gateways, the client and server use 10Gb/s.

Direct Routing (i.e., no VxLAN tunnels) can only use the default path for all plain TCP and plain MPTCP connections due to the constraints of routing. Tunnel Handoff is only available with VxLAN tunnels, and can only aggregate bandwidth when there are multiple connections. Dante proxy offers much greater performance than HPsock proxy, but suffers with VxLAN tunnels due to the smaller MTU and higher CPU load. WaMPTCP performs the best because it uses all available bandwidth in all conditions.

Mb/s	Direct Routing			VxLAN tunnels		
	IPv4	IPv6	10 IPv4	IPv4	IPv6	10 IPv4
Plain TCP	941	928	944	908	893	4550
Plain MPTCP	928	915	931	893	888	4480
MPTCP Tunnel	120	N/A	119	114	N/A	112
HPsock Proxy	740	N/A	1050	752	N/A	876
Dante Proxy	4285	3925	4550	3233	2763	3610
WaMPTCP	4606	4531	4640	4433	4221	4490

Table 3.1: Throughput aggregation comparison

3.5.2 Handoff Traces Evaluation

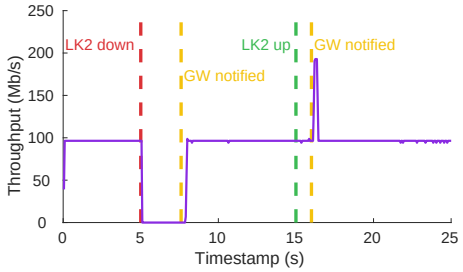


Figure 3.9: Tunnel Handoff

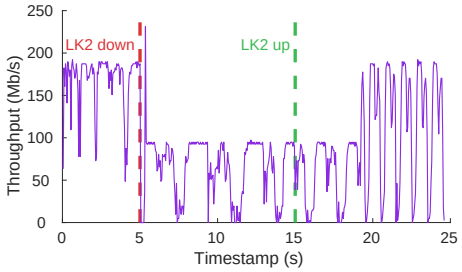


Figure 3.10: MPTCP Tunnel

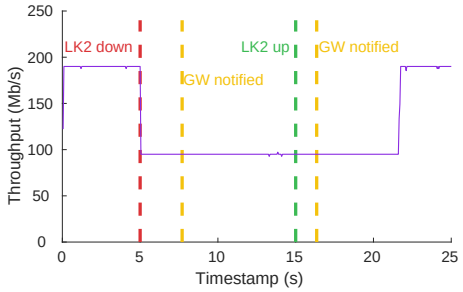


Figure 3.11: WaMPTCP

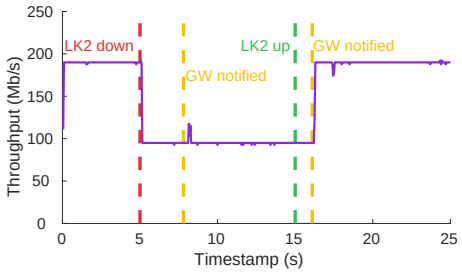


Figure 3.12: WaMPTCP + RO

Figures 3.9, 3.10, 3.11 and 3.12 show the throughput of a single connection during an outage on a link (labeled as "LK2" in the figure). The outage is emulated by blocking the primary link after 5 seconds, and unblocking it after another 10 seconds, whereas the backup link is always healthy. The primary link is blocked using an iptables/netfilter rule [60], which is almost instantaneous. We decompose an outage into two phases: failover (which typically harms throughput) and recovery (which typically benefits throughput). We don't simulate policy changes, since in most cases, it would be similar to the recovery phase of an outage. MPTCP proxy behaves almost identically to WaMPTCP, so it is not shown.

Figure 3.9 shows that tunnel handoff does not aggregate available bandwidth on the two WAN links together, and it experiences more than 3 seconds down-time during failover. Approximately 3 seconds after the link is blocked, BFD considers the path as failed, and tunnel handoff reroutes the affected flow to the healthy link, and it resumes progress. After the link is unblocked, BFD eventually discovers that the path is healthy, and reroutes the flow to its original path. In our experiment, the throughput spikes shortly after recovery. This is because prior to handoff, the flow has filled the send buffers of the NIC and the tunnel endpoint on

the gateway before the bottleneck link. After handoff, the flow moves to an empty link, and can therefore transmit immediately, while the other link is still draining the send buffers. This creates out of order packets [65], however the Linux TCP/IP stack has many optimization to overcome this [64].

Figure 3.11 and 3.12 show that during failover phase, there is no down-time because MPTCP natively supports resilience by maintaining multiple subflows. Just after the failure, the throughput of MPTCP goes down a little bit and quickly reaches fair share. Conceptually, suspension of one subflow does not affect the other one if the other flow has already reached its fair share.

Figure 3.11 shows that when Recovery Optimization in not used, the subflow get stalled (Section 3.3.4). When the path become available again, BFD takes less than 1 second to detect the link recovery, however throughput need further 5 seconds to recover. Further experimentation shows that the L4 throughput repair time increases with the length of the outage and is due to TCP RTO backoff (Figure 3.16).

Figure 3.12 shows that Recovery Optimisation prevents the subflow to get stalled, and that it recovers the bandwidth of the second path very quickly after being moved to the recovered path. The aggregated throughput dips and spikes slightly when the gateway is notified of the failure (at 8s). With the MPTCP proxy, those dips and spike are more pronounced. The two subflows sharing the healthy path interact with each other and create out-of-order packets, causing our measurement application to mistakenly computes instant throughput higher or lower than the real value.

MPTCP Tunnel in Figure 3.10, does not exhibit stable throughput even without any background traffic. We conjecture that it is caused by the interaction between the outer TCP control loop (end-to-end) and inner TCP control loop (tunnel), which is a well know issue of any TCP in TCP encapsulation [70].

3.5.3 Handoff Metric Evaluation

Visual examination of traffic traces does not convey the full picture, and in order to quantitatively study handoff impact on transport layer, we need use the new metrics we designed, *T-throughput* and *T-latency* (Section 3.4). We evaluate those metrics both for the *failover* phase and the *recovery* phase.

For tunnel handoff, the factor that has the biggest impact on handoff performance is the configuration of the **BFD protocol**, used to evaluate the health of the WAN link and the Internet

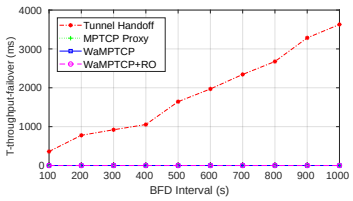


Figure 3.13: BFD impact over T-throughput-failover

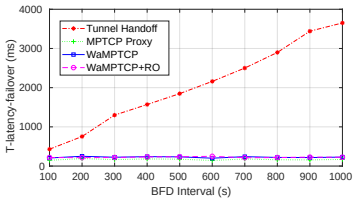


Figure 3.14: BFD impact over T-latency-failover

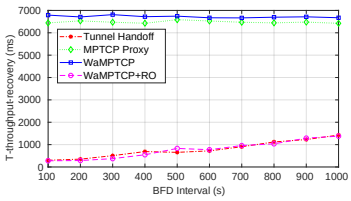


Figure 3.15: BFD impact over T-throughput-recovery

path (Section 3.3.1). Figures 3.13, 3.14, 3.15 and Table 3.2 show both metrics for both phases, with various mechanisms and various setting of the BFD timer.

seconds	**T-throughput-failover**		**T-latency-failover**		**T-throughput-recovery**		**T-latency-recovery**
BFD timer	*1s*	*100ms*	*1s*	*100ms*	*1s*	*100ms*	
Tunnel Handoff	3.77	0.63	3.52	0.56	1.33	0.42	0
MPTCP Tunnel	0		0.34		6.57		0
MPTCP Proxy	0		0.37		6.41		0
WaMPTCP	0		0.32		6.86		0
WaMPTCP+RO	0		0.32		1.28	0.44	0

Table 3.2: Vertical handoff performance of different mechanisms

Tunnel Handoff and Recovery Optimization rely on BFD, and consequently are impacted by the value of the BFD timer. The period of BFD handshakes is equal to the BFD timer, and by default BFD declares the path failed after 3 failed handshake. Our results pretty much confirm this and show that the performance at TCP level when using BFD is mostly dominated by the Layer 3 handoff. Mechanisms not using BFD are obviously not impacted by this setting. Tunnel handoff would need to use a very small BFD timer to compare to how MPTCP handles failures, increasing overhead and the risks of false positives.

The **metrics** do confirm many of our earlier findings. MPTCP provides a much better failover performance, and MPTCP without Recovery Optimization suffer worse recovery performance due to stalled flows. *T-throughput-failover* is zero with MPTCP, which confirm that the throughput of the subflow on the healthy link is not impacted. *T-latency-recovery* is always zero, because there is no retransmissions[3], the handoff is between two healthy link and cause no packet loss.

The traffic traces did show near instant handoff for MPTCP, the metrics are useful to show that this is not the case. *T-latency-failover* shows that for all MPTCP mechanisms, some data get stuck for around 320ms on the failed subflow before being retransmitted on the healthy

[3]Even though packet retransmission may be triggered due to congestion, we do not take it as caused by failure.

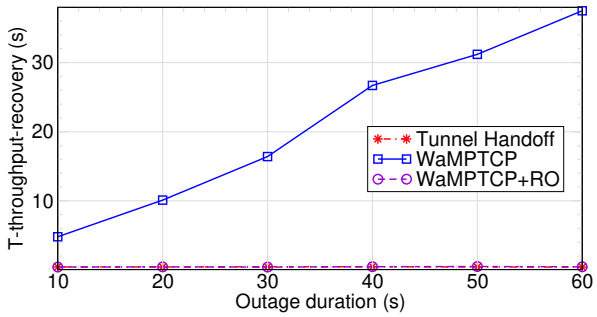


Figure 3.16: Outage impact on T-recovery-time (BFD=100ms)

subflow. This could impact latency sensitive applications.

Figure 3.16 shows the impact of the **outage duration** on the recovery phase and the *T-throughput-recovery* metric. Without Recovery Optimization, MPTCP recovery depends on TCP congestion control, and grows with outage duration. During outage, TCP reconnects after one second by default in Linux, and the retry interval grows exponentially. Linux default number of retries is 15, and thus the maximum outage that can be tolerated is 924.6 seconds. For high-availability server, the suggested number of retries is 3 [71]. In this case, if the outage duration is larger than seconds, a stalled subflow will never resume.

In contrast, Tunnel Handoff and Recovery Optimization are using BFD for recovery, therefore are independent of the outage duration. BFD is using a constant periodic timer, the subflow is not stalled, and therefore *T-throughput-recovery* is approximately layer 3 recovery time plus the delay for congestion window (cwnd) growth. Congestion window growth closely depends on round trip time, bandwidth, and congestion on the forwarding path (Section 3.5.4).

3.5.4 TCP/IP Congestion Control

Figure 3.17 shows that for all mechanisms *T-throughput-recovery* grows as **RTT of the path** grows. Higher RTT makes opportunities to adjust the congestion window (cwnd) less frequent, thus it takes more time to recover the larger cwnd. WaMPTCP+RO and MPTCP proxy+RO do not have to perform an entire cwnd recovery, since the subflow is not stalled (Section 3.3.4), and both subflows recover in parallel. Without Recovery Optimization, the subflow needs to recover the entire tunnel bandwidth from TCP slow start.

Table 3.3 shows that the **TCP congestion control** used for the subflows impacts the performance of WaMPTCP+RO. Six TCP congestion control algorithms are uncoupled, while the

last three - olia, balia, and wvegas - are coupled and specifically designed for MPTCP [38]. We artificially add 1ms or 100ms latency to both WAN links using Linux Traffic Control [72] to emulate a large Bandwidth $\times$ Delay Product (BDP) network. Different congestion control algorithms suit different network characteristics. For example, Reno underutilizes "long fat" paths, and after packet losses it grows cwnd by one every RTT, which explains its slower recovery. Similar to our earlier RTT tests, Reno with Tunnel Handoff is even slower (72.6s). We verified that for the coupled congestion controls, the fairness is proportional to the number of MPTCP sessions, and that for uncoupled congestion controls it is proportional to subflows. High RTT impacts the achievable throughput in all cases. The congestion windows of three coupled congestion controls grow very slowly, two of them even did not recover after 120 seconds. They also have lower throughput at 100ms RTT, which indicates the need of further improvements before they can be used for SD-WAN scenario.

Earlier tests explore only a single MPTCP session in idle links. Figure 3.18 has an increasing number of **TCP background flows** competing for the WAN links. Contention does not impact much *T-throughput-recovery*. *T-latency-failover* increases with contention, because the congestion window size shrinks and there are more retransmitted segments, which causes more delay for retransmitted packets.

3.5.5 Handoff Impact on Applications

A single handoff event won't impact much long lived applications, therefore we evaluate the impact on applications of **intermittent links**, which are frequently blocked or disconnected [73–75], for example a wireless link with slow fading [76]. In our tests, the backup link is

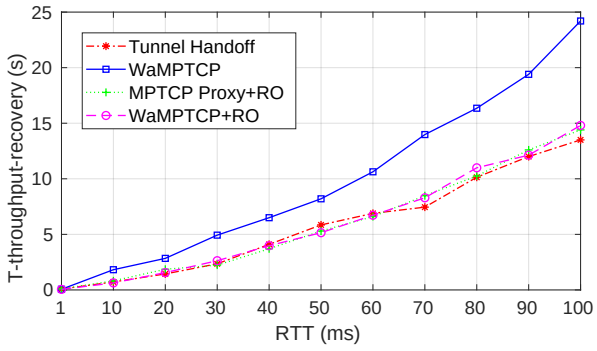


Figure 3.17: RTT Impact over T-throughput-recovery (BFD=100ms, Outage=5s)

seconds	T-latency-fail		T-tput-recover		Throughput	
RTT	*1ms*	*100ms*	*1ms*	*100ms*	*1ms*	*100ms*
cubic	0.43	0.71	1.26	12.3	179	126.0
reno	0.41	1.00	1.65	38.4	179	125.0
vegas	0.22	0.65	1.92	34.9	175	125.0
illinois	0.44	0.86	1.25	23.2	179	124.0
veno	0.33	0.74	1.31	26.8	179	125.0
westwood	0.27	0.55	1.43	14.9	179	126.0
olia	0.38	0.90	1.10	>120	179	83.9
balia	0.62	0.72	1.06	82.7	179	93.5
wvegas	0.64	0.59	1.46	>120	179	54.6

Table 3.3: Impact of TCP congestion control (BFD=1s)

BFD timer	*1s*					*100ms*				
Time between link events (on or off)	*1s*	*2s*	*3s*	*4s*	*none*	*1s*	*2s*	*3s*	*4s*	*none*
Tunnel Handoff	>3600	>3600	>3600	680	177	305	235	205	197	177
MPTCP Tunnel	stall	stall	stall	stall	94	stall	stall	stall	stall	94
MPTCP Proxy	176	175	175	173	91	176	175	175	173	91
WaMPTCP	176	175	175	173	91	176	175	175	173	91
WaMPTCP+RO	176	175	175	168	91	162	155	146	131	91

Table 3.4: HTTP download duration with intermittent link in controlled testbed (seconds for 2GB file)

always healthy, and the primary link alternates between on and off state at periodic interval (from 1s to 4s), it is blocked 50% of the time using netfilter (Section 3.5.2). Half of the handoff are reactive and the other half is proactive, which is more strenuous than frequent policy changes [64] (only proactive handoffs).

Table 3.4 explores the impact of intermittent links on **download** performance. The client downloads a 2GB file from a web server using wget, this represents applications which are

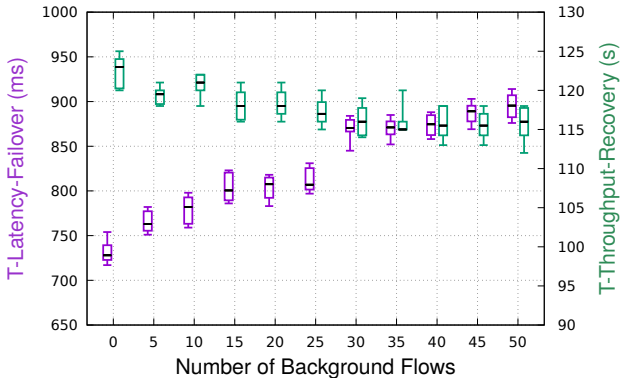


Figure 3.18: Impact of TCP contention

throughput sensitive (Section 3.4).

When using Tunnel Handoff with BFD timer of 1s, downloads are very slow for frequent outages. Our metrics explain this (Table 3.2), the connection is constantly switched back to the bad link, and TCP does not have enough time to fully recover before the next outage or when on the healthy link. TCP can recover only when the period of link events is greater than our metrics. A shorter BFD timer of 100ms produces shorter metrics, increasing performance. The metrics are independent of the outage duration (Section 3.5.3), and therefore the performance is mostly proportional to the outage frequency (for the same overall link duty cycle).

MPTCP mechanisms without Recovery Optimization use only a single link after the first outage, the download time is almost twice the time without outage. *T-throughput-recovery* increases with the outage duration (Section 3.5.3), so only an increase in the duty cycle of the bad link would allow the affected subflow to recover. Recovery Optimization with a short BFD timer enables the affected subflow to exploit the time between outages, with performance mostly proportional to the outage frequency. MPTCP Tunnel is completely stalled in presence of frequent outages and needs to be restarted.

BFD timer	*1s*					*100ms*			
Time between link events (on or off)	*1s*	*2s*	*3s*	*none*		*1s*	*2s*	*3s*	*none*
Tunnel Handoff	754.09	839.01	2021.88	3854.02		1989.44	3437.9	3737.66	3911.91
MPTCP Tunnel	stall	stall	stall	3923.49		stall	stall	stall	3943.24
MPTCP Proxy	3854.02	3883.26	3807.68	3946.42		3847.61	3842.71	3820.95	3876.36
WaMPTCP	3835.75	3822.78	3807.24	4061.85		3835.16	3864.27	3886.63	4048.54
WaMPTCP+RO	3879.77	3854.63	3873.92	4036.86		3828.03	3855.38	3889.34	4054.52

Table 3.5: TCP transaction rate with intermittent link in controlled testbed (transactions / seconds)

Table 3.5 explore the impact of intermittent links on **transaction** performance. The client uses the Netperf TCP_RR test, this measures the number of back to back bidirectional transactions on a TCP connection. This represents applications which are latency sensitive (Section 3.4), higher transaction rate can only be achieved with shorter round trip latency.

Tunnel Handoff with BFD timer of 1 seconds makes slow progress for frequent outages, and BFD timer of 100ms greatly improves performance. MPTCP tunnel is stalled.

Without outages, the performance of MPTCP mechanisms is only slightly higher than Tunnel Handoff, the back-to-back transactions are tiny, therefore they can not exploit the parallelism offered by MPTCP. In the presence of outages, MPTCP only uses a single path, and is mostly unaffected by the outage frequency. The MPTCP scheduler prefers the link with the lowest

Intermittent link	*low-latency link*					*high-latency link*				
Time between link events (on or off)	*1s*	*2s*	*3s*	*4s*	*none*	*1s*	*2s*	*3s*	*4s*	*none*
Tunnel Handoff	350	218	137	125	49	167	113	80	72	49
MPTCP Tunnel	stall	stall	stall	stall	32	stall	stall	stall	stall	32
MPTCP Proxy	90	77	68	61	32	62	58	50	49	32
WaMPTCP	92	78	67	63	32	62	57	51	47	32
WaMPTCP+RO	62	71	62	56	32	58	52	49	42	32

Table 3.6: HTTP download duration with intermittent link in GENI testbed, 100ms BFD (seconds for 20MB file)

RTT, therefore it permanently avoids the link with outage which has higher RTT. In contrast BFD can only do binary evaluation of the link. We only evaluated the default MPTCP scheduler, evaluating other advanced MPTCP schedulers [77, 78] could be future work.

3.5.6 Evaluation in the Wild

We deploy Durga in the real world to confirm the findings of our testbed. We perform experiments over the Internet in both the GENI testbed [79] and Amazon AWS.

In **GENI testbed**, we create a 5 node topology (Figure 3.19). We perform the previous experiments with intermittent links and 100ms BFD timer (Section 3.5.5). One particularity is that both path are heterogeneous and have different latencies, therefore we test with the primary and intermittent link being either the low-latency link or the high-latency link.

Table 3.6 mostly confirms our testbed results. Without outages, MPTCP does not have twice the performance of Tunnel Handoff, it can not exploit both links fully. A higher frequency of outages drastically reduces the performance of Tunnel Handoff, due to penalty of handoffs

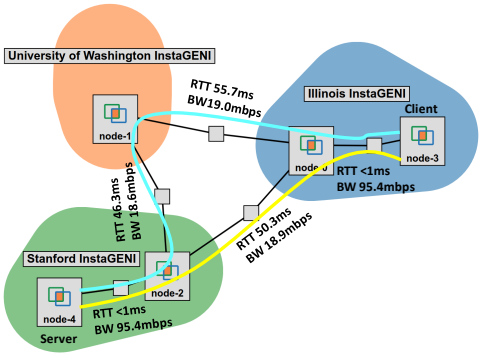


Figure 3.19: GENI Network Setup

Intermittent link	*low-latency link*					*high-latency link*				
Time between link events (on or off)	*1s*	*2s*	*3s*	*4s*	*none*	*1s*	*2s*	*3s*	*4s*	*none*
Tunnel Handoff	7.72	9.67	12.48	14.09	19.55	11.73	14.11	14.78	13.53	9.83
MPTCP Tunnel	stall	stall	stall	stall	19.45	stall	stall	stall	stall	19.36
MPTCP Proxy	10.98	11.22	12.45	13.98	19.56	19.34	19.31	19.33	19.33	19.34
WaMPTCP	11.19	11.65	12.33	14.19	19.32	19.34	19.34	19.31	19.33	19.34
WaMPTCP+RO	11.13	11.52	12.35	14.78	19.34	19.33	19.38	19.33	19.34	19.37

Table 3.7: TCP transaction rate with intermittent link in GENI testbed, 100ms BFD (transactions / seconds)

under high RTT (Section 3.5.4). Frequent outages make MPTCP a bit slower, and outages on the low-latency path are worse, confirming that MPTCP has trouble using the high-latency link effectively.

Table 3.7 mostly confirms our testbed results and highlight the effect of heterogeneous paths. TCP transaction rate mainly depends on latency, consequently without outage Tunnel Handoff perform as well as MPTCP, unless the primary path is high-latency. The default MPTCP scheduler prefers to send packets over low-latency path, therefore when the intermittent path is the high-latency path, MPTCP avoids it almost entirely and does not suffer from outages. When the intermittent path is the low-latency path, performance reduces closer to the performance of the high-latency link alone as outage frequency increases. Finally, when Tunnel Handoff uses the high latency link as the primary and intermittent link, outages increase performance compared to without outages, despite the cost of handoffs, because outages force the traffic onto the low-latency link which has much higher performance.

We instanciate a HTTP server (Apache 2.4.18) in **Amazon Elastic Compute Cloud** (EC2) in eu-west-1 zone. The local gateway uses three types of access networks: cellular network by using the hotspot feature on a mobile phone, university WiFi, and university Ethernet. Those WAN links offer different raw download performance (Table 8). A CNN's homepage consisting of 216 WEB objects is stored on the HTTP server, and the client downloads it.

	RTT	Throughput
Cellular	138∼162 ms	1.2 Mb/s
WiFi	181∼242 ms	1.64 Mb/s
Ethernet	100∼120 ms	6.4 Mb/s

Table 3.9: Parameters to access Amazon EC2

We simulate a single 1-second outage on the Ethernet primary path and investigate the

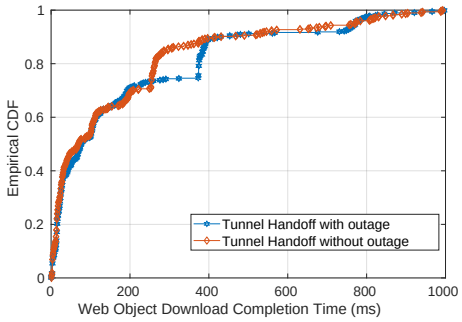


Figure 3.20: Web Browsing via Tunnel Handoff

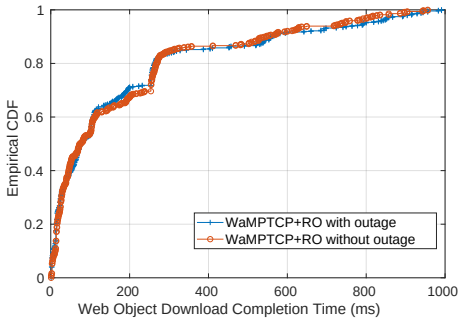


Figure 3.21: Web Browsing via WaMPTCP

cumulative distribution function (CDF) of web object loading time. Figure 3.20 shows that for Tunnel Handoff with 100ms BFD timer, the application can perceive the outage (separation of blue and red curves). Figure 3.21 shows that WaMPTCP+RO successfully hides the outage from the application.

3.6 Related Work

WAN virtualization solutions have been extensively studied in both enterprise and literature. As discussed, existing SD-WAN productions [7] use techniques similar to Tunnel Handoff for reliability, and certain solutions provide more features such as aggregated link utilization and fast failover. However, most implementation details are proprietary. In the literature, the benefits of using MPTCP for throughput and failover are studied [39–43] in different scenarios. Moreover, several MPTCP proxies [45, 46] are implemented for the purpose of link utilization. Our solution focuses on a SD-WAN scenario which has not been fully explored, and studies the metrics directly related to application performance.

On the other side, interests have shown in Layer 4 performance and designing fast failover mechanisms to settle link failure and achieve restoration. N. M. Sahri *et al.* [80] proposes a novel fast failover architecture by having a central controller to compute backup path so as to reduce switching delay. OSP scheme [81] maintains preplanned backup flows at different priorities and achieves fast resilience in optical transport networks. Ranadive *et al.* [65] explore the effect of route fluctuation on TCP. Kim *et al.* [61] propose an improved TCP scheme to deal with outage during handoff. Cârpa *et al.* [64] explore the effect of frequent SDN route changes on TCP. In contrast, our work defines metrics under link failures and takes MPTCP

into consideration.

3.7 Summary

We have presented Durga, a novel SD-WAN solution for fast failover with minimal application performance degradation under WAN link failures. We motivated the design for Durga by illustrating the problem associated with *vertical (tunnel) handoff* commonly employed by existing SD-WAN solutions to handle WAN link failures, and showed that it can lead to significant application performance degradation in the evaluation. Durga combines an innovative *WAN-aware* MPTCP mechanism which enables applications to generate multiple MPTCP flows even with a single physical interface. This is further augmented with an MPTCP proxy to accommodate end systems without native MPTCP support. We also introduced two new metrics to better capture and quantify the impact of WAN link failures on application performance. Through extensive evaluation in emulated testbed and real-world deployment, we demonstrated the superior performance of Durga over existing SD-WAN solutions. While the focus of this project is on providing fast failover under WAN link failures to minimize their impact on application performance, Durga can also be used to further enhance WAN link utilization by intelligently distributing MPTCP subflows to WAN links and support application-aware SD-WAN traffic engineering – an in-depth exploration of these topics will be left to a future project.

Chapter 4

Making Network Functions Aware of Multiple TCP

4.1 Main Results

As facilitating VNFs aware of MPTCP is beneficial to both the performance of MPTCP sessions and the quality of network services described in Section 2.2, we take a first step towards making the network devices MPTCP-aware by investigating how to associate subflows that belong to the same MPTCP session. This is relatively easy to do at a place where all flow records are available, e.g., at the end hosts. In this case one can use MPTCP token in TCP option field carried in the MP_JOIN message of each subflow to identify a MPTCP session. However, the problem of associating MPTCP subflows becomes more challenging in network. For example, it is common that network monitoring devices perform sampling on the data streams before processing them in order to reduce processing load. Moreover, flow paths can also change due to network dynamics and hence the monitoring device may only see a portion of the flow. All such complications may cause the MPTCP packets containing the token to be missing from flow records, and hence a more comprehensive and robust solution is needed for subflow association in network.

We propose SAMPO, an online subflow association mechanism for MPTCP with partial flow record. Our main contribution is a DSN-based algorithm that can associate subflows based on analysis of DSN values of each subflow, their range and overlapping pattern. Through extensive theoretical analysis and experimentation, we find that the DSN-based association is very

effective even when a very small fraction of packets from each subflow are available. For instance, the algorithm reaches close to 100% accuracy when only 1% of packets are sampled in.

The remainder of this project is organized as follows: the workflow of the system is illustrated in Section 4.3. MPTCP subflow association algorithm and its analysis are described in Section 4.4. After that, in Section 6.6, we show the evaluation of the main algorithm and then conclude.

4.2 MPTCP Subflow Association

In this section, we present background information related to the MPTCP subflow association problem. We first describe a token-based solution which is a regular way of associating MPTCP subflows and then explain why such a regular approach, although simple and accurate, may not always work in network.

4.2.1 Token-based Solution

The most straightforward way of solving the problem is to look for signatures in the MPTCP protocol that can be used to identify each session. Figure 4.1 shows the initial MPTCP protocol exchange during connection establishment. When sender A initiates a MPTCP connection with receiver B, it first sends a SYN packet, which contains both MP_CAPABLE flag and sender's key (KEY_A) in the TCP option field of its header. If MPTCP is supported and enabled at the receiver side, the receiver sends back a SYN/ACK that contains a MP_CAPABLE flag with receiver's key (KEY_B). The following ACK packet from the sender to receiver contains the keys of both sides. At this point, the first subflow in MPTCP has been established; this is called meta socket. Later on, when the sender needs to establish an additional subflow, it sends a SYN packet with MP_JOIN. This SYN packet contains a token, which is calculated from the receiver's key that the sender has obtained during meta socket handshake. In MPTCP Linux Kernel implementation, the token is calculated by taking the most significant 32 bits out of 160 bits SHA1 function of the receiver's key [82]. This SYN packet also contains a nonce field for further authentication. The next few handshake packets can be ignored since they are not relevant to MPTCP subflow association. Further details of MPTCP subflow handshake procedure can be found in the MPTCP RFC [8].

Since the token is generated from the receiver's key and the same function is called to compute the token for different subflows, all MPTCP subflows contain the same token. One exception is the meta socket, which only contains keys but not a token. In this case, we can use the token generation function to convert the receiver's key into a token so that we can associate the meta socket with other subflows. Note that in the current MPTCP Linux kernel implementation, only a sender is allowed to initiate subflows so only its receiver's key needs to be used. However, MPTCP specification allows both sides to initiate subflows. In the case where other MPTCP implementations choose to allow a receiver to initiate subflows, its sender's key can then be stored to derive the token for the reverse direction.

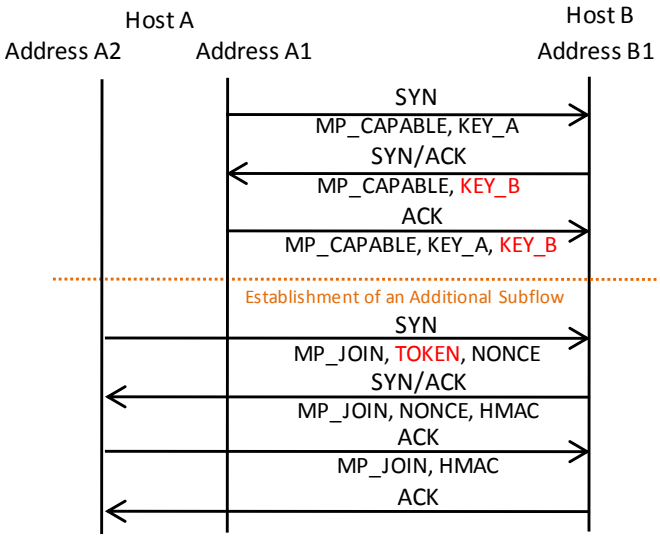


Figure 4.1: Basic Knowledge for Token-based MPTCP Subflow Association

4.2.2 Challenges in Network

Although the token-based approach can effectively associate MPTCP subflows, it relies on the assumption that the entire MPTCP handshake process can be captured, so that each subflow can be identified by using the token or the receiver's key. While this is true at the end hosts, e.g., at the hypervisor where host A or B runs, it is generally not true in network. We describe two common reasons below.

Sampling

Packet sampling is often used in network monitoring systems as a way to reduce processing and memory load [83]. As the link speed continues to increase on routers and switches, it is conceivable that to monitor every packet on each link will be even more challenging. A common sampling approach is to randomly sample a fraction of packets from a link. As a result, only partial flow records may be exposed to the monitoring system. The higher the sampling rate is set in network devices, the more bandwidth to transmit packets and resource to store and process them are required.

Network Dynamics

Network path can be changed due to various reasons. For example, link failure on the Internet is common due to its size and complexity, as shown in the measurement study [84]. Link failure may cause network instability and affect routing convergence on a large scale. For instance, the inter-domain routing protocol BGP could take up to 15 minutes to converge after link failures [85]. Such network dynamics may cause the packets in the same flow to be routed across different paths, and hence a monitoring device may not able to capture the entire flow.

Due to the above reasons, we believe that the token-based association approach will not be a reliable solution for a monitoring device in network. In the next section, we present a more sophisticated algorithm based on statistical characteristics of DSNs in each subflow, which can support online MPTCP subflow association with only partial flow record.

4.3 SAMPO Overview

In this section, we propose SAMPO, an online subflow association system for MPTCP with partial flow records.

Figure 4.2 illustrates the workflow of SAMPO. The input to SAMPO is a set of partial flow records (e.g., sampled packets). The output is the association result of MPTCP subflows, identifying sets of subflows belonging to the same MPTCP session. The pre-processing step is done as follows. First, the flow records are grouped according to 5-tuple information in the packet header. The MPTCP flows are then selected based on the TCP option field. For flows containing MPTCP header information, DSNs and corresponding packet length are extracted

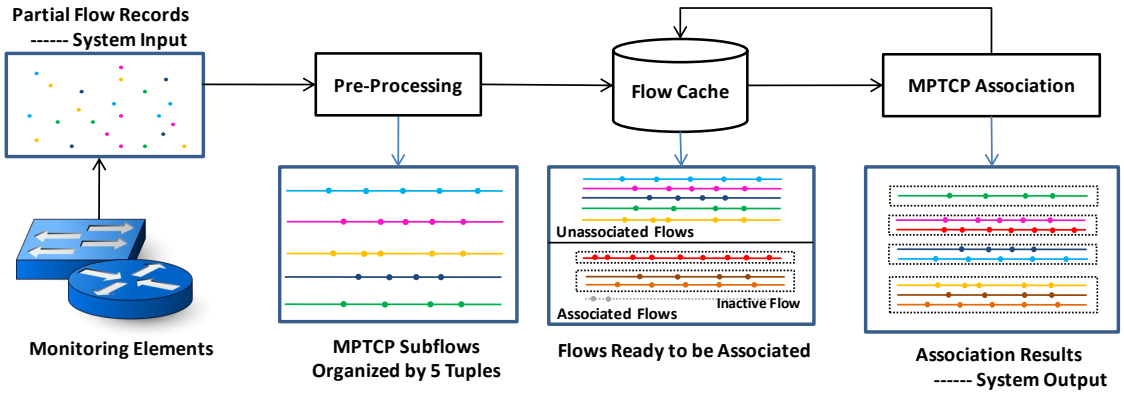


Figure 4.2: SAMPO Workflow

from the option field, along with the token or the receiver's key, if available. Such information is stored in a flow cache as part of the flow record to be fed to the next MPTCP association step. To enable online processing, we use a sliding window mechanism in the flow cache. MPTCP association is triggered every γ second. Flows cached within each time window of γ seconds are processed together. The inactive flows that have arrived before the current time window are timed out and removed.

The MPTCP association step consists of two parts: token-based association and DSN-based association. If a receiver's key or token is available, token-based association is then performed using the method described in Section 4.2. The results of token-based association along with flow records are fed into DSN-based association for further association; details of this step are illustrated in Section 4.4.1. At the end of processing, the system produces a report of flows belonging to the same MPTCP session. For every pair of flows in the report, there is a confidence value Φ to represent how trustworthy the result is. For association results with Φ higher than before, they will be stored back into flow cache for the association of future subflows.

4.4 MPTCP Subflow Association

In this section, we describe the main algorithm in our system and present our analysis. For completeness, the overall SAMPO system still takes advantage of token-based subflow association method (Φ is set to 1 if associated by token) presented in Section 4.2.1, although our main

algorithm presented here only relies on DSN-based association.

4.4.1 DSN-based Association

Recall that DSN is used as a global sequence number for the entire MPTCP data. Since multiple subflows are used for data transmission, DSN is spread across different subflows. We define the *active range* of a subflow as the DSN range from the beginning to the end of this subflow. Similarly, we define the *DSN segment* for a packet as the DSN range between the beginning and end of this packet. Note that the length of each DSN segment is the length of this packet. Hence our main intuition is that if two subflows belong to the same MPTCP session, their active ranges have a high probability of overlapping. Furthermore, within their overlapped active range, the DSN segments of the two subflows should be interleaving, instead of overlapping. This is because upper-layer data should only be assigned to one subflow, except for reinjection packets, which we will address later. DSN-based association actually uses 2-level overlap analysis to determine whether two subflows are generated by the same MPTCP session or not.

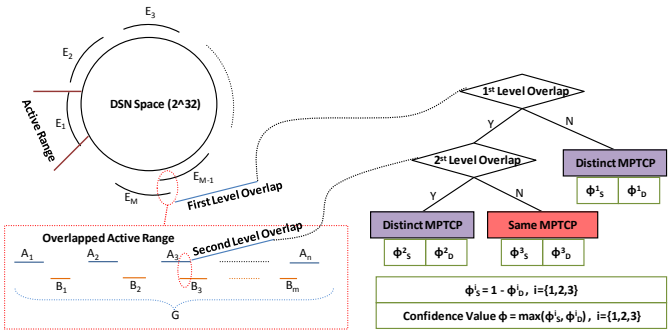


Figure 4.3: DSN-based Association

As shown in Figure 4.3, the algorithm first examines whether the active ranges of two subflows overlap or not. This is called the first-level overlap. If the first-level overlap holds, the algorithm further checks if the DSN segments in the overlapped active range has the second-level overlap. The second-level overlap is defined as overlapped DSN segments in the overlapped active range. The colored boxes on the right side of the figure show steps of the classification logic. Each step of the decision is associated with a confidence value Φ^i_X, where X can be either S (i.e., same MPTCP session) or D (i.e., distinct MPTCP sessions). Label $i = 1, 2, 3$ corresponds to the outcome of each decision step, as shown in the colored box. The confidence

values range between [0,1]. Φ_S^i is defined to be $1 - \Phi_D^i$. For example, given that first-level overlap does not hold, Φ_D^1 is confidence value of the classification result that two subflows belong to distinct MPTCP sessions. Similarly, Φ_S^1 is confidence value of the classification result that two subflows belongs to the same MPTCP session. Since the confidence value, Φ, of an association result for a pair of subflows is defined as $max(\Phi_S^i, \Phi_D^i)$, the essential task of DSN-based association is to calculate $max(\Phi_S^i, \Phi_D^i)$. If Φ_S^i is larger than Φ_D^i, the association result is S for the current round; otherwise, D is. The detailed analysis on the value of Φ_X^i is presented in Section 4.4.2.

In DSN-based association, if a subflow is wrap-around [86] in terms of DSN in a given window, when this subflow is associated with other subflows, a virtual DSN space is used to examine whether two subflows overlap at the first level. In virtual DSN space, wrap-around part [0, n] is mapped to [S, $S + n$] in which S is the total DSN space size.

4.4.2 Analysis of DSN-based Association

The confidence value Φ_X^i generated by DSN-based association can be analyzed by using probability theory. Since DSN-based association is based on a two-level overlap algorithm, we may have the following questions: for subflows belonging to the same MPTCP session, what is the probability that they overlap at the first level? How about subflows belonging to distinct MPTCP sessions? If two subflows overlap at the first level and they belong to the same MPTCP session, what is the probability that they overlap at the second level? How about subflows belonging to distinct MPTCP sessions in this case? Understanding these questions helps us better define Φ_X^i to tune the algorithm. We first present the analysis for the first-level overlap, and then present the analysis for the second-level overlap.

First-Level Overlap

The first-level overlap of two subflows belonging to distinct MPTCP sessions needs to be investigated, i.e., the probability that active ranges of these two subflows overlap. The question is formalized as the following: Given two subflows (length of active range is N_1 and N_2 respectively) passing through a switch, the initialized data sequence number (IDSN) follows a discrete uniform distribution with range [0, s], what is the probability that these two subflows overlap in terms of DSN? ($s = 2^{32} - 1$ gives the problem of calculating probability of first-level overlap).

The probability is easily calculated as $P(E_1E_2) = P(E_1) \times P(E_2|E_1) = \frac{N_1+N_2-1}{s+1}$ in which E_i is the active range of each subflow.

What if these two subflows belongs to the same MPTCP session? In order to understand this problem, we need to understand how MPTCP scheduling algorithm works, i.e., how data segments are scheduled over multiple links. In MPTCP Linux kernel implementation [82], the default scheduler chooses subflow with the lowest RTT until its congestion window is full. Then, the scheduler assigns data segments to the subflow with the next lowest RTT. This scheduler is argued to be the best known till date [87]. Based on the such a scheduling mechanism, it can be inferred that active ranges of two subflows belonging to the same MPTCP session have very high probability of overlapping with each other.

Given that two subflows are not overlapped at the first level, can we guarantee that they must belong to distinct MPTCP? The brief answer is no. There are three reasons. The first reason is that it is possible that active range of one subflow might be too small to overlap with that of another subflow in the same MPTCP session. The second reason is that MPTCP data segments may not be transmitted over multiple paths concurrently. MPTCP end host can specify a path (in fact, end host specify the interface connected to this path) as a backup path which will only be used if no default path is available. A well-known application, Siri, in iOS 7 takes cellular connection as a backup connection [88]. When WiFi goes down, DSN over cellular connection would not be overlapped with that over WiFi. The third reason is that the input to our analysis is partial subflow instead of complete subflow information, so it is possible that overlapped active range is missed due to network dynamics or packet sampling. Despite of various reasons, if subflows belonging to the same MPTCP session are not overlapped at the first level, the gap between two active ranges from two subflows has high probability to be smaller than the largest gap between two consecutive DSN segments from a subflow. Thus, Φ_D^1 is defined as $Sigmoid(\frac{min(gap\_subflow)}{max(gap\_DSN)} - 1)$, in which $min(gap\_subflow)$ is the smaller gap between two active ranges and $max(gap\_DSN)$ is the largest gap between two consecutive DSN segments in the same subflow. Here, we include the sigmoid function as a "cut-off". When $min(gap\_subflow)$ is small, i.e., $min(gap\_subflow) < max(gap\_DSN)$, $Sigmoid(\frac{min(gap\_subflow)}{max(gap\_DSN)} - 1)$ will rapidly get closer to 0, leading to a small probability that two subflows are belonging to distinct MPTCP.

Another question is: given that two subflows are overlapped at the first level, can we guarantee that they must belong to the same MPTCP? The brief answer is also no. Consider that

there are only two subflows passing through a monitoring element, the probability that two active ranges are overlapped is $P(E_1E_2) = P(E_1) \times P(E_2|E_1) = \frac{N_1+N_2-1}{s+1}$ as analyzed before. However, what if there are M subflows belonging to distinct MPTCP sessions passing through the monitoring element?

We generalize above problem from two subflows to M subflows and calculate the probability that at least two subflows have first-level overlap. The question is formalized as given M subflows with length $N_i(i = 1, ..., M)$ passing through a switch, the initialized data sequence number follows a discrete uniform distribution with range [0, s], what is the probability that at least two subflows are overlapped?

We first calculate the probability $\overline{P}(E_1, \ldots, E_M)$ that no subflows are overlapped. Then $1 - \overline{P}(E_1, \ldots, E_M)$ is the probability that at least two subflows are overlapped.

$$\overline{P}(E_1, \ldots, E_M) = \overline{P}(E_M|E_1, \ldots, E_{M-1}) \times \overline{P}(E_{M-1}|E_1, \ldots, E_{M-2}) \cdots \overline{P}(E_2|E_1) \times \overline{P}(E_1)$$

Given $(M - 1)$ subflows with length $N_i(i = 1, ..., M - 1)$ passing through a switch, the IDSN follows a discrete uniform distribution with range [0, s] without any overlap. Given an additional subflow with length N_M passing through this switch, the probability that this subflow will not be overlapped with other $M - 1$ subflows is

$$\overline{P}(E_M|E_1, \ldots, E_{M-1}) = \begin{cases} \max\left(\frac{\sum gap_{M-1} - g_{M-1} \times (N_M - 1)}{S}, 0\right), M > 1 \\ 1 \qquad\qquad\qquad\qquad\qquad , M = 1 \end{cases}$$

with lower bound:

$$\overline{P}_{\min}(E_M|E_1, \ldots, E_{M-1}) = \max\left(\frac{S - \sum_{k=1}^{M-1} N_k - (M-1) \times (N_M - 1)}{S}, 0\right), M > 1$$

and upper bound:

$$\overline{P}_{\max}(E_M|E_1,\ldots,E_{M-1}) = \max\left(\frac{S-\sum_{k=1}^{M-1}N_k-N_M+1}{S},0\right), M>1$$

in which space size S is 2^{32}. $\sum gap_{M-1}$ is the total size of gaps, each of which is equal to or larger than N_M, while g_{M-1} is the number of such gap in total when M-1 subflows have already been placed.

Explanation for lower bound: there are S different choices for each IDSN without considering overlap. First count how many of them are not overlapped. The first IDSN can be chosen freely, in S different ways. Adding E_1 makes N_1+N_M-1 out of S points forbidden as IDSN for other subflows (each of the N_1 points contained in that subflow and N_M-1 points before it). The total legal IDSN for the second subflow is $S-N_1-(N_M-1)$. Choosing the second IDSN will make at most N_2+N_M-1 out of the remaining points invalid. For the IDSN of the third subflow, there are at most $S-N_1-N_2-2(N_M-1)$ valid points. Overall, in the worst case, $S-\sum_{k=1}^{M-1}N_k-(M-1)\times(N_M-1)$ points are forbidden when E_M needs to be placed.

Explanation for upper bound: in the best case, all active ranges are connected end to end in which the most valid places will be available for the next active range. The first IDSN can still be chosen freely, in S different ways. Adding E_1 makes N_1+N_M-1 out of S points forbidden as IDSN for other subflows. Then after adding E_2 which is end to end with E_1, there are only $N_1+N_2+N_M-1$ out of S points forbidden instead of $N_1+N_2+2(N_M-1)$ in the worst case. Therefore, when E_M needs to be placed, there are only $\sum_{k=1}^{M-1}N_k+N_M-1$ places invalid

in the best case.[1]

$$\overline{P}(E_1, \ldots, E_M) = \begin{cases} \max\left(\prod_{k=2}^{M}\left(\frac{\sum gap_{k-1} - g_{k-1} \times (N_k - 1)}{S}\right), 0\right), M > 1 \\ 1 \qquad\qquad\qquad\qquad\qquad\qquad , M = 1 \end{cases}$$

with lower bound:

$$\overline{P}_{\min} = \max\left(\prod_{k=2}^{M}\left(\frac{S - \sum_{t=1}^{k-1} N_t - (k-1) \times (N_k - 1)}{S}\right), 0\right)$$

and upper bound:

$$\overline{P}_{\max} = \max\left(\prod_{k=2}^{M}\left(\frac{S - \sum_{t=1}^{k-1} N_t - N_M + 1}{S}\right), 0\right)$$

We refer Taylor series to calculate the approximations of aforementioned formula. When $S \gg \sum_{t=1}^{M} N_t$, the lower bound can be approximated as:

$$\overline{P}_{\min} \approx \max(e^{-\frac{(M-1) \times (\sum_{t=1}^{M} N_t - \frac{M}{2})}{S}}, 0), M > 1$$

and the upper bound can be approximated as:

$$\overline{P}_{\max} \approx \max(e^{-\frac{\sum_{k=2}^{M} \sum_{t=1}^{k} N_t - (M-1)}{S}}, 0), M > 1$$

The derivation of lower bound and upper bound is shown as follows.

[1] In fact, Birthday Paradox [89] is an extreme case of our analysis in which $N_i = 1 (i = 1, ..., M)$ and S = 365. In this extreme case, the lower bound and upper bound are equal because g_{M-1} is always equal to the number of gaps available.

Key steps of lower bound derivation are:

$$e^{-\frac{\sum_{t=1}^{k-1} N_t+(k-1)\times(N_k-1)}{S}} \approx 1-\frac{\sum_{t=1}^{k-1} N_t+(k-1)\times(N_k-1)}{S}$$

$$\prod_{k=2}^{M}(1-\frac{\sum_{t=1}^{k-1} N_t+(k-1)\times(N_k-1)}{S})$$
$$\approx e^{-\frac{\sum_{t=1}^{1} N_t+(N_2-1)}{S}} \cdots \times e^{-\frac{\sum_{t=1}^{M-1} N_t+(M-1)\times(N_M-1)}{S}}$$
$$= e^{-\frac{(M-1)\times(\sum_{t=1}^{M} N_t-\frac{M}{2})}{S}}$$

Key steps of upper bound derivation are:

$$e^{-\frac{\sum_{t=1}^{M-1} N_t+N_M-1}{S}} \approx 1-\frac{\sum_{t=1}^{M-1} N_t+N_M-1}{S}$$

$$\prod_{k=2}^{M}(\frac{S-\sum_{t=1}^{M-1} N_t-N_M+1}{S})$$
$$\approx e^{-\frac{\sum_{t=1}^{2} N_t-1}{S}} \times e^{-\frac{\sum_{t=1}^{3} N_t-1}{S}} \cdots e^{-\frac{\sum_{t=1}^{M} N_t-1}{S}} = e^{-\frac{\sum_{k=2}^{M}\sum_{t=1}^{k} N_t-(M-1)}{S}}$$

Figure 4.4 shows the rate that two active ranges are overlapped given different subflow size. we can see that given hundreds of subflows, the overlap rate is relatively high. For example, as a switch in the middle of the network, it captures numerous subflows simultaneously. Even though two subflows are not generated by the same MPTCP session, the chance that their active ranges are overlapped could not be ignored. Thus, the algorithm still cannot make a rash classification that two subflows are belonging to the same MPTCP session if their active ranges are overlapped. The algorithm needs to further analyze the DSN segments in the overlapped active range.

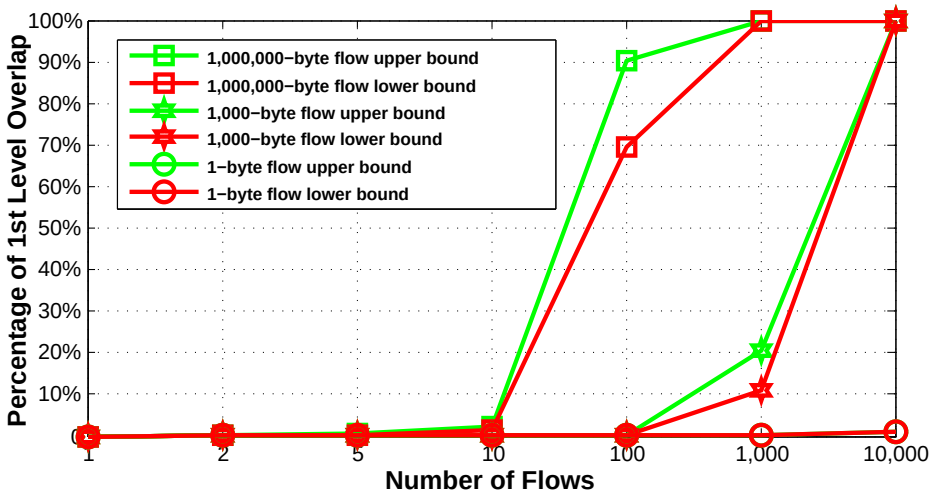


Figure 4.4: First-Level Overlap Rate

Second-Level Overlap

In the second-level overlap, we analyze the DSN segments from two subflows in the overlapped active range and see how this can help the algorithm make better classification. The probability of the second-level overlap is calculated given two subflows belonging to the same MPTCP session or distinct MPTCP sessions.

Given two subflows belonging to the same MPTCP session, what is the probability that they are overlapped at the second level? As we known, for two subflows belonging to the same MPTCP session, their DSN segments are supposed to be interleaved without overlap. It is because MPTCP scheduler assigns packets into different subflows according to scheduler algorithm [87]. It is just like a TCP session. If two packets from a TCP session are not retransmission packets, they would not overlap with each other in terms of sequence number segment which is a segment starting at sequence number and ending at sequence number plus packet length. The story is similar for MPTCP session. The only reason why two subflows belonging to the same MPTCP overlap at the second level is because of reinjection packets which is retransmission over another subflow instead of the original subflow. If the packets are reinjected from a subflow to another, DSN segments of these two subflows would overlap. For example, packets sent over WiFi may be reinjected to cellular network when WiFi signal degrades or WiFi connection is terribly congested. To address this issue, we need to first understand in which condition, packets

would be reinjected to another subflow. After analyzing MPTCP implementation [82], we know that reinjection packets appear if and only if either of the following two conditions hold: 1) link failure; 2) retransmission timer expiration. In order to detect reinjection packets, DSN-based association algorithm includes a heuristic that reinjection packets start with the same DSN and are of the same size. Thus, Φ_M^2 is defined as 0 because the algorithm filters out reinjection packets and then examines whether two subflows overlap at the second level or not.

Given two subflows belonging to distinct MPTCP sessions, what is the probability that they overlap at the second level? As shown in the bottom of Figure 4.4, given two subflows with the length of overlapping active range G, assuming the number of DSN segments within the overlapping segment is n and m ($n \geq m$) respectively, and the size of each DSN segment is $p_{A_i}(i = 1, \ldots, n)$ and $p_{B_i}(i = 1, \ldots, m)$, what is the probability that DSN segment $A_i(i = 1, \ldots, n)$ overlaps with DSN segment $B_i(i = 1, \ldots, m)$? A_i is the ith DSN from subflow A in the overlapping active range, while B_i is the corresponding one from subflow B.

This probability problem can be converted into another form. Given $A_i(i = 1, \ldots, n)$ in space G without overlap, what is the probability of overlapping after all $B_i(i = 1, \ldots, m)$ are put into G, i.e., $P(B_1, \ldots, B_m | A_1, \ldots, A_n)$. Denote $D_{B_{k-1}}$ as the distance between the first DSN of the overlapped active range and the end of DSN segment B_{k-1}. The probability can be calculated as

$$
\begin{aligned}
&P(B_1, \ldots, B_m | A_1, \ldots, A_n) \\
&\quad = 1 - \overline{P}(B_1 | A_1, \ldots, A_n) \times \overline{P}(B_2 | B_1 A_1, \ldots, A_n) \\
&\qquad \cdots \overline{P}(B_{m-1} | B_1, \ldots, B_{m-2} A_1, \ldots, A_n) \\
&\qquad \times \overline{P}(B_m | B_1, \ldots, B_{m-1} A_1, \ldots, A_n)
\end{aligned}
$$

in which

$$
\begin{aligned}
&\overline{P}(B_k | B_1, \ldots, B_{k-1} A_1, \ldots, A_n) \\
&\quad = \frac{\sum gap_{n+k-1} - g_{n+k-1} \times (p_{B_k} - 1)}{G - D_{B_{k-1}}}
\end{aligned}
$$

with lower bound:

$$\overline{P}_{\min}(B_k|B_1,\ldots,B_{k-1}A_1,\ldots,A_n) = \frac{G - D_{B_{k-1}} - \sum_{t=1}^{n} p_{A_t} - n \times (p_{B_k} - 1)}{G - D_{B_{k-1}}}$$

and upper bound:

$$\overline{P}_{\max}(B_k|B_1,\ldots,B_{k-1}A_1,\ldots,A_n) = \frac{G - D_{B_{k-1}} - \sum_{t=1}^{n} p_{A_t} - p_{B_k} + 1}{G - D_{B_{k-1}}}$$

Therefore, the probability that $B_i(i = 1,\ldots,m)$ overlaps with $A_i(i = 1,\ldots,n)$ is

$$P(B_1,\ldots,B_m|A_1,\ldots,A_n) = 1 - \prod_{k=1}^{m} \frac{\sum gap_{n+k-1} - g_{n+k-1} \times (p_{B_k} - 1)}{G - D_{B_{k-1}}}$$

with lower bound:

$$P_{\min}(B_1,\ldots,B_m|A_1,\ldots,A_n) \approx 1 - e^{-\frac{m \times \sum_{t=1}^{n} p_{A_t} + nm \times \sum_{t=1}^{m} p_{B_t} - mn}{G}}$$

and upper bound:

$$P_{\max}(B_1,\ldots,B_m|A_1,\ldots,A_n) \approx 1 - e^{-\frac{-m + m \times \sum_{t=1}^{n} p_{A_t} + \sum_{t=1}^{m} p_{B_t}}{G}}$$

Based on the probability analysis above, Φ_M^3 is defined as the lower bound of $P(B1, ..., Bm|A1, ..., An)$. From a high level understanding, it means that the probability of overlapping at the second level is assigned as the confidence value of judging two subflows belonging to the same MPTCP

session. Given two subflows without overlap at the second level, the higher probability of overlapping is calculated, the more confidence that these two subflows are belonging to the same MPTCP. For example, with more space in G occupied by increasing DSN segments from two subflows, the lower bound of $P(B1, ..., Bm|A1, ..., An)$ grows to 99%, but these two subflows still do not overlap at the second level, and then we can probably have 99% confidence to classify them as belonging to the same MPTCP session. The reason we choose lower bound is because we want to minimize false positive cases in which subflows belonging to distinct MPTCP session is classified as the same MPTCP session. The cost of false positive cases is much larger than that of false negative cases in which subflows belonging to the same MPTCP session is mistakenly classified as distinct MPTCP session. It is because after subflows are classified as the same MPTCP session, there may have some additional operations needed to be performed. For example, the application identification function may assemble subflows belonging to the same MPTCP to perform analysis.

Since partial subflow records could be caused by sampling, it is possible that overlapping DSN segments are not sampled in. Thus, DSN-based association also takes sampling rate ϱ into consideration at the second-level overlap. The solution is quite straight-forward. Every time the probability of second-level overlap, $P(B1, ..., Bm|A1, ..., An)$, is calculated, the algorithm set $G = G/\varrho$.

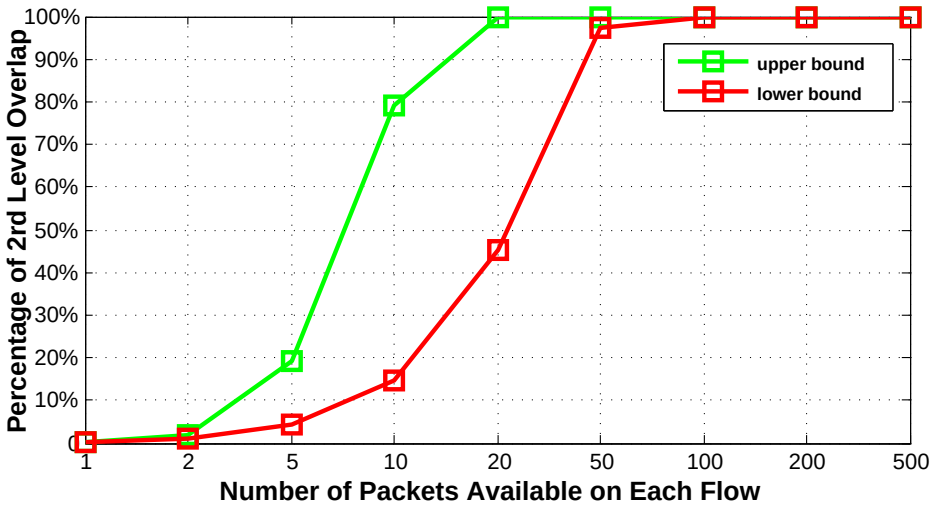


Figure 4.5: Second-Level Overlap Rate

Figure 4.5 shows the second-level overlap rate, we take G as 1,000,000 and 1428 bytes (maximum Ethernet segment size) as size of the packet. We can see that the more DSN segments are in the overlapping segment, the more chance they overlap. Given 50 DSN segments from each subflow in the overlapping active range, it is highly likely that at least two DSN segments overlap in terms of data sequence segment. In this case, if two subflows do not overlap at all, $\Phi = \Phi_S^3$ which is a very large value. It means that DSN-based algorithms classifies these two subflows as the same MPTCP session with very high confidence value.

4.5 Evaluation

SAMPO have been implemented to associate subflows belonging to the same MPTCP session. In this section, we use Mininet experiments to demonstrate the feasibility and effectiveness of SAMPO running in network.

4.5.1 Experimental Setup

Mininet, a Linux container-based emulation tool, is used to conduct experiments. It provides a platform to create a virtual network and generates end hosts and network devices. The benefits of using Mininet are that the topology of the network is flexible and real MPTCP implementation can be used instead of protocol simulation so that experimental results are more practical. Mininet is installed in Ubuntu 64bit LTS directly instead of running it in a virtual machine. The machine running experiments is equipped with Intel Core i7-4770 CPU @ 3.40GHz $\times$ 8, with 32G memory and 512G solid state drives. End hosts in Mininet are installed with Linux kernel implementation of MPTCP v0.89. The reason v0.89 is used because new features in this version provide more flexibility for the experiments. In this version, MPTCP can be switched on/off at application level when a socket is initialized, and thus regular TCP can be generated as the background traffic.

The setup of our experiment is shown in Figure 6.9. It creates a virtual network including a server and 100 clients in which 40 configured to run MPTCP and the rest 60 to run standard TCP. Clients and server are connected with pre-defined bandwidth as shown in the figure. One switch is configured as a gateway to forward packets across different subnets. An Open vSwitch [90] is run on the gateway, and configured to support packet sampling according to experimental requirements. Sampled packets are fed into a virtual host running SAMPO in real-time. Each

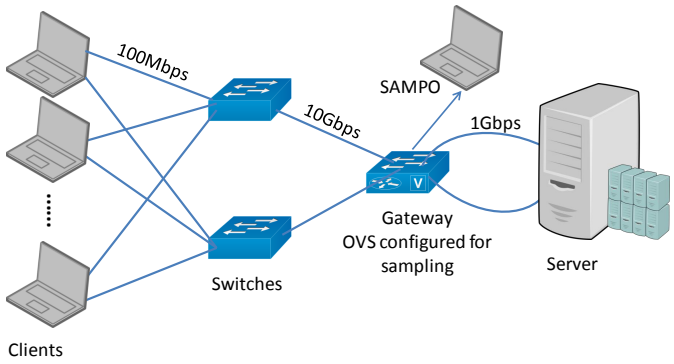


Figure 4.6: Experimental Setup

client or the server has two interfaces, and thus 4 subflows are generated for one MPTCP session between a client and the server. The end hosts running standard TCP randomly choose an interface when experiments start. For each experiment, all clients start a task of downloading a 500 MB file from the server.

In the experiments, we try to classify each pair of MPTCP subflows (i.e., distinct or same). As a ground truth, the total number of MPTCP subflows is 160, which gives rise to $\binom{2}{160}$ pairs of subflows. In these pairs of subflows, 240 pairs belong to the same MPTCP session, and the rest 12480 pairs belong to distinct MPTCP sessions. For each pair of subflow, if it is classified as the same MPTCP session and it actually is, we attribute it as true positive (TP), otherwise it is false positive (FP); if it is classified as distinct MPTCP and it actually is, we count the case as true negative (TN), otherwise it is false negative (FN). The following performance metrics are used in the evaluation:

- Accuracy is defined as $(TP+TN)/(TP+TN+FP+FN)$. It represents the fraction of all flows correctly classified.
- Precision is defined as $TP/(TP+FP)$. It represents how trustworthy the classification result is.
- Recall is defined as $TP/(TP+FN)$. It represents how complete the classification result is.

4.5.2 Experimental Results

In this section, we present experimental results using DSN-based association. Token based association is disabled in the experiments, because association based on token is deterministic and does not produce any error.

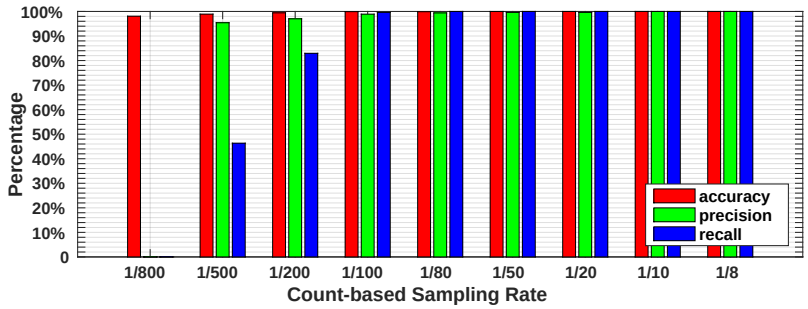


Figure 4.7: Count Based Sampling

We first evaluate DSN-based association by using two different sampling algorithms [83]: count-based sampling (selection triggered at every n packet) and timing-based sampling (selection triggered at every t microsecond)[2]. The sliding window size γ is set to 500 microseconds. Figure 4.7 and 4.8 show the results obtained from count-based sampling and time-based sampling, respectively. Accuracy, precision and recall increase with the sampling rate in the figures and all three metrics approach 100% if one packet is sampled at every 100 packets, or every 50 microseconds. This implies that given the fixed size of γ, as more packets are fed into the algorithm, it can achieve better association results in general. The reason why accuracy is always high is because there are many more negative cases (flows belonging to distinct MPTCP sessions) than positive cases (flows belonging to the same MPTCP session), and the correctness of identifying negative cases is high. Thus, these negative cases (it is a practical setting that most subflows belong to distinct MPTCP sessions) overwhelms the result of subflows belonging to the same MPTCP session. This is the reason why we use precision and recall in addition to accuracy. We can also observe that precision is high. It is because DSN-based algorithm generates very few FP cases even though sampling rate is low. The reason why precision and recall are 0 given that sampling rate is 1/800 is because the number of packets in each flow sampled in to DSN-based algorithm is very few (only 0 or 1). In such an extreme case, the algorithm could not generate even one TP case, and thus the the precision and recall are both 0 according to its

[2]Since Open vSwitch does not support timing-based sampling, its result is generated from simulation.

definition.

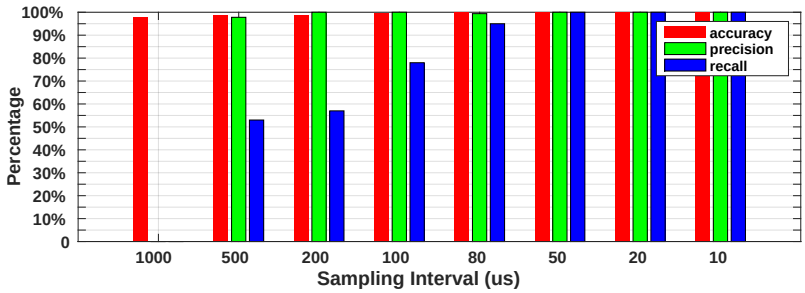


Figure 4.8: Timing Based Sampling

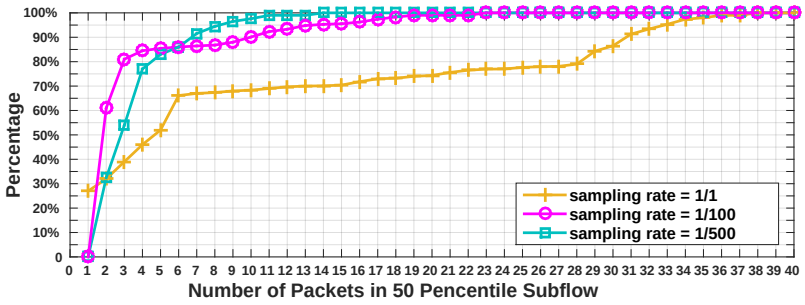


Figure 4.9: Relation Between Packets Number And Recall Rate

Throughout our experiments, we observe that all the performance metrics increase as either the sampling rate or the window size increases. This is quite intuitive as increase in any of the parameters contributes more packets to the association algorithm and that helps in the analysis. Therefore, we set out to investigate the relationship between the number of examined packets in each subflow and the accuracy obtained by the DSN-based algorithm. Since it is difficult to sample exactly the same number of packets from each subflow (due to the disparity in their goodput), we take the 50^{th} percentile number amongst all the subflows as the representative packet count fed into the algorithm. Figure 4.9 plots recall against the number of packets in the 50^{th} percentile over all the subflows. Note that accuracy and precision are both very close to 100% and are omitted to maintain clarity in the plot. Observe that with the increase in the number of packets sampled in, the recall with lower sampling rate reaches 100% faster than that with higher rate. This is due to the manifestation of the first-level overlap. When a fixed number of packets are sampled in, lower sampling rate covers a larger time window into the flow compared to the higher rate. Therefore, with lower sampling rate, the first-level overlap

decision can be made sooner.

The DSN-based algorithm keeps on refining the association decision with newly sampled packets. That is, for the same pair of subflows, the current association decision with lower Φ is always replaced by a higher one. Thus, we investigate how long it takes to get the correct result that is never replaced by the wrong result later. Figure 4.10 shows the resultant cumulative distribution function of the time spent from the start of each subflow. Since the number of packets sampled within a window with a higher sampling rate is larger than that with a lower rate, the time spent to achieve the correct result with a higher sampling rate is much smaller than that with the lower rate.

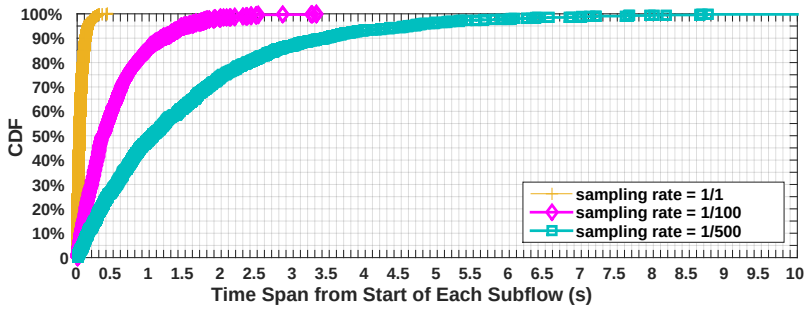


Figure 4.10: Time Spent To Reach Correct Result

Since DSN-based association is an online algorithm, its space and time performance is important too. We have performed the experiments to run SAMPO in real time fed with traffic sampled using Open vSwitch. As DSN-based association only processes data in a window of size γ, it greatly reduces the number of packets to be processed in the association stage. Since around 40 packets in each subflow can produce fairly good result from Figure 4.10 (with higher sampling rate, this number will be smaller), given 160 MPTCP subflows in total (around 6400 packets in a sliding window γ), association of these subflows from scratch takes 237 millisecond on an average. For each subflow, DSN-based association only keeps DSN segment of each packet in the current γ time period. It also maintains a triangular matrix where the relation of each pair of subflows with confidence value Φ is stored. The relation here means whether these two subflows belong to the same MPTCP session. To give an idea of the space requirements of DSN-based association, within a sliding window γ, 20 MB traffic generated by 160 MPTCP subflows costs no more than 1 MB memory. If the algorithm needs to process network traffic collected from multiple sources (e.g. multiple switches), DSN space (2^{32}) can

be split into multiple subspaces and each can be handled by a dedicated distributed node. In this case, MPTCP subflow association can be performed in a parallel.

Last but not least, the impact of 64-bit MPTCP DSN space is discussed. MPTCP specification states that the length of MPTCP DSN space can be set as 64-bit. The use of 64-bit DSN actually benefits the accuracy of DSN-based association. It is because two subflows belonging to distinct MPTCP have even less probability of overlapping in term of active range (first level overlap). For subflows belonging to the same MPTCP connection, since they are more likely to overlap (have been analyzed above), they will not be influenced by DSN space size. Moreover, 64-bit DSN decreases the probability of DSN wrap-around.

From the experimental results, we can see that DSN-based association is able to associate MPTCP subflows with high accuracy even when a very small portion (e.g. high sampling rate) of the entire flow records are available. Note that SAMPO also uses token information to associate MPTCP subflows. Therefore, it is more accurate and effective in online MPTCP subflow association with partial flow records.

4.6 Related Work

MPTCP has generated lots of interest from many researchers in the past few years [10, 91–94]. Olivier has recently written an annotated bibliography on MPTCP [95].

Our work is motivated by the mptcptrace tool developed by Hesmans *et al.*, which is designed to analyze MPTCP flows [96]. The main difference is that mptcptrace works as an offline tool and requires full flow records; it uses token-based approach to associate MPTCP flows. On the other hand, SAMPO is designed to be an online tool that can work with either full or partial flow records.

Sandri *et al.* [12] have designed a method to improve MPTCP performance by distributing subflows of the same MPTCP connection across different paths. An OpenFlow controller is used to associate MPTCP subflows and hence it relies on reactive flow processing, which may bring scalability concerns. In addition, their subflow association algorithm is based on token only, ignoring MPTCP meta socket. SAMPO does not assume a centralized point where all flows would pass through and hence can be used in a more flexible setting.

Relatively less work has been done to support MPTCP based services. Greory *et al.* have designed a middlebox that supports translation between MPTCP and TCP. In addition, Zubair

et al. have investigated the security issues exposed by the multiple MPTCP subflows. They have discovered a new MPTCP cross-path attack [97], which allows a service provider to infer the path quality of its competitors if its customer's MPTCP subflows go through multiple provider's networks. However, their work does not explicitly address the subflow association issue.

4.7 Summary

Making network devices MPTCP-aware may benefit both the performance of MPTCP sessions and the quality of network services. In this project, we explore a first step towards this goal by solving online MPTCP subflow association problem. Associating MPTCP subflows in network is more challenging than doing it at end hosts because the identity of a MPTCP subflow can be missing due to network dynamics or packet sampling. SAMPO solves this problem by using both tokens (when available) and analysis of overlapping in DSN space. Both our theoretical analysis and experimental results show that SAMPO can detect and associate MPTCP subflows with high accuracy even when only a very small portion of each MPTCP subflow is available.

Chapter 5

Taking Consensus as a Service in Software Defined Network

5.1 Main Results

We first introduce a brief overview of Raft and P4 [5] in Section 5.2. We then illustrate a few network failure scenarios that may arise when applying Raft to a distributed SDN control cluster (see Section 5.3). We demonstrate how these failure scenarios can severely affect the *correct* or *efficient* operations of Raft: in the best case they significantly reduce the available "normal" operation time of Raft; and in the worst case, they render Raft unable to reach consensus by failing to elect a consistent leader. It is worth noting that the problems highlighted here are different from those addressed by, *e.g.,* the celebrated CAP Theorem in distributed systems [98, 99], which establishes impossibility results regarding simultaneously ensuring availability and (strong) consistency under network partitions. This result has been recently generalized in [100] to SDN networks in terms of impossibility results regarding *ensuring network policy consistency under network partitions*. In contrast, we argue that thanks to the inter-dependency between the network OS as a distributed system and the network it attempts to control, *SDN introduces new network failure scenarios that are not explicitly handled by existing consensus algorithms such as Raft, thereby severely affecting their correct or efficient operations.*

In Section 5.4.1, we discuss possible "fixes" to circumvent these problems. In particular, we argue that in order to fundamentally break this inter-dependency, it is crucial to equip the

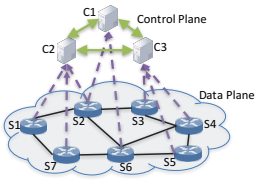


Figure 5.1: SDN Control Plane Setup.

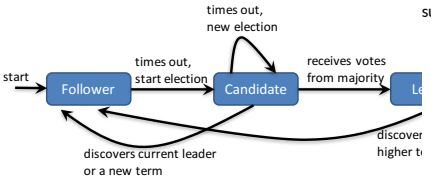


Figure 5.2: Raft States.

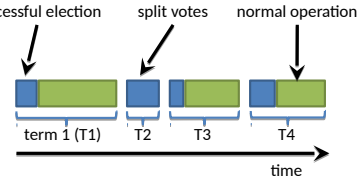


Figure 5.3: Raft Terms.

SDN control network with a resilient routing mechanism such as *PrOG* [101] that guarantees connectivity among (non-partitioned) SDN controllers under arbitrary failures. We then propose a network-assisted Raft consensus algorithm that takes advantage of programmable network and offloads certain Raft [4] functionality to P4 [5] switches. Our goal is to improve the performance of Raft without sacrificing scalability. Using a vanilla Raft implementation [102], *PrOG*, and a P4 switch simulator, we provide preliminary evaluation results in Section 5.5. Finally, we discuss the related work in Section 5.6 and conclude the project in Section 5.7.

5.2 Fundamentals

5.2.1 Raft Overview

Raft [4] is a consensus algorithm designed as an alternative to (multi-)Paxos [17, 103], and it meant to be more understandable than Paxos, and it is also formally proven safe. Raft is as efficient as Paxos, but its structure is different from Paxos. This makes Raft more understandable and it also provides a better foundation for building practical systems. To enhance understandability, Raft separates the main consensus components into the following subproblems: 1) Leader election: a new leader is elected when the current leader fails; 2) Log replication: the leader accepts log entries from clients and replicates them, forcing other logs to be consistent with its own log; and 3) log commitment: few restrictions on leader election are enforced to ensure safe log commitment, *i.e.,* if any member applied a particular command to its state machine, then no other member may apply a different command for the same entry. Raft starts by electing a *strong leader*, then it gives the leader full responsibility for managing the replicated log. The leader accepts log entries from clients, and replicates them to other servers. When it is safe to apply log entries to the state machines, the leader tells the servers to apply them to their local state machines.

Raft States. Raft clusters typically contains odd number of members (*e.g.,* five servers allows two failures). As illustrated in Figure 5.2, a server can be in one of three states: follower, candidate, or leader. Normally, there is one leader in the cluster and others are just followers passively receiving RPCs from the leader or candidates. A leader responds to clients' requests and replicates the log entries to followers. In case a client contacts a follower, it will be redirected to the leader.

Raft Terms. As shown in Figure 5.3, time is divided in terms of arbitrary length. Terms are monotonically increasing integers, where each term begins with an election. If a candidate wins an election, it will serve as the leader for the rest of the term. Terms allow Raft servers to detect obsolete information such as stale leaders. Current terms are exchanged whenever servers communicate using RPCs. When a leader or a candidate learns that its current term is out of date (*i.e.,* there exists a higher term number), then it immediately reverts to the follower state. If a server receives a request, either a vote request or a replicate log entry from the leader, with a stale term number, it rejects the request.

Raft Leader Election. A leaders in Raft send periodical heartbeats to all followers, and all other servers remain in the *follower* state as long as they are receiving heartbeats from the current leader. If a follower receives no heartbeat messages over a predefined period of time (*election timeout*), then it assumes that there is no leader and starts a new election. To start a new election, the follower that encountered the *election timeout* increments its current term, vote to itself, and moves to candidate state. Then, it send *request-to-vote* RPCs to other servers. Three possible outcomes can happen: *Win Election:* if it receives votes from a majority, it sends heartbeats to all servers to prevent new elections and establish its authority for its term. *Lose Election:* While waiting for votes, the candidate server may receive a heartbeat message from another server claiming to be the leader. If the received term number is at least as large as the candidate's current term, then it surrenders as a follower. *Split Votes:* If no candidate server receives majority of votes, then one of the servers will timeout for not receiving heartbeat messages from any leader and start a new election. Raft uses randomized timeouts to ensure Split Votes is a rare event. Raft enforces restrictions on elected leaders *e.g.,* a server votes to a candidate if its term is higher, and the candidate's log is at least as up-to-date as its own; otherwise the server rejects the vote request. Therefore, receiving a majority of votes means that the new leader's log contains all committed entries.

To ensure safe log commitment, Raft enforces restrictions on elected leaders to guarantee

that all committed entries from previous terms are present on the new leader. During the election process, the candidate must receive majority from the cluster. A server in the cluster will vote to the candidate if the term is higher and the candidate's log is at least as up-to-date as its own log. Otherwise, the server rejects the vote request. Therefore, receiving a majority means that the new leader log contains all committed entries.

5.2.2 P4 Overview

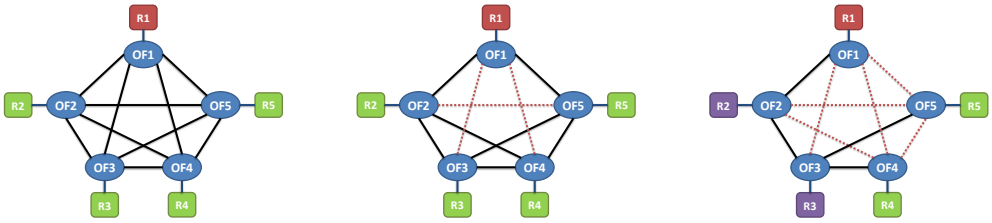
P4 [5] is a language to program data-plane behavior of network devices. It can be used to support customized functionality, e.g., evolving OpenFlow standard [13], specific data-center packet processing logic, etc. The P4 language composes an abstract forwarding model in which a chain of tables are used for packet processing. The tables match pre-defined packet fields, and perform a sequence of actions. Then P4 compiler takes charge of the abstract forwarding model to a concrete implementation on a particular target platform, e.g., software switches, FPGAs, etc.

There are five major components in a P4 program: 1) control blocks which specify a way of composing tables; 2) tables which specify packet processing logic, a high-level behavior representation about field matching and corresponding actions; 2) customized packet header fields which are a collection of packet bytes; 3) packet header parser which describes a way of transforming incoming packets to field matching instances; 4) actions which forward or drop packets; modify fields; perform stateful memory operations; encapsulate or decapsulate headers.

Beyond these five major components, P4 offers additional mechanism for performing stateful packet operations. Network-based Raft algorithm uses registers to keep track of Raft states like logs and state machines. Registers provide persistent state which is organized as an array of cells. We need to specify the size of each cell and the number of cells in the array, when declaring a register for Raft state.

5.3 Raft Meets SDN

In distributed network OS such as ONOS and ODL, multiple controllers must maintain a *consistent* global view of the network. This is achieved by employing a consensus protocol such as Raft to ensure consistency among the replicated network states maintained by each controller.



(a) Control Cluster under Normal Operations.

(b) Oscillating Leadership.

(c) No Leader Exists.

Figure 5.4: Motivating Examples.

The connectivity among these controllers can be provided either via a dedicated *control* network ("out-of-band") or via "in-band" (virtual) control network through the data plane under their control [14, 15, 100, 104]. In either case, we refer to the (dedicated or virtual) network connecting the controllers as the *control network*. We assume that it consists of OpenFlow switches with flow rules installed by the *same* controller cluster to which it provides connectivity.

Figure 5.4(a) shows an SDN control cluster with three controllers in a full-meshed control network with five OF switches. The controllers run Raft to ensure consistency among the replicated (critical) network states they maintain. We will use a few toy (contrived) examples to illustrate the new failure scenarios that may arise when applying Raft to a distributed SDN control cluster for consensus. In the following scenarios, we assume that initially **R1** is the leader (in red) and green indicates that the logs of a member (controller) is up-to-date. The current term is **T1** as seen by all members. Up links are black, while down link are red, and the cluster is not partitioned.

Scenario 1: Oscillating Leaders. Figure 5.4(b) shows a Raft cluster, where the links (R1, R3), (R1, R4), and (R2, R5) fail. Either **R3** or **R4** will time out for not receiving heartbeats. Assuming **R3** times out first, then it will increment its term number to **T2**, vote for itself, and request votes from **R2**, **R4**, and **R5**. After getting the votes, **R3** will be the leader, and the current term vector becomes **Term(R1=1,R2=2,R3=2,R4=2,R5=2)**. After that, **R1** will step down after learning about the higher term **T2** from **R2** or **R5** through heartbeat messages, and update its term to **T2**. **R1** and **R3** cannot communicate, because the link (R1, R3) is down. **R1** will time out, and increase its term number. Thus, it can get **R2** and **R5** votes to become a leader for term **T3**, force **R3** to step down, and snatches the leadership, because its term number is larger than **T2**. The term vector becomes **Term(R1=3,R2=3,R3=2,R4=2,R5=3)**. Then, we are back to the initial settings, and the whole scenario can be repeated.

Assuming after **R3** became the leader again for **T4**, it receives some requests and updates the logs for all nodes except **R1**. Thus, when **R1** tries to become the leader for **T5**, it will not receive votes from **R2** and **R5**, because their logs are more recent than **R1**'s log, and the current term vector will be **Term(R1=5,R2=5,R3=4,R4=4,R5=5)**. Thus, **R3** will step down. Currently, there is no leader, so whoever times out first can be the leader except **R1**. If **R3** or **R4** times out first, what we just discussed will be repeated.

Assuming **R2** times out first, increments its term to **T6**, and becomes a leader. In this case, **R5** will not receive heartbeats ((R2, R5) is down), and will try to become a leader. Thus, we can notice the leadership will be oscillating between either (R2, R5) or (R1, R3, and R4). In the worst case, the cluster can be dead, since clients sending requests to the current leader will be redirected to the new leader. By the time they contact the new leader, it will change again.

Condition. Up-to-date nodes have a quorum, but they cannot communicate with each other.

Scenario 2: No Leader Exists. Figure 5.4(c) shows a Raft cluster, where the leader **R1** successfully updated the logs of **R4** and **R5**, but failed to update the logs of **R2** and **R3** (in purple), due to link failure, packet drop or network congestion. Assuming **R2** times out first for not receiving heartbeats, it will increase its term to **T2**, forcing **R1** to step down. In this case, even though **R2** and **R3** have a quorum (**R2** connected to **R1** and **R3**; **R3** connected to **R2**, **R4**, and **R5**), they will not be able to become a leader, because their logs are not up-to-date. **R1**, **R4**, and **R5** cannot be the leader also, because they do not have a quorum. Therefore, the cluster is not live anymore, even though the underlying network is *not partitioned*.

Condition. Nodes have a quorum, but they have obsolete logs, and nodes having up-to-date logs, do not have a quorum.

In summary, consensus distributed systems are designed agnostic from underlying network, with the assumption of all-to-all communication between network entities, as long as the underlying network is not partitioned. In SDN, these consensus systems are used to manage the underlying network. Therefore, making progress depends on updating the OpenFlow rules, which depends on the connectivity between servers (chicken and egg situation.) Hence, a novel approach needs to be designed to solve this issue in SDN networks.

5.4 System Design

5.4.1 Possible Solutions During Leader Election

In this section, we illustrate the solution requirements and discuss the limitations of some possible solutions. Then, we briefly introduce a prospective solution for this problem.

Solution Requirements. The problem with Raft, and distributed protocols in general, is the assumption of *all-to-all connectivity among cluster members as long as the network is not partitioned*. In SDN, switches in the data plane forward traffic based on decisions made by the control plane, which uses a consensus protocol like Raft to ensure the state is replicated correctly. Upon failures, controllers may lose connection to each other and to switches in the network. Moreover, in SDN failures can be *physical*, i.e., physical link/node failures, or *logical*, e.g., two servers are physically connected, but there are no corresponding rules installed, thus preventing them from communicating. Therefore, with unpredictable network failures, the solution should be resilient against arbitrary link/node failures and ensure that controllers retain reachability among them whenever the underlying network is physically connected.

Gossiping. One common distributed systems solution to restore connectivity is via gossiping. Thus, to achieve connectivity, Raft can be extended to enable servers to gossip and forward heartbeats through other servers to overcome the failures affecting their direct communication. This solution may help avoid the scenarios mentioned in Section 5.3 by probabilistically forwarding Raft messages (heartbeat, replicated logs, ... etc.) through some other servers, which may have a path to the original message's destination. If the number of these servers is large enough, there is a high probability one of them is able to forward the message to its destination. As long as followers receive these messages from the leader, the cluster can still be live, as they will renew their heartbeatTime, and avoid starting a new election process.

However, the problem with such a solution is that it may work in some cases only, as it depends on the underlying network connectivity and which servers are selected to forward messages. Since it assumes uncorrelated link failures and might be affected by new link failures and Raft timeouts, it is not guaranteed to work in all cases and scenarios. Alternatively, flooding can be used instead to ensure message delivery. However, it may lead to network congestion, which may not only increase packet delay and affect data plane flows, but create additional packet losses/network failures. We believe that *built-in resiliency* in the (control network/data plane) is essential for high-availability of SDN controllers. The control network (data plane)

should not rely on the control plane to recover from failures, and should have pre-installed rules in switches for automatic failure recoveries. Fortunately, this can be achieved via a new routing paradigm proposed in [101], known as "routing via preorders", which provides adaptive resilient routing to ensure all-to-all communications among servers regardless of network failures, as long as the underlying network is not partitioned. We briefly discuss it as a prospective solution next.

Routing via Preorders. Considering a network $G = (V, E)$ representing the control plane, and a flow F from a source s to a destination d, where s, d are SDN control cluster servers. A preorder is defined on a node set V' where $s, d \in V'$, and $V' \subseteq V$. This preorder specifies the relation between any two nodes u, v, where $v \rightarrow u$ means v is a child of u, and $v \leftrightarrow u$ means u, v are siblings. The result is a directed connected (sub)graph $G' = (V', E')$, where each edge in E' ($\subseteq E$) is oriented either uni-directional or bi-directional, and G' is called a preordered graph (*PrOG* in short.) At each node, packets can be forwarded to any of its parents or siblings, (*i.e.,* follow any directed path from s to d without enumerating all paths between them.) Bi-directional links are only activated along one direction upon failures. Through its construction, *PrOG* includes all possible paths between s, d. Therefore, it ensures controllers retain connectivity, as long as the underlying graph is not partitioned, allowing Raft participants to exchange data even if direct communication links are unavailable. *PrOGs* are constructed for each source-destination pair, using a modified version of breadth-first search.

Upon failures, the affected parts of *PrOG* are deactivated, and traffic is routed along the remaining part, where each node uses its alternative outgoing links if they exist. Upon link/node recovery, relevant links are activated. The original *PrOG* is restored, when all links/nodes recover. *PrOG* also provides an additional feature, where the constructed graph can have a bounded threshold for the cost of the included paths from s to d (see [101] for more details.) Thus, a threshold can be defined for normal operations, and a relaxed threshold to be used upon failures. Therefore, Raft members can communicate within a predefined threshold even when there are failures. *PrOGs* are then converted to OpenFlow rules pre-installed in switches. Switches are provided with a small functionality required to maintain and update an internal state. For example, each switch maintains the state of its outgoing links, whether they are active or not, and sends activation/deactivation messages upon link recovery/failure.

Summary. "Routing via Preorders" uses local data plane operations to achieve resiliency under arbitrary link/node failures, without any involvement from the control plane to recompute

routes, since they are pre-computed and pre-installed. Thus, it avoids the cyclic dependency between control network connectivity and management, where controllers need to setup rules to recover from failures, but cannot reach switches because of failures. Therefore, it provides all-to-all communications among cluster members for a stable Raft leadership and enables the cluster to progress regardless of failures. Finally, it is a general solution as it does not require any modifications to Raft, and can be used by other distributed protocols as well. The correctness and overheads of *PrOG* are discussed in [105]. We will expand its design for *control network resiliency* in a future project.

5.4.2 Offloading Raft to P4 Switch

There are three key requirements for our network-assisted Raft. First, the correctness of Raft algorithm should not suffer by offloading processing logic to the network. Second, the Raft logic on a switch should be able to respond most requests directly for improved performance. Third, the Raft logic on a switch should safely discard obsolete log entries and even state machine for scalability. In the basic Raft consensus algorithm, there are three major elements: *leader election*, *log replication*, and *log commitment*. In order to satisfy the above requirements, our system consists of two major components: front-end implemented in a switch taking care of *log replication* and *log commitment*, and back-end in a server having a complete Raft implementation[1]. Front-end enhances Raft in two aspects. First, it is able to perform Raft-aware forwarding. Second, it can quickly respond to Raft requests by rewriting the incoming packets. The job of back-end maintains complete states on the server for responding certain requests (described in the following subsection) which might not be fulfilled by front-end. The unique feature of our proposed solution is that we duplicate only the necessary logic to switches which act as a cache to reduce consensus latency, and we minimize the storage of replicated log entries and state machine in switches. The entire Raft algorithm is still running on the server machines. This partial offloading architecture helps improve the performance of Raft without sacrificing scalability.

Raft-aware P4 Switch. Front-end runs in a P4 switch which parses Raft requests and caches Raft states using P4 primitive actions. Upon the reception of a request, front-end parses the request and rewrites the packets to construct the corresponding response. It also forwards the

[1] *leader election* and *log replication* are duplicated at front-end and back-end to improve performance and scalability.

original packet to back-end for liveness check which is part of leader election. Normal response would be generated from back-end and sent to the switch as well, but front-end does not need to forward the message and only extracts necessary flow control information from it. For certain request, front-end may not be able to generate response due to limited information available on the switch, and such a request will be served normally by back-end. For example, when a new server joins the cluster, it needs to fetch all logs which may not be available at front-end, and then back-end would serve the request.

Now we discuss how front-end can forward certain Raft message without involving the back-end. In Raft, the requests from a client can only be handled by the leader. In the bootstrap phase, the client would randomly choose a server in the cluster to talk with. If the selected server is not a leader, it would notify the client of the leader's IP address if it knows. Then the client can issue a new request to the leader. In our design, since front-end is aware of Raft, front-end of the selected server can forward the request to the leader directly and reduce the communication overhead.

In a practical system, especially when *log replication* and *log commitment* are implemented in switches, the log and state machine of Raft cannot grow without a bound. To solve this problem, we adopt a similar but simpler approach as snapshot mechanism described in Raft [4]. In our scenario, front-end just needs to discard obsolete information without storing it because back-end always keeps all necessary information. However, the mechanism for discarding state machine is different from discarding obsolete log entries because front-end needs to know whether a requested item is already in swtich's state machine or server's. Thus, before discarding state machine cached in a switch, it needs to maintain a dictionary for existing keys in the system.

Handling Failures. We now discuss how to handle failures in this partial offloading architecture. Because front-end records the gap between two consecutive Raft heartbeat, if the gap is larger than a timeout value, front-end knows that back-end is down. Front-end will not discard any log and state machine from now on and sync them after the failed back-end is recovered. Communication failure between front-end and back-end is equivalent to a back-end failure.

If front-end fails, more specifically, the switch running front-end fails, back-end times out and issues a RequestVote message which cannot reach others. Back-end then times out and increases its term (defined as a representation of virtual time in Raft) indefinitely. Later on, when the switch is recovered, it will retrieve previous state by forwarding requests to back-end

and observing its response. At this point, back-end may have the largest term number, but may not have up-to-date log. Thus, it would step down the existing leader, and propose a new round of leader election.

5.5 Evaluation

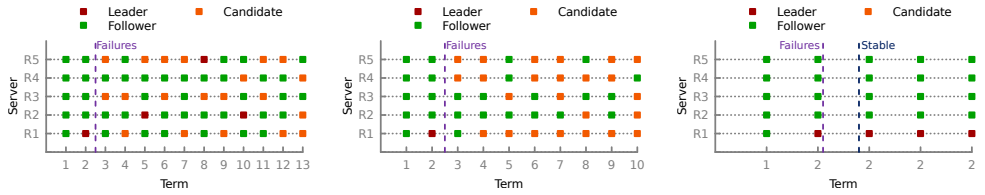
5.5.1 Raft with Routing via PrOG

In this section, we compare the results of vanilla Raft and *PrOG*-assisted Raft to show that *PrOG* resolves the issues presented in Section 5.3 and enables a more resilient and robust SDN distributed control platform.

Experiment Setup. Standard Raft C++ implementation in LogCabin [102] is used in the experiments. LogCabin is a distributed storage system which supports all major Raft features like log replication, membership changes, *etc.*. The basic setting of our experiment is to create six Docker containers [106] (Ubuntu 14.04) in which five containers serve together as a Raft cluster, while the other one serves as a client which reads logs from the cluster or writes logs to the cluster. Topology is setup as illustrated in Figure 5.4 and Open vSwitch [47] (OVS) instances are used as software switches. Data written to the Raft cluster is replicated across all cluster members (*i.e.,* containers) by a Raft leader. We disable vote withhold [107] and simulate the two failure scenarios described in Section 5.3.

Results. To demonstrate the effectiveness of *PrOG*, we present three types of results: 1) leadership shifting diagram; 2) statistics of clients' failed attempts when accessing cluster leader; and 3) statistics of cluster availability time. Leadership shifting diagram is a straight-forward way of observing the states of Raft servers at each term before and after failure occurs. In Figure 5.5, the x-axis shows the current term number, and the y-axis shows the raft server. Different colors shows different states: leader (red), follower (green), and candidate (orange), *e.g.,* Figure 5.5(a) `R1` starts as follower in term 1, then leader for term 2, then follower again for term 3, and so on. Note that for leaders, we don't show the transition to candidate, since it is implied by successful transition to leader.

Figure 5.5(a) shows the results for Figure 5.4(b). It shows the leadership keeps oscillating and is not stable. In our experiments, we noticed that this blocks the client from being able to read or write to the storage for a large number of trials (detailed later.) Figure 5.5(b) shows the results for Figure 5.4(c), in which no viable leader exists, since servers cannot *directly*

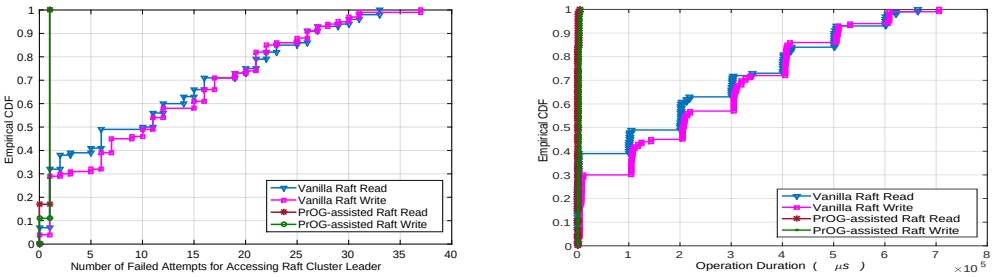


(a) Raft: leadership keeps oscillating between servers (unstable).

(b) Raft: no viable leader (liveness lost).

(c) PrOG: leadership is stable.

Figure 5.5: Results for simulating the motivation examples using vanilla Raft and *PrOG*-assisted Raft.



(a) Client suffers much more failed attempts for accessing cluster leader in vanilla Raft.

(b) Latency of a request operation increases accordingly.

Figure 5.6: Vanilla Raft vs. PrOG-assisted Raft.

communicate with each other, therefore the client cannot read or write to the storage anymore. Figure 5.5(c) shows that our extension resolves the issues in Figure 5.5(a), because servers can indirectly communicate with each other through other servers, therefore the client can read from and write to the storage quickly. The system is stable from term 2.

When a client performs read/write operations on a Raft cluster, it will randomly select one server in the cluster. If the selected server is not the current leader, it replies with the current leader's IP. However, if there is no leader at the moment, no suggestion is returned, and the client will randomly select a server again. The client issues 100 read/write requests with 1-second interval under the failure scenario in Figure 5.4(b) for each round of experiments. Then, we count the number of failed attempts before the client successfully reaches the current cluster leader as shown in Figure 5.6(a). Moreover, we measure the latency of each read/write request as shown in Figure 5.6(b). The results demonstrate that *PrOG*-assisted Raft is more robust to network failures. In terms of why most durations in Vanilla Raft are close to $N \times 10^5 \mu s$ (N is positive integer), it is because the client is set by default in LogCabin to wait $10^5 \mu s$ before trying another server's IP.

We also carefully analyzed the log of the five Raft servers to sum up availability time of the whole Raft system under the oscillating leadership scenario. The availability time in our experiments is defined as the total time period in which a leader is available, because the distributed system can only serve clients when a leader exists. We perform five rounds of three-minute experiments, and calculate system availability time. The average availability time is 75.67s (42.04% out of 180s), and 248 times of leadership shifting happen, even though the network is not partitioned.

5.5.2 Raft-aware P4 Switch

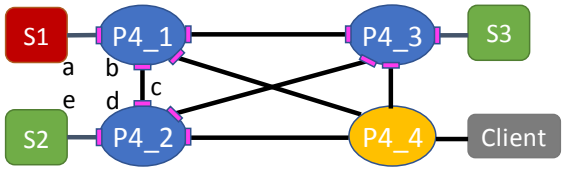


Figure 5.7: Experimental Setup

In this section, we present the preliminary experimental results to demonstrate the feasibility and efficiency of our HydraNF. We show the experimental setup in Figure 5.7, which consists of four Docker containers and P4 switches (simulated by p4factory [108]). We run LogCabin [102] in three containers (leader in red; followers in green) and run three Raft-aware P4 switches (in blue), serving together as a Raft cluster. The box in grey acts as a client, and the P4 switch in yellow performs regular forwarding. We instrument LogCabin to measure the interval of RPC calls and run tcpdump on the NICs in pink to record timestamp.

Table 5.1: Decomposed Latency between Raft Leader and a Follower.
a: RPC latency at leader side + bidirectional latency between *S1* and *P4_1*; b: bidirectional latency in *P4_1*; c: bidirectional latency between *P4_1* and *P4_2*; d: bidirectional latency in *P4_2*; e: bidirectional latency between *P4_2* and *S2* + latency at Raft follower

	a	b	c	d	e	sum
Heartbeat (Vanilla)	106	233	6	231	94	670
Heartbeat (ours)	110	244	4	243	n/a	601
Write (Vanilla)	779	243	8	251	1206	2487
Write (ours)	n/a	479	7	451	n/a	937

We measure the latency (in µs) between a Raft leader and a follower for heartbeat message and client's write request, as shown in Table 5.1. The latency is decomposed into fine-grained

segments. We can observe latency benefits for both heartbeat and write request. Moreover, the proof-of-concept does not add significant memory usage compared to the one performing regular forwarding. Note that the current P4 switch is running as a simulator, we expect better performance when running front-end on a real hardware P4 switch.

5.6 Related Work

SDN availability. SDN controllers may take advantage of established techniques from the distributed systems literature. For example, a control cluster with multiple controllers could use a distributed storage system for durable state replication. Distributed SDN controller designs rely on consensus algorithm, such as Paxos used by ONIX [104] and Raft used by ONOS [14], and even stronger consistency guarantees are required by Ravana [109]. LegoSDN [110] focuses on controller crash failures caused by software bugs. Statesman [111] demonstrates incrementally mitigating up-to-date state to switch with obsolete state when a control master fails. Liron *et al.* [112] propose a model for designing distributed control plane, which maintains connectivity between a distributed control plane and the data plane. In contrast, we study the availability issues in consensus algorithm, and propose *PrOG* to enhance Raft. We assume that Network OS employs consensus algorithms, such as Raft, to maintain the correctness of control logic managing the network, and enhaces it with a "self-healing" resilient control network.

Robust message exchange. Robust message exchange in SDN is fundamental for controller availability. Webb *et al.* [113] propose a way of deploying tightly-coupled distributed system in wide area in a scalable way. It preserves efficient pairwise communication through an overlay network with gossip-based communication protocol. Schiff *et al.* [114] propose a synchronization framework for control planes based on atomic transactions, implemented in-band, on the data-plane switches. Akella *et al.* [115] tackle the problem of in-band network availability and synthesize various distributed system ideas like flooding, global snapshots, etc. HyperFlow [116] utilizes publish-subscribe messaging paradigm among controller instances to replicate network events, and local state is built solely by controller application based on subscribed event. Muqaddas *et al.* [117] quantify the traffic exchanged amongst controllers running Raft and summarize that the inter-controller traffic scales with the network size. *PrOG* provides a general robust and resilient message exchange mechanisms for both in-band and out-of-band control channels.

5.7 Summary

SDN controllers use distributed consensus protocols, like Raft, to manage the network state and provide a highly available cluster to the underlying networking elements. Therefore, SDN controller liveness depends on all-to-all message delivery between cluster servers. In this chapter, we use Raft to illustrate the problems which may be induced by this inter-dependency in the design of distributed SDN controllers. We discuss possible solutions to circumvent these issues. Also, by leveraging programmable devices, we propose to partially offload certain Raft functionality to P4 switches for improving latency, while not sacrificing scalability. Our evaluation results show the effectiveness of *PrOG* and Raft-aware P4 switches in improving the availability of leadership in Raft used by critical applications like SDN controller clusters.

Chapter 6

Parallelizing Network Functions For Accelerating Service Function Chains

6.1 Main Results

We present HydraNF, a parallelism mechanism to accelerate SFCs spanning multiple servers. Instead of parallelizing NFs as much as possible (*e.g.,* [1, 2]), HydraNF employs a controller that converts a sequential chain into a hybrid chain [1] and parallelizes packet processing only if it is *beneficial*. Additionally, the controller adopts traffic level parallelism to distribute traffic in an optimized way to satisfy service level objectives (SLOs) of target traffic. The controller programs both software and hardware switches to activate parallelism across NFs spanning multiple physical servers. HydraNF employs a customized data plane to support hybrid chains without modifying the implementation of existing NFs. Based on the instructions from the controller, HydraNF data plane *mirrors* packets to parallelized NFs and then *merges* their outputs to ensure correctness – *i.e.,* traffic and NF states changed by a hybrid chain must be identical to what would have been produced by the original sequential SFC. We make the following contributions:

• We identify the motivations and challenges in SFC parallelism (§ 6.2), and design HydraNF, a mechanism to support SFC parallelism across multi-core servers (§ 6.3).

• We present HydraNF controller (§ 6.4) to enable both NF level and traffic level parallelism,

[1] A hybrid chain consists of both parallel segments (with NFs processing packets in parallel) and sequential segments (with NFs processing packets sequentially).

which effectively converts sequential SFCs into hybrid ones, and calculates optimized paths over multiple servers for traffic steering. In addition, we present key building blocks in HydraNF data plane, and explore the placement choices for a high-performance data plane (§ 6.5).

• To demonstrate the effectiveness of HydraNF (§ 6.6), we implement a prototype and evaluate it via both synthetic SFCs generated by simulation, and practical SFCs constructed by off-the-shelf open-source and production-grade NFs.

6.2 Challenges in Parallelizing VNFs

In this section, we present a few examples that highlight the motivations in designing a system for accelerating SFC with parallelism. We also articulate some of the key challenges and issues in exploiting SFC parallelism.

NF Order Dependence, State Poisoning, and Function-Level Parallelism. The basic premise of SFC *function-level* parallelism is that given a pair of NFs, if the operations of two NFs applying to the same traffic stream do not conflict, then they can be executed in parallel. For example, if both NFs simply perform read operations, they can be parallelized. If one performs a read operation and another performs a write operation, they can be parallelized if and only if they do not operate on the same header field. Likewise, if both perform write operations (including inserting/removing headers/bits in the packet), they cannot be executed in parallel if the data portions (header or payload) they modify potentially overlap. Given such NF order independence, a sequential SFC can be converted to a *hybrid* one with both sequential and parallel NF segments.

However, order dependency is necessary but not sufficient in determining whether it is *safe* to parallelize an SFC, one must also take into account the states maintained by NFs. Otherwise this may lead to *state poisoning*. In Figure 6.1, consider a simple SFC consisting of a L4 (layer-4) FW (which does not modify packet headers, but may drop packets from certain sources) followed by a *dynamic* NAT (which re-writes the public destination IP address carried in a packet to the next available private IP address in a pool). Since L4 FW only reads packet headers while NAT (re-)writes destination IP address header, based on NF order dependency

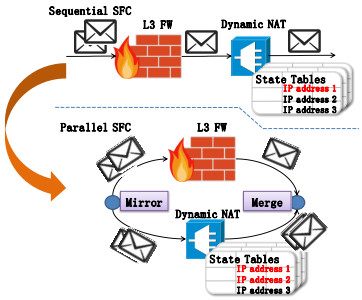


Figure 6.1: NF State Poisoning in Function-level Parallelism

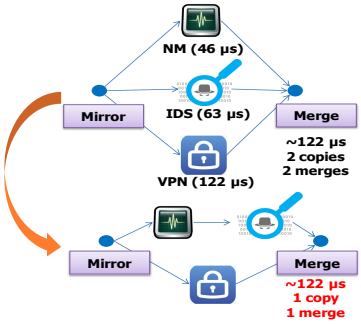


Figure 6.2: Reducing Function-level parallelism with SFC latency constraints

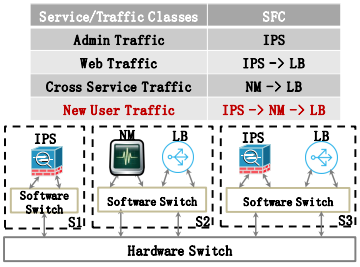


Figure 6.3: Traffic Distribution Across Multi-core Servers.

alone, these two NFs can be executed in parallel[2]. In the case of a *static* NAT[3], it is indeed safe to parallelize the sequential FW → NAT SFC by executing the two vNFs in parallel. However, in the case of a *dynamic* NAT, the operation of NAT on a new flow alters its internal state – the availability of private IP addresses is reduced by one. This can potentially lead to exhaustion of available IP addresses – thus *state poisoning* – when in fact some of these addresses are not being used.

Reducing Function-level Parallelism Overhead with SFC latency constraints. If NFs are executed in a sequential way, traffic SLOs may break due to accumulated NF processing delays. Thus, function-level parallelism is adopted to satisfy latency constraints. However, parallelizing NFs as much as possible may not gain benefits, instead wasting available resources. For example, in Figure 6.2, after dependency analysis [1, 2], network monitoring (NM), intrusion detection system (IDS), and VPN gateway could be executed in parallel. Since the processing delays of these NFs are different, the overall processing delay is determined by the longest NF processing delays plus parallelism overhead. In the bottom part of Figure 6.2, the overall processing delay is similar to the upper part of the figure, but it only uses fewer CPU cores, and fewer packet mirror and merge operations. Thus, when a sequential SFC is converted

[2]In order to deal with the issue that FW may drop/block certain packets/flows, ParaBox [1] uses a timeout mechanism in the *merge* function whereas NFP [2] generates a "nil" packet to signal the merge function to drop the packet already processed by the NAT.

[3]A static NAT employs a *fixed* mapping, e.g., a hash function, to translate the public destination IP address to a private one.

into a hybrid SFC, how to reduce parallelism overhead needs to be considered.

Traffic Distribution across Multi-core Servers. In a cluster of multi-core servers, there are many factors to consider when deciding *whether* and *how* to parallelize an SFC. For example, it might be beneficial to execute an SFC entirely within a single server, or even within a CPU core – *run-to-completion* [118, 119] to avoid network latency or core switching overheads. Alternatively, one might want to distribute the vNFs across multiple servers and execute them in parallel, or replicate some heavily loaded vNFs to execute them in parallel on multiple servers. Which options are best in terms of the SFC processing latency and overall system throughput will depend on individual vNF performance, traffic volumes, server capabilities, switch capabilities, and network bandwidth. Moreover, *real-world* operational constraints of vNF placement (e.g., security concerns requiring some third-party vNFs be placed on certain servers running as VMs) further limit the options of SFC parallelism. After NF provisioning, given multiple SFCs as exemplified in Figure 6.3, how to distribute traffic in an optimized way by taking parallelism into consideration is also challenging.

All in all, when developing mechanisms for accelerating SFC processing in a multi-server environment, we must not only explore opportunities and constraints in parallelizing SFC processing along both the *(network) function* and the *traffic*-levels, but also need to *co-design* the rules needed (in hardware and software switches) for traffic distribution to appropriately steer and load balance traffic among multiple servers while minimizing the network latency. HydraNF is designed to address these challenges while exploring the opportunities afforded by operating NFV in multi-core, multi-server environment to accelerate SFC processing for the high scalability, availability and performance.

6.3 HydraNF Overview

In this section, we present the system architecture and introduce the key components of HydraNF.

Realizing parallel packet processing is by no means straightforward. First, HydraNF must guarantee the correctness of the generated chain by carefully analyzing the *order dependency* of NFs in a chain. The dependency relies not only on the semantics of NFs, but also their configurations and operational rules. Second, HydraNF needs to automatically program both virtual

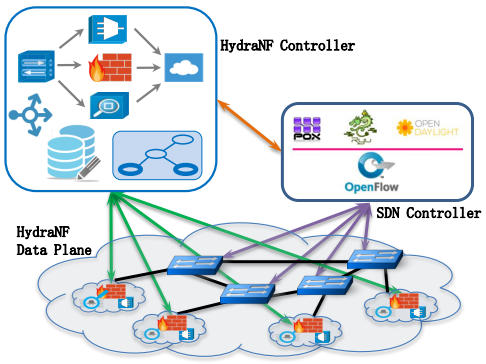


Figure 6.4: System Overview of HydraNF.

switches in servers and hardware switches connecting the servers by creating appropriate data-plane forwarding rules to perform parallel packet processing across multiple servers. Third, the data plane functions of HydraNF to support SFC parallelism should be lightweight, avoiding adding too much processing overhead. Finally, to enable incremental deployment, HydraNF should not require modifications to existing NFs.

We schematically depict the overall system architecture of HydraNF in Figure 6.4. HydraNF consists of a **HydraNF Controller** (HRC) and a **HydraNF Data Plane** (HDP) running on a cluster of multi-core servers equipped with DPDK. These servers are connected via an SDN network. The primary role of the HRC is two-fold. First, based on dependency analysis, HRC constructs a service processing graph (SPG) from each sequential SFC. SPG is defined as a hybrid SFC with optimized processing latency and optimized parallelism overhead (number of mirror and merge operations). Second, based on the knowledge about vNF performance profiles and placement constraints as well as information about server and network resources, HRC provisions an SPG to servers and cores to satisfy SLOs of target traffic load. The HydraNF data plane engine consists of two key building blocks, a *mirror* module which duplicates packets for parallel processing and *merge* module which combines and processes duplicated packets after parallel processing. Its primary role is to enforce SFC parallelized and distributed processing based on the forwarding and processing rules installed by HRC.

6.4 HydraNF Controller

In this section we present HRC design as shown in Figure 6.5. It comprises two key modules: SPG construction and SPG provisioning. We first illustrate how an SPG is constructed from a sequential SFC along with each NF Profile to enable NF-level parallelism which accelerates SFC processing with minimized overhead. After SPGs are generated, we demonstrate how each SPG is provisioned into physical machines with various constraints to support traffic-level parallelism. Finally, the output of HRC is software and hardware switch rules.

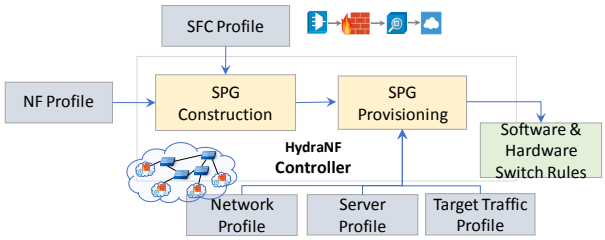


Figure 6.5: HydraNF Controller Design.

6.4.1 SPG Construction

NF-level parallelism is to parallelize packet processing in multiple NFs so that overall processing latency can be minimized. In order to enable NF-level parallelism, we construct SPG based on dependency between any two NFs. After NF dependency is analyzed, a dependency graph can be generated from NF profiles as exemplified in the upper part of Figure 6.6. Next, we present how an SPG is constructed.

Problem Description. SPG construction can be expressed as the following problem:

• *Input*: a finite collection of NFs; processing delay of each NF; dependency between any two NFs [4].

• *Goal*: constructing an SPG with two objectives: 1) minimized overall latency (high priority); 2) minimized parallelism overhead (low priority).

Overall latency is defined as the packet delay caused by SFC processing. When we enable NF-level parallelism, packets needs to be mirrored and merged so that NFs can process the

[4]We assume that dependency between any two NFs is given, because both Parabox [1] and NFP [2] have described methods to obtain NF dependency.

same packet in parallel. Thus, parallelism overhead is defined as the number of mirror and merge operations. Given SPGs with the same overall latency, we prefer an SPG with lowest parallelism overhead as mentioned in § 6.2.

Analysis. The key observation of our analysis is that the problem can be converted to a Bin Packing problem with bin size as critical path in a NF dependency graph. Intuitively, if the processing delays of two parallizable NFs do not exceed the processing delay of the critical path, there is no benefit to parallelize these two NFs. The critical path is defined as a chain with longest processing delay in dependency graph (*e.g.,* 1,7,5,8,10 in Figure 6.6)

Proposition 1. Let $\varphi(g)$ be the processing delay of critical path in dependency graph g. The processing delay of g is bound by $\varphi(g)$ without taking mirror and merge overhead into consideration.

Hence, in order to construct an SPG satisfying two objectives with different priorities, we divide this task into two steps. The first is to identify the minimied overall latency which is determined by the processing delay of a critical path. Then we try to eliminate unnecessary parallelism.

Step 1: finding a critical path. We first apply forward and backward pass algorithm to find the critical path [120] in a dependency graph. It identifies the earliest start and finish times, and the latest start and finish times for packets going through each NF. In Figure 6.6, path in yellow is critical path. An SPG with NF processing latency as large as critical path satisfies the first objective in problem description.

Step 2: eliminating unnecessary parallelism. Demonstrating critical path alone in the dependency graph is insufficient, because we have the second objective to achieve, minimizing pallelism overhead.

Proposition 2. The SPG construction problem is strongly NF-complete.

Proof. This is proved by establishing a polynomial-time reduction from the *Bin Packing* problem, known to be strongly NP-complete [121], which is defined below:

Given a set of bins $S_1, S_2, ...$ with the same size V and a list of n times with sizes $a_1, ..., a_n$ to pack. Find an integer number of bins B and a $B - partition$ $S_1 \cup ... \cup S_B$ of the set $1, ..., n$ such that $\sum_{i \in S_k} a_i \leq V$ for all $k = 1, ..., B$.

Lemma 1. The SPG construction problem with $\gamma_{i,j} = 0$ and the Bin Packing problem are equivalent (note that $\gamma_{i,j} = 0$ implies that there is no dependency between NF i and NF j).

Proof. Given any instance of the Bin Packing problem, we construct a specific instance

of the SPG construction problem as follows. We create a separate object for each NF, $N = 1, 2, ..., |S|$. Certain NFs are shared across multiple chains, we create multiple objects, *e.g.,* node 5 in Figure 6.6. The sizes of the objects match processing delays of NFs. The bin size matches critical path in dependency graph. If there exists an SPG, the latency of which is bound by the critical path, and the number of mirror and merge operations (number of chains - 1) is minimized, then there will also be an n-way partition ρ of objects such that $\sum_{i \in S_k} a_i \leq V$ for all $k = 1, ..., B$. The objects in each bin will correspond to NFs in each chain. The number of chains in SPG will be equal to the number of bins. Hence, a solution to the SPG construction problem yields a solution to the Bin Packing problem. It is known that it is NP-hard to approximate the Bin Packing problem to any factor better than 1.7 [122]. Therefore, *Lemma 1* indicates that the same statement holds for the SPG construction problem, *i.e.,* the SPG construction is at least as hard as the Bin Packing problem, which is strongly NP-complete.

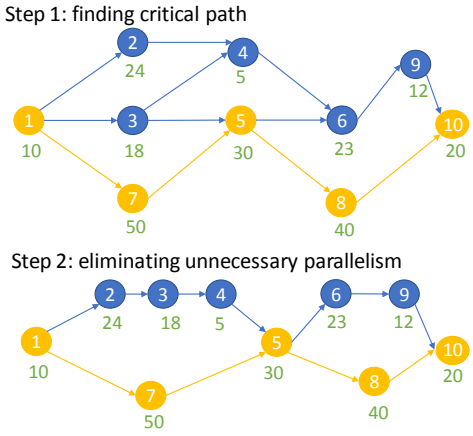


Figure 6.6: An example of SPG construction.

Problem Formulation. An SFC consists of N NFs. NF i ($1 \leq i \leq N$) has a processing delay D_i. z_k indicates whether bin k is used for packing or not. We use B to denote the processing delay bound of the SFC. The SFC can be divided into M chains (*e.g.,* 1,2,4,6,9 in Figure 6.6) based on their dependency. If NF i ($1 \leq i \leq N$) is on chain j ($1 \leq j \leq M$), we have $a_j^i = 1$; otherwise $a_j^i = 0$. If chain j ($1 \leq j \leq M$) is in bin k ($1 \leq k \leq N$), we have $y_k^j = 1$; otherwise $y_k^j = 0$. If NF i ($1 \leq i \leq N$) is in bin k ($1 \leq k \leq N$), we have $x_k^i = 1$; otherwise $x_k^i = 0$. Our problem can be formulated as follows:

$$\min_{x,y,z} \sum_{k=1}^{N} z_k \tag{6.1}$$

$$\text{s.t.} \quad \sum_{k=1}^{N} y_k^j = 1, \qquad \forall j, \tag{6.2}$$

$$z_k \geq x_k^i, \qquad \forall i,\ k, \tag{6.3}$$

$$\sum_{i=1}^{N} (D_i * x_k^i) \leq z_k * B, \qquad \forall k, \tag{6.4}$$

$$x_k^i \geq a_j^i * y_k^j, \qquad \forall i,\ j,\ k, \tag{6.5}$$

$$x_k^i,\ y_k^j,\ z_k \in \{0, 1\},\ \forall i,\ j,\ k. \tag{6.6}$$

Eq. (6.1) minimizes the number of bins. Eq. (6.2) assigns each chain to exactly one bin. Eq. (6.3) ensures that an bin is used as soon as it contains one chain. Conversely, thanks to the objective function, every used bin contains at least one chain. From Eq. (6.4), an SPG's processing time is limited by the bound B. Eq. (6.5) guarantees that, when a chain belongs to an bin, this bin includes all its NFs. Eq. (6.6) indicates the integrality constraints of the variables.

Heuristic Algorithms. Optimal solutions of the above problem could be achieved by existing integer program solvers. However, it may take a quite long time to calculate an optimal solution if input SFCs are complex. Inspired by Pagination problem [123], a variation of Bin Packing problem supporting overlapping items, we take each chain (*e.g.,* 1,2,4,6,9 in Figure 6.6) in dependency graph, instead of each NF, as an basic unit to pack into bins. Certain NFs are duplicated in multiple chains, and thus we take these NFs as overlapping items in the Pagination problem.

We implement 3 heuristics to construct SPG.

• First Fit. The algorithm rescans sequentially each bin already created, and put the new chain into the first bin where it fits.

• Overload-and-remove. The algorithm adds a chain c to the bin b on which c has maximum overlapping size with existing c_b in b, even if this addition actually overloads b. If overloading, the algorithm tries to unload b by removing chain(s) c' which has minimal overlapping size in b. The removed chains c' are rescheduled by putting them to the end of a FIFO queue, and

forbidden to reenter to the same bin b.

• Genetic. The algorithm runs with the cost function to reduce the number of distinct NFs in the last nonempty bin, and penalize invalid bin packing (*i.e.,* exceeding bin size). During mutation, the algorithm transfers one randomly selected chain from one bin to another. The crossover process applies standard two-point crossover without requiring any repair process.

After bin number and chains in each bin are calculated, we finally merge chains in the same bin (serializing parallelizable NFs), and stitch merged chains in multiple bins together to construct an SPG. For example, in Figure 6.6, chains except the critical path can be packed into one bin, the size of which is 150. Then the chains in each bin will be merged into one chain, *i.e.,* 1,2,3,4,5,6,9,10. After that, two chains (the merged chain and the chain on critical path) are stitched together as a constructed SPG as shown in the bottom part of Figure 6.6.

6.4.2 SPG Provisioning

Given an SPG, HRC takes the performance profiles of each vNF, profiles of servers and network resource, and constraints (e.g., vNF placement policy considerations) to decide on SPG provisioning, weighing various options by considering their performance benefits and service level objectives/agreements. These options include a) executing the SPG within a single server using one core or multiple cores, b) parallelizing and executing the SPG within a single server using multiple cores, c) distributing and striding the SPG processing across multiple servers, but executing each SFC sequentially; d) distributing and executing SPG processing in parallel across multiple servers. For each of these options, one can further explore traffic-level parallelism (if available) by running multiple instances of the same SPG.

Due to inter-core communication, switching overheads, and network latency, the best option in terms of SFC processing latency and overall system throughput will hinge on specific vNFs involved, the current server and network resource availability, traffic load and NF placement constraints. Furthermore, and perhaps more importantly, in real-world operational scenarios, there are often multiple concurrent SFCs with vNFs placed on certain servers. When mapping SPG to the physical system substrate, we must also account for these scenarios. For this reason, we conduct SPG mapping to jointly decide vNF placement and traffic distribution based on vNF placement policy constraints, server resource availability, and existing placement of vNFs. The vNF placement decides the number of instances of NFs and the location of instances in servers, while the traffic distribution decide whether certain vNFs (either residing on the same server or

different servers) should be executed in parallel (on different cores within the same server or on different servers).

Problem Statement. Given a set of servers $\mathcal{S}$ and an SFC that consists of a set of NFs $\mathcal{NF}$, we should place each NF $\in \mathcal{NF}$ into a server $\in \mathcal{S}$ by deciding N, the number of instances of the NF, and $\mathcal{X}$, the placement of N instances of the NF into $\mathcal{S}$, and $\mathcal{R}$, the traffic rate distribution among instances of NFs. We use $\mathcal{D}$ to denote the set of transmission delay among servers. Our objective is to minimize the SFC's overall transmission overhead TO. The TO between servers equals $r \times d$, where r denotes the traffic rate between two servers and d denotes the transmission delay between the two servers. The overall TO equals the total TO between any two servers. That is

$$TO^{SPG} = \sum_{r \in \mathcal{R}} \sum_{d \in \mathcal{D}} (r \times d).$$

Problem Formulation. The profile of an SFC is its target throughput requirement T_{SFC}. The profile of NF $i \in \mathcal{NF}$ consists of three parts: processing rate pr_i, CPU requirement cpu_i, and memory requirement mem_i. To meet the throughput requirement of the SFC, each NF has to be realized by N_i instances, where

$$N_i = \lfloor \frac{T_{SFC}}{pr_i} \rfloor, \ \forall i \in \mathcal{NF}.$$

If i and i' ($i, i' \in \mathcal{NF}, i \neq i'$) have dependency, $\alpha_{i,i'} = 1$; otherwise $\alpha_{i,i'} = 0$. The traffic rate from instance $j \in \mathcal{I}_i$ of NF i to instance $j' \in \mathcal{I}_{i'}$ of NF i' is $y_{i,j,i',j'}$.

The profile of a server $\in \mathcal{S}$ consists of two parts: CPU resource CPU and memory MEM. The transmission delay between servers s_k and k^* ($s_k, k^* \in \mathcal{S}$, $k \neq k^*$) is d^{k,k^*}. The two instances in the same server do not have the transmission delay. That is $d^{k,k} = 0$. If server s_k is used, we have $z^k = 1$; otherwise $z^k = 0$. If instance j of NF i is placed in server s_k, we have $x^k_{i,j} = 0$; otherwise, we have $x^k_{i,j} = 1$.

Our problem can be formulated as follows:

$$\min_{x,y,z} \sum_{k\in\mathcal{S}}\sum_{k^*\in\mathcal{S}}\sum_{i\in\mathcal{NF}}\sum_{i'\in\mathcal{NF}}\sum_{j\in\mathcal{I}_i}\sum_{j'\in\mathcal{I}_{i'}}(z^k * z^{k^*} \tag{6.7}$$

$$* d^{k,k^*} * y_{i,j,i',j'} * (x^k_{i,j} + x^{k^*}_{i',j'}))$$

s.t.

$$N_i - 1 = \sum_{k\in\mathcal{S}}\sum_{j\in\mathcal{I}_i} x^k_{i,j} * (1 - z^k),\ \forall i, \tag{6.8}$$

$$CPU \geq \sum_{i\in\mathcal{NF}}\sum_{j\in\mathcal{I}_i}[(1 - x^k_{i,j}) * z^k * cpu_i],\ \forall k, \tag{6.9}$$

$$MEM \geq \sum_{i\in\mathcal{NF}}\sum_{j\in\mathcal{I}_i}[(1 - x^k_{i,j}) * z^k * mem_i],\ \forall k, \tag{6.10}$$

$$\alpha_{i,i'} * T_{SFC} = \sum_{j\in\mathcal{I}_i}\sum_{j\in\mathcal{I}_{i'}} r_{i,j,i',j'},\ \forall i, i', \tag{6.11}$$

$$x^k_{i,j},\ z^k \in \{0,1\}, pr_i \geq y_{i,j,i',j'} \geq 0, \tag{6.12}$$

$$\forall k, k^* \in \mathcal{S}, i, i' \in \mathcal{NF}, j \in \mathcal{I}_i, j' \in \mathcal{I}_{i'}.$$

Eq. (6.7) minimizes the TO of the SFC. Eq. (6.8) assigns each instance of an NF to exactly one server. Eq. (6.9) ensures that the CPU consumption of a server does not exceed its bound. Eq. (6.10) guarantees that the memory consumption of a server does not exceed its bound. From Eq. (6.11), if two NFs are dependent, the traffic between the instances of two NFs should equal to T_{SFC}. Eq. (6.12) indicates the integer constraints of the variables.

Heuristic Algorithm. The above problem is a nonlinear integer problem with a high computation complexity. To provide the high quality service, we need to place the SPG quickly. Therefore, we need a heuristic algorithm to efficiently solve the problem. The basic idea of our algorithm is to greedily minimize the transmission rate and transmission delay iteratively. Recall the goal of the above problem is to minimize the multiple of transmission rate and transmission delay, and the instances in the same server do not have transmission delay. Thus, our algorithm finds the NF instances with the high transmission rate and place them into the server with the highest CPU and memory resource among the rest of servers. This procedure iteratively process until all instances are placed into servers.

Forwarding Rule Generation. Once the traffic distribution algorithm determines the paths for traffic steering, forwarding (including packet duplication) rules for both software and hardware switches are generated as part of the output. HRC configures hardware switches through

an SDN controller (*i.e.,* Ryu [124]). HydraNF does not rely on a specific SDN technology, such as OpenFlow, and can operate on top of any programmable switches [125] with their corresponding controllers. For software switches on servers, an agent in HydraNF data plane engine is responsible for communicating with HRC and installing rules. This agent also reports the runtime state of HydraNF data plane and system load information to HRC which will be used for auto-scaling and other control & management plane operations.

6.5 HydraNF Data Plane

We present the data-plane design of HydraNF, focusing on the mirror and merge modules and their placement.

6.5.1 Mirror and Merge Modules

We show the main building blocks and data structures of the HydraNF data plane in Figure 6.7. The data-plane execution engine enforces SFC parallelism based on the forwarding and processing rules installed by HRC. We first present the design of data plane within a single server as an extension of software switches. We then present the multiple server case in § 6.5.2. Before presenting the details of the mirror and merge modules, we first describe the structure of two tables, *Flow Steering* table and *Packet State* table, that these modules operate on.

The Flow Steering table contains information for packet processing of SFC segments consisting of NFs residing within a given server. Each entry represents an SFC segment, *e.g.,* {A,{B,C}}, along with the corresponding NF operations denoted as OPS, and FID which shows the flow to which the SFC segment applies. HRC installs these entries in the software switch. The mirroring module uses the Flow Steering entries to steer packets among NFs, duplicating them if needed. For an example SFC segment {A,{B,C}}, mirroring module would duplicate packets processed by A, sending them to both B and C for parallel processing.

The Packet State table is primarily used by the merge module and contains four fields: 1) PID (packet ID, identification field in IP header), 2) PKTR (reference pointer to the memory address of the original packet), 3) BUF (packet buffer for saving the intermediate results), and 4) CNT (counter array for parallel SFC segments). The unique PID keeps track of duplicate packets that are processed by parallelizable NFs. The CNT records the number of NFs in each segment of a local hybrid SFC. For instance, CNT for {A, {B, C}} is {1, 2}. The count

decrements by one after a packet goes through an NF in the segment. HydraNF performs the merge operation when the count reaches zero.

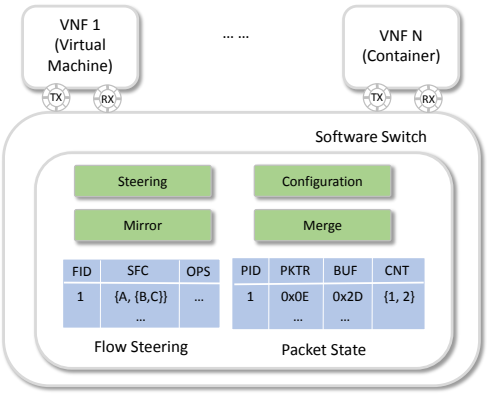


Figure 6.7: Building blocks and data structures of HydraNF data plane.

In the merge operation, we treat a data packet as a sequence of bits, namely a $\{0|1\}^*$ string. If an NF in a parallel segment adds L extra bits into packets, we insert a string of L zeros at the corresponding location in the outputs from other NFs before the merge. In a similar vein, if an NF removes L bits from packets, we delete the same bits from the outputs of other NFs. Moreover, if packets are dropped, a no-op packet with corresponding configured information is sent back to merge module.

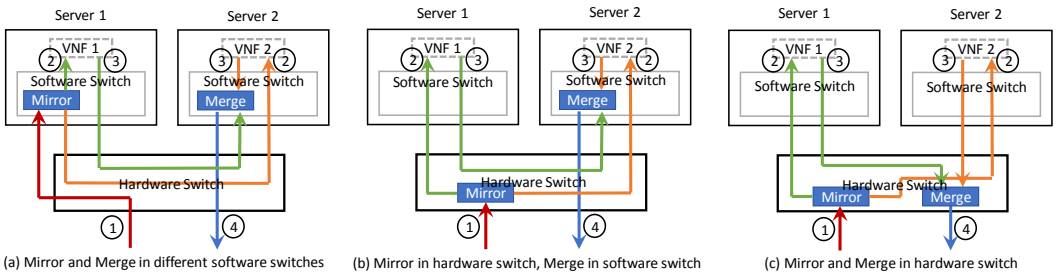


Figure 6.8: Various placements of the mirror and merge modules for the multi-server scenario. In the first step, the mirror module receives the packets. It then duplicates them to two NFs in the second step. In the third step, the NFs send back the packets to the merge module which generates the final output.

Assume P_O is the original packet and there are two NFs A and B in the chain. Assume that

P_A and P_B are their respective outputs. The packet output after passing two network functions will be $P_O \oplus P_A \oplus P_B$. Note that correctness is guaranteed as parallelizable NFs do not modify the same portion of a packet. Finally, we recalculate the checksum before steering/mirroring the packet to the next NF(s) – either residing within the same server or different servers. In order to avoid merge function to be bottleneck, multi-queue NIC with 5-tuple based load balance is enabled.

6.5.2 Placement of Mirror and Merge

When parallelizing packet processing for NFs spanning multiple machines, a natural question is where to place the mirror and merge functions. For SFC parallelism within a server, the mirror and merge functions are placed with software switches, as in ParaBox [1], or the merge function can run as a dedicated container in NFP [2]. However, when considering an SFC spanning multiple servers, if we naively place mirror and merge modules on one of the servers, we may not only waste bandwidth, but also potentially increase the SFC latency.

For example, consider an SFC: NAT $\rightarrow$ FW $\rightarrow$ IPS $\rightarrow$ WANX, and suppose we can parallelize the packet processing for NAT, FW and IPS, but not the WANX. Further, assume that NAT and FW are placed on server 1, while IPS and WANX are on server 2. If we place the mirror and merge functions on server 1, the mirror function will have to duplicate packets to IPS on server 2. After IPS examines the packets, it will have to send them back to server 1 for the merge, and then back to server 2 again for the last-hop WANX processing. As a result, placing the mirror and merge functions on server 1 ends up increasing latency instead of reducing it. Instead if we place mirror and merge intelligently, *i.e.,* mirror on server 1 and merge on server 2 then we can reduce the extra traversal of packets caused by placing both mirror and merge on same server.

Hence, when parallelizing SFC processing across multiple servers (with NF placement constraints), the location of the mirror and merge functions is crucial. Using two parallel NFs placed on two servers as an example, Figure 6.8 illustrates several design choices for the placement. (a) *mirror and merge in software switches (on servers) only*: This placement strategy can, to certain extent, avoid sending packets back-and-forth between the two servers, but there will still be two outgoing flows for server 1 and two incoming flows for server 2 (in contrast, processing two NFs sequentially generates only one incoming and outgoing flow for each server). (b) *mirror on hardware switches and merge on software switches*: This placement can reduce

the number of outgoing flows for server 1 from two to one; it still cannot improve the situations for server 2. This design choice is what is currently implemented in HydraNF. (c) *both mirror and merge on hardware switches*: This is the ideal case which achieves the same bandwidth utilization as the sequential chain.

Although it is feasible to place the mirror function on hardware switches, it is more challenging to design and implement the merge function even on programmable hardware switches. The reason is that the merge function requires relatively complex operations (*e.g., xor*) and needs extra memory to store the intermediate results.

6.6 Evaluation

In this section, we evaluate HydraNF through a prototype implementation, and conduct experiments for the following scenarios to demonstrate its efficacy and benefits.

• We first evaluate the SPG construction algorithms using both synthetic and practical SFCs to show their correctness and benefits.

• In realistic chains, we demonstrate HydraNF indeed reduces SFC latency in various setups (Figure 6.12).

• HydraNF improves packet processing performance in multi-server scenarios (Figures 6.13, 6.14, and 6.15).

• The overhead introduced by HydraNF is manageable (Figures 6.16, 6.17, and 6.18).

Experimental Setup. The experimental setup is shown in Figure 6.9. HydraNF prototype uses an Openflow-enabled switch with 48 10Gbps ports. We connect a cluster of six servers to the switch, four of which are equipped with two 10Gbps links, and the other two are equipped with four 10Gbps links. Each server uses an Intel(R) Xeon(R) CPU E5-2620 with 6 cores, for a total of 36 cores. On each server, one core is dedicated to HydraNF data plane. The HydraNF controller runs on a standalone machine that connects to each server and to the management port of the switch on a separate 1Gbps control network. Pktgen-dpdk [126] is used as the traffic generator to generate 64B Ethernet packets.

We choose the Berkeley Extensible Software Switch (BESS) [127] mainly due to its high performance and programmability. To optimize the bandwidth utilization for the multi-server scenario, we have offloaded the mirror module inside an OpenFlow switch. Due to the more complex operations in the merge module, it is challenging to implement it using OpenFlow

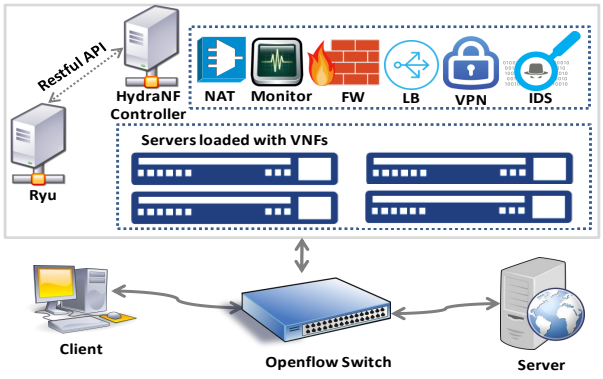


Figure 6.9: Experiment Setup

switches. As ongoing work, we are exploring the feasibility of offloading the merge module to P4 [128] switches.

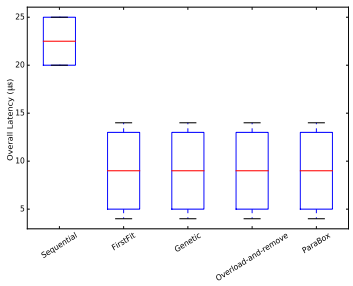


Figure 6.10: Overall latency in synthetic SFCs

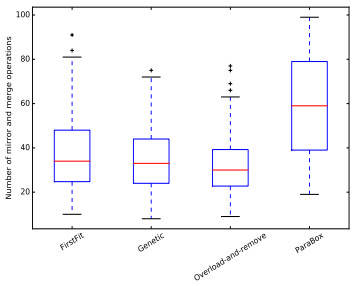


Figure 6.11: SPG benefit in synthetic SFCs

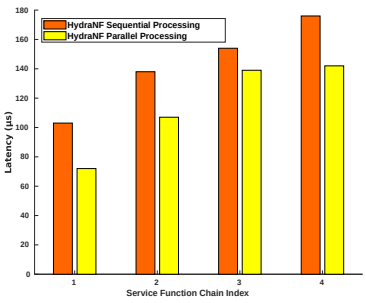


Figure 6.12: Performance of Different Service Function Chains with Different Complexity

Regarding the control plane, we have implemented HRC in Python using the Flask framework [129]. We store the table information as discussed in § 6.4 using pickleDB [130] which is a light-weight key-value store. HRC communicates with the OpenFlow controller Ryu [124] through its RESTful APIs. The local daemon implemented on each server also uses Flask framework.

NFs Used in Experiments. Each NF is running in either a Docker container or a KVM-based VM. We dedicate a CPU core to each container or VM. We use seven types of NFs: Layer 2 forwarder (L2FWD), NAT, FW, IDS, Monitor, Load Balancer, and VPN gateway. L2FWD

is used for NF benchmarking. For NAT and FW running in VM, we use product-level NFs with real operational rules from a carrier network. We also use open-source iptables running in containers as NAT and FW. We use BRO [131] as IDS, Nload [132] as Monitor, Linux Network Load Balancing [133] as Load Balancer, and OpenVPN [134] as VPN gateway in the experiments. Moreover, we create customized L2FWD, NAT, and FW to invoke BESS zero-copy API and OpenNetVM zero-copy API respectively for benchmarking purpose.

6.6.1 HydraNF Performance

Synthetic and Practical SFCs. We use both synthetic and practical SFCs in experiments. Pagination database [135] is used to construct synthetic SFCs. For each SFC, it is constructed with various NF dependencies (we take each tile in pagination problem as a chain in NF dependency graph). The processing cost of each simulated NF is set as 1. Each chain in NF dependency ranges from 2 to 13. We compare the overall processing delay of SFC sequential execution and parallel execution. Moreover, in parallel execution, we compare parallelism overhead between SPG construction algorithms and Parabox [1].

Figure 6.10 shows the overall processing latency of SFC sequential execution and parallel execution. Since the overall latency is determined by the critical path, and thus for any SFC input, critical paths calculated in parallel processing (*i.e.,* FirstFit, Genetic, Overload-and-remove, and ParaBox) are the same. In contrast, the overall latency of parallel processing is reduced.

Figure 6.11 presents SPG benefits compared to ParaBox which parallelizes NFs based on topological sorting (parallelizing NFs as much as possible). The number of mirror and merge operations in constructed SPG is less than that of ParaBox.

Index	Deployed Chain in One Machine	Hybrid Chain in One Machine	Deployed Chain in Two Machines	Hybrid Chain in Two Machines
1	{IDS, NAT, FW}_container	{(IDS, NAT, FW)}	{IDS ‖ NAT, FW}_container	{(IDS, ‖ NAT, FW)}
2	{IDS, NAT, FW}_vm	{(IDS, NAT, FW)}	{IDS ‖ NAT, FW}_vm	{(IDS, ‖ NAT, FW)}
3	{VPN, Monitor}_vm, {FW, LB}_container	{(VPN), (Monitor, FW), (LB)}	{VPN, Monitor}_vm ‖ {FW, LB}_container	{(VPN), (Monitor ‖ FW), (LB)}
4	{NAT, FW, IDS, LB}_vm	{(NAT, FW, IDS), (LB)}	{NAT, FW ‖ IDS, LB}_vm	{(NAT, FW ‖ IDS), (LB)}

Table 6.1: Service Function Chains Used in the Experiments

In experiments over synthetic SFCs, we did not consider processing delay of mirror and merge modules. We only calculate process latency and SPG benefits over simulated NFs with synthetic processing delay. Next, we conduct experiments on practical SFCs with real NFs in practical SFCs to further evaluate the performance benefits. Table 6.1 presents sequential and hybrid chains generated by Overload-and-remove Algorithm 6.4.1. NFs in () mean that they

can be processed in parallel, and NFs separated by || mean that they are on different servers. HydraNF can reduce the SFC latency by up to 31.7% as shown in Figure 6.12. Moreover, since HydraNF provides unified virtual interfaces to both Docker containers and KVM-based VMs, mixed-technique NFs can be integrated into a chain and processed by HydraNF.

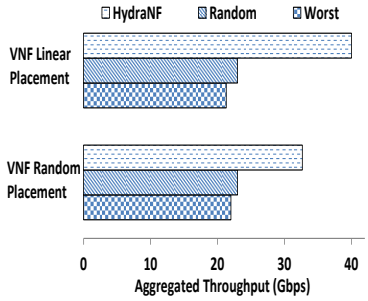


Figure 6.13: Throughput with different placement solutions

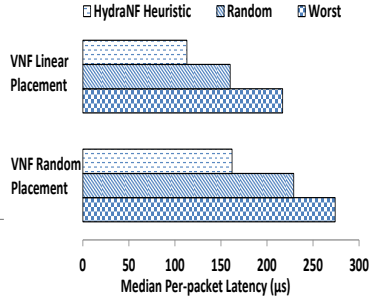


Figure 6.14: Latency with different placement solutions

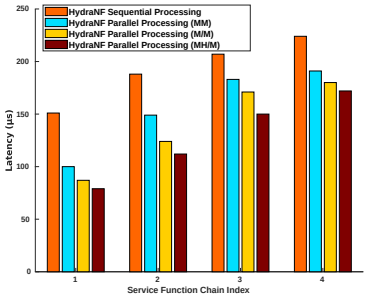


Figure 6.15: Performance of Different Core Module Placements

Impact of Different Module Placements. Figure 6.15 shows latency of different core module placements. $HydraNFParallelProcessing(MM)$ means that mirror and merge functions are placed on the same server; $HydraNFParallelProcessing(M/M)$ indicates that mirror and merge functions are running on different physical servers. $HydraNFParallelProcessing(MH/M)$ means that mirror function is implemented in a physical switch. Figure 6.15 shows latency gains achieved by NF parallelism in various SFCs. We observe that HydraNF controller can correctly distribute forwarding rules into hardware and software to enable parallelism, and NF parallelism among multiple servers can still achieve latency gains. Among three system implementation choices for NF parallelism, putting mirror functions into physical switch achieves the best results in most cases.

However, it is not always the case that putting mirror functions into physical switch can achieve the best results. For example, in the chain that VPN and Monitor are in the same server, while FW and LB are on the other server. Monitor and FW can be processed in parallel. However, putting mirror function into a physical switch is not as good as putting it into a physical server in such a scenario because traffic processed by VPN needs to be mirrored into Monitor and FW. Thus, the traffic needs to be detoured back to physical switch for mirroring.

Impact of Different NF Placements. We investigate the impact of NF placement spanning

over multiple servers in Figure 6.13. Three customized zero-copy NFs (L2FWD, NAT, FW) are loaded into each of four servers in linear placement scenario, while these 12 NF instances are placed randomly in NF random placement. We configure generated traffic into four SFCs (L2FWD→NAT→FW, L2FWD→NAT, L2FWD→FW, NAT→FW). 500 flows are generated to follow each chain. We present the results of HydraNF, random distribution approach, and the worst case. The aggregated throughput is shown in Figure 6.13, and latency result is shown in Figure 6.14. HydraNF achieves 1.74x-1.87x throughput improvement in NF linear placement while 1.42x-1.48x throughput improvement in NF random placement. Similarly, we can also observe great latency benefits of HydraNF compared with others.

Furthermore, in order to verify the correctness of HydraNF, we always send traffic through traditional SFCs and replay the same traffic in HydraNF. We then compare the packets at the receiver side and network state maintained in NFs using log information, in order to confirm that the HydraNF generates the same output as the sequential ones.

6.6.2 HydraNF Overhead

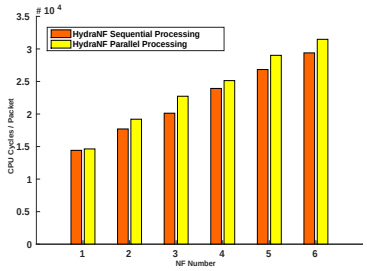


Figure 6.16: CPU Overhead in Sequential Processing and Parallel Processing

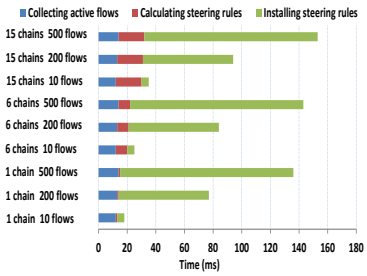


Figure 6.17: Break Down of Controller Processing Time

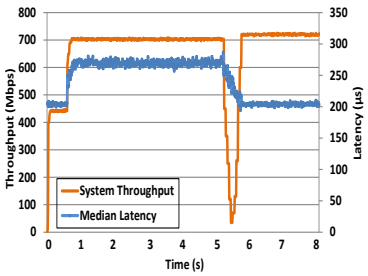


Figure 6.18: Overhead during NF Scaling

We evaluate HydraNF overhead in this section. We first show the overhead of data plane modules; then controller overhead is showed; and finally the performance overhead during NF scaling is demonstrated.

To understand the overhead introduced by HydraNF, especially the mirror and merge modules, we measure the CPU cycles to process each packet in benchmarking experiments which uses L2FWD as NF. Figure 6.16 shows the CPU cycles per packet increasing with the SFC length for both sequential and hybrid chains. However, CPU cycles per packet in HydraNF

parallel processing is up to 7% more than the sequential SFCs.

HydraNF controller calculates appropriate rules to install into hardware and software switches. Controller overhead is segmented into three parts: 1) collecting active flows; 2) calculating steering rules; 3) installing steering rules. We create 9 scenarios with different complexity, and manually trigger topology update which forces the controller to recalculate traffic distribution rules. We measure the overhead of each segment, as shown in Figure 6.17. We observe that the overhead of calculating steering rule depends on the number of chains in SFC configuration, while the overhead of installing steering rules depends on the number of active flows in existing network. Since controller only recalculates steering rules when topology is changed, the relative costly operation is affordable. We have been exploring more efficient approaches without requiring NF modifications.

During NF scaling, we want to further investigate the overhead and observe how throughput and latency change. We deploy two machines, one of which only runs an IDS, while the other one runs an IDS and a L2FWD. Then we generate traffic going through IDS $\rightarrow$ L2FWD. As shown in Figure 6.18, the throughput of the system increases immediately and reaches the bottleneck of IDS processing. After that, HydraNF controller gets notification from local daemon running on the NF server and utilizes candidate SFC calculated by traffic distribution algorithm. The throughput of the system goes up because two IDS instances are utilized, but median latency increases as well because certain traffic is detoured into a sub-optimal SFC which spans over two servers. At the fifth second, we spin up another L2FWD in the server running only one IDS, and trigger the controller to recalculate the traffic distribution rules. In our current implementation, after controller recalculates traffic distribution rules, it will refresh existing rules. This will cause temporary system unavailability, and thus the system throughput drops down rapidly, but recovers quickly. This is a trade-off of NF performing scaling without NF modification, and we have been exploring an effective mechanism of conquering it. Now both paths (IDS $\rightarrow$ L2FWD on servers) are optimized, so latency drops.

6.7 Related Work

SFC Parallelism. Although using parallelism for accelerating an SFC has been explored, *e.g.,* in ParaBox [1] and NFP [2], these solutions are limited in terms of real-world applicability. We differentiate HydraNF with ParaBox and NFP in Table 6.2 on key features desired for SFC

parallelism.

Features	ParaBox	NFP	HydraNF
NF Level Parallelism	✓	✓	✓
Optimized SPG	✗	✗	✓
Inter-Server Traffic Distribution	✗	✗	✓
Incremental Deployment	✓	✗	✓

Table 6.2: Comparison of HydraNF with ParaBox [1] and NFP [2] on the desirable features of parallel packet processing for NFs.

First, these solutions only work in deployments where all parallelizable NFs of a chain are on the same server. While running NFs on the same server reduces bandwidth consumption [118], in real-world networks, it is difficult to allocate required resources using a single server for all NFs in a chain. Many existing NFs are resource intensive when handling large volumes of traffic [37], and a server can often support a very limited number of concurrent NFs [136]. Dynamics of network services and elastic scalability offered by NFV also prevent single server deployments.

Second, during SFC analysis, the existing two works try to parallelize a sequential SFC as much as possible. However, maximal parallelism may not be beneficial since it could waste some resources. Hence, HydraNF constructs an SPG to reduce parallelism overhead.

Third, enterprise-grade NFs are constructed with large amount of engineering efforts, and some companies even design customized operating systems to support NF functionalities [137, 138]. Hence, HydraNF enables parallelism without NF modification.

SFC Optimization. The SFC optimization has attracted a flurry of research interests in recent years. NetVM [36] is the first to utilize DPDK to provide zero-copy packet delivery across a chain of vNFs running as VMs on the same physical server. Inspired by Click-Router [139], BESS [127, 140] develops a modular software switch tailored for NFV by utilizing DPDK to achieve more than 10 Gbps processing speed using a single CPU core. Flurry [30], NFVnice [29], and E2 [118] exploit flow-level traffic parallelism for smart CPU core scheduling to improve scalability of NFV, whereas OpenBox [141] decomposes vNFs into re-usable modules to further speed up the packet processing pipeline. Metron [119] synthesizes vNFs in an SFC and executes the SFC in a single core with hardware offloading to achieve (near) line-speed packet processing. The statefulness of vNFs is a major hurdle in the SFC optimization. Many

studies have been carried out to address these issues, e.g. [142–148]. For example, CHC [148] leverages an external state store coupled with state management algorithms and metadata maintenance for correct operation even under a range of failures. S6 [147] supports elastic scaling of NFs without compromising performance. In contrast, HydraNF focuses on parallelism at both NF and traffic level to optimize SFC. In addition, unlike [2, 118, 119, 141, 149], HydraNF can accommodate to both open-source and proprietary vNFs with no NF modifications.

SFC Provisioning. The SFC provisioning has also attracted a plethora of research studies, as partially summarized in [150]. For example, Steering [28] proposes a network function placement algorithm in a given network topology with the goal of minimizing the delay or distance to be traversed by all subscriber's traffic. Sun *et al.* [151] model the placement of SFC in SDN networks as an integer linear programming problem to minimize the overall energy consumption. Li *et al.* [152] construct a deep learning model to provision on-demand SFCs. However, these works do not consider SFC parallelism optimization as HydraNF does.

6.8 Discussion

In this section, we discuss how to extend HydraNF and highlight several open issues.

Implementing Merge in Programmable Switches. We have currently implemented mirror function in hardware switches, but merge function in software switches. As discussed in § 6.5.2, placing both functions in hardware switches is the ideal solution which introduces almost no overhead in terms of bandwidth utilization. Even for OpenFlow switches, it is challenging to implement the merge functions, as the required *xor* and *or* operations are not yet supported by the OpenFlow specification 1.3. However, it is possible to implement the merge function with P4 [128] switches. P4 exposes a standard set of primitive actions, including bit_or and bit_xor [125]. Given this support, we are currently implementing the merge function of HydraNF on P4 switches.

NF Scaling In/Out. To support NF scaling, HRC needs to integrate with the NF application controller/orchestrator. One possible approach is to inform HRC about the creation/deletion of NF instances and their locations. Based on the notification, the mirror function would be aware of the new instances and start mirroring traffic to them, or stop duplicating packets for the removed NF instances. This will require adding the support for NF instance tracking inside the Flow Steering table as well as modifications to the merge function. Furthermore, we plan to

support asymmetric NF scaling, *i.e.,* HydraNF can dynamically scale in/out any hop of a chain, instead of scaling the whole chain.

NF Decomposition. Recent proposals virtualize and decompose NF at different granularity. For example, EdgePlex [153] assigns a single VM for the control plane and a dedicated VM for the data plane of each customer provisioned on a provider edge router. HydraNF should be able to handle this type of decoupling of control and data planes due its ability to work at flow-level. OpenBox [141] performs fine-grained decomposition at the software module level of NFs. In this case, the components of a service chain will be NF modules, instead of NFs. Despite this, it should still be feasible for HydraNF to parallelize packet processing for these NF modules in a chain.

The Gain of Parallelization depends on the dominating NFs in a chain in terms of processing latency. For example, if a traffic shaper adds 100 μs extra latency to a service chain with two other NFs having 10 μs processing latency each, employing parallelize packet processing will reduce the latency from $\sim$ 120 μs to $\sim$ 100 μs (*i.e.,* merely a 16.7% reduction). However, if the traffic shaper adds only 20 μs, HydraNF can reduce the latency by $\sim$ 50% (from 40 μs to 20 μs). In cases where parallelization gain is not high enough, it would be prudent to revert back to the sequential chain.

Other Limitations. We have not considered other types of NFs, *e.g.,* mobility NFs [154] which include virtual Packet Data Network Gateway, Serving Gateway, Mobility Management Entity *etc.*, for parallel data processing. We have also not investigated the parallelization for layer-3 virtual routers. These NFs either create various tunnels or determine the next hop, creating order dependency with other NFs.

6.9 Summary

In this project, we have designed, implemented, and evaluated HydraNF, a novel mechanism to utilize parallelism for SFC acceleration. In contrast to existing approaches, HydraNF performs both NF and traffic level parallelism without NF modification. HydraNF supports not only NFs within a multi-core server, but also those spanning multiple machines by provisioning SPG in an optimized way. The data plane of HydraNF runs as an extension of the BESS virtual switch to achieve high performance and programmability. Our evaluation through synthetic and practical SFCs demonstrates that HydraNF can improve SFC performance with manageable overhead.

Chapter 7

Conclusion, Lessons Learned & Thoughts for the Future

In this dissertation, we have argued that by following the blueprint of joint design, networks equipped with VNFs can be made more effective and efficient.

7.1 Summary of Contributions

Our main contributions in this dissertation are as follows:

In **Durga** (Chapter 3), we designed, implemented, and evaluated a novel SD-WAN solution for fast failover with minimal application performance degradation under WAN link failures. We mitigated the impacts of WAN failures on application performance by joint architectural innovation at both VNFs running on SD-WAN gateways and connected end systems. Durga combines an innovative *WAN-aware* MPTCP mechanism which enables applications to generate multiple MPTCP flows even with a single physical interface. This is further augmented with an MPTCP proxy to accommodate end systems without native MPTCP support. Through extensive evaluation in emulated testbed and real-world deployment, we demonstrated the superior performance of Durga over existing SD-WAN solutions.

In **SAMPO** (Chapter 4), we facilitated VNFs aware of MPTCP for the benefits of network performance and the quality of VNFs. Instead of designing a customized solution in every VNF to conquer this common challenge (facilitating VNFs aware of MPTCP), we implemented SAMPO as an online service to be readily integrated into VNFs. SAMPO is a first step towards

this goal by solving online MPTCP subflow association problem. Both our theoretical analysis and experimental results show that SAMPO can detect and associate MPTCP subflows with high accuracy even when only a very small portion of each MPTCP subflow is available.

In **Network-Assisted Raft** (Chapter 5), following the same principle, we implemented consensus as a service in software defined network. We use Raft to illustrate new failure scenarios in the design of distributed SDN controllers. We discussed *PrOG* to circumvent these issues. Also, by leveraging programmable devices, we proposed to partially offload certain Raft functionality to P4 switches for reducing Raft processing latency, while not sacrificing scalability. Our evaluation results show the effectiveness of *PrOG* and Raft-aware P4 switches in improving the availability of leadership in Raft used by critical applications like SDN controller clusters.

In **HydraNF** (Chapter 6), we re-considered VNFs deployed in a network from the perspective of network administrators. HydraNF explores parallelism in service function chains composing a sequence of VNFs that are typically traversed in-order by data flows. The evaluation through synthetic and practical SFCs demonstrates that HydraNF can improve SFC performance with manageable overhead.

7.2 Lessons Learned & Thoughts for the Future

We now discuss a few broad lessons learned over the course of this dissertation, and what they suggest about future VNF development and deployment.

Jointly designing VNFs for enhancing networks involves interactions among VNFs and hence requires unified programming interfaces and platform.

In 2012, the European Telecommunications Standards Institute (ETSI) issued a proposal named as Network Functions Virtualization (NFV) [155]. The incentives of proposing NFV is that modern telecoms networks contain an ever increasing variety of proprietary hardware, and thus the launch of new services often demands network reconfiguration and on-site installation of new equipment which in turn requires additional floor space, power, and trained maintenance staff. NFV accelerates and requires greater flexibility and dynamism than hardware-based appliances allow. Hard-wired network with single functions boxes are tedious to maintain, slow to evolve, and prevent service providers from offering dynamic services similar as the motivation for joint design blueprint proposed in this dissertation.

Along these lines, several projects were designed to conquer the open challenges in the

joint design blueprint. For example, VNF orchestraters [118, 156] were proposed for automatically instantiating or closing VNF instances as traffic load changes. The SFC working group in IETF [25] is actively investigating how to best implement routing through multi-middlebox topologies and enforce policies about which traffic receives processing by which VNFs. Many research studies have been carried out to address statefulness of VNFs, e.g. [142–147]. In terms of NFV behavior modeling, synthesis, testing as well as policy analysis and traffic steering, various novel techniques have been proposed, see, e.g., [26–28, 157–163]. The NFV placement problem has also attracted a plethora of research studies, mostly employing mathematical optimization techniques [150]. While we can see the benefits brought by joint design, how to design unified programming interfaces for supporting interactions among VNFs and implement a general platform for the integration of VNFs is required.

Jointly designing can be extended to offload certain functions down to hardware programmable switches.

Several recent projects investigate offloading consensus algorithms to either switches [22] or FPGA devices [23]. NetPaxos [22] proposes to implement the Paxos consensus algorithm in network by leveraging programmable switches. Besides the Paxos roles implemented on servers, NetPaxos requires one switch serving as a Paxos coordinator and several others as Paxos acceptors. NetPaxos can be implemented using P4 [5], a domain specific language that allows the programming of packet forwarding planes. However, Paxos consensus algorithm is very difficult to understand and implement due to its notoriously opaque explanation and lack of details for building practical systems [4]. Thus, offloading such a complex consensus algorithm to network is error-prone. István *et al.* [23] takes the efforts of implementing the entire ZAB consensus algorithm [24] on FPGA devices using a low-level language which is difficult to program. Moreover, this hardware-based solution may not be scalable as it requires the storage of potentially large amounts of consensus states, logic, and even the application data. Even though offloading VNFs to networks is a promising area to explore, it would be demanded to formally validate the correctness of such a decoupled architecture. Moreover, it is also interesting to compare the solution implemented in real P4 switches with other existing FPGA-based or RDMA-based solutions.

Decomposing VNFs opens the door to pursue further performance enhancement.

Recent proposals virtualize and decompose NF at different granularity. For example, EdgePlex [153] assigns a single VM for the control plane and a dedicated VM for the data plane

of each customer provisioned on a provider edge router. OpenBox [141] performs fine-grained decomposition at the software module level of NFs. In this case, the components of a service chain will be NF modules, instead of VNFs. It would be more challenging to parallelize packet processing at a finer grained level in a chain.

Enhancing networks equipped with VNFs in the context of 5G technologies.

In the emerging 5G technologies – besides innovations in radio technologies such as 5G new radio [164, 165], NFV will be a key enabling technology [166–171] underpinning the envisioned 5G "Cloud RANs", MECs and packet core networks for support of *network slicing* and diverse services ranging from enhanced mobile broadband to massive machine type communications and ultra-reliable low latency communications. For example, upon a request for a service (e.g., from a mobile user or a machine, say, an autonomous vehicle or an industrial controller), a SFC will be dynamically constructed using a series of VNFs such as firewalls, mobility managers, network address translators, traffic shapers and so forth that are deployed on demand at appropriate locations within a (dynamic) network slice to meet the desired service requirements. It would be challenging to jointly design VNFs in the context of 5G technology, and leverage various 5G VNFs and SFCs to support the development of 5G end-to-end facilities, network slicing, 5G services and vertical trials. Through end-to-end evaluations and 5G service trials, NFV platforms can be further refined and expanded.

References

[1] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *Proc. SOSR*, 2017.

[2] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling Network Function Parallelism in NFV. In *Proc. SIGCOMM*, 2017.

[3] National Research Council. *Realizing the Information Future: The Internet and Beyond*. The National Academies Press, Washington, DC, 1994.

[4] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proc. USENIX ATC*, 2014.

[5] Pat Bosshart et al. P4: Programming Protocol-Independent Packet Processors. In *CCR*, 2014.

[6] Aryaka, 2017 State of the WAN Report. `https://info.aryaka.com/state-of-wan-2017.html`, 2017.

[7] Comparison of the SD-WAN vendor solutions. `https://www.netmanias.com/en/post/oneshot/12481/sd-wan-sdn-nfv/comparison-of-the-sd-wan-vendor-solutions`, 2017.

[8] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. Tcp extensions for multipath operation with multiple addresses. RFC 6824, IETF, January 2013.

[9] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. Exploring mobile/wifi handover with multipath tcp. In *Proceedings of the 2012*

ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design, CellNet 12, pages 31–36, New York, NY, USA, 2012. ACM.

[10] Costin Raiciu, Christoph Paasch, Sbastien Barr, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *USENIX Symposium of Networked Systems Design and Implementation, San Jose, CA*, NSDI 12, 2012.

[11] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI 11, pages 99–112, Berkeley, CA, USA, 2011. USENIX Association.

[12] M. Sandri, A. Silva, L.A. Rocha, and F.L. Verdi. On the benefits of using multipath tcp and openflow in shared bottlenecks. In *Advanced Information Networking and Applications*, AINA 15, pages 9–16, March 2015.

[13] Nick McKeown et al. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 2008.

[14] Pankaj Berde et al. ONOS: Towards an Open, Distributed SDN OS. In *Proc. HotSDN*, 2014.

[15] OpenDaylight: Open Source SDN Platform. `https://www.opendaylight.org/`, 2017.

[16] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 1998.

[17] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 2001.

[18] Sushant Jain et al. B4: Experience with a Globally-deployed Software Defined WAN. *Proc. SIGCOMM CCR*, 2013.

[19] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: an Engineering Perspective. In *Proc. PODC*, 2007.

[20] Avinash Lakshman and Prashant Malik. Cassandra: a Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 2010.

[21] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. OSDI*, 2006.

[22] Huynh Tu Dang et al. Netpaxos: Consensus at network speed. In *Proc. SOSR*, 2015.

[23] Zsolt István et al. Consensus in a box: Inexpensive coordination in hardware. In *Proc. NSDI*, 2016.

[24] Patrick Hunt et al. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. ATC*, 2010.

[25] Joel M. Halpern and Carlos Pignataro. Service Function Chaining (SFC) Architecture, 2015.

[26] Zafar Ayyub Qazi et al. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. SIGCOMM*, 2013.

[27] Seyed Kaveh Fayazbakhsh et al. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proc. NSDI*, 2014.

[28] Ying Zhang et al. StEERING: A software-defined networking for inline service chaining. In *Proc. ICNP*, 2013.

[29] Sameer G Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proc. SIGCOMM*, 2017.

[30] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. Flurries: Countless fine-grained nfs for flexible per-flow customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 3–17, New York, NY, USA, 2016. ACM.

[31] Pamela Zave, Ronaldo A. Ferreira, Xuan Kelvin Zou, Masaharu Morimoto, and Jennifer Rexford. Dynamic Service Chaining with Dysco. In *Proc. SIGCOMM*, 2017.

[32] Surendra Kumar et al. Service Function Chaining Use Cases In Data Centers. Technical report, IETF, 2016.

[33] Thomas Nadeau and Paul Quinn. Problem Statement for Service Function Chaining. RFC 7498, 2015.

[34] Jeffrey Napper et al. Service Function Chaining Use Cases in Mobile Networks. Technical report, IETF, 2016.

[35] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 2015.

[36] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. NSDI*, 2014.

[37] Virtual Network Functions (VNFs). `https://www.business.att.com/content/productbrochures/network-functions-on-demand-vnf-brief.pdf`, 2016.

[38] MultiPath TCP - Linux Kernel implementation. `https://www.multipath-tcp.org/`, 2018.

[39] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 99–112, Berkeley, CA, USA, 2011. USENIX Association.

[40] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412, San Jose, CA, 2012. USENIX Association.

[41] Costin Raiciu, Dragos Niculescu, Marcelo Bagnulo, and Mark James Handley. Opportunistic mobility with multipath tcp. In *Proceedings of the Sixth International Workshop on MobiArch*, MobiArch '11, pages 7–12, New York, NY, USA, 2011. ACM.

[42] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. Exploring mobile/wifi handover with multipath tcp. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, CellNet '12, pages 31–36, New York, NY, USA, 2012. ACM.

[43] Andrei Croitoru, Dragos Niculescu, and Costin Raiciu. Towards wifi mobility without fast handover. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 219–234, Oakland, CA, 2015. USENIX Association.

[44] Gregory Detal, Christoph Paasch, and Olivier Bonaventure. Multipath in the middle(box). In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '13, pages 1–6, New York, NY, USA, 2013. ACM.

[45] MPTCP Proxy: OpenMPTCProuter. `https://www.openmptcprouter.com/`, 2018.

[46] MPTCP Proxy: Over The Box. `https://github.com/ovh/overthebox`, 2018.

[47] Open vSwitch. `http://openvswitch.org/`, 2017.

[48] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks). RFC 7348, August 2014.

[49] J. Tourrilhes, P. Sharma, S. Banerjee, and J. Pettit. Sdn and openflow evolution: A standards perspective. *Computer*, 47(11):22–29, Nov. 2014.

[50] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, June 2010.

[51] HPsockd. `https://tracker.debian.org/pkg/hpsockd`, 2002.

[52] Dante. `https://www.inet.no/dante/`, 2006.

[53] Transparent Proxy Howto. `https://www.tldp.org/HOWTO/TransparentProxy.html`, 2018.

[54] tproxy. `https://www.kernel.org/doc/Documentation/networking/tproxy.txt`, 2018.

[55] Franck Le, Erich Nahum, Vasilis Pappas, Maroun Touma, and Dinesh Verma. Experiences deploying a transparent split tcp middlebox and the implications for nfv. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 31–36, New York, NY, USA, 2015. ACM.

[56] S. Alexander and R. Droms. DHCP Options and BOOTP Vendor Extensions. RFC 2132, March 1997.

[57] ISC DHCP. `https://www.isc.org/downloads/dhcp/`, 2006.

[58] Dhcpy6d. `https://dhcpy6d.ifw-dresden.de/`, 2006.

[59] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862, September 2007.

[60] Netfilter. `https://netfilter.org/documentation/`, 2018.

[61] Sungeun Kim and John A. Copeland. Tcp for seamless vertical handoff in hybrid mobile data networks. In *GLOBECOM*, 2003.

[62] B. Oh and J. Lee. Feedback-based path failure detection and buffer blocking protection for mptcp. *IEEE/ACM Transactions on Networking*, 24(6):3450–3461, December 2016.

[63] Feng Wang, Zhuoqing Morley Mao, Jia Wang, Lixin Gao, and Randy Bush. A measurement study on the impact of routing events on end-to-end internet path performance. *SIGCOMM Comput. Commun. Rev.*, 36(4):375–386, August 2006.

[64] Radu Carpa, Marcos Dias de Assunão, Olivier Glück, Laurent Lefèvre, and Jean-Christophe Mignot. Evaluating the impact of sdn-induced frequent route changes on tcp flows. *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–9, 2017.

[65] Ulka Ranadive and Deep Medhi. Some observations on the effect of route fluctuation and network link failure on tcp. In *ICCCN*, 2001.

[66] Shahram Jamali. Enhanced fast tcp by solving rerouting problem. *International Journal of Integrated Engineering*, 9(3), 2017.

[67] T. Goff, J. Moronski, D. S. Phatak, and V. Gupta. Freeze-tcp: a true end-to-end tcp enhancement mechanism for mobile environments. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 3, pages 1537–1545 vol.3, March 2000.

[68] Xiaolan Liu, Danfeng Shan, Ran Shu, and Tong Zhang. MPTCP Tunnel. *Wirel. Commun. Mob. Comput.*, 2018, March 2018.

[69] MPTCP Tunnel. `https://github.com/dfshan/mptcp-tunnel`, 2017.

[70] TCP over TCP. `http://sites.inka.de/bigred/devel/tcp-tcp.html`, 2018.

[71] tcp_retries2 parameter. `https://access.redhat.com/solutions/726753`, 2018.

[72] Linux Traffic Control. `https://linux.die.net/man/8/tc`, 2018.

[73] A. Zimmermann and A. Hannemann. Making TCP More Robust to Long Connectivity Disruptions (TCP-LCD). RFC 6069, December 2010.

[74] M. Brent Reynolds. Mitigating tcp degradation over intermittent link failures using intermediate buffers. *Air Force Institute Of Technology*, 2017.

[75] Purvang Dalal, Nikhil Kothari, and Kankar S. Dasgupta. Improving TCP performance over wireless network with frequent disconnections. *CoRR*, abs/1112.2046, 2011, 1112.2046.

[76] Jean Tourrilhes. Fragment adaptive reduction: coping with various interferers in radio unlicensed bands. In *ICC*, 2001.

[77] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 147–159, New York, NY, USA, 2017. ACM.

[78] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli. Daps: Intelligent delay-aware packet scheduling for multipath transport. In *2014 IEEE International Conference on Communications (ICC)*, pages 1222–1227, June 2014.

[79] GENI Network. `http://www.geni.net/`, 2018.

[80] N. M. Sahri and Koji Okamura. Fast failover mechanism for software defined networking: Openflow based. In *Proceedings of The Ninth International Conference on Future Internet Technologies*, CFI '14, pages 16:1–16:2, New York, NY, USA, 2014. ACM.

[81] Andrea Sgambelluri, Alessio Giorgetti, Filippo Cugini, Francesco Paolucci, and Piero Castoldi. Openflow-based segment protection in ethernet networks. *IEEE/OSA Journal of Optical Communications and Networking*, 5:1066–1075, 2013.

[82] Mptcp - linux kernel implementation. `http://www.multipath-tcp.org/`.

[83] T. Zseby, M. Molina, N. Duffield, S. Niccolini, and F. Raspall. Sampling and filtering techniques for ip packet selection. RFC 5475, IETF, March 2013.

[84] Gianluca Iannaccone, Chen-nee Chuah, Richard Mortier, Supratik Bhattacharyya, and Christophe Diot. Analysis of link failures in an ip backbone. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, IMW 02, pages 237–242, New York, NY, USA, 2002. ACM.

[85] Dan Pei, X. Zhao, D. Massey, and Lixia Zhang. A study of bgp path vector route looping behavior. In *Distributed Computing Systems*, pages 720–729, 2004.

[86] L. Zhang V. Jacobson, R. Braden. TCP Extension for High-Speed Paths. RFC 1185, October 1990.

[87] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath tcp schedulers. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*, CSWS, pages 27–32, New York, NY, USA, 2014. ACM.

[88] Mptcp in apple siri. `https://support.apple.com/en-us/HT201373`.

[89] Philippe Flajolet, Danile Gardy, and Los Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207 – 229, 1992.

[90] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 15, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[91] Andrei Croitoru, Dragos Niculescu, and Costin Raiciu. Towards wifi mobility without fast handover. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 15, pages 219–234, Oakland, CA, 2015. USENIX Association.

[92] Felicián Németh, Balázs Sonkoly, Levente Csikor, and András Gulyás. A large-scale multipath playground for experimenters and early adopters. In *Proceedings of the ACM SIGCOMM Conference on SIGCOMM*, pages 481–482, New York, NY, USA, 2013. ACM.

[93] Yung-Chih Chen, Yeon-sup Lim, Richard J. Gibbens, Erich M. Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of multipath tcp performance over wireless networks. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC, pages 455–468, New York, NY, USA, 2013. ACM.

[94] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM 11, pages 266–277, New York, NY, USA, 2011. ACM.

[95] Bonaventure. Multipath tcp : An annotated bibliograph. 2015.

[96] Benjamin Hesmans and Olivier Bonaventure. Tracing multipath tcp connections. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 361–362, New York, NY, USA, 2014. ACM.

[97] M. Zubair Shafiq, Franck Le, Mudhakar Srivatsa, and Alex X. Liu. Cross-path inference attacks on multipath tcp. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 15:1–15:7, New York, NY, USA, 2013. ACM.

[98] Eric A. Brewer. Towards robust distributed systems. 2000.

[99] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[100] Aurojit Panda, Colin Scott, Ali Ghodsi, Teemu Koponen, and Scott Shenker. Cap for networks. 2013.

[101] Eman Ramadan, Hesham Mekky, Braulio Dumba, and Zhi-Li Zhang. Adaptive resilient routing via preorders in sdn. In *Proc. DCC*, 2016.

[102] Diego Ongaro. Logcabin: A distributed storage using raft. `https://github.com/logcabin`, 2016.

[103] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 1998.

[104] 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. OSDI*, 2010.

[105] Eman Ramadan, Hesham Mekky, Cheng Jin, Braulio Dumba, and Zhi-Li Zhang. Provably resilient network fabric with bounded latency. In *Under Submission*, 2017.

[106] Docker Containerization Platform. `https://www.docker.com/`, 2017.

[107] Jehan-Francois Paris and Darrell D. E. Long. Pirogue, a lighter dynamic version of the raft distributed consensus algorithm. In *Proc. IPCCC*, 2015.

[108] P4 behavioral simulator. `https://github.com/p4lang/p4factory`, 2017.

[109] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proc. SOSR*, 2015.

[110] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating sdn application failures with legosdn. In *Proc. HotNets*, 2014.

[111] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *Proc. SIGCOMM*, 2014.

[112] L. Schiff, S. Schmid, and M. Canini. Ground Control to Major Faults: Towards a Fault Tolerant and Adaptive SDN Control Network. In *Proc. DSN-W on IFIP*, 2016.

[113] Kevin C. Webb, Bhanu C. Vattikonda, Kenneth Yocum, and Alex C. Snoeren. Scalable coordination of a tightly-coupled service in the wide area. In *Proc. SOSP TRIOS*, 2013.

[114] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. In-band synchronization for distributed sdn control planes. *SIGCOMM CCR*, 2016.

[115] Aditya Akella and Arvind Krishnamurthy. A highly available software defined fabric. In *Proc. HotNets*, 2014.

[116] Takayuki Dan Kimura. Hyperflow: A uniform visual language for different levels of programming. In *Proc. CSC*, 1993.

[117] Abubakar Siddique Muqaddas, Andrea Bianco, and Paolo Giaccone. Inter-controller traffic in onos clusters for sdn networks. 2016.

[118] Shoumik Palkar et al. E2: A Framework for NFV Applications. In *Proc. SOSP*, 2015.

[119] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 171–186, Renton, WA, 2018. USENIX Association.

[120] Hsu-Hao Yang and Yen-Liang Chen. Finding the critical path in an activity network with time-switch constraints. *European Journal of Operational Research*, 2000.

[121] Bin Packing Problem. `https://en.wikipedia.org/wiki/Bin_packing_problem`, 2019.

[122] György Dósa and Jiri Sgall. First Fit bin packing: A tight analysis. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, 2013.

[123] Aristide Grange, Imed Kacem, and Sbastien Martin. Algorithms for the bin packing problem with overlapping items. *Computers & Industrial Engineering*, 2018.

[124] RYU SDN Framework. https://osrg.github.io/ryu/, 2017.

[125] The P4 Language Specification, version 1.0.3. https://p4lang.github.io/p4-spec/p4-14/v1.0.3/tex/p4.pdf, 2016.

[126] Pktgen DPDK. http://pktgen-dpdk.readthedocs.io/en/latest/, 2017.

[127] Berkeley Extensible Software Switch. http://span.cs.berkeley.edu/bess.html, 2017.

[128] Pat Bosshart et al. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, 2014.

[129] Flask Python Framework. https://www.fullstackpython.com/flask.html, 2017.

[130] PickleDB: Python Key Value Store. https://pythonhosted.org/pickleDB/, 2017.

[131] BRO: Network Intrusion Detection System. https://www.bro.org/, 2017.

[132] Network Monitor. https://linux.die.net/man/1/nload, 2017.

[133] Linux Network Load Balancing. http://lnlb.sourceforge.net/, 2017.

[134] OpenVPN: VPN Gateway. https://openvpn.net/, 2017.

[135] Pagination Dataset. https://github.com/pagination-problem/pagination, 2019.

[136] AT&T FlexWare. https://www.business.att.com/solutions/Service/network-services/sdn-nfv/virtual-network-functions, 2017.

[137] Junos OS. https://www.juniper.net/us/en/products-services/nos/junos/, 2017.

[138] Cisco IOS. https://www.cisco.com/c/en/us/products/ios-nx-os-software/index.html, 2017.

[139] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. In *ACM Transactions on Computer Systems*, 2000.

[140] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. *Berkeley TechReport*, 2015.

[141] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proc. SIGCOMM*, 2016.

[142] Shriram Rajagopalan et al. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. NSDI*, 2013.

[143] Aaron Gember-Jacobson et al. OpenNF: Enabling Innovation in Network Function Control. In *Proc. SIGCOMM*, 2014.

[144] Junaid Khalid, Aaron Gember-jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Implementation Nsdi. Paving the Way for NFV : Simplifying Middlebox Modifications Using StateAlyzr. *Proc. NSDI*, 2016.

[145] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association.

[146] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 227–240, New York, NY, USA, 2015. ACM.

[147] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium*

on Networked Systems Design and Implementation (NSDI 18), pages 299–312, Renton, WA, 2018. USENIX Association.

[148] Junaid Khalid and Aditya Akella. Correctness and performance for stateful chained network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[149] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proc. of HotMIddlebox*, 2016.

[150] Xin Li and Chen Qian. A survey of network function placement. In *2016 13th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 948–953, Jan 2016.

[151] Jian Sun, Yue Chen, Miao Dai, Wanting Zhang, Arun Kumar, Gang Sun, and Han Han. Energy efficient deployment of a service function chain for sustainable cloud applications. *Sustainability*, 2018.

[152] Baojia Li, Wei Lu, Siqi Liu, and Zuqing Zhu. Deep-learning-assisted network orchestration for on-demand and cost-effective vnf service chaining in inter-dc elastic optical networks. *Journal of Optical Communications and Networking*, 10, 05 2018.

[153] Angela Chiu, Vijay Gopalakrishnan, Bo Han, Murad Kablan, Oliver Spatscheck, Chengwei Wang, and Yang Xu. EdgePlex: decomposing the provider edge for flexibilty and reliability. In *Proc. SOSR*, 2015.

[154] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal. Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc). *IEEE Network*, 2014.

[155] ETSI NFV Standard. `http://www.etsi.org/deliver/etsi_gs/nfv/001_099/001/01.01.01_60/gs_nfv001v010101p.pdf`, 2017.

[156] Open platform for nfv. `https://www.opnfv.org`.

[157] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proc. HotNets*, 2016.

[158] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network: The Magazine of Global Internetworking*, 2008.

[159] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proc. SIGCOMM*, Proc. SIGCOMM, 2016.

[160] G. P. KATSIKAS, M. ENGUEHARD, M. KUNIAR, G. Q. MAGUIRE JR., and D KOSTI. SNF: Synthesizing high performance nfv service chains. `http://dx.doi.org/10.7717/peerj`, 2016.

[161] Seyed K Fayaz et al. Buzz: Testing context-dependent policies in stateful networks. In *Proc. NSDI*, 2016.

[162] Renaud Hartert et al. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. *Proc. SIGCOMM*, 2015.

[163] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. Programming Slick Network Functions. In *Proc. SOSR*, 2015.

[164] Qualcomm-5G-NR. `https://www.qualcomm.com/invention/technologies/5g-nr`, 2018.

[165] Sarah Yost. Decoding 5G New Radio: The Latest on 3GPP and ITU Standards. `https://spectrum.ieee.org/telecom/wireless/decoding-5g-new-radio`, 2018.

[166] C. J. Bernardos, A. de la Oliva, P. Serrano, A. Banchs, L. M. Contreras, H. Jin, and J. C. Zuniga. An Architecture for Software Defined Wireless Networking. In *IEEE Wireless Communications Magazine*, 2014.

[167] Dario Sabella, Peter Rost, Albert Banchs, Valentin Savin, Marco Consonni, Marco Di Girolamo, Massinissa Lalam, Andreas Maeder, and Ignacio Berberana. Benefits and challenges of cloud technologies for 5G architecture. In *Proc. of Vehicular Technology*, 2015.

[168] Peter Rost, Albert Banchs, Ignacio Berberana, Markus Breitbach, Mark Doll, Heinz Droste, Christian Mannweiler, Miguel A. Puente, Konstantinos Samdanis, and Bessem

Sayadi. Mobile network architecture evolution toward 5G. In *IEEE Communications Magazine*, 2016.

[169] M. Moradi, Y. Lin, Z. M. Mao, S. Sen, and O. Spatscheck. Softbox: A customizable, low-latency, and scalable 5g core network architecture. pages 1–1, 2018.

[170] 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. SOSR*, 2016.

[171] Charalampos Rotsos, Daniel King, Arsham Farshad, Jamie Bird, Lyndon Fawcett, Nektarios Georgalas, Matthias Gunkel, Kohei Shiomoto, Aijun Wang, Andreas Mauthe, Nicholas Race, and David Hutchison. Network service orchestration standardization: A technology survey. In *Computer Standards and Interfaces*, 2017.