IMPROVING NETWORK DIAGNOSTICS USING PROGRAMMABLE NETWORKS

Pravein Govindan Kannan

NATIONAL UNIVERSITY OF SINGAPORE
2020

Improving Network Diagnostics using Programmable Networks

Pravein Govindan Kannan
(M.Comp, National University of Singapore)
(B.Eng., College of Engineering, Guindy, Anna University)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2020

Supervisor:
Associate Professor Chan Mun Choon

Examiners:
Associate Professor Ben Leong Wing Lup
Assistant Professor Richard Tianbai Ma
Professor Boon Thau Loo, University of Pennsylvania

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all sources of information which have been used in the thesis.

This thesis has not been submitted for any degree in any university previously.

Pravein Govindan Kannan
April 27, 2021

Acknowledgements

I would like to thank Prof. Chan Mun Choon for his guidance and support throughout my graduate study at NUS. I am really fortunate to have worked with him during my Master's as well where he introduced me to networking research. He closely works his students on the research ideas from the inception and till paper deadlines. He motivated a lot even when I was discouraged by results or paper reviews, and taught me the ways to improve. I truly owe him a lot for being the guiding light throughout this endeavor. My sincere thanks to Prof. Ben Leong, Prof. Richard.T.B.Ma and Prof. Boon Thau Loo for their insightful comments to improve each part of this thesis.

I was fortunate to work with Prof. Ananda.A.L as well. He motivated me to pursue PhD when I was never confident of stepping on the long journey. Looking back, I thank him truly for the discussions on motivating me to do PhD. I would like to thank Prof. Chang Ee-Chien for his guidance and giving me the opportunity to work in NCL (National Cybersecurity Lab).

I got the opportunity to work with Prof. Anirudh Sivaraman. I thank him for giving me the opportunity to visit his research lab at NYU during summer of 2018. I enjoyed working with him on interesting problems, and it truly increased my horizon in research. A major part of this thesis is performed using Barefoot's switches. I would like to thank Vladimir Gurevich and the Barefoot support team for their generous help and support.

A big shout-out to all my lab colleagues Padmanabha, Xiangfa, Mostafa, Kartik, Nimantha, Girisha, Mobashir, Sumanaruban, Hande, Vara, Anuja, Raj, Nishant, Ebram, Ahmad. My journey would have not been fun without them. I cannot forget the crazy discussions and funny arguments during Tea sessions. Even though everyone works on different research problems, their comments have been very valuable during group meetings to improve the presentation.

I am grateful to the National University of Singapore and the Singapore government for giving me the opportunity to live and pursue my Ph.D. at one of the world's best universities. I would like to thank School of Computing for awarding me President Graduate Fellowship.

I cannot thank enough my wife Sneha, my parents (Jayanthi and Kannan) and my parents-in-law (Radha and Muralidharan). Without their support, I couldn't have sailed through this journey. Thanks for bearing with me during paper deadlines and providing continuous love and blessings. A special thanks to my three-year-old daughter Dhrithi for making my Ph.D. journey cheerful and memorable. I dedicate this thesis to my wonderful family.

Contents

Abstract

Network Diagnostics (monitoring, debugging and testing) in data centers has always been difficult. The problem is only getting more challenging with link speeds reaching 400 Gbps, the number of end-points crossing 100K and data center topologies getting more complex. Furthermore, with an increase in virtualized and diverse applications, network interactions have become more complicated. Recent studies have noted that network faults are extremely hard to diagnose due to their transient nature. Debugging such hard-to-diagnose issues require a global snapshot of the network to understand and rectify the problems accurately. However, obtaining a consistent global state of the network is extremely difficult with network metrics changing in the order of few nanoseconds.

Recent advances in programmable networking have led to better control, management and programmability of networks in the control-plane as well as data-plane. In this thesis, we study how network diagnostics of data-center networks can be enhanced by leveraging programmable networks.

To enable consistent and fine-grained monitoring of network-wide events in the data-plane, precise time-synchronization is essential in the network data-plane. We design and implement a time-synchronization protocol, *DPTP* that leverages high-resolution clocks, stateful memory and flexible header parsing available in programmable switches to maintain the clock in the network data-plane of each data-center switch. The network acts as the master clock for the end-hosts thus enabling time-synchronization within sub-RTT timescales. Our evaluation on a multi-switch testbed shows that *DPTP* can achieve median and 99^{th} percentile synchronization error of 19 ns and 47 ns between 2 switches, 4-hops apart, in the presence of clock drifts and under heavy network load.

Leveraging the synchronized network data-plane clocks, we design and implement *SyNDB*, a framework to consistently record network-wide events at a packet-level granularity to debug ephemeral events. We leverage the programmable switches' SRAM to provide a temporal storage of packets recordings and enable network-wide offline debugging using SQL queries. Additionally, *SyNDB* provides a programming abstraction for the network operators to change configuration of metrics to be collected along with the packet recordings. We evaluate *SyNDB* using a realistic topology with programmable switches, and achieve consistent ordering of packet

records, to correlate and find root cause of various network faults without affecting line-rate traffic.

Finally, it is important to test and validate the fixes, protocols and research ideas on a network environment with high fidelity and scale to prevent unexpected failures in the production network. However, the available emulators fail to address the fundamental need of test scenarios requiring diverse and scalable set of network topologies. To facilitate this, we propose a novel approach to embed arbitrary data center topologies on a substrate network of programmable switches using our network virtualization technique, called Bare-metal Network Virtualization (*BNV*). *BNV* leverages the control-plane APIs of programmable switches to build a network hypervisor. Our evaluations show that *BNV* can support various data center topologies with fewer switches and can facilitate building a high fidelity, repeatable and isolated experimentation platform for data center network operators and networking research.

In summary, this thesis demonstrates that it is possible to develop precise monitoring, debugging and testing frameworks to enhance network diagnostics using programmable networks.

List of Publications

1. **Pravein Govindan Kannan**, Nishant Budhdev, Raj Joshi, Mun Choon Chan, *"*SyNDB*: Synchronized Network Monitoring and Debugging in Data Centers"* (Under Submission) [Chapter 4]

2. **Pravein Govindan Kannan**, Mun Choon Chan, *"On Programmable Network Evolution"*, CSI Transactions on ICT 2020 [Chapter 2]

3. **Pravein Govindan Kannan**, Raj Joshi, Mun Choon Chan, *"Precise Time-synchronization in the Data-Plane using Programmable Switching ASICs"*, Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR) 2019 (Awarded Best Paper) [Chapter 3]

4. **Pravein Govindan Kannan**, Ahmad Soltani, Mun Choon Chan, Ee-Chien Chang, *"Network Virtualisation Method, Computer-Readable Medium, and Virtualisation Network"*, Patent Application No. PCT/SG2018/050352, Pub No. WO/2019/017842 [Chapter 5]

5. **Pravein Govindan Kannan**, Ahmad Soltani, Mun Choon Chan, Ee-Chien Chang, *"BNV: Enabling Scalable Network Experimentation through Bare-metal Network Virtualization"*, Proceedings of USENIX Workshop on Cyber Security Experimentation and Test (CSET) 2018 [Chapter 5]

6. **Pravein Govindan Kannan**, Mun Choon Chan, Richard T.B. Ma, Ee-Chien Chang, *"Raptor: Scalable Rule Placement over Multiple Path in Software Defined Networks"*, Proceedings of IFIP TC6 Networking Conference, Networking 2017

7. Sivaramakrishnan Ramanathan, **Pravein Govindan Kannan**, Mun Choon Chan, Jelena Mirkovic, Keith Sklower, *"Enabling SDN Experimentation in Network Testbeds"*, Proceedings of ACM SDNNFV Security Workshop 2017.

List of Figures

List of Tables

Chapter 1

Introduction

Cloud services are becoming more popular with several web service providers like NetFlix, Spotify, Twitter, etc. moving to public cloud services to host their platform. Further, Cloud service industry is expected to grow at 17.5% in 2019 to $214 billion [2]. These public cloud services are supported by data centers at the scale of several thousands of devices with complex inter-connects at peta-bit speeds [3]. Managing these super-scale data center networks are extremely challenging due to complex interactions between network services [4, 5] under different network topology conditions [6–10].

With cloud service providers promising SLAs (Service-Level Agreement) greater than 99.5% [11] and in some cases even 99.99% [12], failures are extremely sensitive. A failure can result in hurting customers' revenues and additional several billions of loss in revenue [13] for the cloud provider in the form of compensation [14], losing customers to competitors and crashing stocks. The infamous British Airways outage which left 75,000 passengers stranded was caused by a data center systems failure [15].

Several studies have noted that most of the data center outages are caused by network failures [16–18]. Hence, understanding the root cause of such network failures is extremely important. Further, a recent study from a large data center has reported that network failures could take several days to be properly triaged and fixed [16]. While

network maintenance updates, misconfiguration, bugs and insufficient capacity account for more than 50% of the network failures, a relatively large proportion (29%) of the failures go without establishing the root cause due to their transient nature. To understand why such a huge proportion of failures go undetermined, we need to first inspect how the typical work-flow in data-center operations upon encountering faults look like. When a network fault occurs (refer Figure 1.1), the first step incorporated by the operators is to gather the monitored data from the underlying networking devices. Next step would be identifying the root-cause or debugging from the monitored data. Finally, upon identifying the root-cause, the fixes have to be tested to make sure the fault has been addressed completely. The main question is *Why is this process hard?*

1. **Monitoring.** First, data center networks are high-speed at several tera-bits , very dynamic with constantly changing configurations and distributed with several thousands of devices. It is not possible to predict where the fault will occur. Hence, monitoring has to be performed throughout the network. To understand the network-wide context, the monitoring has to reveal correlation between devices and events. For this, we need to have synchronized measurements.

2. **Debugging.** Next problem is that network debugging must be unified to debug the network as a whole and enable fine-grained visibility of each part of the network. Thus, we need network-wide, correlated and fine-grained debugging to understand and pin-point problems in the network.

3. **Testing.** Finally, once the problem is identified, the solutions need to be tested high-fidelity environment which can mimic the scale of the production network before they are applied.

Over the past few years, programmable networks have revolutionized networking by providing abstractions to program the control-plane and data-plane of networks. The

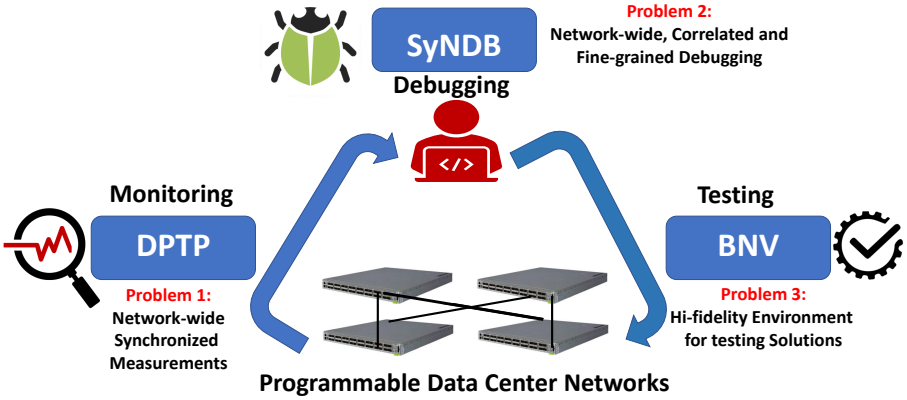


Figure 1.1: Overview of the Proposed solutions in this thesis

data-plane of a network device can be programmed using languages like P4 [19], PX [20], etc., while Control-plane of the entire network can be configured using APIs like OpenFlow [21] and P4Runtime [22]. Several research in the recent past have leveraged programmable networks to develop better management, monitoring and programmability for networks.

In this thesis, the main question is : *Is it possible to develop better network diagnostic solutions by leveraging programmable networks?*. To enable synchronized measurements at the monitoring layer (e.g. increase in switch queue due to bursty flows), we need to have nanosecond-level synchronization in the data-plane. We develop *DPTP*, a time-synchronization protocol for network data-plane which ensures nanosecond-level synchronization between network devices (switches and NICs). Next problem is that network debugging must be unified to debug the network as a whole, instead of individual devices and must enable fine-grained visibility of each part of the network. We develop *SyNDB*, a framework which leverages *DPTP* to provide synchronized packet-level network-wide recording of events at data-plane to perform debugging. Finally, once the problem is identified, we need a high-fidelity environment to test the solutions before they are moved to production networks. To facilitate this, we develop *BNV*, which

provides a scalable way of creating arbitrary topologies for network experimentation and testing.

1.1 Overview of Proposed Solutions

In recent years, Programmable networks (Software Defined Networking) have created a fundamental change in how we manage or program networks. The network model in programmable networks is modified in following ways: 1) First, de-coupling the control-plane and data-plane in the switches, 2) Logically centralizing the programmable control-plane, and 3) Defining language and abstractions to program the data-plane. Recently, many switch vendors have started shipping switches which use programmable networks [23–26]. This offers network researchers a unique opportunity to develop ASIC-based solutions which could be deployed in upcoming data centers with just a software upgrade. This is precisely the reason why we focus on programmable networks in this thesis. We leverage programmable networks to develop novel network diagnostic solutions. We depict the overview of this thesis in Figure 1.1 and explain the individual solutions in the following sections.

1.1.1 DPTP

Network-wide monitoring tools rely on correlating network events happening in individual network devices. Correlating network events have a fundamental requirement of time synchronization between network devices. The link speeds in modern data centers are 100 Gbps and moving towards 400 Gbps. At 100 Gbps, an 80-byte packet can be received every 6.4 ns. Thus to monitor fine-grained correlated events in the network, it is necessary to have nanosecond-level time synchronization between network devices. Upon network failures, a root cause analysis is initiated by traversing the observed stream of events. Each of these events is indexed with the network timestamp affixed by the re-

porting network device. If these timestamps are synchronized, the proper network order can be established, and the root cause quickly identified. However, root cause analysis is extremely difficult without accurate network synchronization [27]. Existing standards such as PTP (Precise Time-synchronization Protocol) [28] in standard industry-grade switches handle the synchronization protocol stack in the slow-path (control-plane). However, time-synchronization in the data-plane is necessary for monitoring events (e.g. switch queue depths, packet drops) in the network data-plane.

With the ability of programmable switches to perform high resolution timestamping and stateful computation on a per-packet basis, we try to answer the question: how far can we go and what does it take to achieve nanosecond-level time synchronization using the data-plane programmability framework? Clearly, implementing network-wide time synchronization in the data-plane is a much more natural way to support existing and future data-plane based distributed applications like monitoring, In-Network Telemetry [29], consistency, caching, etc.

We design a time synchronization protocol, *DPTP*, with the core logic running in the data-plane. We perform comprehensive measurement studies on the variable delay characteristics in the switches and NICs under different traffic conditions. Based on the measurement insights, we design and implement *DPTP* on a Barefoot Tofino switch using the P4 programming language. Our evaluation on a multi-switch testbed shows that *DPTP* can achieve median and 99^{th} percentile synchronization error of 19 ns and 47 ns between 2 switches, 4-hops apart, in the presence of clock drifts and under heavy network load.

1.1.2 SyNDB

Network monitoring and debugging in data centers has always been difficult. The problem is only getting more challenging with link speeds reaching 400 Gbps, and the number of end-points crossing 100K. Furthermore, with an increase in virtualized and diverse

applications, network interactions have become more complex. An incorrectly configured load-balancing or routing policy at a switch can cause a sudden increase in link utilization several hops away. In this case, it is not possible to find the root cause using just the localized information from the switches and therefore network-wide visibility is required. At the same time, fine-grained packet-level visibility is required to detect ephemeral events such as microbursts that last at most few hundred microseconds.

One way to attain the visibility needed to debug a complex network fault is to observe the progression of network-wide metrics (e.g. packet arrivals and departures at all ports), over a time period *before* and *after* the fault has occurred. Such information would allow one to know what *transpired* in the entire network leading to the fault and its impact afterwards. However, since network faults occur unpredictably, we need to record the progression of network-wide metrics and packet headers continuously. The main challenge lies in doing this continuous recording, synchronously and at a packet-level granularity, given the sheer size of a data center network and the traffic volume.

Replay-based debugging are known to benefit in debugging hard-to-reproduce and obscure issues [30–32] in distributed systems and operating systems. Taking an inspiration from these systems, we design and implement *SyNDB*, which enables recording of packets in the data-plane using programmable switches for a short duration and enables offline queries on the packet events in a coordinated and synchronized fashion when a problem occurs. *SyNDB* leverages *DPTP* to provide accurate timestamps in the network data-plane. *SyNDB* provides a programming abstraction for network operators to configure what to be monitored and when the network-wide collection of monitored data is to be performed. Additionally, we provide a query-based framework to simplify network debugging. We evaluate using a realistic topology with programmable switches, and observe consistent and accurate recording of packet histories to debug root cause of various network faults.

1.1.3 BNV

It is important to make data centers resilient. However, large data centers are challenging to keep up and running in a reliable manner due to their scale and complexity. Any changes to the network like modifying topology, routing configuration, load balancing policy, etc. need to be tested in a reliable staging environment before they are actually implemented in the production to ensure reliability and high availability. Hence, there is a need for platforms that can enable researchers and operators to experiment with various network scenarios.

In most cases, experiments rely on network emulation tools like Mininet [33], MaxiNet [34], etc. These tools allow experimenters to create network topologies instantly on their host machines. However, since they run on the CPU of the machines they are hosted on, the network performance depends on the deployment environment. Hence, it is hard to achieve fidelity and repeatability in the experiments. Additionally, these tools cannot scale to emulate production networks [35, 36].

In this work, we propose a novel approach to embed arbitrary topologies on a substrate network of programmable switches using our network virtualization technique, called Bare-metal Network Virtualization (*BNV*). *BNV* uses a novel loop-back technique to emulate virtual switch-to-switch links, and uses burst-metering to approximate virtualization of switch buffers. *BNV* is entirely software configurable and has been implemented on open source software and unmodified OpenFlow-enabled switches. The system has been deployed in a production testbed in National Cybersecurity Laboratory (NCL[1]) and also used by recent research works [37, 38] to emulate multi-switch topologies. Our evaluations show that BNV can support various data center topologies with less number of switches which can facilitate building a high fidelity, repeatable and isolated experimentation platform for data center, SDN and other research in computer

[1] https://ncl.sg

networks.

1.2 Contributions

We make the following research contributions in this thesis:

1. We perform comprehensive measurement of variable delays occurring in the network. These studies help in understanding and designing new network protocols in the data-plane. Based on the measurements, we design and implement *DPTP*, the first time-synchronization protocol for the network data-plane. *DPTP* can achieve nanosecond-level synchronization between network devices and hosts. Based on thorough evaluation, we observe *DPTP* can synchronize switches separated separated by 4-hops by less than 50 ns even under heavy network load.

2. We design and implement *SyNDB*, the first synchronized packet-level network-wide monitoring and debugging framework. We demonstrate that *SyNDB* can help in debugging ephemeral and obscure network faults. We design a programming abstraction for network operators to configure when to collect network records and what to collect in the records. We design an easy-to-use SQL based query interface for network debugging.

3. We design *BNV* using a novel topology wiring using loop-back links in switches, to emulate multiple arbitrarily connected virtual switches. To embed arbitrary topologies on the substrate topologies, we formulate an ILP (Integer Linear Program) to maximize the scalability and fidelity of network testing. We implement *BNV* using unmodified OpenFlow-based switches and demonstrate the ability to create network topologies with over 100s of switches with just five Top-of-the-Rack (ToR) switches.

Together, these tools provide precise network monitoring, debugging and testing

frameworks which help in continuous operation of data center networks. Further, they demonstrate the strength of network programmability in both control-plane and data-plane.

1.3 Thesis Structure

The rest of this thesis is structured as follows. Background and Related work is reviewed in Chapter 2. We present the motivation, design, implementation and evaluation of *DPTP*, *SyNDB* and *BNV* in Chapter 3, 4 and 5 respectively. Finally, Chapter 6 concludes the thesis with directions for future work.

Chapter 2

Background and Related Work

As background for the rest of this thesis, we first present the detailed evolution of programmable networks over the years (§2.1). We then review the work closely related to the work presented in this thesis (§2.2, §2.3 and §2.4). We conclude this chapter by discussing how programmable networks are leveraged in the rest of this thesis (§2.5).

2.1 Evolution of Programmable Networks

Computer networks play a critical role in modern technology. For a long time, computer networks were hardware based solutions with rigid and proprietary solutions. Hence, majority of the end-to-end network applications had been designed with that assumption of no network control [39]. The minimum functionality that was assumed from the network (routers) was best-effort packet forwarding. Functionalities like network monitoring, diagnostics were absent from router's feature set.

Mid 1990s saw the development of active networks [40, 41]. There are two forms of active networking: 1) capsule model and, 2) switch/router model. In the capsule model, programs can be embedded in packets to deploy new services in the network. However, the capsule model rises lots of security concerns as it is possible for malicious

end-host to infect a router and hence the network. The other form is the switch/router model, whereby the switch/router can be programmed to perform different tasks. This is conceptually similar to the data-plane programmability model to be discussed in §2.1.3.

The concept of an open programmable network was also proposed in works [42, 43] in the late 90s. The idea was to provide a set of application programming interfaces (APIs) that abstract network resources and services. By providing access to the network hardware via open, programmable network interfaces, new services, in particular, multimedia that require QoS guarantees can be built on top of these APIs through a distributed programming environment.

Another similar effort is the IETF General Switch Management Protocol (gsmp) [44] working group that was active from 1999 to 2003. The gsmp working group aimed to provide an interface that can be used to separate the data forwarder from the routing and other control-plane protocols. One of the objective of this separation is that it will allow easier addition of network services.

A more recent effort is the IETF Forwarding and Control Element Separation (ForCES) [45] working group (2001 - 2015) which also looks at the separation of control-plane and data-plane. The ForCES base protocol (RFC5810) is used to maintain the communication channel between the control element (CE) and forwarding element (FE). SoftRouter [46] is one of the earlier proposal to use logically centralized and separate control elements to control multiple forwarding elements. It uses the ForCES protocol for communication between the CEs and FEs.

Unfortunately, the above proposals were not widely adopted. Besides the fact that it is difficult to get vendor support, these ideas also received a push-back because exposing the underlying hardware through API, a more complex control-plane and logically centralized route control could be argued as going against the simple/dumb network design philosophy of the internet that prefers a minimalist approach [39].

Despite the previous difficulties in incorporating programmable networking ideas in

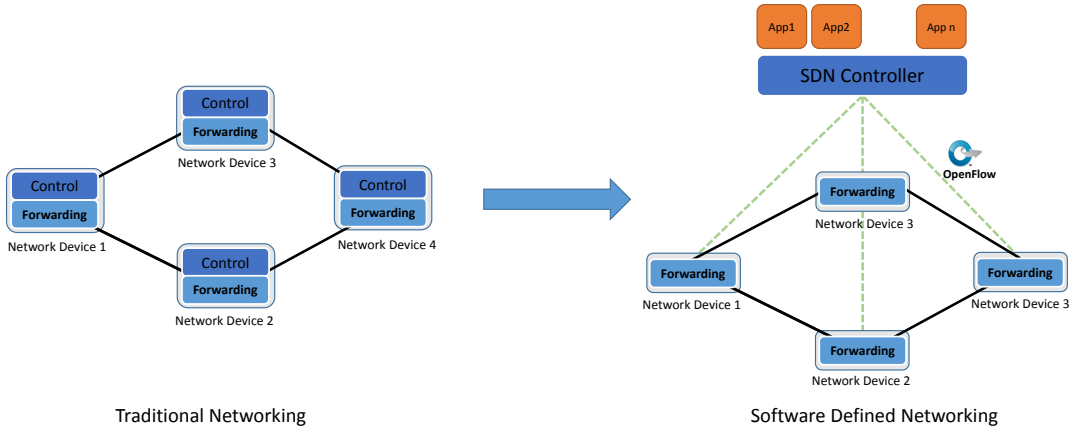


Figure 2.1: Software Defined Networking model compared to traditional networking

actual network deployment, programmable networks have recently enjoyed much more success in the form of Software Defined Networking (SDN). The success of SDN has created a fundamental change in how we manage or program networks in the following ways : 1) First, de-coupling the control-plane and data-plane in the switches. 2) Logically centralized programmable control-plane, and 3) Programmable data-plane. We investigate the aspect of each of the change in the upcoming sections.

2.1.1 Decoupling Control and Data plane

Proprietary networking devices have for long coupled control-plane and data-plane with proprietary solutions. Control-plane is the intelligence layer, which implements protocols like OSPF, BGP, Multicast, Traffic Engineering, etc. Data-plane performs the high-speed packet forwarding based on the rules and policies programmed by the control-plane. Each network device is an independent entity that needs to be managed separately in the network. Also, each network device contains complex functionality baked in, making it very complex to configure, debug and manage. SDN decouples the control/data-plane and provides an API for the control-plane to program the data-plane in the form of Match-Action rules. OpenFlow [21] provides this API as an open standard interface

for the network operators to program the network devices. Many switch vendors such as Cisco, Juniper, HP, Arista, etc. support OpenFlow API to allow their devices to be programmed.

2.1.2 Logically Centralized Programmable Control-plane

Apart from decoupling control and data-plane, the Control-plane is now a logically centralized entity (a.k.a controller) as shown in Figure 2.1. The Controller being centralized, can see the network view, and provide a network-level abstraction for the applications to be developed over the controller. The controller, typically running on commodity servers, can install rules to network device, aggregate statistics from the network device to provide a One-Big-Switch [47] abstraction.

While the idea of control/data plane separation and logical centralized control may have already been proposed in previous work, SDN, in particular OpenFlow [21], is the first successful and widely deployed framework for network programmability. The popularity of OpenFlow may have been due to the ease of adoption by switch vendors using just software (OS) upgrades. Another strength of OpenFlow is the availability of many SDN controllers based on OpenFlow APIs targeting both academia (NOX [48], POX [49], Floodlight [50], Ryu [51], etc.) and industry (ONOS [52], OpenDayLight [53], etc.). OpenFlow APIs allow the SDN controllers to program rules in switches that match several fields on the packet, and take actions like forwarding, meter, queuing, filtering, etc. OpenFlow 1.0 started simple with matching on 12 fields (Ethernet, TCP/IPv4), while OpenFlow 1.5 supports matching on 44 fields [54].

OpenFlow was initially deployed in campus networks. Early commercial successes, such as Googles wide-area traffic-management system [55] and adoption by large telcos such as AT&T [1] has also boosted its attractiveness. The emergence of data center as the key workhouse in modern internet service provisioning also helped to make SDN

[1] https://www.onap.org/wp-content/uploads/sites/20/2019/07/35752846-0-Lumina-Orchestration-2.pdf

deployment attractive. The data center network is a large network managed by a single entity. It demands advance feature to deal with the scale and can be managed centrally.

Much of the early vision for SDN focused on control-plane programmability. The next step is data-plane programmability.

2.1.3 Programmable Data-plane

While the control-plane programming allows a control program to modify the Match-Action rules in the data-plane, the matching fields and parsing logic are still pre-defined and relatively limited in scope. On the other hand, data-plane programming allows flexible parsing and matching on non-standard fields. This enables faster and easier network evolution. New protocols (or headers) addition which traditionally are big hardware upgrades that take 4-5 years, could be done by a software upgrade in a few months.

Programmable switches expose new data path primitives that are previously not available, including:

1. Transactional Memory (SRAM) with stateful ALUs that can perform simple computations like add, subtract and approximate multiply/divide.

2. Precise timestamps at different stages (ingress/egress pipeline), useful in network telemetry like queue duration and depth.

3. Packet cloning/replication useful for flexible mirroring, reporting postcards and conditional multicast.

Recent networking architectures [56,57] have enabled the following to be programmable:

- Parsing of a packet;
- Ingress/egress processing using flexible Match-Action tables;
- Stateful maintenance of network states using SRAMs (registers).

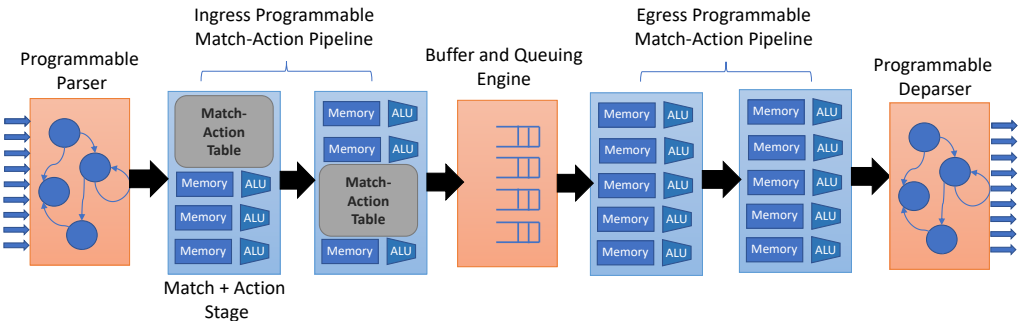


Figure 2.2: Portable Switch Architecture [1]

The model of programmable data-planes are standardized in the form of a Portable Switch Architecture (PSA) as shown in Figure 2.2.

Given all these additional capabilities and features, programming the data-plane is challenging considering that all processing *must* be performed at line-rate with no slowdown in the packet processing pipeline. This constraint comes at the cost of the programming flexibility. For example, in the program to be run in the data-plane, there is no loop construct, no floating point computations and no complex computations like multiply/divide (only approximation possibly using bit-shifts). A memory unit is mapped to a single stage and it can be read/write only once in the single pass of the packet. In spite of these limitations, developing applications in data-plane provides an attractive option to achieve high processing throughput at low latency.

Data-plane programmable switches have gained major traction in the networking industry with programmable switching ASICs being released by major switch vendors such as BareFoot Tofino [58], Cavium [59], Intel Flexpipe [57], Cisco Nexus 3000 [60], Broadcomm Trident [61], Innovium [62] and Xilinx SDNet [63]. P4 programming language [54], which emerged from academic and industry efforts is currently widely used to program these devices.

In this thesis, we leverage both the control-plane and data-plane programmability of this emerging network paradigm to improve network diagnostics in data centers. In

the subsequent sections, we will be discussing the existing work in literature related the frameworks proposed in this thesis as follows:

1. *DPTP*: Data-Plane Time-synchronization Protocol (§2.2),
2. *SyNDB*: Synchronized Network-wide Monitoring and Debugging (§2.3),
3. *BNV*: Bare-metal Network Virtualization for Network Testing (§2.4)

2.2 Network Time-Synchronization Protocols

In this section, we discuss the existing works which provide time synchronization. We discuss popular techniques from NTP [64] to recent works like HUYGENS [65], and why they are not enough to provide synchronized measurements.

NTP [64] is the most popular time synchronization protocol which is widely used. It runs on the end-hosts and uses software-based timestamping before sending the requests and after receiving the response and then calculates the one-way delay by halving the RTT. It achieves synchronization accuracy in the range of few milliseconds [66]. It additionally leverages statistical corrections to maintain the synchronization accuracy. While NTP is easy to deploy at the end-hosts, its accuracy is relatively low.

The IEEE 1588 Precise Time Protocol (PTP) [28] is a popular time synchronization protocol for networked devices in a data-center as it can provide a precision of up to 10's of nanoseconds. PTP uses a Best Master Clock Algorithm to identify the Master Clock. It forms a tree with the Master Clock as the root. At each level of the tree a switch or a server acts as master to its child nodes, and slave to its parent node. PTP is an open standard, but it is haunted by implementation-specific artifacts due to their implementation on CPUs. Even though, PTP utilizes hardware timestamping available in switching ASICs or end-host NICs, the protocol stack still needs to deal with generating requests and handling responses. Most of the proprietary implementations of PTP implement

the PTP stack in the host software [67] or in the switch control-plane [24]. This has three main disadvantages. First, the client requires several synchronization rounds and statistical filtering to offset clock drifts suffered during software processing delays. This results in clients requiring up to 10 minutes to achieve an offset below 1 μs [68]. Second, the precision of such implementations has been reported to degrade up to 100's of μs under heavy network load [68, 69]. And finally, a software implementation cannot scale to many clients without adding delays and affecting precision. There do exist dedicated PTP hardware appliances [70]. However, they are expensive and are not amenable to a datacenter-wide deployment.

GPS [71] can be used to achieve nanosecond precision [72] by connecting each device to a GPS receiver. However, deployment is cumbersome and not easily scalable since each device is directly synchronized to the GPS satellites.

Datacenter Time Protocol (DTP) [68] leverages the synchronization mechanism in the Ethernet PHY layer to achieve time synchronization. DTP modifies the PHY in each device to use the control messages in the physical layer to exchange time-synchronization messages without affecting the data traffic. Since it uses the PHY clock, its accuracy depends on the PHY characteristic. With 10Gbps link rate, the PHY frequency is 156.25MHz, and a single clock cycle is 6.4 ns. The synchronization precision achievable for 10G NIC is bounded by 25.6 ns (4 clock cycles) for a single hop. In addition, DTP requires each device (NIC and switch) in the network to have the modified PHY. HUYGENS [65] performs software clock synchronization between the end-hosts using coded probes sent during a time interval. It selects the right set of data samples using SVM and leverages network effect with measurements among many node pairs to achieve precise synchronization. However, HUYGEN's methodology cannot be used to synchronize network data-plane due to complex algorithm. Additionally, it incurs a non-negligible amount of bandwidth and processing. A summary of the comparison of different previous works is shown in Table 2.1. We observe that almost all the synchronization protocols

aim to synchronize end-host CPUs or NICs. However, for synchronized network measurements, we need synchronization in the network data-plane that *DPTP* offers.

Table 2.1: Summary of previous works on network time-synchronization

Protocol	Precision	Approach	Pros	Cons
NTP [64]	sub-ms	End-to-End Host	Simple, easy to deploy	Measurement overhead, low accuracy due to network jitters
PTP [28]	ns to sub-us	Network provides feedback on queuing delays	Support available in most NICs and switches	PTP behaviour is deployment specific, synchronized clock is not available at network data-plane.
GPS [71]	sub-ns	GPS-based clock	Accurate	Network-wide deployment is challenging
DTP [68]	ns	PHY maintains the clock	High accuracy, not affected by cross-traffic	Need DTP-enabled PHY at host-NICs and switches, accuracy depends on NIC bit rate
HUYGENS [65]	ns	Implemented at end host using probes and SVM	High accuracy, easy to deploy, no hardware dependency	Synchronization only available on end host, not available at network data-plane.
DPTP (§3)	ns	Network data-plane maintains the clock	High accuracy, synchronization available at host and network data-plane	Data-plane programmability support needed

2.3 Network Monitoring and Debugging

Traditional Network Monitoring have relied on sampling techniques like NetFlow [73], sFlow [74], SNMP [75] which collect flow information by sampling techniques. These techniques are useful in monitoring the network over a period of time, however cannot monitor the network in shorter time-scales (milliseconds). Additionally, due to their extremely low sampling rates at 0.1%, they cannot capture an accurate picture of the network.

Recently, several research works have leveraged programmable networking to develop monitoring tools in the context of a single network device and also network-wide. We group them into three categories : 1) Local Monitoring: The monitoring context is local to a device. 2) Network-wide Monitoring: The monitoring context is to provide network-wide statistics. 3) Coordinated Monitoring & Debugging: The monitoring context is to provide coordinated network-wide measurements.

2.3.1 Local Monitoring

HashPipe [76] provides heavy-hitter detection in the network data-plane using a pipeline of hash-tables. BurstRadar [77] and Snappy [78] implement techniques in the network data-plane to detect short-term events such as microbursts. BurstRadar [77] sends post-cards of the queue contents when microbursts occur to facilitate further debugging. Snappy [78] provides a probabilistic technique to identify the heavy flows in the network data-plane. *flow [79] implements mechanism to export telemetry information from switching ASICs in a flexible format.

2.3.2 Network-wide Monitoring

FlowRadar [80] provides per-flow counters for all the flows in short time scales by encoding the flows and their counters with a small memory in the switches and performs

aggregation of counters across switches to build network-wide information. UnivMon [81] uses sketch data structure in the data-plane and aggregate statistics in the control-plane to estimate network-wide statistics like heavy-hitters, DDos victim detection, entropy estimation. Mozart [82] uses temporal co-ordination to measure the flows at the right time subjected to the switch constraints. NetSight [83] uses packet-histories in the form of postcards for every normal packet sent to end-hosts to build a debugger which can be used to debug a packet's history. Marple [84] provides data-plane design for network-wide monitoring by providing a scalable Key-Value store hardware primitive. It additionally allows network operators to specify the network monitoring intents in the form of queries. Sonata [85] provides a declarative interface to express queries for network-wide telemetry tasks by offloading the primitives of queries into network data-plane.

While existing works [81–86] provide network-wide monitoring solutions, they do not provide synchronized metrics to correlate events at high resolution. We discuss the earlier works that perform either coordinated monitoring or replays.

2.3.3 Coordinated Monitoring & Debugging

Network Based. OFRewind [87] enables coordinated replay of control-plane (OpenFlow [21]) events by recording controller messages and packets to diagnose OpenFlow configuration issues and bugs. Although it helps in debugging configuration issues, it cannot replay the data-plane events in a time synchronized manner at granular time scales to debug issues like congestion. Speedlight [88] performs synchronized network snapshots using consistent snapshot algorithm in data-plane and control-plane clocks. It requires advance scheduling of snapshots every few milliseconds. Hence, it cannot aid in debugging ephemeral faults in the network. Additionally, since it uses PTP [28] to synchronize control-plane clocks, it achieves snapshots of a network where devices are synchronized up to a few microseconds.

End-Host Based. Trumpet [89] performs end-host based monitoring of packets

based on specific triggers to perform coordinated monitoring of network events. In-band Network Telemetry [29] appends switch-telemetry information like queue depth, duration to packet headers as they pass the switch. These packet headers are then analyzed at the end-hosts to further analysis to categorize congestion, microbursts etc. SwitchPointer [90] leverages end-host storage and programmability to collect packet telemetry data while switches maintain a directory or links to the storage. These approaches which are based on post-cards or appending information in the packet headers are expensive to be enabled all-time due to overhead of the disk write space and speed apart from network throughput overheads at high packet-rates. Hence, they can be enabled only when an operator observes a problem. Thus, they may end up missing the historical information of a problem. Apart from that, they do not capture synchronized data to debug timing and load imbalance issues at minuscule timescale. MahiMahi [91] builds a record and replay framework for HTTP-based applications to understand their behaviour in certain conditions. SIMON [92] applies Network Tomography to reconstruct the queuing in the switches using packet timestamps collected at the NICs. DETER [93] performs TCP replay for diagnosis of fine-grained TCP events. It constructs switch queues using the recorded TCP Packets in the end-hosts. It is helpful in debugging the effect of multiple TCP flows however, it cannot diagnose problems inside the network at small time-scales. is complementary to such higher layer replay debugging frameworks. Confluo [94] provides an end-host stack to diagnose network-wide events. However, it cannot provide the nanosecond-level correlation of events inside the network. SIMON [92] applies network tomography technique to reconstruct the switch queuing behaviour based on NIC packet timestamps.

We note that the existing frameworks discussed above do not provide fine-grained, synchronized and network-wide monitoring and debugging. While NetSight [83] and INT [29] could be augmented along with data-plane time synchronization (DPTP), they have several disadvantages like high overhead and low accuracy which we discuss in §4.2.

2.4 Network Testing Frameworks

In this section, we discuss the existing works which provide a platform for network emulation and testing. We group them into three categories : 1) Network hypervisors, 2) Switch abstractions, and 3) Network embedders.

2.4.1 Network hypervisors

Existing network slicing hypervisors like FlowVisor [95], OpenVirteX [96], VeRTIGO [97] and FSFW [98] are used in major testbeds like GENI [99], CloudLab [100] and Deterlab [101] to slice a portion of the network for experiments. Additionally, they provide ways to program the sliced network to create different scenarios. However, these hypervisors can only provide either the corresponding physical topology or a subset of it to the programmers. OpenVirtex provides a big switch abstraction, which combines multiple physical switches to one virtual switch (many-to-one). However, None of them allow the programmers or operators to create arbitrary network topologies.

2.4.2 Switch abstractions

NVP [102,103] performs network virtualization using virtual switches in the hypervisor. Since, it is based on hypervisor, it is hard to produce arbitrary topologies using software switches due to packet processing CPU overhead at the VMs or servers. Trellis [104], VINI [105] propose creation of arbitrary network topologies using containerization and tunnel abstractions. However, they cannot achieve line-rate needed for heavy workload testing. Similarly, Mininet [33] and Maxinet [34] are popular network emulation tools used to construct SDN-based virtual topologies on server/clusters. These are excellent tools for functional testing, however they cannot scale to emulate large networks carrying traffic at line-rate. CrystalNet [35] provides an extensible testing environment to mimic production networks using a large number of virtual machines from public cloud infras-

tructures. However, it does not offer fidelity of network data-plane. *BNV* can bridge this gap by providing fidelity of the network data plane. MiniReal [36] has proposed an approach to achieve multi-switch abstraction using a single bare-metal switch by modifying the software of the switches and also loopback link. *BNV* is different in that it performs virtualization functioning as a network hypervisor and does not need to modify the switch software. Additionally, it can dynamically modify topologies at run-time. CoVisor [106] enables one-to-many abstraction as part of their topology virtualization. However, since the abstraction is mainly done for programmability and security purposes, it does not cater for accurate network behavior emulation. Another approach to provide network virtualization in bare-metal cloud is to use Smart NICs [107, 108], which are programmable to perform host based networking. However, supporting arbitrary topologies using Smart NICs remains a challenge.

2.4.3 Network embedders

Other experimental setups like DeterLab [109] use Assign [110] and Assign+ [111]. Assign [110] uses simulated annealing to get an embedding for a particular network topology. BNV Mapper, since it tries to embed arbitrary topologies, needs exact port mapping. There exists a variety of virtual network embedding techniques in literature [112,113]. However, *BNV* models the virtual network using loopback links and has a different objective from the works in [112] to reduce the burden on the backbone links, and use loopback links.

2.5 Leveraging Programmable Networks

Programmable networks is an emerging trend with several programmable switching ASICs [58, 62, 114, 115] available today. These Programmable switches provide flexible packet parsing and header manipulation through reconfigurable match-action pipelines.

They also provide transactional stateful memory (in SRAMs) that allows stateful processing across packets at line-rates. Most importantly, they provide access to high-resolution clocks (∼1 ns) in the data-plane using 42-48 bit counters [1] which could maintain timestamps over several hours before *wrap around*. This combination of data-plane programmability and high-resolution clocks have made it possible to add high-resolution timing information to the packets at line-rates enabling some novel applications such as the In-band Network Telemetry (INT) [29].

SRAM sizes in switching ASICs have been increasing with recent switches having more than 100 MB of SRAM [116–118][2]. SRAM in programmable switches have been earlier used for in-network caching [120–122], application load-balancing [119] and coordination [123].

Apart from programmability in the data-plane, the Match-Action tables of programmable switches can be programmed at run-time (network operation) through control-plane APIs like OpenFlow [21] or P4Runtime [22].

In this thesis, we leverage *programmable networks* in the following ways :

1. In Chapter 3, we leverage flexible packet processing, transactional stateful memory, and **high-resolution clocks** provided by programmable switches to design and implement a precise **D**ata-**P**lane **T**ime-synchronization **P**rotocol (*DPTP*) to enable synchronized monitoring of data-plane events.

2. In Chapter 4, we leverage the **SRAM** in switching ASICs and precise time-synchronization to design and implement *SyNDB*, a synchronized packet-level recording of network-wide events to enable debugging transient network faults.

3. Finally, in Chapter 5, we leverage the **control-plane programmability** to design and implement *BNV*, a network hypervisor which enables scalable network testing and experimentation.

[2]Size estimated and extrapolated using technique in [119]

Chapter 3

DPTP: Data-Plane Time-synchronization Protocol

Fine-grained Network diagnostics in the network data-plane require precise time synchronization between network devices to accurately correlate events from distributed network measurements. Current implementations of time synchronization protocols (e.g. PTP) in standard industry-grade switches handle the protocol stack in the slow-path (control-plane). However with the necessity to monitor and precisely correlate fine-grained events in the data-plane (e.g. switch queues, packet telemetry), global time-synchronization in the data-plane is very much necessary. In this chapter, we explore the possibility of using programmable switching ASICs to design and implement a time synchronization protocol, *DPTP* , with the core logic running in the data-plane. We perform comprehensive measurement studies on the variable delay characteristics in the switches and NICs under different traffic conditions. Based on the measurement insights, we design and implement *DPTP* on a Barefoot Tofino switch using the P4 programming language. Our evaluation on a realistic topology shows that *DPTP* can achieve median and 99^{th} percentile synchronization error of 19 ns and 47 ns between 2 switches, 4-hops apart, in the presence of clock drifts and under heavy network load.

3.1 Introduction

Precise network clock synchronization plays a vital role in solutions that tackle various consistency problems related to maintaining distributed logging, databases, applications in the context of e-commerce, trading, and data-mining that run in datacenter settings. Existing standards like NTP [64] achieve only millisecond-level accuracy and require a large number of message exchanges. Synchronization mechanisms based on the IEEE 1588 Precise Time Protocol (PTP) [28] can achieve nanosecond-level accuracy under idle network conditions, but achieve only sub-microsecond level accuracy under network congestion [68]. Additionally, PTP's behavior is very tightly coupled to proprietary implementations [69], and hence it is unable to achieve the theoretical performance.

The link speeds in modern datacenters are moving towards 100 Gbps. At 100 Gbps, an 80-byte packet can be received every 6.4 ns. Hence, it is imperative that clock synchronization achieves nanosecond-level precision. Recent works like DTP [68] and HUYGENS [65] have shown that it is possible to provide nanosecond-level precision in datacenters. DTP implements their synchronization logic in the Ethernet PHY. Even though it incurs minimal traffic overhead and is highly precise, DTP needs to be supported by the PHYs across the entire network. On the other hand, HUYGENS implements time synchronization protocol between end-hosts by sending coded probes across the network and using SVM to perform clock estimation. However, HUYGENS synchronizes only the hosts, and while efficient, still incurs a non-trivial amount of bandwidth and processing overhead.

With the arrival of programmable switching ASICs, many distributed algorithms and applications [88, 120, 123–129] are readily implementable in the data-plane of the switch. These algorithms leverage the line-rate processing speed and stateful memory available in the switches. Our work draws its inspiration from these data-plane approaches. With the ability to perform high resolution timestamping and stateful computation on a per-

packet basis, we try to answer the question: how far can we go and what does it take to achieve nanosecond-level time synchronization using the data-plane programmability framework? Clearly, implementing network-wide time synchronization in the data-plane is a much more natural way to support existing and future data-plane based distributed algorithms and applications like monitoring, consistency, caching, load balancing, network updates, and tasks scheduling.

Our contributions are as following:

1. We perform a comprehensive measurement study on the variable delay characteristics of different stages in the switch pipeline, NICs as well as cables under different network conditions.
2. Taking insights from the measurement study, we design and implement a precise **D**ata-**P**lane **T**ime-synchronization **P**rotocol (*DPTP*) which leverages the flexible packet processing, transactional stateful memory and high-resolution clocks available in programmable switching ASICs. *DPTP* stores the time in the data-plane and responds to *DPTP* queries entirely in the data-plane.
3. We have implemented and evaluated *DPTP* on a multi-hop testbed with Barefoot Tofino switches [58].

Our evaluation shows that, in the absence of clock drifts, *DPTP* can achieve median synchronization error of 2 ns and 99^{th} percentile error of 6 ns for two directly connected switches. In the presence of clock drifts across the network switches, *DPTP* can achieve median synchronization error of 19 ns and 99^{th} percentile error of 47 ns for switches 4-hops apart. *DPTP* achieves 99^{th} percentile error of 50 ns between two host NICs 6-hops apart even under heavy network load.

This chapter is organized as follow. §3.3 presents the measurement study performed on the switch and end hosts. §3.4 presents how the insights from measurement study is used to design the time-synchronization protocol. The implementation and evaluation of *DPTP* are presented in §3.5 and §3.6 respectively. We highlight some discussion

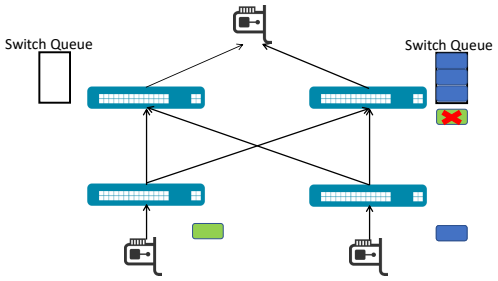


Figure 3.1: A network that load-balances traffic across multiple paths experiencing a packet drop due to burst flow on a single path

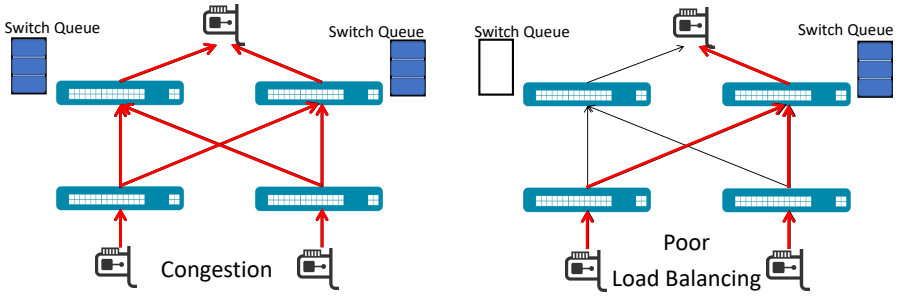


Figure 3.2: Two scenarios where the symptom (packet drop) is same, however caused due to different causes.

regarding the implementation of *DPTP* in §3.7. Finally, we conclude in §3.8.

3.2 Motivation

The main question one would ask is *why do we need nanosecond-level synchronization in the network data-plane?* We explain this using an example of performing synchronized measurements in the network in Figure 3.1. Consider a network that load-balances traffic equally across multiple paths to the destination. However, due to load-balancing issues, let's assume a bursty flow (blue packets) occupies a single path and fills up the switch queue. Now a small flow (green packet) observes momentary packet loss due to full switch queue. The reason for the packet drop could be genuine congestion, in

which case the network experienced more than provisioned capacity traffic and hence both the switch-queues are full (left part of Figure 3.2). However, the reason could also be poor load balancing (right part of Figure 3.2), in which case the network has capacity however the skewness in load balancing algorithm has caused an intermittent packet drop. Asynchronous measurements on a single device/path cannot differentiate between the two cases. To correlate the two switch queues and identify the root cause in these scenarios, synchronized measurements are required. Since network states change over nanosecond intervals at high-throughput, nanosecond-level time-synchronization is needed in the network data-plane to be precise.

3.3 Measurements

Designing a time synchronization protocol requires understanding the various delay components involved in the communication between a reference clock and a requesting client. In particular, it is important to understand which delay components could be accounted for using the timestamping capabilities in the overall system and which delay components remain unaccounted for. Achieving nanosecond-scale synchronization requires accounting for nanosecond-level *variability* in the various delay components. Further, for *maintaining* the nanosecond-level synchronization, it is necessary to keep the total synchronization delay low so that more requests could be processed per-second (for tighter synchronization) and a large number of clients could be supported.

In this section, we consider a request-response model where a *client* sends a request to synchronize with a *server* who maintains the reference clock. We perform measurements to understand the various delay components involved in a request-response packet timeline under different traffic scenarios. The request-response timeline starts when a request packet[1] is generated by a *client* network switch (or host) and ends when the correspond-

[1] 60 bytes in size comprising of the Ethernet header and a custom *DPTP* header for storing individual component delays

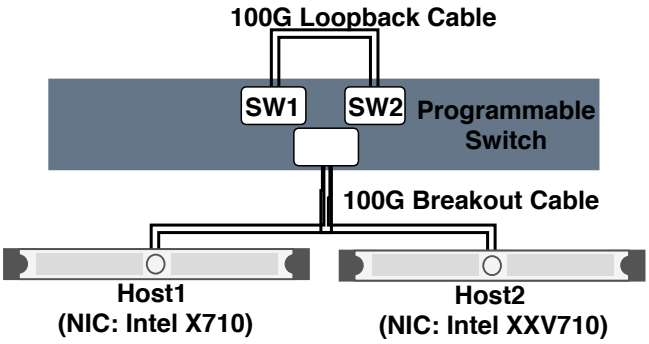


Figure 3.3: Measurement setup with a Programmable switch, two servers and a loopback cable

ing response packet is received from the *server* (another switch or host). Specifically, we measure the delays involved in NIC Tx/Rx, switch's data-plane pipeline processing, and wire propagation. We break down the overall delay into smaller components that can be measured using the precise timestamps provided by the switch data-plane.

Testbed Setup. Our measurement testbed (shown in Figure 3.3) consists of a Barefoot Tofino switch connected to two Dell servers with Intel XXV710 (25G/10G) and Intel X710 (10G) NIC cards using a 100G breakout cable configured as 4 x 10G. Two switches SW1 and SW2 are emulated on a single physical switch and connected by a 100G QSFP loopback cable (similar to [130]), which can be configured as 4 x 10G or 4 x 25G or 1 x 40G or 1 x 100G for different experiment scenarios. All cables are direct attached copper (DAC) and 1 m in length. To account for the delays involved in switch data-plane pipeline processing, we use the various high-resolution timestamps available in Portable Switch Architecture [1] (Figure 3.4): (i) T_{Rx}: timestamp when the entire packet is received by the Rx MAC, (ii) T_{Ig}: timestamp when the packet enters the ingress pipeline, (iii) T_{Eg}: timestamp when the packet enters the egress pipeline, and finally (iv) T_{Tx}: timestamp when the packet is transmitted by the Tx MAC. The switch is programmed with a custom P4 program to parse the Ethernet and a custom *DPTP* header. Based on the Ethernet type, our P4 program parses the *DPTP* header and

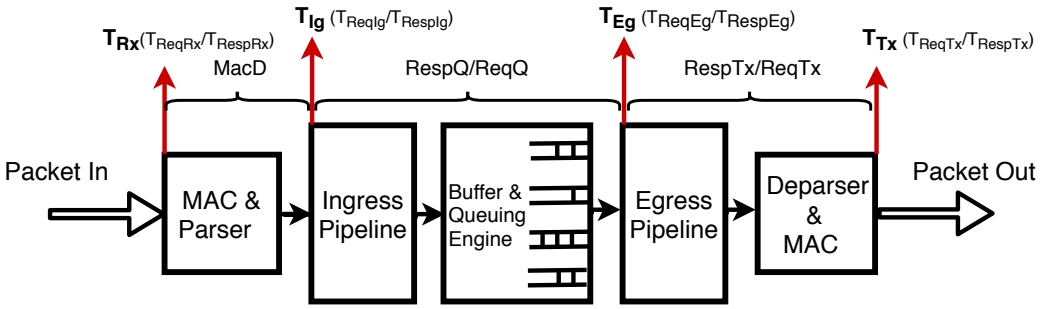


Figure 3.4: Timestamps in Portable Switch Architecture [1]

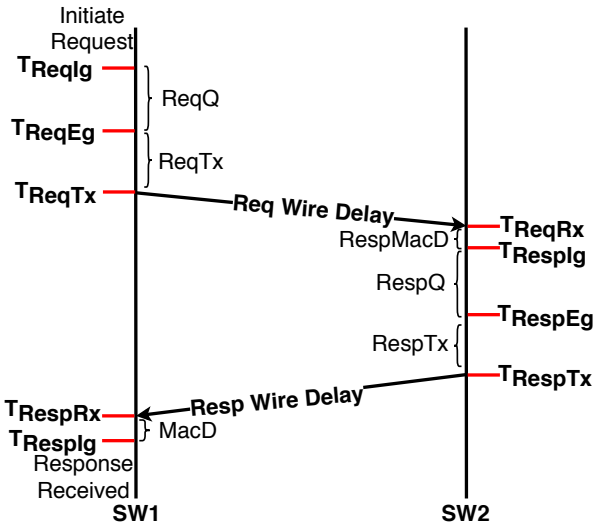


Figure 3.5: Packet Timeline and delay measurement

adds timestamps at different stages. Packet forwarding is done based on the destination MAC address. We note that the latency numbers reported in the rest of this section correspond to our custom P4 program[2].

3.3.1 Switch-to-Switch Synchronization

Figure 3.5 shows the request-response timeline when a switch SW1 sends a synchronization request to switch SW2. We list the different timestamps and intervals (in bold) of

[2]the actual switch latency can vary based on the P4 program being used.

the request-response timeline below (in chronological order):

1. T_{ReqIg}: *Timestamp* (Request packet arrival at ingress pipeline of SW1).

   ReqQ: *Duration* (Ingress pipeline processing and packet queuing at buffer engine).
2. T_{ReqEg}: *Timestamp* (Request packet arrival at egress pipeline of SW1).

   ReqTx: *Duration* (Egress pipeline processing and deparsing).
3. T_{ReqTx}: *Timestamp* (Request packet serialization and Tx from SW1).

   ReqWD: *Duration* (Wire-delay across DAC cable).
4. T_{ReqRx}: *Timestamp* (Request packet arrival at SW2).

   RespMacD: *Duration* (Input buffering and parsing).
5. T_{RespIg}: *Timestamp* (Request packet arrival at ingress pipeline and response generation).

   RespQ: *Duration* (Ingress pipeline processing and packet queuing in the buffers).
6. T_{RespEg}: *Timestamp* (Response packet arrival at egress pipeline of SW2).

   RespTx: *Duration* (Egress pipeline processing and deparsing).
7. T_{RespTx}: *Timestamp* (Request packet serialization and Tx from SW2).

   RespWD: *Duration* (Wire-delay across DAC cable).
8. T_{RespRX}: *Timestamp* (Response packet arrival at SW1).

   MacD: *Duration* (Input buffering and parsing).
9. T_{RespIg}: *Timestamp* (Response packet arrival at ingress pipeline of SW1).

From this list, only the wire delays i.e. ReqWD and RespWD cannot be directly measured using the switch data-plane timestamps. However, since there is a single physical clock shared by SW1 and SW2, we can directly compare the timestamps of the request and response packets to precisely construct the request-response timeline shown in Figure 3.5.

We perform each measurement using 10,000 request packets. Timestamps are added to the packet's header at different points in the switch-pipeline by the P4 program and the delays of various components are calculated when response arrives. We summarize the

individual delays in Table 3.1 under three network conditions: (1) **Idle links**: No other traffic is being sent on the links except for the request-response packet. (2) **Line-rate recv traffic**: Cross-traffic (64/1500 byte UDP packets) at line-rate along the direction of the response packet. (3) **Oversubscribed recv traffic**: Heavy incast cross-traffic (64/1500 byte UDP packets) that induces queuing along the direction of the response packet.

Table 3.1: Delay profiling of components in Switch-to-Switch measurements over 10G links [min - max (avg)]

Component	**Idle Links**	**Line-rate recv traffic(1500b)**	**Oversubscribed recv traffic(64/1500b)**
Request processing and Queuing (**ReqQ**)	306 - 313 ns (307)	306 - 313 ns (306.7)	306 - 313 ns (307)
Request Egress Tx Delay (**ReqTx**)	304 - 356 ns (340.8)	300 - 361 ns (341.1)	296 - 361 ns (341)
*[Wire Delay (1M) (**ReqWD**)]*	67 - 79 ns (72.6)	66 - 79 ns (72.7)	66 - 79 ns (72.4)
MAC & Parser Delay at Server (**RespMacD**)	0 - 57 ns (26.1)	0 - 57 ns (26)	0 - 59 ns (27)
Reply processing and Queuing (**RespQ**)	313 - 323 ns (317)	313 - 481 ns (317.3)	1.552- 1.556 ms (1.554)
Reply Egress Tx Delay (**RespTx**)	296 - 365 ns (341)	298 - 803 ns (555)	1747 - 1914 ns (1783)
*[Wire Delay (1M) (**RespWD**)]*	64 - 77 ns (70)	64 - 76 ns (70)	63 - 76 ns (70)
MAC & Parser Delay at Client (**MacD**)	0 - 55 ns (27)	0 - 58 ns (27)	0 - 90 ns (27))
Total RTT	1462 - 1637 ns (1473)	1460 - 1976 ns (1667)	1.555 - 1.559 ms (1.556)

Measurement results under different load conditions. In the scenario with idle links, major delays occur in the queuing and pipeline processing components (ReqQ, ReqTx, RespQ and RespTx) which add up to 84% of the total delay. Somewhat surprisingly, wire delay over the 1 m DAC cable (ReqWD and RespWD) takes around 70 ns, much more than the MAC Delay (RespMacD or MacD). Interestingly, the MAC Delay fluctuates from 0 - 55 ns since the parser takes a variable amount of time to parse the packet and feed it into the ingress pipeline [131]. RespTx also experiences similar amount of fluctuation. RespTx includes egress pipeline processing, deparsing, Tx MAC delay and serialization. Since egress processing has a fixed delay [131], this variation is contributed by the other three components of RespTx. In the line-rate traffic scenario, the RespTx is higher because the line-rate cross-traffic with 1500 byte packets fully saturates the link. The serialization delay for these packets is higher than the egress pipeline processing delay. As a result, the *extra* synchronization packets get queued for serialization in a small buffer after the egress processing. This queuing causes the Egress Tx Delay (RespTx) to increase. Line-rate cross-traffic with 64-byte packets does not observe this behaviour due to faster serialization. All other delay components are similar to the idle link scenario because they result from data-plane (hardware) processing which has similar performance under all conditions.

Under oversubscribed conditions, delay increases in two components – RespQ and RespTx. RespQ increases to 1.55 ms because the queue buffer is always full due to the oversubscribed cross-traffic (1.55 ms is the default maximum per queue buffer in our setup). RespTx in this scenario is even higher than the line-rate traffic scenario. This is because with the oversubscribed traffic, the aforementioned small pre-serialization buffer remains mostly filled with the cross-traffic packets which causes additional delay to the synchronization packets. Due to this, the range of RespTx values falls to a small window of 167 ns. Note that ReqQ and ReqTx remain unaffected in our scenarios due to cross-traffic only along the response direction. However, under the presence of cross-traffic

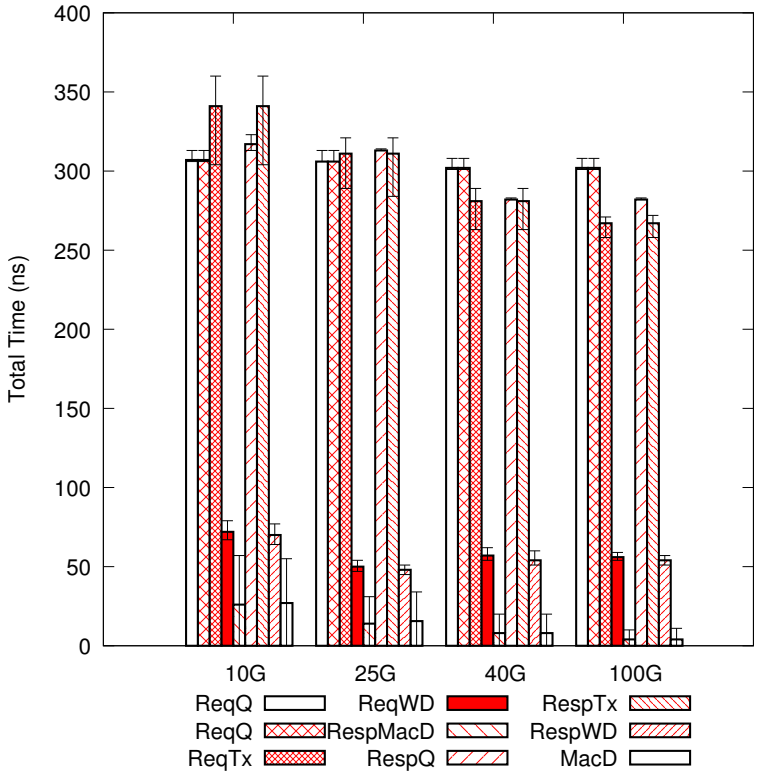


Figure 3.6: Delays for various link-speeds (idle links)

in the request direction, ReqQ and ReqTx exhibit the same behaviour as RespQ and RespTx.

Effect of different link speeds. We re-configured the link speed of the loopback cable between SW1 and SW2 (Figure 3.3) to 25G, 40G, and 100G and plot the eight delay components as bars for 10G/25G/40G/100G in Figure 3.6 for the idle traffic condition. We observe that the components involving MAC Delay (ReqTx, RespMacD, RespTx and MacD) reduce significantly with higher link speeds. We also observe decrease in wire delay of up to ∼20 ns with 100G compared to 10G. Most importantly, the variation of delays (errorbars) reduce significantly with higher linkspeeds. The average RTTs observed for 25G, 40G and 100G are 1356 ns, 1264 ns and 1044 ns respectively.

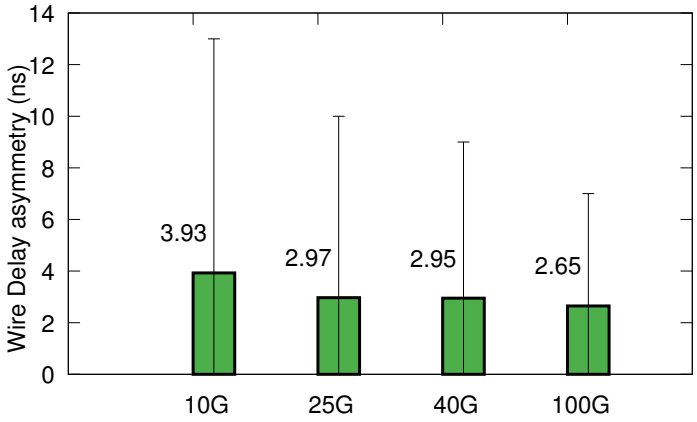


Figure 3.7: Asymmetry of wire delay between request and response for different link speeds

Accounting for the delay components. Normally, it would be reasonable to assume the wire delays to be symmetric between two directly connected switches. However, at a nano-second scale we found that the wire delays are asymmetric. We plot the wire delay asymmetry ($abs(ReqWD - RespWD)$) for different link speeds in Figure 3.7. For 10G link speed, we observe the asymmetry variation to be ranging from 0-13 ns with an average of 3.93 ns. The asymmetric wire delays could be due to the low-frequency clocks at PHY [68]. This variation drops to 0-7 ns with an average of 2.65 ns at 100G since PHY clocks operate at higher frequency clocks at higher speed. Thus, at higher link speeds, clock synchronization would be more accurate, since the unaccounted wire delay would have less asymmetry.

Measuring ReqTx and RespTx. To accurately measure and account for ReqTx/RespTx, we need to know the exact time when the request/response packet left the respective switch i.e. T_{ReqTx}/T_{RespTx}. The timestamp of the exact instant when the packet serialization starts cannot be embedded into the packet itself. This is a fundamental limitation and thus this timestamp needs to be communicated separately via a *follow-up* packet. To avoid such separate follow-up packet, several commercial switches support embedding a *departure timestamp* into the departing packet. The departure

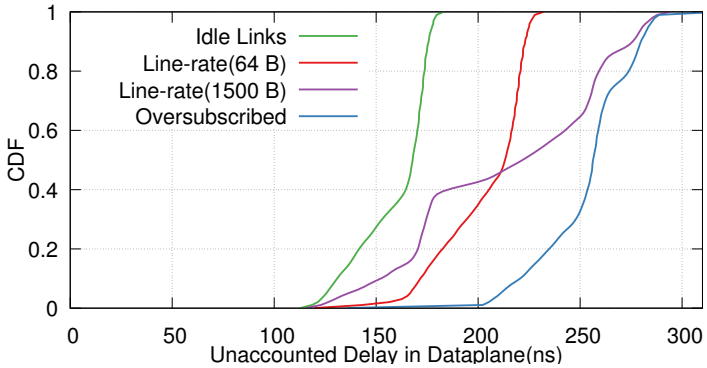


Figure 3.8: Unaccounted Delay (in ns) when using packet departure timestamp in data-plane

timestamp is added when the packet is received by the egress MAC and thus differs from the exact transmit time. The use of this departure timestamp to avoid the follow-up packet thus leads to unaccounted delay during synchronization. We measured this unaccounted delay in our setup under different cross-traffic conditions and the results are shown in Figure 3.8. At the 99^{th} percentile, oversubscribed traffic leads to unaccounted delay of about 289 ns while it remains about 175 ns for idle links.

Reducing the delays. Even if most of the delay components could be accounted for in a synchronization request-response, it is still necessary to keep the total delay low. A lower total delay allows for more requests to be processed per unit time for tighter synchronization in face of clock drifts. It also enables scaling to a large number of clients. Table 3.1 shows RespTx and RespQ to be the major contributors of the total delay. We observed that using prioritized queues for synchronization packets doesn't reduce RespTx under line rate or oversubscribed conditions. This is because the small pre-serialization buffer (the cause for higher RespTx) still remains mostly filled with cross-traffic packets. Using prioritized queues does however reduce RespQ significantly (as expected) under oversubscribed conditions. We measured RespQ to be about 2000 ns for 1500-byte packets and 650 ns for 64-byte packets with oversubscribed cross-traffic. RespQ does not come down to the same value as with idle/line-rate traffic conditions be-

cause the small pre-serialization buffer once full, back-pressures the port-level scheduler and the egress processing.

Effect of FEC. When FEC (forward error correction) is enabled, we observed that the wire delay (ReqWD and RespWD) component increases to 344 - 356 ns with idle links. The overall RTT is about 2030 ns. At line-rate traffic we observe an increase in the delay for RespQ (314 - 4452 ns) and RespTx (298 - 1914 ns) in addition to the wire delay components. This behaviour can be attributed to the additional back-pressure created by the serialization overhead of FEC. The overall RTT for line-rate traffic varies from 2026 - 7589 ns. However, it is important to note that the wire-delay asymmetry still remains the same i.e. bounded by 12 ns.

Take-aways:

1. Hardware supported timestamps and the ability to embed them in packets make programmable data-planes immensely useful to account for the various delay components.
2. Even when the link is idle, we observe a 175 ns (1462 - 1637 ns) delay variation in the processing time.
3. Even though there is much variability in the various delay components, since accounting is available for most of them, the variability does not affect clock synchronization. The only unaccounted delay is the wire-delay.
4. At a nanosecond-scale, the wire-delay exhibits asymmetry (between request and response). The asymmetry reduces with higher link-speeds thus enabling better precision at higher link-speeds. This asymmetry forms the lower bound on the achievable synchronization accuracy.
5. To accurately account for a packet's exit timestamp, a follow-up packet is required. The follow-up packet could be avoided if the packet *departure timestamp* could precisely capture packet's exit time and could be embedded in the data-plane. This has important implications for the design of a time synchronization protocol.

3.3.2 Switch-to-Host Synchronization

Similar to the earlier switch-to-switch measurement, we profile a request-response packet timeline, where the request packet originates from a host (rack server) and a switch responds to the request. We use MoonGen, a packet generation tool based on DPDK (ver 17.8) to generate request packets. The NIC is configured to capture the hardware timestamp when the packet is sent out and received. We perform the measurement on two NICS: (i) Intel X710 (SFP+), and (ii) Intel XXV710 (SFP28/SFP+). We configure both NICs to operate at 10G. For each NIC, we perform the measurements under three scenarios: 1) Idle host-switch link, 2) Line-rate receive traffic (64/1500-byte packets) from switch to host, and 3) Line-rate send traffic (64/1500-byte packets) from host to switch. For each scenario, we record the following delay components: 1) MAC and parser delay at the Switch (RespMacD), 2) Reply processing and queuing (RespQ) and 3) Reply egress Tx delay (RespTx). By subtracting the accounted components from the overall RTT using the timestamp captured at NIC hardware, we obtain the total unaccounted delay. Table 3.2 and Table 3.3 summarize these measurements for Intel X710 and Intel XXV710, respectively. We report only 64-byte packet measurements in the tables. For 1500-byte packets, we observe the measurements to be same for all components except $NicWireDelay$, which will be discussed later in this section.

Table 3.2: Delay profiling of components in Switch-to-Host measurements over 10G links (SFP+)

Component	Idle Links	Line-rate recv traffic(64b)	Line-rate send traffic(64b)
Reply MAC & Parser Delay (RespMacD)	0 - 56 ns (26.1)	0 - 58 ns (26.9)	0 - 57 ns (28.1)
Reply processing and Queuing (RespQ)	299 - 306 ns (300.6)	299 - 319 ns (303.8)	299 - 306 ns (300.3)
Reply Egress Tx Delay (RespTx)	297 - 360 ns (328)	303 - 411 ns (356.8)	297 - 360 ns (326)
[Un-accounted (NicWireDelay)]	402 - 429 ns (417.5)	405 - 430 (419) ns	625 - 699 (652) ns
Total RTT	1059 - 1084 ns (1072.4)	1069 - 1149 ns (1107.1)	1280 - 1354 ns (1307.3)

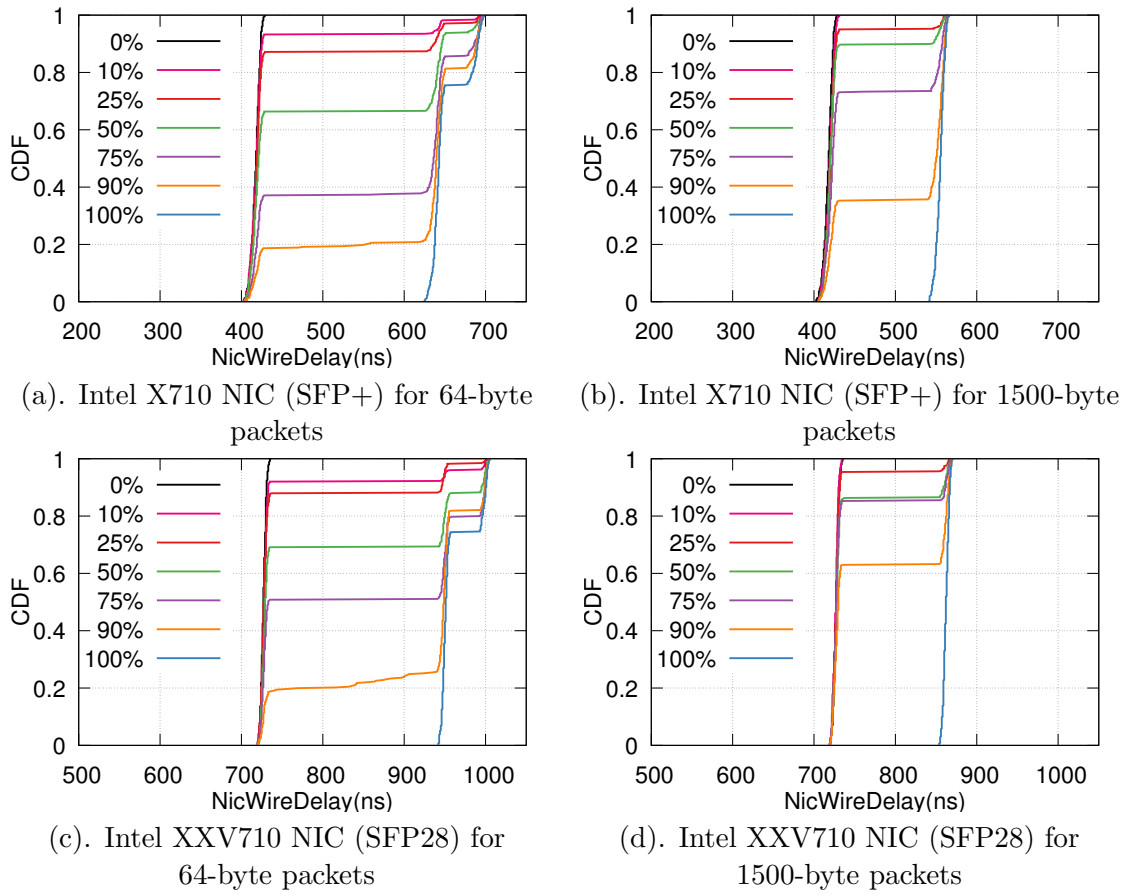
Table 3.3: Delay profiling of components in Switch-to-Host measurements over 10G links (SFP28)

Component	Idle Links	Line-rate recv traffic(64b)	Line-rate send traffic(64b)
Reply MAC & Parser Delay (RespMacD)	0 - 56 ns (27.1)	0 - 60 ns (28.4)	0 - 59 ns (27.8)
Reply processing and Queuing (RespQ)	297 - 304 ns (297)	297 - 316 ns (298.4)	297 - 303 ns (297)
Reply Egress Tx Delay (RespTx)	283 - 356 ns (328)	299 - 356 ns (358)	299 - 364 ns (331)
[Un-accounted (NicWireDelay)]	720 - 736 (727) ns	716 - 735 (725.7) ns	941 - 1005 (962.3) ns
Total RTT	1373 - 1389 ns (1381)	1379 - 1453 ns (1411)	1593- 1661 ns (1661)

The overall RTT on NIC X710 is similar to the switch-to-switch measurements without the ReqQ component. The RTT on NIC XXV710 is higher than X710 by about ∼300 ns.We suspect that this could be because of running the SFP28 NIC in the SFP+ compatibility mode[3]. The unaccounted delay consists of the two-way wire delay and the MAC/serialization delay incurred at the NIC/transceiver. To validate this, we perform an additional experiment in which we make a loopback connection between two ports of the NIC from the same host. We measure the timestamps captured when sending and receiving the packet, and observe that the total delay incurred is roughly ∼348 ns for Intel X710 and ∼660 ns for Intel XXV710. This value is close to the unaccounted delay minus one-way wire delay (roughly ∼70 ns). We refer to this unaccounted delay as $NicWireDelay$ in future references.

The Case of NicWireDelay. One interesting observation is that when we *send* cross-traffic of 64-byte packets at line-rate from the same interface as the request packets, we observe an increase in the $NicWireDelay$ by about ≈250 ns in both the NICs. We can infer that this is a one-way delay increase since we do not observe an increase in $NicWireDelay$ when we *receive* line-rate cross traffic in the same interface. In order to understand this delay, we further vary the amount of traffic (using 64-byte packets) being sent out of the host interface from 0% to 100%. Figure 3.9(a) and 3.9(c) respectively show the CDF of the $NicWireDelay$ under different volumes of sent traffic for the two NICs. We observe that the overall $NicWireDelay$ fluctuates even under 10% traffic, and the fluctuation (tail) increases with an increase in traffic. The overall range of variation is about 300 ns. Further, Figure 3.10(a) shows the average ***increase*** in the $NicWireDelay$ wrt 0% traffic for different traffic volumes. The $NicWireDelay$ increases almost linearly with an increase in the sent traffic. We repeat the same experiment with bigger packets of size 1500-bytes for the sent traffic. As earlier, we plot the CDF of $NicWireDelay$ for the two NICs with 1500-byte packets in Figures 3.9(b) and 3.9(d). We observe that the

[3]We couldn't measure XXV710 in 25G mode due to NIC timestamp issues.



(a). Intel X710 NIC (SFP+) for 64-byte packets

(b). Intel X710 NIC (SFP+) for 1500-byte packets

(c). Intel XXV710 NIC (SFP28) for 64-byte packets

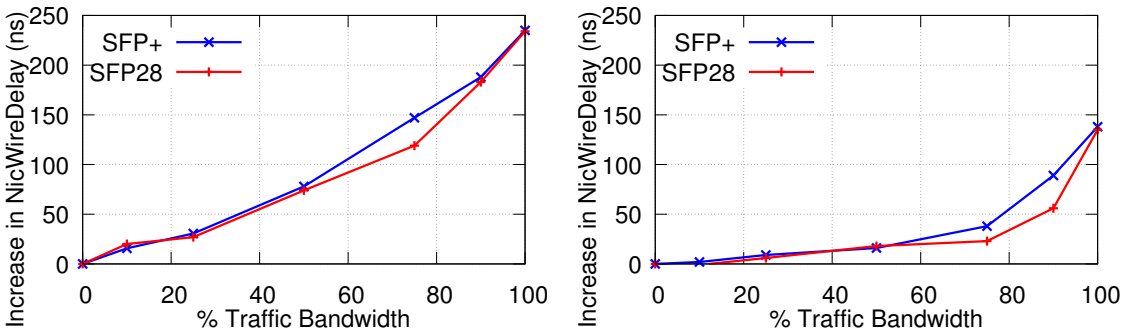
(d). Intel XXV710 NIC (SFP28) for 1500-byte packets

Figure 3.9: CDF of NicWireDelay for idle and cross-traffic conditions with 64-byte and 1500-byte packets

variation is less (∼150 ns) compared to the sent traffic with 64-byte packets. This is likely because, with 64-byte packets, there is repeated MAC/serialization delay which adds to the variability. Similar to Figure 3.10(a), Figure 3.10(b) shows the average *increase* in the $NicWireDelay$ with 1500-byte packets in the sent traffic. We observe a sluggish increase in the $NicWireDelay$ of about 40 ns till 75% traffic, and then it increases by about 135 ns at line-rate. This could be due to reduced I/O operations with 1500-byte packets as compared to 64-byte packets.

Take-aways:

1. In spite of NIC hardware timestamping, there is an unaccounted and variable $NicWireDelay$.



(a). 64-byte packets (b). 1500-byte packets

Figure 3.10: Increase in NicWireDelay w/ cross-traffic

2. The $NicWireDelay$ varies up to 27 ns in Intel X710 (SFP+) and 16 ns in Intel XXV710 (SFP28) under idle conditions.
3. In the presence of cross-traffic from the host, $NicWireDelay$ increases for both the NICs in a similar fashion.
4. Time synchronization error can increase in the presence of upstream cross traffic due to asymmetry of $NicWireDelay$. Hence, it is ideal for the host to have knowledge of the cross-traffic volume to minimize the error.

3.4 Design

DPTP is designed to be a network-level service with networking devices (switches) providing accurate time to hosts. To start, one of the switches in the network is designated the master switch. Next, the switches in the network, communicate with the master switch (directly or indirectly) to get the global time. This step happens periodically since the clocks are known to drift up to 30μs per second [65]. Once the switches in the network are continuously synchronized, they can respond to the time-synchronization queries from the end-hosts. Hence *DPTP* requests from hosts can be completed in a sub-RTT timeframe[4] and in just one hop.

[4]sub-RTT wrt requests being responded by a master switch/host

3.4.1 *DPTP* in Operation

DAG Construction. The network operator designates a switch as the master. Once the master switch is identified, a network-level DAG is constructed leading to master switch in the same way as done by DDC [132]. By leveraging DDC [132], link-failures and fast re-routing can be taken care of at the data-plane. Upon construction of DAG, each switch will query their parent switch for time-synchronization. Eventually, the TOR switches can also maintain accurate global timing in the data-plane and respond to *DPTP* queries of hosts.

Clock Maintenance. The master switch SW stores the reference clock timestamp from an external global clock source[5] to the data-plane ASIC as $T_{Ref_{SW}}$ and stores the current data-plane timestamp (T_{RespIg}) to a register T_{offset}. Note that $T_{Ref_{SW}}$ and T_{offset} are registers available in the data-plane. We do not disturb (or reset) the existing data-plane clock due to multiple reasons: 1) rewriting the internal data-plane clock is an expensive operation since it involves consistency checks, and 2) other applications like INT [29] may be using the data-plane clock in parallel. Hence, it is necessary to leave the value unchanged to be non-intrusive to other applications.

Era Maintenance. The switch data-plane's internal clock counter used to obtain T_{RespIg} or T_{RespEg} (Figure 3.5) is usually limited to x bits. For example, a 48-bit counter can account for only up to 78 hours before rolling over. In order to scale above the counter limit, the switch's control-plane program probes the data-plane counter periodically to detect the wrap around. Upon detection, it increments a 64-bit era register T_{era} by $2^x - 1$.

Switch Time-keeping. On receiving an incoming request, the master switch SW reads the initially stored external reference time $T_{Ref_{SW}}$, the era T_{era} and the correction offset T_{offset} in ingress pipeline. At the egress pipeline, it reads the current egress

[5]The details of the interface and maintaining sync between the external global clock and the master switch are beyond the scope of this work.

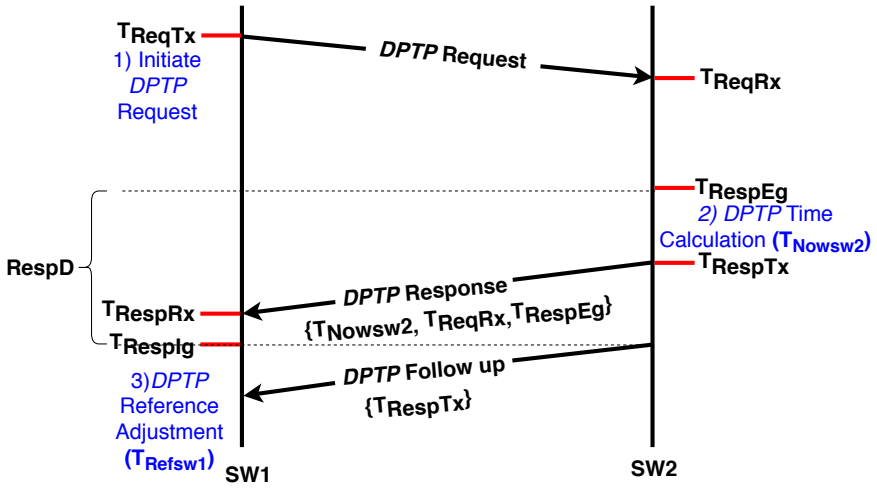


Figure 3.11: *DPTP* Request-Response timeline.

timestamp T_{RespEg} and calculates the current time $T_{Now_{SW}}$ by adding to the reference time, the time that has elapsed since initially storing the reference time at T_{offset}:

$$T_{Now_{SW}} = T_{Ref_{SW}} + (T_{era} + T_{RespEg} - T_{offset}) \tag{3.1}$$

Note that, T_{Now} is calculated at the egress pipeline using T_{RespEg} to avoid queuing delays following the T_{Now} calculation. Every time the switch SW receives *DPTP* queries from its child switch/host, it calculates $T_{Now_{SW}}$ using the current T_{RespEg}. The above steps of Clock Maintenance, Era Maintenance, and Switch Time-keeping are executed by every switch in the network.

3.4.2 Switch-to-Switch *DPTP*

Figure 3.11 shows the request-response timeline between switch SW1 and the master switch SW2, which has the correct $T_{Ref_{SW2}}$. SW1 initiates *DPTP* Request and sends the packet out at T_{ReqTx}. The Request message is received by SW2 at T_{ReqRx}. SW2 calculates the reference $T_{Now_{SW2}}$ as per Equation 3.1 in the egress pipeline. $T_{Now_{SW2}}$,

T_{ReqRx} and T_{RespEg} are embedded in the packet by the data-plane. The packet is then forwarded back to SW1 after swapping source and destination MAC addresses. The response message is received by SW1 at T_{RespRx} and it starts processing the message at T_{RespIg}. It now stores T_{RespIg} as T_{offset}. To calculate the correct reference time $T_{Ref_{SW1}}$, SW1 also needs to know T_{RespTx} which is the time just before the serialization of the response packet. For the reasons mentioned in §3.3.1, SW2 can accurately capture T_{RespTx} only *after* sending the response message. SW2 thus sends a *Follow-up* message containing T_{RespTx}. Once SW1 knows T_{RespTx}, it sets the correct reference time $T_{Ref_{SW1}}$ as following:

$$T_{Ref_{SW1}} = T_{Now_{SW2}} + RespD \tag{3.2}$$

where the response delay $RespD$ is defined as,

$$\begin{aligned} RespD = & \frac{(T_{RespRx} - T_{ReqTx}) - (T_{RespTx} - T_{ReqRx})}{2} \\ & + (T_{RespTx} - T_{RespEg}) + (T_{RespIg} - T_{RespRx}) \end{aligned} \tag{3.3}$$

In Equation 3.3, the first term calculates the approximate one-way wire-delay by removing the switch delays from the RTT. The second term is the time between T_{RespTx} and T_{RespEg}, since $T_{Now_{SW2}}$ is based on T_{RespEg}. The third term adds up the delay at SW1 before the response message is processed. When there is a follow-up packet for obtaining T_{RespTx}, SW1 would initially record the T_{RespIg} (as T_{offset}) and T_{RespRx} when it receives the DPTP response, and then use them for calculation once the follow-up packet arrives with T_{RespTx}. Hence, SW1 does not need to account for the delay due to follow-up packet. Once the reference time $T_{Ref_{SW1}}$ and the corresponding T_{offset} is stored in SW1, T_{Now} can be calculated on-demand as per Equation 3.1 for use by other data-plane applications or DPTP clients.

3.4.3 Switch-to-Host *DPTP*

Switch-to-Host *DPTP* is mostly similar to Switch-to-Switch *DPTP* . However, due to variable delay involved when there is outgoing cross-traffic from the host, we design Switch-to-Host *DPTP* to operate in two phases. Phase 1 happens only once during the initialization of the host and phase 2 is the operational phase.

Phase 1: Profiling. During this phase, the host sends *DPTP* probe packets to the switch, which are in the same format as *DPTP* query packets. The switch replies back the current time along with the switch delays ($T_{Now_{SW}}$, T_{ReqRx}, T_{RespTx}). Additionally, the switch replies the current traffic-rate (R) it is receiving from the host. If this rate is close to 0%, then the host calculates the idle $NicWireDelay$ as per Equation 3.4 (c.f. §3.3.2), and maintains its average, $AvgNicWireDelay$ over a few seconds of profiling.

$$NicWireDelay = (T_{RespRx} - T_{ReqTx}) - (T_{RespTx} - T_{ReqRx}) \tag{3.4}$$

Rate Maintenance (R). The incoming traffic-rate from each host is maintained by the ToR switches. For the traffic-rate calculation, we leverage the low-pass filter (essentially used for metering) available in the switch's data-plane. This outgoing traffic rate can also be maintained in the host's smart-nic. However, it is advisable not to calculate the outgoing traffic-rate by the application at the host since a physical interface may be virtualized to several applications. Alternatively, this statistic could be obtained with hypervisor support too.

Phase 2: Synchronization. During this phase, Switch-to-Host synchronization is performed. The host sends *DPTP* query packets, and receives the current time (T_{Now}), and delays incurred in the switch (T_{ReqRx}, T_{RespTx}). Then it calculates the response delay ($RespD$) as following:

$$RespD = OWD + (T_{RespTx} - T_{RespEg}) \tag{3.5}$$

where, OWD is the one-way $NicWireDelay$ calculated as:

$$OWD = \begin{cases} \frac{NicWireDelay}{2} & ; \text{if R} \approx 0\% \\ \\ \frac{AvgNicWireDelay}{2} & ; \text{Otherwise} \end{cases} \tag{3.6}$$

If the outgoing traffic-rate from the host (R) is close to 0%, $DPTP$ can use the $NicWireDelay$ calculated from the current $DPTP$ query to compute the OWD. However, if R is not close to 0%, it should use the $AvgNicWireDelay$ calculated during the profiling phase. This helps to bound the synchronization error, because the $NicWireDelay$ variation is small (27 ns on X710 and 16 ns on XXV710) when the traffic-rate is close to 0% (§3.3.2). The host then uses the response delay to calculate the correct reference time as per Equation 3.2.

3.5 Implementation

We have implemented $DPTP$ on a Barefoot Tofino [58] switch in about 900 lines of P4 code which performs the reference lookup, calculation and era maintenance using stateful memories and ALUs in the data-plane. We store the reference timestamp in the form of two 32-bit registers. We also implement a control-plane program in about 500 lines of C code that runs two threads to perform: (i) Era Maintenance, and (ii) $DPTP$ request generation. A Follow-up message handler registers a *learning digest* callback with the data-plane. Each time a $DPTP$ request is received, the Follow-up handler gets a learn digest from the data-plane with the following information: 1) host mac-address, 2) $DPTP$ reply out-port. The handler then probes the port's register for the transmit timestamp (T_{RespTx}), crafts a $DPTP$ follow-up packet containing T_{RespTx} destined to the host mac-address, and sends to the data-plane via PCIe. Note that, the programmable switch allows the transmit timestamp to be recorded for specific packets. Hence, we record it

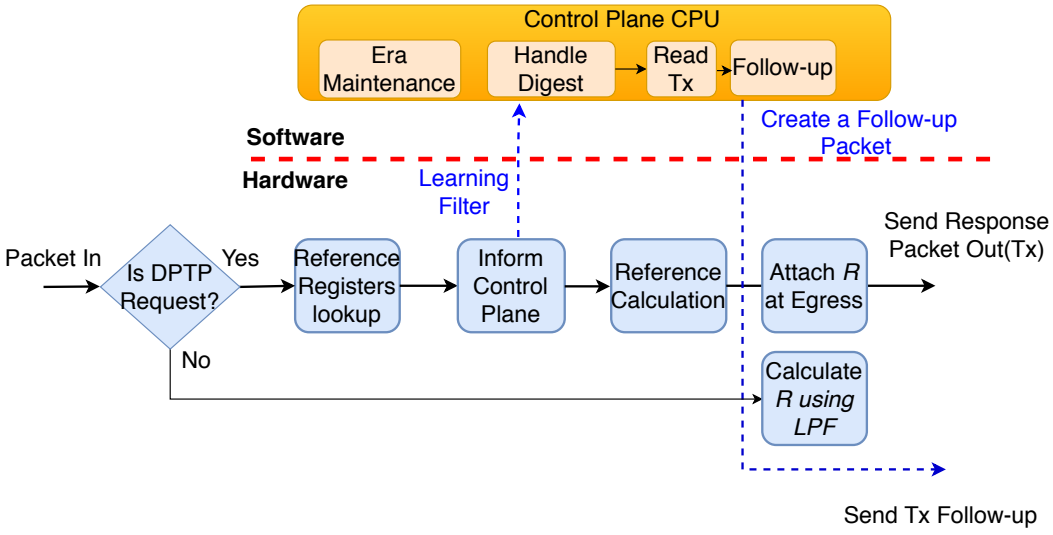


Figure 3.12: *DPTP* Implementation using P4

only for the *DPTP* packets. We implement the Follow-up handler in the control-plane since the accurate T_{RespTx} which is available in a port-side register is readable only from the control-plane.

The overall implementation of *DPTP* on the programmable switch is shown in Figure 3.12. The *DPTP* packets are assigned to the *highest priority queue* in the switches to minimize queuing delays (§3.3.1). The host client is implemented using MoonGen [133].

3.6 Evaluation

We perform the evaluation of *DPTP* using two Barefoot Tofino switches, and four servers. Each Tofino switch connects to two servers via two network interfaces per server. Using this physical setup, we form a virtual topology as shown in Figure 3.13. Switches S1, S3, S5 and M (Master) are implemented on Tofino2. The links S1-S5, S5-M are implemented using loopback cables. Similarly, S2, S4, and S6 are implemented on Tofino1 and the link S4-S6 is implemented using a loopback cable. The links S2-S5, S3-S6 and S6-M are formed using 100G cables connecting Tofino1 and Tofino2. Note that the virtualization

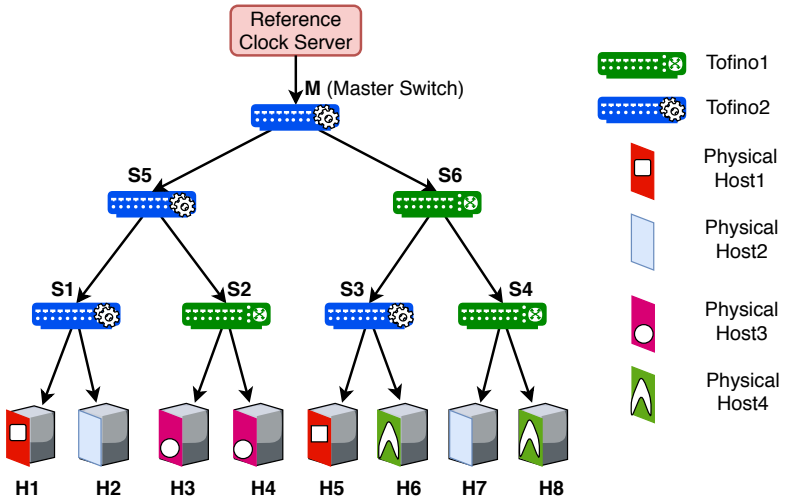
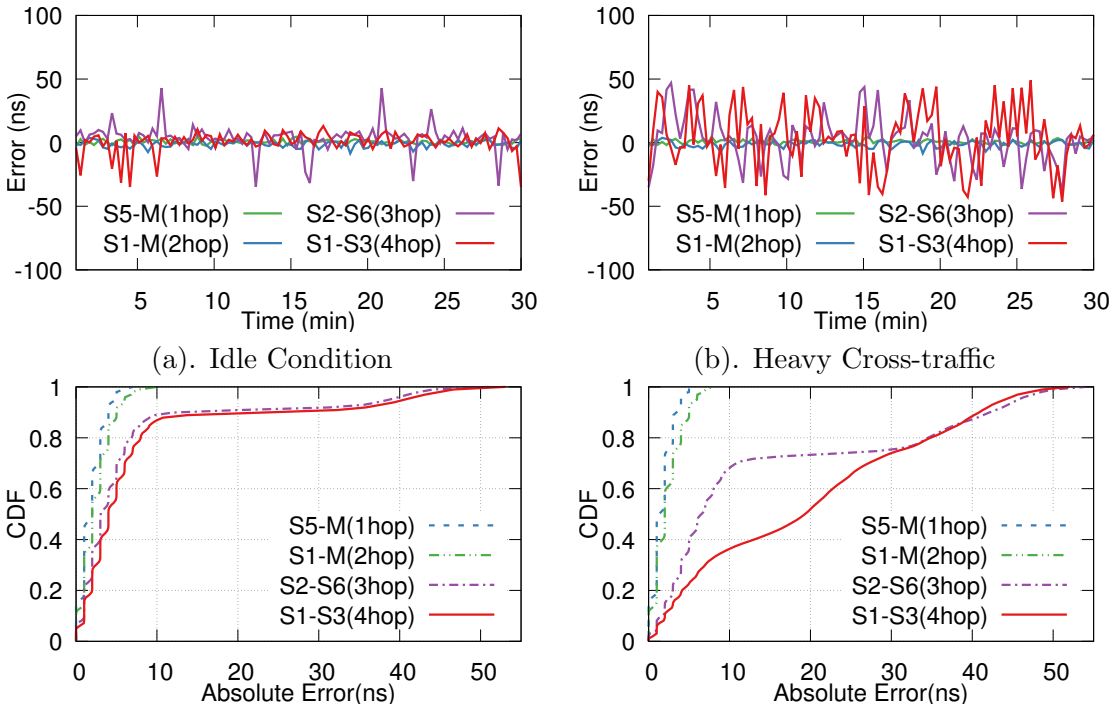


Figure 3.13: Evaluation Topology

is done in data-plane using Match-Action rules similar to [130], and so there is no virtualization overhead. All cables are Direct Attach Copper (DAC) and configured with 10 Gbps link speed. The four physical hosts are virtualized into 8 virtual hosts, with each virtual host (H1, H2, ... H8) assigned a NIC interface, which is connected to either Tofino1 or Tofino2.

The Master-switch (M) is at the core of the network and is assumed to be synchronized to an external time reference. Other switches synchronize their time with their parent switch. For example, S1 queries S5, S3 queries S6 and so on. Further, S1, S2, S3, and S4 respond to synchronization requests from H1, H2, ... H8.

The experiments are run for at least 1800 secs and each switch synchronizes by sending 500 *DPTP* packets/sec unless otherwise mentioned. We limit to 500 packets/sec (every 2ms) due to accuracy reporting at the control-plane for each synchronization. Each call to accuracy reporting involves calculation of current timestamp from different virtual switches on the same switch and writing to a file. This process takes approximately 1.5 ms. We later show in our evaluation that we can scale up to $\sim$2000 DPTP packets/sec/port.



(a). Idle Condition (b). Heavy Cross-traffic

(c). Idle Condition CDF (d). Heavy Cross-traffic CDF

Figure 3.14: Error in *DPTP* Switch-to-Switch synchronization

3.6.1 Switch-to-Switch Synchronization

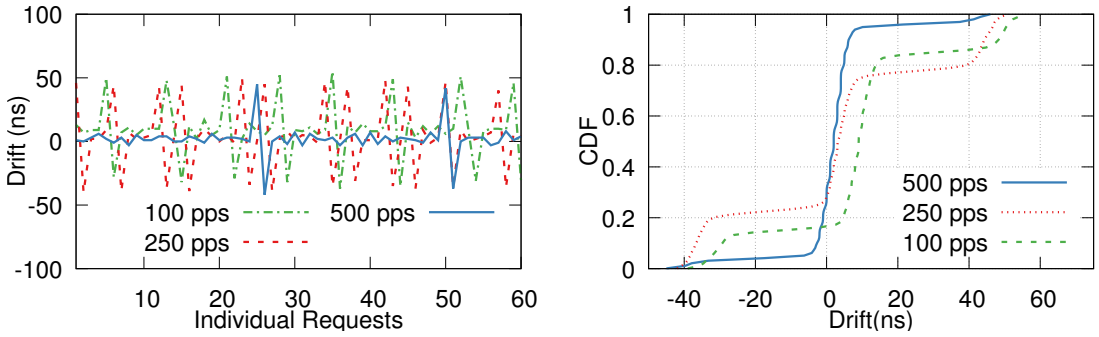
The virtual switches share a common ground-truth clock when they belong to the same physical switch. As a result, synchronization accuracy calculation is possible at the same time instant in the two virtual switches who belong to the same physical switch. We plot the synchronization error between S5-M (1-hop), S1-M (2-hop), S2-S6 (3-hop) and S1-S3 (4-hop) under idle link condition in Figure 3.14(a) and with heavy cross-traffic in Figure 3.14(b). Note that the synchronization still happens over a single hop between adjacent parent-child switches (§3.4). Figure 3.14 only captures the accumulated effect of synchronization error in between switches which are 1, 2, 3 and 4 hops away.

Idle Condition. We observe that the synchronization error does not go beyond 6 ns for single-hop (S5-M) and 12 ns for two-hop (S1-M) under both idle and heavy

cross-traffic condition. In the case of S2-S6 (3-hop) and S1-S3 (4-hop) we experience higher synchronization error bounded by about ∼50 ns due to compounded wire-delay asymmetry error with an increase in hop-count. To understand the distribution of synchronization error, we plot the CDF of the absolute value of error in Figure 3.14(c). We observe the median error to be about 2 ns and the 99^{th} percentile to be about 6 ns for single hop (S5-M). For two-hop (S1-M), the median is about 3 ns and 99^{th}percentile is about 8 ns. For three-hop (S2-S6) and four-hop (S1-S3), we observe the median to be about 4 ns and a long tail with the 99^{th} percentile about 47 ns. This is due to the effect of drift between the clocks of two different physical switches (explained later). Note that the synchronization paths from the master switch till the switches S1, S2, S3 and S6, include at least one segment where the two synchronizing virtual switches belong to two different physical switches.

Heavy Cross-traffic. When there is heavy cross-traffic across the topology, in Figure 3.14(b) we observe no effect on the accuracy between S1-M and S5-M when compared to Figure 3.14(a). Figure 3.14(d) shows the CDF of the absolute value of error under heavy cross-traffic. While S5-M and S1-M remain unchanged, we observe a higher variation of error in S2-S6 and S1-S3. The median error is about 8 ns for S2-S6 and 19 ns for S1-S3. Note that the maximum buffer size is 1.5 ms. Changing the buffer size will not affect the accuracy since we use prioritized queueing for DPTP packets and queuing delays are accountable. The higher variation could be due to the effect of non-linearity of the clock during stressed conditions [65] and higher variation of drift (explained next).

Effect of Drift. To understand the drift behavior between two physical switches, we measure the drift of the clock at S6 when it synchronizes with the Master. S6 sends a *DPTP* request and uses the T_{Now_M} from the response to calculate T_{Ref1}. At the same time, it stores the current T_{RespIg} as $T_{Elapsed1}$. S6 then sends another *DPTP* request, and calculates T_{Ref2} while storing the current T_{RespIg} as $T_{Elapsed2}$. Now the drift can



(a). Drift (in ns) between individual DPTP requests

(b). CDF of Drift

Figure 3.15: Drift captured at S6 for different DPTP synchronization intervals

be calculated as T_{Ref2} - (T_{Ref1} + ($T_{Elapsed2} - T_{Elapsed1}$)). We verified that this value is almost zero in the case of two virtual switches implemented on the same physical switch because there is no drift between the two. While our method to measure the drift may not be as accurate as using an external clock reference, the expected errors due to unaccounted wire delay asymmetry are very small. Since we generate the *DPTP* request packets from the control-plane, the packet-arrival rate may not be accurate as configured. In order to make the drift measurement as accurate as possible, we log the interval ($T_{Elapsed2} - T_{Elapsed1}$) for each drift data-point and normalize. When S6 is synchronized every second (1 DPTP pkt/sec), we observe an average drift of about 775 ns (min:664 ns, max:886 ns). In Figure 3.15(a), we plot the observed drift at S6 between individual DPTP request/response with the Master at different rates of 100/250/500 requests/sec. We observe that drift between the switches fluctuates and does not increase linearly. While Figure 3.15(a) shows a 'zoomed in' picture, the trend in the drift remains the same even over a longer period (∼30 mins).

Additionally in Figure 3.15(b), we plot the CDF of drift values observed at S6. We observe a high variance in drift values. The median drift is around 11 ns at 100 pkts/sec, 7 ns at 250 pkts/sec and 3 ns at 500 pkts/sec. The drift is about 43 ns at 99^{th} percentile with 500 DPTP pkts/sec and 90^{th} percentile with 250 DPTP pkts/sec.

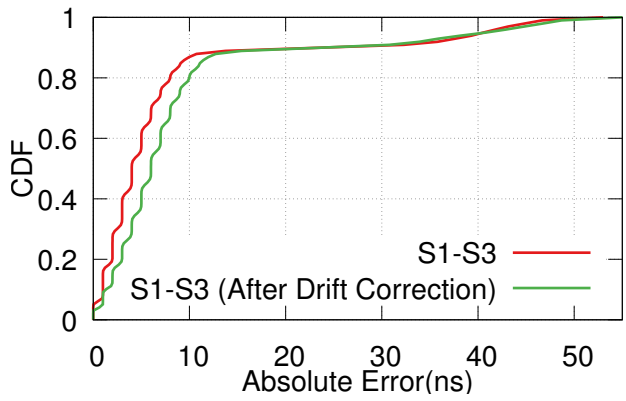


Figure 3.16: *DPTP* w/ and w/o linear drift correction

These drift statistics provide insight in to why the synchronization errors between S1-S3 and S2-S6 reach ∼50 ns occasionally during idle and heavy cross-traffic conditions (Figures 3.14(c), 3.14(d)).

Linear Extrapolation of Drift. HUYGENS performs synchronization every 2 seconds and uses linear extrapolation of past estimates to correct drifts. Based on our measurements of S6's drift wrt Master (Figure 3.15), the drift may not be linear across small time-scales. We tried applying HUYGEN's method of drift correction, by averaging the past drift estimates (2 ms window) to do drift correction at S6 before responding to requests from S3. Ideally, this should compensate S6's drift (wrt Master) between its consecutive synchronizations with the Master. This should, in turn, improve the synchronization between S1-S3 due to less impact of drift on S3. In Figure 3.16, we plot the CDF of synchronization error between switches S1-S3 with and without drift correction at S6. We observe that there is no improvement and in fact drift correction reduces the accuracy slightly. This can be explained by our observation that the clock drift over small time-scale is not linear. Hence, we do not use any statistical drift adjustments.

Effect of Link-speeds. We perform switch-to-switch synchronization by changing the link-speeds to 25/40/100G between all the switches. We plot the CDF of error

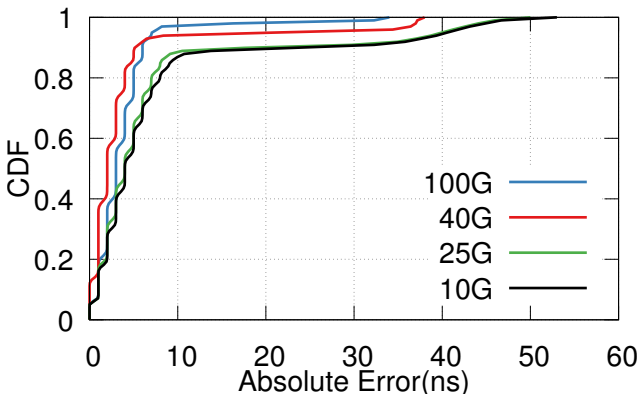
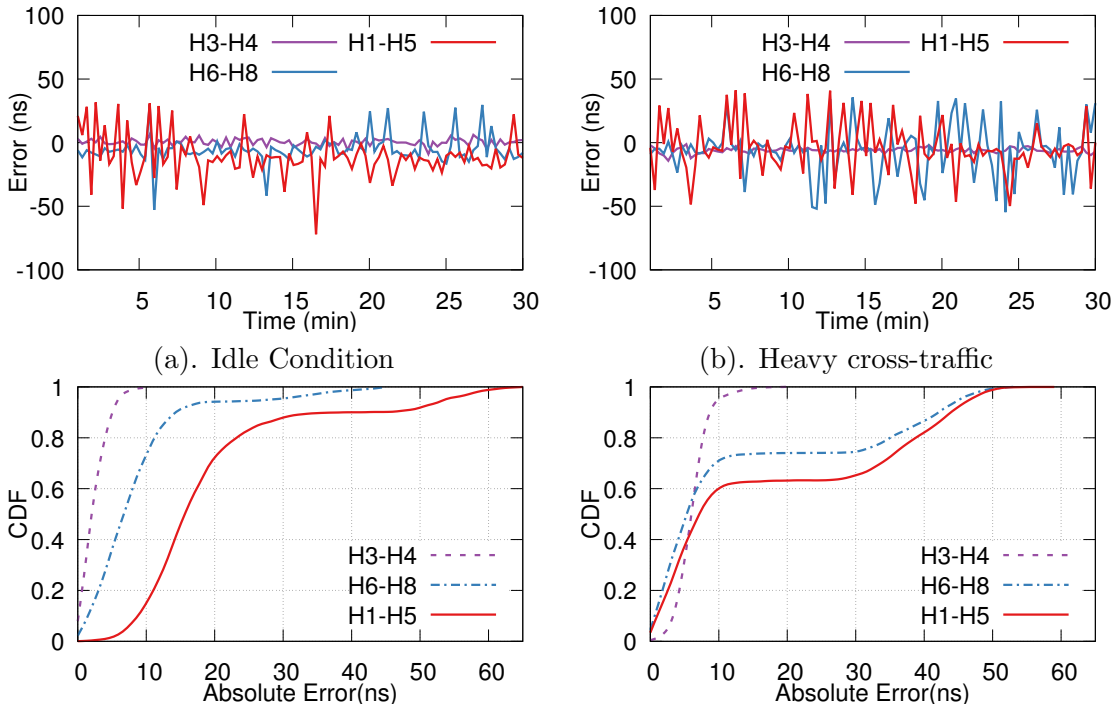


Figure 3.17: Error of Clock Synchronization (S1-S3) for different link-speeds

between S1-S3 (longest path in the topology) in Figure 3.17. As shown in our analysis in §3.3, we observe a reduction in the error with higher link-speeds. We observe a similar trend for 40G and 100G, with 100G marginally better at the 95^{th} percentile.

3.6.2 Switch-to-Host Synchronization

We send *DPTP* requests from hosts to their parent switches at 2000 requests/sec. By using NIC timestamps, we capture the synchronization error in between H3-H4 (hosts connected to the same switch), H6-H8 (hosts sharing the same aggregation switch) and H1-H5 (inter-pod hosts). We plot the synchronization error (smoothened) over a 30-minute run under idle condition (Figure 3.18(a)) and when there is a line-rate upstream cross-traffic (Figure 3.18(b)). We also plot the CDF of the errors in Figure 3.18(c) and Figure 3.18(d) for the two conditions. We observe a median error of 3 ns between H3-H4 under idle condition and 7 ns under heavy cross-traffic condition. The error seems to be higher for the hosts connected to different physical switches due to the drift between the switches as noted earlier. Between H1-H5, we observe a 75^{th} percentile error of 20 ns under idle condition. Under heavy cross traffic, we observe an increase in the error to 31 ns. However, there is minimal change in the 99^{th} percentile error. We observe similar trend in the case of H6-H8 which are separated by four hops. The increase in error



(a). Idle Condition (b). Heavy cross-traffic

(c). Idle Condition CDF (d). Heavy cross-traffic CDF

Figure 3.18: Error in *DPTP* Switch-to-Host synchronization

from 60^{th} percentile in heavy cross-traffic conditions can be attributed to the usage of the profiled value $AvgNicWireDelay$, upon the feedback of high traffic-rate(R) from the switch.

Drift at NIC. Following similar methodology as in §3.6.1, we measure the drift at the host's NIC clock. This drift gives an idea of how many requests/sec need to be generated from the host to keep the synchronization error low. At 1 DPTP request/sec, we observe that the host NIC drifts at $\sim$21μs/sec. We also observe that the overall range of the drift is 20874-21061 ns/sec. To understand the drift behavior in the NIC, we plot the drift occurred between individual DPTP requests at different rates in Figure 3.19. We observe that at short time-scales the drift in the NIC is also not linear and has fluctuations. Therefore, to maintain the error in the order of 10s of ns, it is necessary to perform *DPTP* requests more often. While Figure 3.19 shows a 'zoomed in' picture,

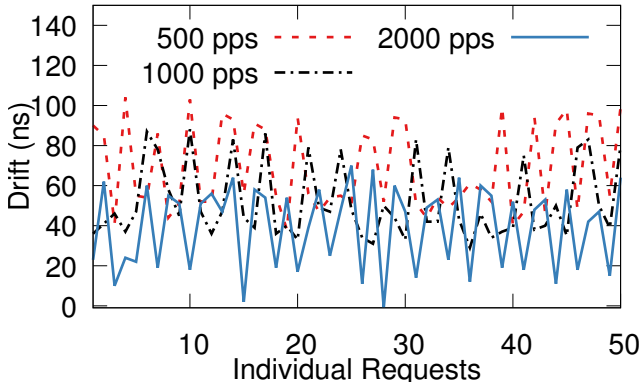


Figure 3.19: Drift (in ns) at the Host NIC Clock between individual DPTP requests

the trend in the drift remains the same even over a longer period (∼30 mins).

3.6.3 Scalability

The scalability and accuracy of *DPTP* is hugely dependent on the follow-up delay. Suppose, there are X *DPTP* request/sec on a specific port. Therefore, the control-plane has to read the port Tx register within $\frac{1}{X}$ sec to get T_{RespTx}. If the control-plane fails to read the T_{RespTx} within that deadline, the next response packet would be sent out through the same port, thus overwriting the value of the previous T_{RespTx}. We use learning filter from data-plane to deliver digests to the control-plane. The learning filter is configured to immediately send the digest notification to the control-plane. We observe that the follow-up delay[6] is about ∼429 μs *regardless* of the load of packets on various ports. This means that the control-plane can keep up with requests at the rate, (1 second/ 429μs) = 2331 requests/sec/port. Considering the maximum drift of 30μs/sec (reported by [65]), 2331 requests/sec/port can keep the drift of the worst-case clock under 13 ns. Since, the follow-up delay directly impacts the accuracy (in terms of drift), it is ideal to prioritize the follow-up packets to avoid buffer delays.

Figure 3.20 shows the CPU utilization of the control-plane program with 1000 *DPTP*

[6]Digest notification to control-plane + follow-up packet to reach data-plane

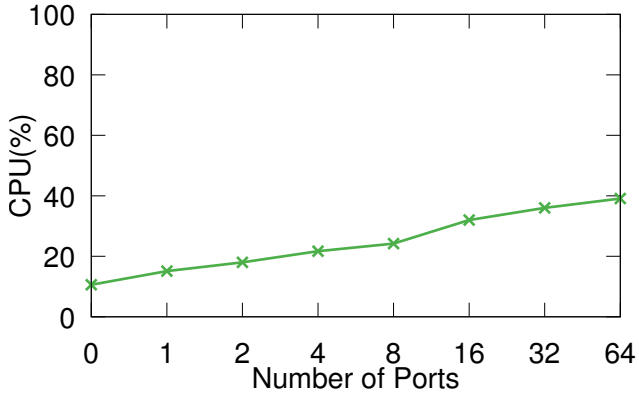


Figure 3.20: CPU utilization to handle follow-up packets

requests/sec from each port. The baseline CPU utilization is about 10 %. We observe a steady increase in CPU utilization and it reaches only up to 40% with incoming requests from 64 ports concurrently.

3.6.4 Resource Utilization

We evaluate the hardware resource consumption of *DPTP* compared to the baseline switch.p4 [134]. The switch.p4 is a baseline P4 program that implements various common networking features applicable to a typical datacenter switch. We illustrate the percentage of extra hardware resources consumed by *DPTP* in Table 4.1. We observe that while *DPTP* consumes a relatively higher proportion of stateful ALUs to maintain the reference clock and to perform the arithmetic operations to calculate and adjust the reference. The requirement of other resources is less than 7%. Hence, *DPTP* can fit easily into datacenter switches on top of switch.p4.

Bandwidth Consumption. A DPTP packet is an Ethernet frame packet with a DPTP header [$Type$, $T_{Now}(hi)$, $T_{Now}(lo)$, T_{ReqRx}, T_{RespEg}, R] of size [1+4+4+4+4+4] bytes = 21 bytes. Due to minimum Ethernet frame size, this translates to 64 bytes "on wire" including the Ethernet FCS. 2000 DPTP requests/sec could keep the clock drift

Table 3.4: Hardware resource consumption of *DPTP* compared to the baseline switch.p4

Resource	switch.p4	*DPTP*	Combined
SRAM	29.79%	6.25%	36.04%
Match Crossbar	50.13%	4.62 %	54.75%
TCAM	28.47%	0.0%	28.47%
Stateful ALUs	15.63%	15.63%	31.26%
Hash Bits	32.35%	3.99%	36.34%
VLIW Actions	34.64%	4.43%	39.07%

errors low to about 15 ns considering the worst-case clock drift of 30 μs/sec. Hence, total bandwidth usage for 2000 DPTP requests/sec is about 6000 (req + reply + follow-up packets) * 64 bytes = 3.07 Mbps per link, which is 8 times lower than the bandwidth usage (25 Mbps for T-1 testbed) of HUYGENS [65].

3.7 Discussion

Currently *DPTP* works the best if the entire network supports *DPTP* in the data-plane. In future, we would like to gauge the behaviour of *DPTP* under partial deployment of programmable switches which could increase the synchronization error. Tackling these uncertainties with statistical extrapolation in the network data-plane can help reduce the error. It would be interesting to implement sophisticated extrapolation techniques like Linear/Polynomial/Log Regression using bit-shifts in data-plane and exponential smoothing using low pass filters. Such statistical extrapolation could further help in reducing the number of *DPTP* requests thereby reducing switch CPU and bandwidth consumption. Future works could also look into more sophisticated ways of estimating $NicWireDelay$ based on the traffic-rate. Currently, the timestamps for follow-up packets are obtained from a port-side register by the control-plane. The reason is because the port-side register is not integrated with the egress pipeline and is accessible only from the control-plane. With hardware enhancements to make the port Tx timestamp accessible at the egress pipeline, a follow-up packet could be generated at the data-plane itself and

the Tx timestamp could be embedded in the follow-up packet in the egress pipeline. We believe this is not the case currently since there was no application that demanded this. However, *DPTP* could hugely benefit from this minor hardware enhancement: *Zero* CPU consumption and an *order of magnitude* reduction in the follow-up packet latency. In future, we also plan to expand our measurement study to include other NIC standards, different cable lengths, and cable medium (optic cables).

Network Monitoring. *DPTP* can provide the critical functionality that current in-network telemetry systems like INT [29] lack : "The ability to correlate events in the network data-plane". In the next Chapter (4) we describe in detail how the data-plane time synchronization provides the foundation to fine-grained network-wide monitoring and debugging of network faults.

3.8 Summary

In this chapter, we design and implement *DPTP*, which to the best of our knowledge, is the first precise time synchronization protocol for the network data-plane. To understand why *DPTP* performs better than the existing techniques, we summarize the design princples in terms of challenges faced by any time synchronization protocol.

1. **Variable network delays.** *DPTP* addresses this challenge by using precise timestamps available in programmable to account nanosecond-level variable delays at component-level granularity in the switches. To address the variability at the NICs due to *NicWireDelay*, *DPTP* performs one-time profiling to accurately estimate one-way delays under no cross-traffic conditions. The one-time profiling in turn helps in correcting the one-way RTT skewness in NIC during cross-traffic conditions.

2. **CPU scheduling delays.** *DPTP* addresses this challenge by maintaining the clock in the network data-plane which does not have arbitrarily long scheduling

delays. The delays due to queueing (or congestion) during synchronization is addressed by sending *DPTP* packets at high priority. Maintaining the clock in the data-plane makes global-time available for each packet at line-rate.

3. **Clock drifts.** *DPTP* addresses clocks drifts between two physical clocks by running the synchronization protocol continuously between two physical clocks of devices every millisecond. Since, this synchronization happens only between two physically connected nodes based on a computed DAG, the Request/Response cycles are in the order of few microseconds. This is short by a magnitude compared to techniques that synchronize CPUs.

Finally, we evaluate *DPTP* on a hardware testbed and observe synchronization error of within ∼50 ns between switches and hosts separated by up to 6 hops and under heavy traffic load.

Chapter 4

SyNDB: Synchronized Network-wide Monitoring and Debugging

Network monitoring and debugging in data centers has always been difficult. The problem is only getting more challenging with link speeds reaching 400 Gbps, and the number of end-points crossing 100K. Furthermore, with an increase in virtualized and diverse applications, network interactions have become more complex. An incorrectly configured load-balancing or routing policy at a switch can cause a sudden increase in link utilization, several hops away. In many cases, it is not possible to find the root cause by observing only the local information in a switch. In fact, in order to find the root cause, it is required to have fine-grained packet-level and network-wide visibility. In this chapter, we present the design and implementation of *SyNDB*, which enables packet-level synchronized recording of network-wide traffic in the data-plane. *SyNDB* enables network-wide offline debugging in a synchronized fashion to find the root cause of network faults. We implement and evaluate *SyNDB* using a realistic topology with programmable switches, and achieve consistent ordering of packet records, to correlate and find root cause of various network faults without affecting line-rate traffic. Finally, we present three case studies to demonstrate how operators can debug networking faults using *SyNDB*.

4.1 Introduction

Debugging network failures occurring in modern data centers is extremely challenging due to the scale and complexity of interactions in a dynamic environment. A recent study [16] has noted that 29% of network failures go without establishing the root cause due to their transient nature. Furthermore, network faults come in many forms. During network configuration, updating of routes can lead to unexpected packet drops due to temporal inconsistency [135, 136] or intermittent congestion [137]. A load balancing policy for handling multi-path routes can lead to short term load imbalance [138, 139] resulting in packet drops. A sudden burst in traffic from multiple sources can lead to microburst that last at most few hundred microseconds [140].

Systems programmers have long benefited from debugging tools like GDB [141] to get the visibility of what is going on "inside" (stack trace, register values) the program during execution. In the networking domain, to attain the visibility needed to debug a complex network fault, we need a tool that can (1) *observe* the network-wide metrics (e.g. packet arrivals and departures at all ports for all switches) and (2) *correlate* these network-wide observations over a time period *before* and *after* the fault has occurred.

To address the issue of *observability*, approaches such as per-packet postcards [83] or always-on telemetry approaches (INT [29], PathDump [86]) have been proposed recently as possible solutions. However, these approaches incur very high overhead and do not scale well. The main bottleneck, in these approaches, is the limited network bandwidth available for exporting the recorded data from the switches (via management/mirror port), and the slow write speeds of cold data storage devices (magnetic disk drives) where the data is stored eventually. Additionally, we observe that continuously recording to the cold data storage is wasteful, as network faults occur infrequently [16]. Another drawback of using approaches such as INT which store network information in the packet header, is the fact that they are unable to reproduce information of dropped packets.

This makes debugging failures due to packet drops infeasible in most cases due to lack of network information.

There are two challenges that make it difficult to reduce the amount of data to be collected. First, it is not possible to predict where a fault will occur. Hence, monitoring needs to be enabled on the entire network. Second, network faults are unpredictable and hence, monitoring needs to be enabled at all times. The key lies in designing a monitoring system that can perform continuous monitoring at packet-level granularity across the network, yet have low overheads in terms of network, compute and storage.

To address the issue of *correlating* data from network-wide observations, approaches such as NetSight [83] and INT do not provide explicit timestamps but assume that ordering information is available. For example, NetSight assumes that postcards are received in-order. Hence, these approaches can only correlate packets within a single flow at best. SpeedLight [88] provides causal consistency and timestamps with accuracy in the order of tens of microseconds. At 100 Gbps, a 500-byte packet can be received every 40 ns and measurements have shown that RTTs in modern data center vary between 5μs to 100μs [108]. Therefore, a much higher time resolution is needed for precise network-wide event correlation.

In this chapter, we present, *SyNDB*, a packet-level, synchronized network-wide debugging framework that addresses the challenges listed above. Our key idea is to leverage the switch data-plane as a fast temporal storage to perform continuous recording of packet-level telemetry information (*packet records*) over a moving time window (*recording window*). When no network fault is detected, the recording window moves ahead and older data beyond the record time-length is discarded. When a network fault is detected, packet records stored in the recording window the fault are sent to a monitoring server (collector) for permanent storage. At the same time, new packet records are stored in a fixed size buffer. Once this buffer is full these packet records are also sent to the collector and the switch stores new packet records in the recording window as usual. In this way,

SyNDB provides an efficient way to capture packet-level network metrics before and after the fault. Further, the switch that detects the fault broadcasts a (high priority) message to all other switches, which in turn triggers these switches to export the data from their recent recording window to the collector for storage and analysis. To correlate the data collected from different switches, the switch data-planes are time-synchronized using DPTP [37]. DPTP is a recently proposed network time synchronization protocol that runs in the data-plane to provide precise time synchronization. At the monitoring server, the synchronized, network-wide packet-level data enables root cause analysis. In summary, we make the following contributions:

1. We present the design of *SyNDB*, the first framework for synchronized recording of network-wide events at packet-level granularity. *SyNDB* captures metrics, detects events and achieves time synchronization, all in the data-plane (§4.3).
2. We develop an abstract interface and a run-time support for the user to configure and dynamically change the operating parameters of *SyNDB* such as fault detection conditions and the metrics to be recorded without the need to re-program the data-plane (§4.5). This is unlike prior data-plane based systems such as Marple [84].
3. We have implemented *SyNDB* on Barefoot Tofino [58] switches using P4. The packet records at the collector are stored in a relational DBMS facilitating debugging of network faults using SQL queries (§4.4 and §5.5).
4. We demonstrate the effectiveness of *SyNDB* by showing how it can be used to identify the root cause for several network faults (§5.6).

4.2 Motivation

Constructing a fine-grained network-wide debugger requires highly precise data-plane time synchronization across network devices to correlate events at the granularity of packets. However, given a synchronized data-plane, achieving fine-grained network-wide

recording is not straightforward with existing approaches:

Per-Packet Post-cards. NetSight's approach is to continuously send per-packet postcards from every switch in the network [83]. Such a proactive approach would not miss any information and help to debug every network fault. However, this approach is very expensive and does not scale to multi-petabit datacenter networks [142]. The main bottleneck is the limited network bandwidth available for exporting the recorded data from the switches (via management/mirror port) and the slow write speeds of cold data storage devices (magnetic disk drives) where the data would eventually end up. Other than scalability, this approach is also wasteful since network faults don't occur all the time, and the mean time between faults is observed to be in the order of several hours [143]. Hence, most of the recorded data is of no use.

In-Network Telemetry. A diametrically opposite approach would be to append telemetry information in the packet headers in data-plane [29, 86, 90]. These approaches use the destination end-hosts to store the appropriate per-packet telemetry information after extracting the headers. Although, it's very simple and scales easily to support collecting telemetry by storing across the end-hosts, it has the following disadvantages: 1) Increases the packet size by 10's of bytes[1]. 2) Need to handle packet segmentation issues. 3) Requires CPU/NIC resources to parse and store telemetry in each host. 3) Loses critical telemetry information upon packet losses, and 4) Finally, to debug network-wide issues, it has to collect telemetry information across end-hosts. To overcome these drawbacks, it's possible to collect telemetry information *after* a fault has been detected for the first time. Such a reactive approach misses important historical information required for finding the root cause, especially in the case of ephemeral faults [143]. In *SyNDB*, we tackle this challenge by leveraging the high-speed SRAM in switching ASICs to record packet-level telemetry at line-rate.

[1]For a 5-hop diameter network, INT increases packet size by at least 54 bytes

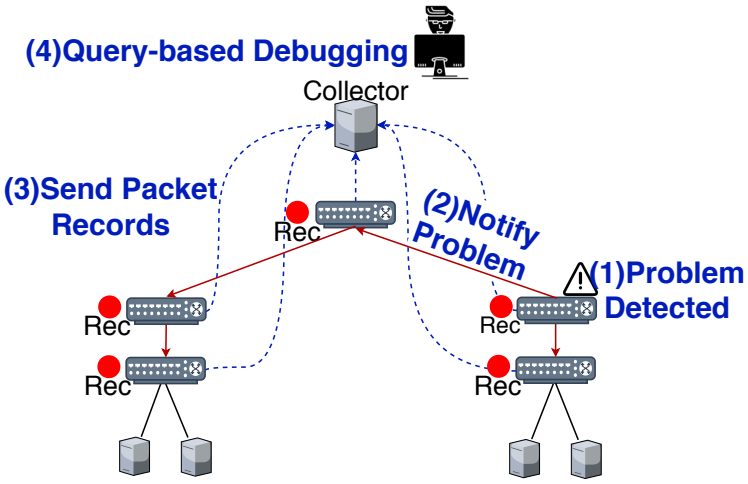


Figure 4.1: *SyNDB* Overview : Switches continuously maintain packet records, but send them to collector for debugging only upon detecting a problem

4.3 Design

SyNDB's goal is to provide recording of fine-grained (packet-level, nanosecond resolution) and network-wide telemetry information in a synchronized manner to enable debugging of network faults. The key idea is to leverage the switch data-plane as a fast temporal storage to do continuous recording of packet telemetry information over a moving time window (*recording window*). In addition, a data-plane time synchronization protocol such as DPTP [37] is used to implement a time synchronized data-plane.

We depict how *SyNDB* works in Figure 4.1. The switches continuously record the packet records in the data-plane. When no network fault is detected, the recording window moves ahead and the older telemetry data beyond the record time-length is discarded. Hence, this temporary buffer always maintain the recent history. When a configured trigger condition (e.g. packet loss or high latency) is observed in any switch (Step 1), high priority trigger packets are broadcast to all switches (Step 2). Upon reception of trigger packets, collection of p-records is performed by the switch control-plane. The collected p-records are forwarded to the collector, which stores the p-records in a database (Step 3). Once data collection is completed, the p-records collected both

before and after the fault, are available for debugging (Step 4). The debugging can be performed by operators using queries to answer questions like: *1) What are the packets observed during queue build-up?* 2) *Which path did they take?* 3) *Is there a load imbalance in any of the path?* 4) *Is the traffic synchronized?* 5) *Was there a network re-configuration?* and so on. We explain the detailed design in the following sections.

4.3.1 Data-plane Recording

The recording is performed in the switch data-plane and hence can record at line-rate. *SyNDB* consists of four main components viz. time synchronization, packet record generation, trigger initiation and packet record collection.

Time-synchronization. In *SyNDB*, global timing information is used to correlate packet records from multiple switches, and to construct a network-wide ordering of events. Additionally, clocks across switches are synchronized to a fine granularity to avoid timing inconsistencies in the packet records. We derive the necessary condition for consistency.

Let's consider two directly connected switches X and Y, with internal clocks C_X and C_Y respectively. We denote the synchronization error $(C_X - C_Y)$ between the internal clocks by T_{err}. Packet A is transmitted from switch X to switch Y. Packet A leaves switch X at $TimeOut_X$, and enters switch Y at $TimeIn_Y$, after a propagation delay D. $TimeOut_X$ corresponds to the time packet A enters the egress pipeline in switch X after queuing. This is the latest available time in the data-plane for a packet [1]. Similarly, $TimeIn_Y$ corresponds to the time packet A enters the ingress pipeline at switch Y. Consequently, propagation delay now becomes,

$$D = EgressDelay + DeparserDelay + MACDelay + WireDelay \tag{4.1}$$

To ensure *causal consistency* [144] in the packet record events, we should see packet A leave switch X before reaching switch Y. In short, $TimeOut_X$ should be less than

$TimeIn_Y$. This will be true if the synchronization error between the internal clocks is less than the propagation delay.

$$T_{err} < D \tag{4.2}$$

If the condition stated in this equation can be met, we can ensure consistency between any set of packets transmitted between two adjacent switches.

Previous works on network time synchronization have shown that the T_{err} between neighbouring switches is in the order of tens of ns [37, 68]. Additionally, real world data shows that D between two adjacent switches ranges between 360 ns to 1900 ns under varying traffic conditions [37]. Thus it is possible to achieve packet level consistency between adjacent switches, using current time synchronization techniques. The same principle can be extended between two switches separated by several hops in the network. In such cases, we observed that the increase in propagation delay is higher than the increase in T_{err} as number of hops increase, thus ensuring packet level consistency across multiple hops.

Packet Records. To generate packet-level events, we record information for each packet that enters a switch. We call this information p-record . To model a packet's arrival and exit, a p-record should contain a unique identifier for each packet, the time it enters a switch, and the time spent queuing. Hence, we define p-record to contain 3 basic fields: $[Hash, Time_{in}, Time_{queue}]$. $Hash$ is the hash value of the packet's headers (IPv4, TCP). It is used to identify and track a packet across the network. Although hash collisions are possible, we can resolve them using topology and timing information. $Time_{in}$ captures the time when the packet enters the switch. $Time_{queue}$ is the duration spent in the switch buffers. Additional fields are appended by the network operator to a p-record to capture statistics, such as queue depth, link utilization, forwarding table version, queue congestion, port counters, etc. An operator can specify such additional fields via a *SyNDB* configuration (§4.5).

After a p-record is generated in the data-plane for each packet, it can be stored either

in the switch's control-plane or data-plane. To store the p-record in the control-plane, it must be transferred over the PCIe interface and written to the DRAM. The control-plane DRAM provides significantly larger storage capacity, compared to the SRAM in the data-plane. However, the write speed is severely limited by the of PCIe throughput and DRAM latency. Even at low packet rates ($\sim$10 Mpps), the control-plane cannot keep up with the p-records generated. *SyNDB* therefore stores each p-record in a ring buffer array in the switch data-plane. This ring buffer array continuously maintains only the recent p-records for the recording time window.

Trigger Initiation. While *SyNDB* collects p-records for each packet in the data-plane, it requires a trigger to detect potential network faults. For example, a *congestion at a link* can be caused due to a wide variety of faults such as synchronized incast, load balancing policy issues, under-provisioned network, etc. We configure such trigger conditions in the data-plane. Once a trigger is hit, the p-records can be transmitted to the collector.

To initiate network-wide p-record collection, we create a trigger packet to be broadcast (with priority) to other switches through the data-plane. In *SyNDB*, when a trigger condition is hit, we clone the current packet and insert a new header type for trigger packet after stripping the payload and other headers. The trigger header consists of: 1) Trigger ID: Auto-incremental ID, 2) Trigger Type: unique type to classify trigger, and 3) Trigger Time: time when the trigger was hit.

The switches receiving the trigger packets further broadcast it to their neighboring switches and so on. Due to redundancies in trigger packet broadcast (multiple paths in data center topology), unless the network is partitioned, trigger packets reach the entire network. On receiving a trigger packet in the data-plane, the switch stops storing p-records in the ring buffer and instead uses a post-trigger fixed buffer for subsequent storage of p-records. If a switch had previously received a trigger packet with the same ID, then it's dropped. The ring buffer contains p-records of packets before the trigger

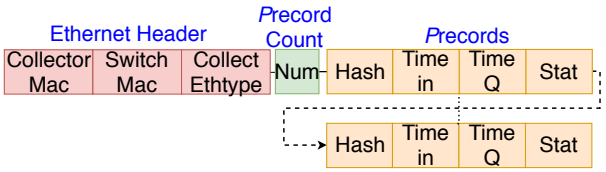


Figure 4.2: Packet Record (p-record) Header Format

condition and the post-trigger buffer contains that of packets after.

Conceptually, the size of the ring buffer that is needed to be stored for debugging purpose depends on the round-trip-time (RTT). In a data center context, recent measurements show that VM-to-VM RTTs vary between 5μs to 100μs [145]. For a packet rate of 1 Bpps, 1 million p-record entries could store up to 1 millisecond duration of history. This translates to 10's of historical RTTs available for debugging.

***p*-record Collection**. Upon receiving a trigger packet with a new trigger ID in the data-plane, collection of p-records has to be performed. Ideally, the collected p-records have to be sent to control-plane so that it doesn't interfere with data-plane traffic. Sending the p-records from data-plane to control-plane can be performed by packetizing p-records . An alternate approach like polling registers from CPU is very slow and expensive. The control-plane injects collection packets (or set up packet generators available in programmable switches) to flush the p-records from the data-plane to the collector. The collection packet can read only one p-record each time it traverses through the switch [1], before being forwarded to the switch control-plane. Consequently, each switch will generate a large number of packets, equal to the number of p-records. This would overwhelm the switch control-plane CPU due to the large serialization overhead incurred from the flood of packets (one per p-record) received from the data-plane. We mitigate this problem by coalescing multiple p-records into a single packet as shown in Figure 4.2. Each collection packet is recirculated multiple times in the data-plane to coalesce multiple p-records .

Once the number of p-records in a packet has reached a threshold (configured by

switch control-plane), the collection packet is forwarded to the switch control-plane. The control-plane further transfers the p-records and triggers to the collector via a slower management port using TCP. In case of continuous triggers, the control-plane can use DRAM/disk to buffer p-records . A collection cycle ends when all the p-records stored in the data-plane have been transmitted or sufficient time has elapsed since the trigger. The collection cycle repeats upon a new trigger hit. It is important to note that regular traffic forwarding is not disrupted during trigger and collection process. For cases when an additional trigger is hit during collection process of the previous trigger, a new trigger packet is generated and collection period is extended. Note that, since each p-record contains the global time, the collection process does not require any ordering (as needed by previous approaches [29, 83]). The pseudocode for recording, trigger and collection are shown below.

```
precordArray   : Register Buffer Array
writeIndex     : Current index to write
N              : Size of the ring buffer
POST_TRIG_SIZE : Size of buffer for post trigger
pwriteIndex    : Current index to write post trigger
Time_Now       : Current Global Time
-------------------------------------------------------
 Packet Record Logic
if packet is normalPacket:
    if collectInProgress == False:
        Store Hash, Time_now , Time_queue ,
            CustomStats in precordArray[writeIndex]
        writeIndex = (writeIndex + 1) % N
    else :
        if pwriteIndex < POST_TRIG_SIZE:
            Store Hash, Time_now , Time_queue ,
            CustomStats in precordArray[pwriteIndex]
            pwriteIndex = (pwriteIndex + 1)
```

```
    if triggerHit is True:
         clone(packet)


if packet is clonedPacket:
    add_header(trigger)
    remove_header(ipv4/tcp/udp)
    trigger.time = Time_now
    trigger.id   = triggerId
    trigger.type = triggerType
    recirculate()
```

Trigger Packet Logic

```
if packet is triggerPacket:
    if trigger.id != lastSeenId:
        collectInProgress = True
        broadcast()
    else:
        drop()
```

Collection Packet Logic

```
if packet is collectPacket:
    if collectPacket.entries < MAX_ENTRIES_PKT:
        p-record =  precordArray[readIndex]
        readIndex = (readIndex + 1) % N
        add_header(p-record)
        collectPacket.entries++
        recirculate()
    else:
        l2fwd_to_collector()
```

4.4 *S*yNDB Debugger

The collector composes of multiple servers which store, analyze, and replay the p-record. We store the p-records in a relational database (RDBMS) which allows the p-records to

be queried using SQL. The collector also stores information regarding the trigger events, network topology, and position of switches within the topology. Before storing p-records in database, it performs hash collision removal using topology and timing information. We organize these tables in a relational database as shown in Figure 4.3.

1. Packet records: This table stores basic and custom fields within each p-record. Each p-record stores: 1) Switch ID, 2) Packet Hash, 3) Time In, 4) Time Queued, 5) Time Out and, 6) Operator-specified statistics.

2. Triggers: This table stores information regarding each trigger event. Each trigger event stores: 1) Trigger Type, 2) Trigger Time, and 3) Trigger Origin Switch. This enables *SyNDB* to classify network faults based on the trigger type.

3. Links: This table stores the topology of the data center, as specified by the network operator instead of inferring from the Packetrecords table, because it is impossible to guarantee non-zero utilization of all links in the topology. Each link stores the endpoints and the link capacity.

4. Switches: This table stores the position of a switch in the topology, for example, ToR, Aggregation, Core etc. The position can be customized by the network operator based on their topology.

To determine the root cause of a network fault, we use SQL queries on the above tables. For example, in the case of an incast, culprit packets and their routes can be obtained by combining information from Packetrecords, Triggers, and Links tables. The output of these queries can also be used to replay or build dashboards using tools [146–148], which are beyond the scope of this work (refer §4.8).

We list some example queries below (scenarios in §4.7.3):

1) List packet events in the order of their occurrence:

```
Select * FROM packetrecords ORDERBY time_in ASC;
```

2) List events in the switch which triggered a fault:

```
Select * FROM packetrecords JOIN triggers
        ON packetrecords.switch = triggers.switch;
```

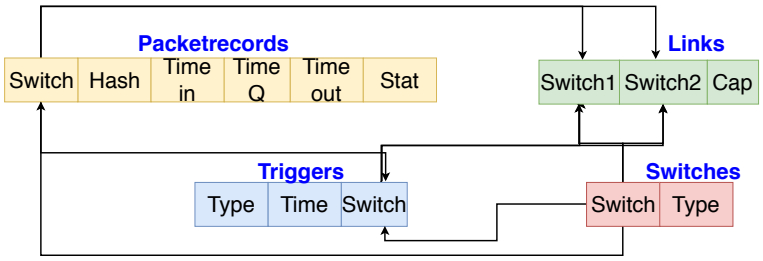


Figure 4.3: *SyNDB* Debugger Database Schema

3) Check to see where the problem-causing packets came from and the path they took:

```
Select * FROM (packetrecords as P) WHERE hash
        IN (select hash from packetrecords JOIN triggers
        ON p.switch = triggers.switch
        AND p.time_in < triggers.time) ORDERBY time_in ASC;
```

4.5 *SyNDB* Configuration

SyNDB provides an interface for defining *p*-records and triggers which can be used along with any P4 program to specify the configuration of *p*-records and trigger conditions. The programmer configures the following parameters in *SyNDB*: 1) the network statistics (fields) to be collected in *p*-records , 2) the number of *p*-record entries to be collected, and 3) the trigger (fault) conditions to initiate a collection of *p*-records . The fields specified in the configuration could be: 1) switch-provided metadata (queue depth, ingress port, egress port), 2) packet header data (flow-id), and 3) data that is computed and stored in user metadata by the programmer (link_utilization, counters, EWMA). The *SyNDB* configuration is compiled, then translated to P4 and finally embedded with the original switch P4 program. Figure 4.4 shows the interfaces to define *p*-records and trigger conditions.

A *p*-record defines a list of field_lists. Each field_list contains one or more (metadata)

```
precord {
  fields {
    field_list_1;
    field_list_2;
    ...
  }
  default_field : field_list_{x};
  entries       : {y};
  post_trigger  : {z};
}
trigger {
  conditions {
    condition_1;
    condition_2;
    ...
  }
}
```

Figure 4.4: *SyNDB* Configuration Syntax

fields [149] from the Packet Header Vector (PHV) [1] supported by the switch architecture and defined in the user's P4 program. A "default_field" list is specified by the programmer which is the active field_list to be included in each p-record . The current *active* field_list can be changed during runtime. The "entries" refer to the total history of p-records to be collected (*recording window*) while "post_trigger" refers to the number of p-records to be collected after a trigger occurs. Finally, the user needs to configure a list of triggers, which are predicates operating on header/metadata fields and can be changed during runtime. For example, $link\_utilization > 90$.

***SyNDB*-Runtime**. In practice, there is a need to make changes to the p-record structure and trigger configurations while *SyNDB* is running without the need to recompile and load a new P4 program. *SyNDB*-Runtime facilitates changing the configuration in following ways: 1) Adding a new field_list or editing the active field_list, and 2) Adding/removing trigger conditions. Note that these changes are restricted to the available PHV contents in the data-plane as there is no modification to the underlying P4 program.

When the *SyNDB* configuration is compiled, the compiler enumerates all the PHV

contents (packet headers, switch and user-defined metadata) of the P4 program. It then creates template tables with actions for each PHV container to be stored in the *p*-record . This facilitates the runtime to dynamically add/remove the fields to be recorded in each *p*-record . The fields can be flow 5-tuple or TCP sequence number which are part of packet headers, or ingress_port, queue_depth, etc. which are part of the switch metadata. The field to be added cannot be a statistic (e.g. EWMA) that is not defined or a packet header that is not parsed by the already compiled P4 program. Since PHV contents are limited, enumerating and storing them in actions do not significantly increase data-plane resource consumption. The maximum bytes in a *p*-record and the number of *p*-record entries (*recording window*) is fixed at compile-time based on the available hardware resources (stateful ALUs and SRAM). To facilitate addition/removal of trigger conditions at runtime, *SyNDB* configuration compiler uses similar enumeration technique and generates range-based match-action tables. Additionally, *SyNDB*-Runtime updates the collector each time the *SyNDB* configuration is changed, to ensure that *p*-records are stored correctly.

Figure 4.5 summarizes the *SyNDB* workflow. A network programmer configures the statistics to be recorded and the fault triggers. The configuration can be continuously tweaked to suit the statistics that the programmer wants to keep an eye on using *SyNDB*-Runtime.

4.6 Implementation

We have implemented *SyNDB* on Barefoot Tofino [58] switches using P4 (∼1900 LoC). We use DPTP [37] for time synchronization between switches. We use DPTP since it is implemented on PSA architecture and provides a global timestamp in the data-plane. We store the baseline contents of *p*-record in both ingress and egress pipeline of the PSA [1] architecture. Ingress pipeline maintains the write_index of the ring buffer array

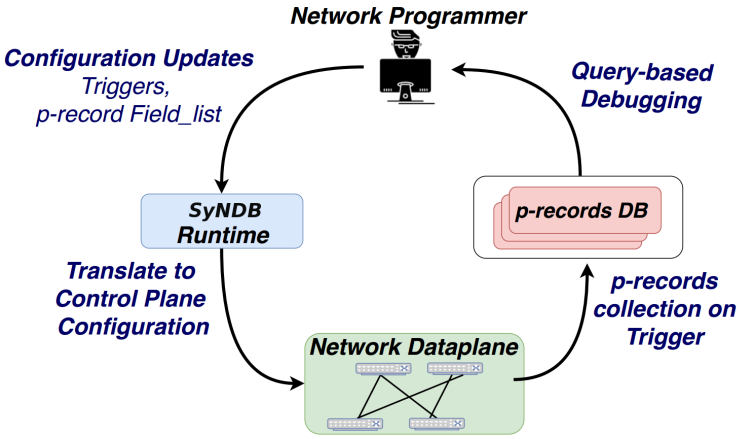


Figure 4.5: *SyNDB* from Programmer's perspective

upon a packet arrival and stores the $Hash$ and $Time_{in}$. Egress pipeline stores $Time_{queue}$ and custom field_list to be captured. $Hash$ is a 32-bit hash of a packet's headers. Each switch in the network applies the same 32-bit hash algorithm on a packet to store this information in a p-record. For a long lived TCP connection, two packets may have the same the hash value in their p-record due to TCP seq number wrap around. *SyNDB* uses the time cut-off (e.g. 1 second) and topology information to distinguish, since a packet lasts for less than a second within the network and a TCP seq number cannot wrap around within a second.

$Time_{in}$ is a 32-bit global timestamp (at nanosecond granularity) of the packet when it enters ingress pipeline of the switch. On the other hand, $Time_{queue}$ is a 24-bit field which captures time spent by the packet in the Traffic Manager queue. 24-bit $Time_{queue}$ allows capturing up to 16 ms of queuing delay which is much larger than normal queuing times observed in the switches [37].

We have implemented the compiler for *SyNDB* configuration using Rust (∼4000 LoC). It takes as input the configuration and the switch P4 program, and generates a P4 code that implements p-record storage and collection logic. p-record storage and trigger conditions are executed using stateful ALUs. Additionally, the runtime environment (implemented in Python) accepts commands to modify configuration such as: 1)

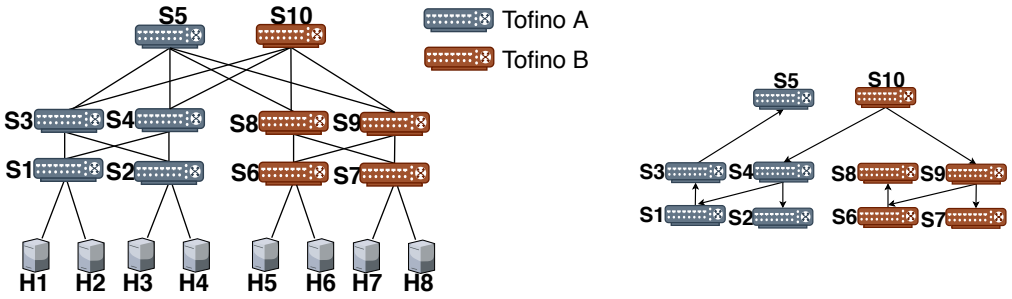
changing the active field_list, 2) adding new field_list, and 3) adding/removing trigger conditions. These configurations translate to control-plane configuration updates of the composed switch P4 program.

Finally, we implement the collector using `n2disk` utility (with *PF_RING* [150]) to store collection packets as PCAP files in the local disk. Additionally, we implement a Python program to parse the PCAP files and store individual *p*-records in a MySQL database. The collector also takes as input the *SyNDB* configuration to parse and name the custom statistics fields correctly from the PCAP files. Everytime the active active field_list is modified through the *SyNDB* runtime, the update is also passed on to the collector.

4.7 Evaluation

We evaluate *SyNDB* using 4 physical servers, and 2 Barefoot Tofino Wedge100BF-32X switches [151]. The servers and the switches are virtualized to create a fat-tree topology [152] as shown in Figure 4.6(a). Switches S1 to S5 are virtualized on the first switch ("Tofino A") using 10G loopback links. Similarly, S6 to S10 are virtualized on the other switch ("Tofino B"). Additionally, we synchronize each virtual switch's data-plane to S10 using DPTP (see Figure 4.6(b)). We create a register array of 50K entries in SRAM to implement the ring buffer for storing *p*-records on switches A and B. Each virtual switch stores its *p*-records in a slice of this register array. For example, S1 uses 0-10K, S2 uses 10K-20K and so on. We also provision 5K entries per virtual switch for capturing *p*-records after a trigger condition is hit. Figure 4.7 shows the *SyNDB* configuration that we use in the evaluation. Each *p*-record entry is 16 bytes in size.

As a result, 10K *p*-record entries, occupying 160 KB of SRAM, are sufficient to capture 1 ms packet history at 10 Mpps packet rate for each virtual switch. However, we show in §4.7.2 that there is sufficient SRAM available to support 1 ms packet history for



a. Evaluation Topology

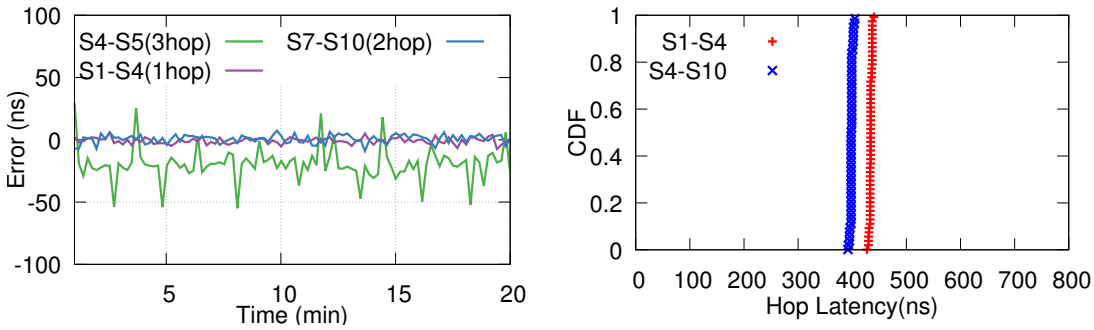
b. Network-level DAG used for DPTP synchronization

Figure 4.6: Evaluation Topology Testbed

```
field_list SyNDB_verification {
  meta.packet_sequence;
}
field_list SyNDB_scenario {
  meta.ingress_port;
  meta.link_utilization;
  meta.drop_counter;
}
precord {
  fields {
    SyNDB_verification;
    SyNDB_scenario;
  }
  default_field : SyNDB_scenario;
  entries       : 10000;
  post_trigger  : 5000;
}
trigger {
  conditions {
    meta.packet_sequence == 5000;
    meta.drop_rate > 250;
    meta.time_queue > 10000;
  }
}
```

Figure 4.7: *SyNDB* Configuration for Evaluation

state-of-the-art pipeline speeds of 1 Bpps. Note that, we do not modify the application in anyway to support *SyNDB*.



a. DPTP Synchronization error

b. Hop Latency between switches

Figure 4.8: DPTP Synchronization error

4.7.1 Design Verification

We verify *SyNDB*'s design by evaluating its ability to capture network information along three dimensions: time, scale and granularity.

Time. To ensure p-records captured by *SyNDB* are consistent, the time synchronization error should be less than the propagation delay between adjacent switches (equation (4.2) from §4.3). We plot the DPTP time-synchronization error between switches 1-hop, 2-hop, and 3-hops away in Figure 4.8(a). The synchronization error between switches separated by 3-hops is less than 50 ns. Additionally, the synchronization error is significantly lower, under 10 ns, in case of switches 2-hops apart. Next, in Figure 4.8(b) we evaluate the propagation delay between two pairs of switches S1-S4, virtualized using the same physical switch, and S4-S10, virtualized using different physical switch. We observe that the propagation delay between two switches varies between 400-450 ns. The difference in delay (≈30 ns) between S4-S10 and S1-S4, is attributed to the effect of clock-drift between the physical switches. Thus, synchronization error is much lower than the propagation delay between two switches and hence captured p-records should be consistent with the ground truth.

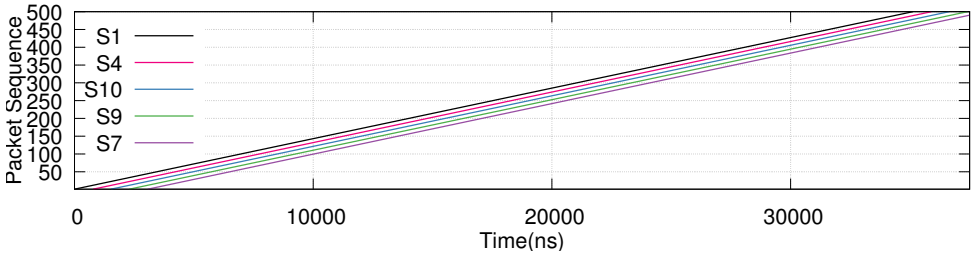
Scale and Granularity. We evaluate *SyNDB*'s ability to capture information at a network-wide scale and packet-level granularity. For this, we use a Constant-Bit Rate (CBR) of 10 Mpps (limited due to 10G host links), with each packet annotated with

a sequence number. The packets are sent from S1 to S7 along the path S1-S4-S10-S9-S7. At each switch, we record the "packet_sequence" number of each packet. Next, we configure S1 to generate a trigger and broadcast the trigger packet after receiving 5000 packets. After receiving a trigger packet, each switch sends the *p*-records to the controller where we plot the arrival time of packets based on their sequence number as shown in Figure 4.9.

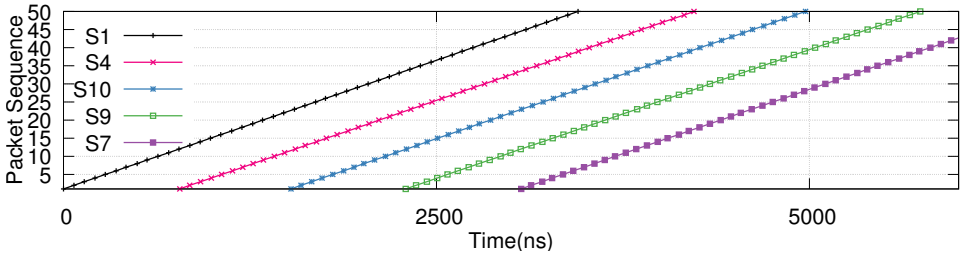
We show the time of arrival for a 500 packet sequence at each switch on the path in Figure 4.9(a) for CBR traffic. Figure 4.9(b) provides us a closer look for the corresponding traffic patterns by plotting a 50 packet sequence. We observe that for every packet, it is recorded in the next switch only *after* it has left the previous switch. Furthermore, the timestamps increase linearly as expected. This behaviour confirms that records captured by *SyNDB* across all the switches in the network are consistent and at the expected intervals. Furthermore, *SyNDB* is able to capture *p*-records for all 5000 packets on all switches traversed by the packets through the network. This verifies *SyNDB*'s ability to capture network information at a packet-level granularity.

Correlating *p*-records. *SyNDB* helps to identify the root cause for network faults by accurately presenting the ordered events in the network leading to the failure. Therefore, the ability to track the progression of events across a packet's end-to-end route using *p*-records, is not just advantageous but crucial for identifying the cause. However, it is impossible to collect the same set of *p*-records from all switches. This is because *SyNDB* stores *p*-records in a ring buffer, where old *p*-records may be discarded to accommodate incoming packets during the time the trigger packets travel across the network. Therefore, the difference in *p*-records collected depends on the distance from the origin trigger switch, the number of *p*-record entries, and network utilization. In this evaluation, we study the correlation between the *p*-records seen by the collector after a trigger is generated.

We use a similar setup, with 10 Mpps CBR traffic transmitted along the path S1-



a. 500 Packet Sequences(CBR)



b. Zoom of 50 Packet Sequence(CBR)

Figure 4.9: Validation of Traffic pattern observed at different switches for 10Mpps CBR traffic

S4-S10-S9-S7 with each switch storing up to 10K *p*-record entries. In this case, we configure S7 to generate a trigger after it has received 10,000 packets. Once the trigger is generated at S7, it is broadcast to other switches in the network to initiate *p*-record collection. Based on the *p*-records available at the collector, in Figure 4.10 we plot the percentage of common *p*-records seen by other switches compared to those seen by S7 (the switch that initiates the trigger). We observe that with increasing number of hops from S7, the percentage of correlating *p*-records reduces to 99.44%. This implies that *SyNDB* can accurately record the progression of nearly all packets leading up to the fault.

4.7.2 *SyNDB* Overhead

SRAM Overhead. We estimate the total historical duration of *p*-records and total amount of SRAM consumption based on a *p*-record size of: 11 bytes (baseline *p*-record), 16 bytes (our evaluation configuration), 24 bytes (baseline *p*-record + 5-tuple flow ID), and 32 bytes (baseline *p*-record + 5-tuple flow ID + 8-bytes switch metadata) in

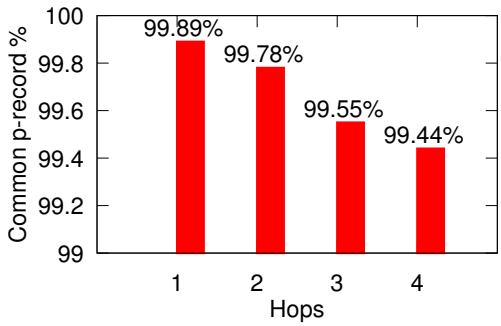
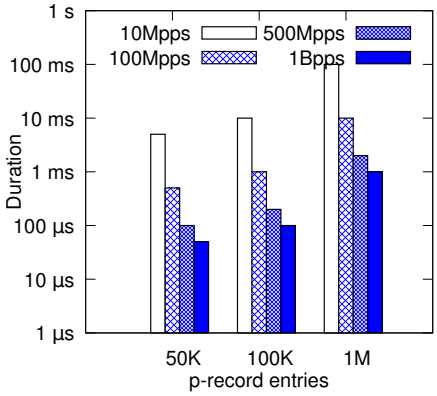
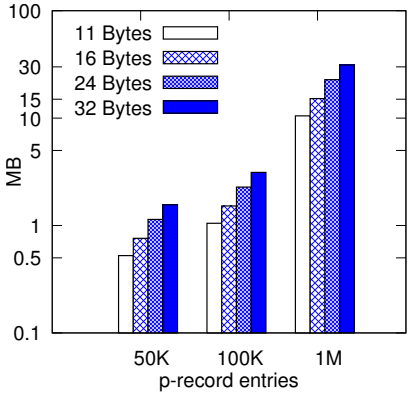


Figure 4.10: Percentage of common p-records



a. Duration of packet histories captured



b. SRAM consumption for different p-record size

Figure 4.11: SRAM consumption and packet record histories for different packet rates

Figure 4.11. At packet rates of 1 Bpps, *SyNDB* consumes about 22 MB of SRAM to record 1 ms of packet histories using 1 million p-records of size 24-bytes in the data-plane. This can be easily accommodated by latest switching ASICs [116–118] which contain SRAM greater than 100 MB. Recent studies [140] have observed high utilization only across few switch ports during congestion events. Thus the pipeline utilization has always been lower than its capacity. At lower packet rates like <500 Mpps, it is possible to store packet records for several milliseconds. The programmer can trade-off between the number of p-records and the size of each p-record depending on the memory budget. Although 5-tuple flow ID can be stored in a p-record , to keep the p-record size small, such static per-flow data could be temporally stored only in the ToR switches and

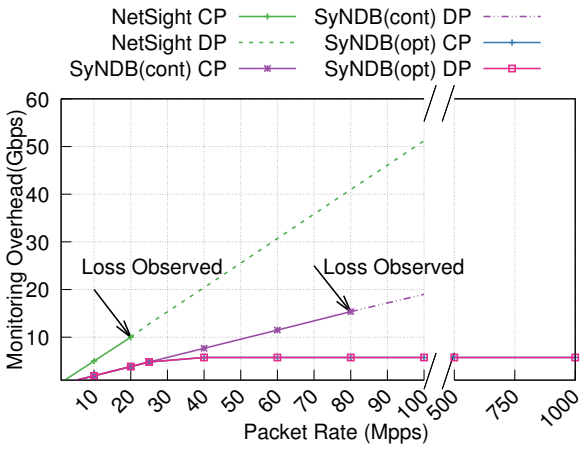


Figure 4.12: Traffic overhead comparison with NetSight

collected when there is a trigger.

Collection Overhead. We measure the additional overhead to collect the p-records and ship them to the collector. To perform collection, the switch control-plane injects empty packets (typically 1 pkt/1μs). With 64 p-records per packet, collection of 10000 p-records requires 157 packets of 1060 bytes each and takes a total time of 230μs. We calculate the pipeline overhead by (total number of packets injected $\times$ p-records per packet) / total time taken for collection. The overall pipeline overhead is about 43 Mpps (4 % of the overall pipeline processing) and the switch bandwidth consumption is about 5.7 Gbps during collection. This bandwidth consumption is localized to the re-circulation port and the PCIe channel, thus not affecting regular data-plane traffic. We observed that the total time and size of packetized p-records linearly increases with the number of p-records and takes about 15.7 ms to collect 1M p-records . Hence *SyNDB* can support upto ≈66 triggers/sec at 1 Bpps traffic rate, and ≈ 600 triggers/sec at 100 Mpps. This is enough to handle network incidents which are usually separated by hours [16]. The collection packets can be injected from control-plane or generated in data-plane using packet generators (available in programmable switches) for faster collection, subject to available PCIe bandwidth.

Comparison with Other Debugging Tools. Next, we compare the overhead of

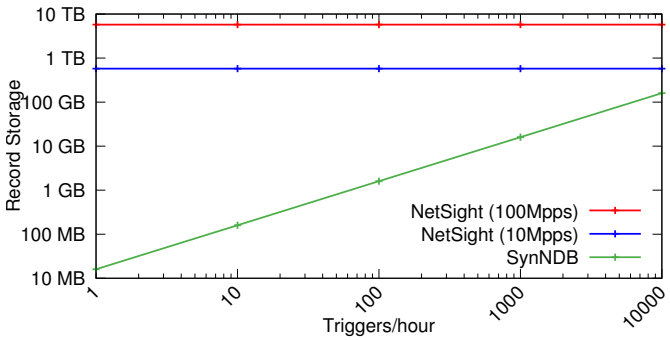


Figure 4.13: Storage overhead comparison with NetSight

SyNDB to that of NetSight [83], which is the closest comparable work based on individual packet post-cards. NetSight creates a post-card by stripping the packet payload, and attaching switch ID and ingress port to the post-card. Since *SyNDB* sends p-records via the control-plane, for fairness of comparison we configure NetSight to send out the post-cards via the control-plane channel. Each packet post-card is of 64-byte due to minimum Ethernet frame size. We compare NetSight with two versions of *SyNDB*: 1) *SyNDB* (cont): Triggers occur continuously. In this version, collection packets are generated in the data-plane continuously to perform continuous collection of packet-records. 2) *SyNDB* (opt): Triggers occur at optimal rate, i.e. exactly after a previous trigger collection has finished (∼66 triggers/sec at 1 Bpps).

We plot the traffic overhead of these approaches for increasing packet rates in Figure 4.12. In case of NetSight, we denote the total traffic sent from data-plane to control-plane as NetSight-DP, and total traffic received by control-plane as NetSight-CP. With NetSight, a post-card is sent to the control-plane for each packet in the data-plane, thus sending same packet-rates to the control-plane. Beyond 20 Mpps, the control-plane fails to receive all post-cards since the control-plane CPU cannot handle the high packet-rate.

With *SyNDB* (cont), we observe that the control-plane (*SyNDB* (cont)-CP) receives p-records until 80 Mpps of data-plane traffic, and fails to scale beyond that due to CPU overhead in handling larger packet sizes (1060 bytes per packet). At this point, the control-plane experiences a packet rate of about 3 Mpps even though the data-plane

traffic is 80 Mpps. This is because *SyNDB* coalesces multiple *p*-records into a single packet. When triggers occur at an optimal rate with *SyNDB* (opt), we observe a consistent bandwidth of 5.7 Gbps (at *SyNDB* (opt)-DP and *SyNDB* (opt)-CP) with no *p*-record loss even with the data-plane packet rate of 1 Bpps. Also, if the mirror port is used to export post-cards/*p*-records , this result holds qualitatively as *SyNDB*'s packet I/O overhead is lower due to coalescing.

Next, we compare the overall storage overhead incurred at the collector from a single switch for *SyNDB* compared to NetSight. Even though NetSight does not scale beyond 20 Mpps packet rate, we assume the availability of infinite resources to scale NetSight. *SyNDB* performs collection only upon fault triggers while NetSight performs collection throughout the network operation. In Figure 4.13, we plot the overall storage incurred for an hour of network operation with increasing number of triggers/hr. We assume both NetSight and *SyNDB* store 16-byte post-cards/*p*-records per packet. Irrespective of the frequency of faults, NetSight collects about 500 GB and 5 TB of data per hour from a single switch at 10 Mpps and 100 Mpps packet rates, respectively. *SyNDB* on the other hand collects only 160 GB per hour for 10000 triggers/hr. Here *SyNDB*'s exported data rate is agnostic to the packet rate since it exports *p*-records from a fixed size ring buffer (Figure 4.7) upon a trigger.

Switch Resource Overhead. We evaluate the total hardware resource consumption of *SyNDB* (with configuration shown in Figure 4.7) compared to the baseline switch.p4 [134]. switch.p4 is a baseline P4 program that implements various common networking features applicable to a typical datacenter switch. As we implement *SyNDB* along with DPTP, we show the total resources consumed by all the components (switch.p4, DPTP and *SyNDB*) in Table 4.1. The majority of resources required for *SyNDB* arise from the need to store *p*-records in the dataplane. We observe that *SyNDB* consumes 34% of stateful ALUs and 9% of SRAM to store *p*-records and trigger conditions. Thus, *SyNDB* can be implemented on top of switch.p4 in programmable

Table 4.1: Hardware resource consumption of *SyNDB* compared to the baseline switch.p4

Resource	switch.p4	DPTP	*SyNDB*	Combined
SRAM	29.79%	6.25%	9.06%	45.1%
Match Crossbar	50.13%	4.62 %	9.83%	64.58%
TCAM	28.47%	0.0%	1.04%	29.51%
Stateful ALUs	15.63%	15.63%	34.37%	65.63%
Hash Bits	32.35%	3.99%	10.49%	46.83%
VLIW Actions	34.64%	4.43%	4.16%	43.23%

switch ASICs available today.

4.7.3 Network Debugging Scenarios

In this section, we show how *SyNDB* can be used to debug 2 kinds of common network faults [16] (1) microburst caused by incast traffic and (2) congestion caused by multi-path routing. We use the same configuration setup as defined in Figure 4.7 . Each p-record is configured to contain the custom "field_list: SyNDB_scenario". It contains three metrics : 1) Ingress Port 2) Link Utilization and 3) Drop Counter.

Ingress port of a packet is provided by switch meta-data. Link utilization is calculated at fine time-scales ($10\,\mu$s) in the data-plane using a low-pass-filter. Drop counter is calculated based on packets which missed the forwarding table. Additionally, we configure *SyNDB* to perform collection of p-records based on two triggers : 1) High Drop Rate, and 2) High Queuing Delay. In each of the following case studies, we generate data and control traffic to emulate the corresponding network faults.

4.7.3.1 Case 1: Microbursts

Microburst is common in data-centers in which congestion is caused by a short burst of packets lasting for at most a few hundred microseconds [140, 153]. Traffic bursts occur due to various reasons like application (e.g. DFS, MapReduce), TCP behavior and also NIC (Segmentation offload, receive offload) [154]. The complex interactions and traffic patterns make microbursts debugging extremely complicated. In this experiment,

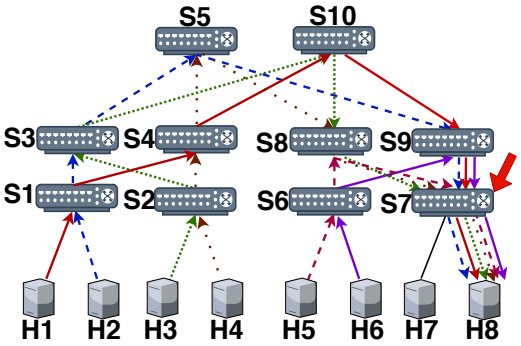
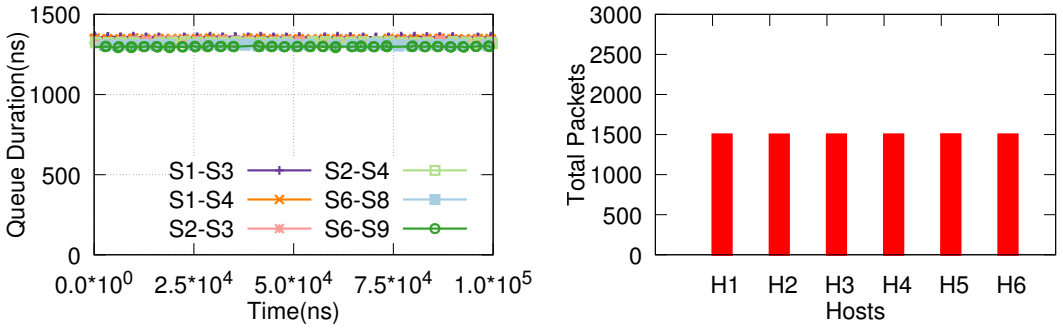


Figure 4.14: Microburst at S7 due to synchronized fan-in



a. Leaf Links queuing delay

b. Packet Distribution from Hosts

Figure 4.15: Queueing at ToR switches and packet distribution observed before the congestion at S7

we demonstrate how packet-level time-synchronized network-wide recording of *SyNDB* helps in understanding the traffic patterns causing microbursts.

We consider the commonly known fan-in traffic pattern of datacenter networks exhibited by applications such as MapReduce and Distributed File System (DFS). This is a incast traffic pattern where many sources transmit to a small number of destinations within a short time window. These short bursts of traffic increase the queuing delay at microsecond time-scales. The challenge in identifying the root cause of such incast traffic is that many sources contribute to the total traffic and the burst occurs only for a very short time.

We setup the experiment with host H1 to H6 sending data to H8 as shown in Figure 4.14. Each host sends a burst of 10 1500-byte packets at an average rate of 1Gbps to H8 via ToR switch S7. All links have capacity of 10 Gbps. In the experiments, due to

random time delays in the rate-control mechanisms running on the hosts, transmissions from different hosts synchronized over short periods causing sudden spikes in queuing delays on S7.

Approaches such as NetSight, INT and SpeedLight can detect the microbursts. However, since NetSight and INT look only at per-flow statistics, they will not be able to identify the root cause by correlating information across flows. Although SpeedLight can correlate across flows, it works at coarse granularities. Figure 4.15(a) shows the queuing delay for all the ToR links and Figure 4.15(b) shows the packet distribution from hosts. Both the queuing delays and packet distribution do no show any anomaly.

On the other hand, with *SyNDB*, query on network-wide and time sensitive behavior is possible. To determine if the issue of microburst is caused by synchronized fan-in traffic, a query of the queuing delay at S7 together with the packet arrival information at the ToR switches before the microburst detection can be performed at the collector as shown below:

```
SELECT switch, ingress_port, time_in  FROM packetrecords
    WHERE hash IN (SELECT hash FROM packetrecords AS A
    JOIN triggers as T  ON (A.time_in < T.time AND A.switch = T.switch))
    AND switch IN (SELECT switch FROM switches WHERE type = "tor");

SELECT time_queue FROM packetrecords where switch=7;
```

Listing 4.1: Query to list the packet arrival times at ToR switch ports and queuing delay at S7

The answer to the query is shown in Figure 4.16. The top plot in the figure illustrates the queue buildup over time. When we correlate the packet arrivals from different hosts before the bursts occurred, we see that the packets that make up the bursts are transmitted by hosts H1 to H6 at about the same time (see arrow in Figure 4.16). In this way, the root cause of incast from H1 to H6 can be determined.

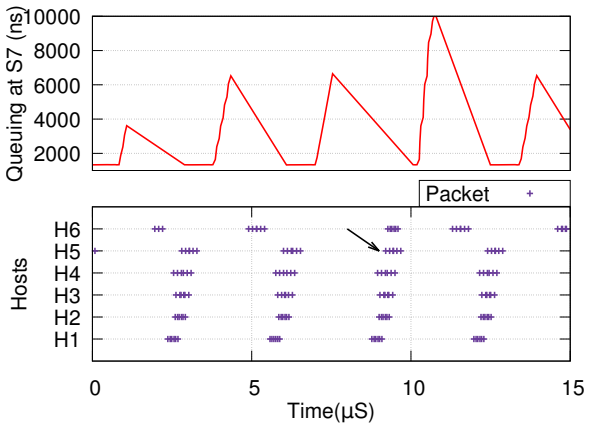


Figure 4.16: Queuing at S7 and Packets arrival sequence at ToR Switches leading to microburst at S7

4.7.3.2 Case 2: Transient Load Balancing Issues

Modern datacenter topologies such as fat-tree provide redundant paths between a source-destination pair. ECMP [138, 139, 155] is a common load balancing policy for handling multi-path routes. However, it has a lot of inefficiencies in distributing the load evenly [138, 139]. As a result, it has been observed that a subset of core-links regularly experience congestion while there is spare capacity on other links [156].

In this scenario, we setup ECMP based load balancing. Each switch calculates the hash of the 5-tuple and redirects the flow via one out of the two links. We experiment with a variety of combinations of 5-tuple flows, and use a set of combinations which can lead to load imbalance in the network. In one such combination, S9-S7 is congested, even though spare capacity is available at S8-S7. We create multiple flows in the network originating from H1 to H6 with the destination as H7 and H8 (Figure 4.17). The traffic (containing faulty combination) is sent at short bursts, with an overall throughput of 1 Gbps per flow. The load imbalance happens when both the core switches (S5 and S10) direct too many flows to S9, resulting in congestion on the S9-S7 link.

With only the congestion indication, it is be difficult to determine the root cause.

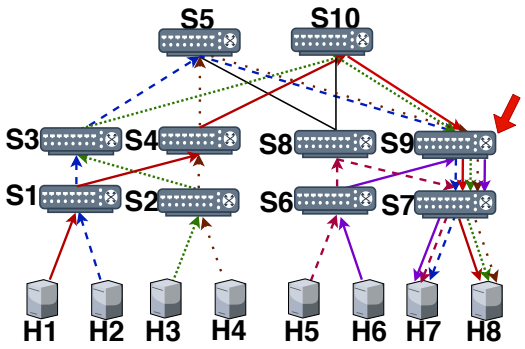


Figure 4.17: Congestion at S9 due to Link Load Balancing problem

To determine if load imbalance is the root cause, one would have to observe the queuing duration and link utilization of various links at the same time. These network metrics are not available with both NetSight and INT. SpeedLight [88] can measure only coarse-grained link utilization (several μs). With *SyNDB*, we can plot the utilization of the links measured at the same time at packet-level granularity using the query shown in Listing 2.

```
SELECT switch1, switch2, link_utilization*8, time_queue
    FROM (SELECT switch1, switch2 FROM links WHERE (switch1
    IN (select switch FROM switches  WHERE type !="tor")  AND switch2
    IN (SELECT switch FROM switches WHERE type !="tor"))) AS L
    JOIN (SELECT * FROM packetrecords) AS A
    JOIN (SELECT * FROM packetrecords) AS B
    ON (A.hash = B.hash AND A.switch = L.switch1 AND B.switch = L.switch2);

SELECT forwarding_rule_ver FROM packetrecords WHERE switch=10;
```

Listing 4.2: Query for link utilization and queue depths

The result is shown in Figure 4.18. We can observe that there is high link utilization at S9-S7 while link S8-S7 sees no significant utilization. Furthermore, the congestion trigger at the link S9-S7 is preceded by higher than normal link utilization in links S5-S9

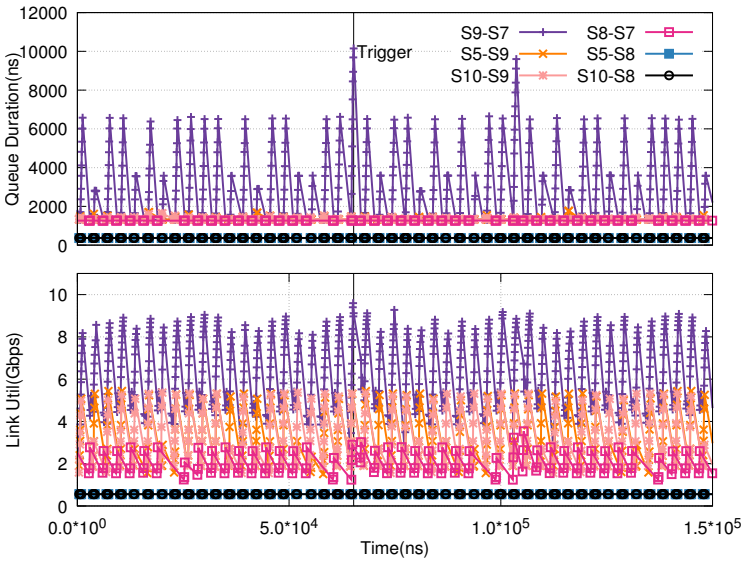


Figure 4.18: Link Utilization and Queuing delays observed at the core links points to load imbalance

and S10-S9. Thus, the load distribution from the core switches (S5 and S10) to S8 and S9 is heavily skewed, with most flows being routed via S9 during some time intervals. Based on this observation, one can infer that the root cause for the congestion at the link S9-S7 is the load imbalance cause by the load balancing scheme.

4.7.3.3 Case 3: Dynamic Network Configuration

Networks operate in a dynamic environment where operators frequently modify forwarding rules and link weights to perform tasks from fault management, to traffic engineering, to planned maintenance [157]. However, dynamic network configurations are complex and prone to error. For example, updating the route for a flow(s) can lead to unexpected packet drops if the updates are not applied consistently or efficiently [135, 136]. In this case study, we pick two categories of network updates and use *SyNDB* to identify the offending update. We add forward_rule_version to the *field_list SyNDB_scenario*. We assume that each forwarding rule indicates a version number and route based on destination MAC address as shown below.

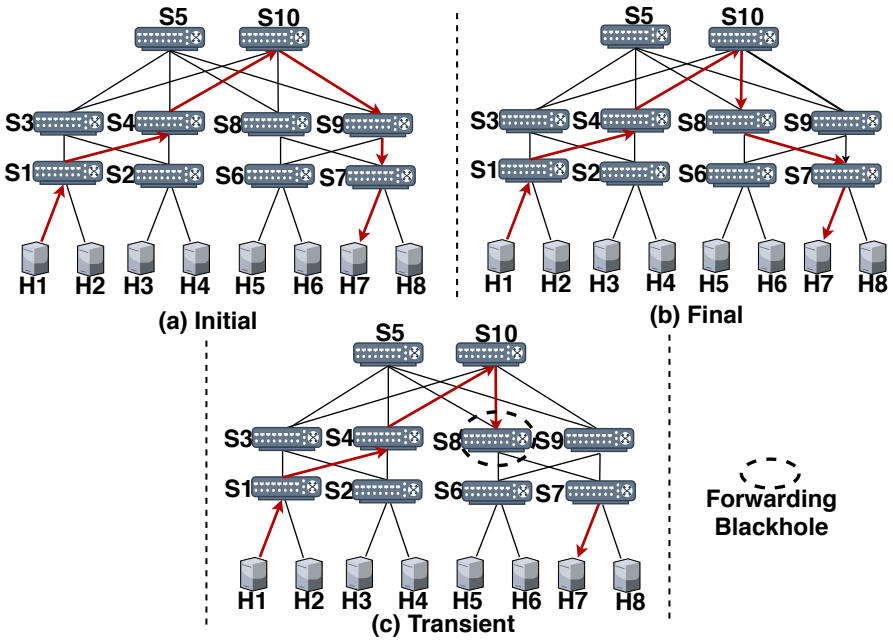
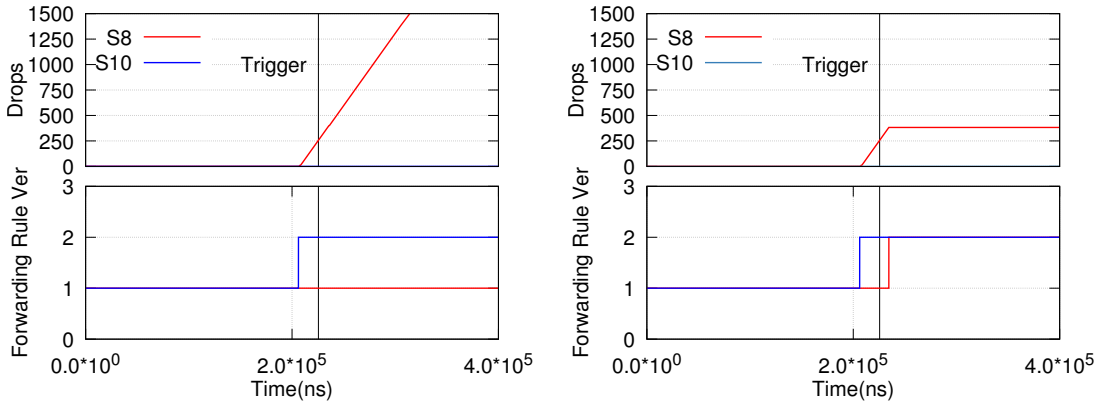


Figure 4.19: Network Update Scenario causing a Forwarding Blackhole at S8

```
table_add forward
    send_to_port ethernet_dstAddr <dstMac> =>
    output_port <num> entry_ver <num>
```

Forwarding Blackhole. A forwarding blackhole occurs when an out-of-order execution of a network update gives rise to non-deterministic network behavior leading to packet loss temporarily/permanently [158]. We evaluate forwarding blackholes using a setup shown in Figure 4.19. Figures 4.19(a) and (b) depict the initial and final state of the network after the updated route. The routing of a flow from H1 to H7 is updated by rerouting traffic from S10 to S8. However, transitioning from configuration (a) to (b) requires updates to both S10 and S8.

In this network update, first S8 needs to add a new rule to route the flow and then S10 needs to update the policy to route flow from S9 to S8 instead. At this point, there are two scenarios which can cause a forwarding blackhole. In the first scenario, S8 never receives the update message and hence all packets routed from S10 are dropped



a. Permanent Blackhole b. Transient Blackhole

Figure 4.20: Forwarding rule updates (S8 and S10) leading to drops at S8

as there is no rule for handling these packets. This is called a permanent blackhole. In the second scenario, S8 receives the update message albeit after a short delay. This is called a temporary blackhole where packets are dropped during this short delay before the new rule is added. In both scenarios, a trigger will be raised at S8 as the number of packet drops increases beyond a threshold.

Based on the packet drop trigger, the root cause can be mostly attributed to blackholes or link failure. To check if a network rule update has occurred before the drops, the forwarding rule version and the drop counters can be queried as below in Listing 1 and it is plotted in Figure 4.20.

```
SELECT forwarding_rule_ver, drop_counter FROM packetrecords
    WHERE switch=8 OR switch=10;
```

Listing 4.3: Query for correlating network update with drops

Observe that the forwarding rule update in S10 corresponds with simultaneous packet drops in S8. Additionally, the number of packet drops continues to increase over time. This leads to conclude the root cause as a permanent blackhole, caused by an inconsistent network update in S10 and S8. In the other scenario, we observe packet drops in S8 for a short duration. However, the packet drops stop immediately after the forwarding rule

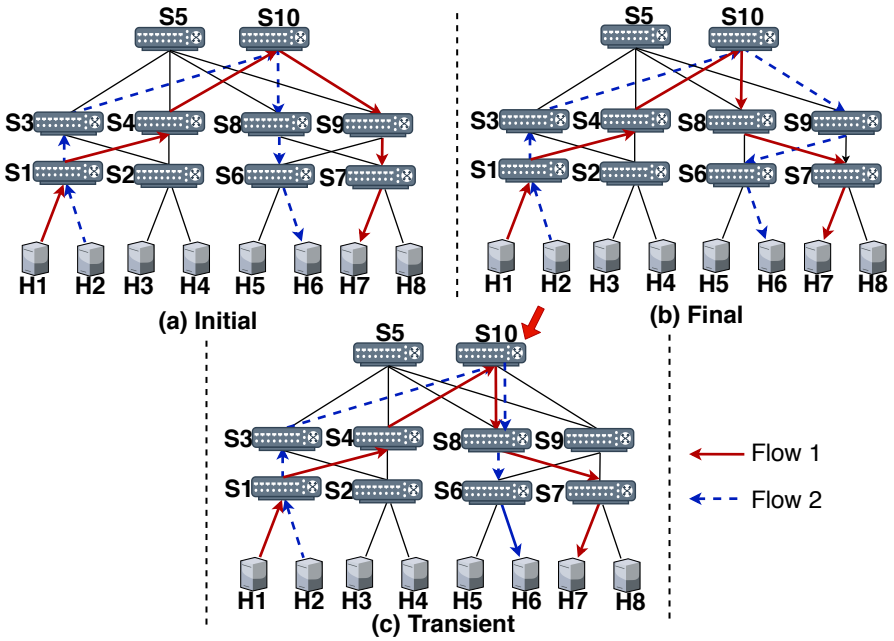


Figure 4.21: Network Update Scenario causing a link congestion at S10-S8

is updated in S8. This leads to conclude the root cause as a transient blackhole due to delay in updating the forwarding rule in S8.

Network updates leading to Congestion. Network updates to the forwarding rules can cause not only packet drops, but also cause intermittent congestion in the network [135, 137]. One such scenario is depicted in Figure 4.21. There exist two flows in this scenario, H1 to H7 and H2 to H6, of 6 Gbps each. The initial routing for these flows is shown in Figure 4.21(a). To reduce the traffic on links S8-S6 and S9-S7, routing is changed to the configuration shown in Figure 4.21(b). This requires the network to update two forwarding rules in S10: 1) Route flow 1 via S8, and 2) Route flow 2 via S9. While the rules are being updated, we can end up in a scenario as shown in Figure 4.21(c), where the two flows share the link S10-S8. This causes over-utilization of the S10-S8 link as the link capacity is 10 Gbps whereas the combined traffic rate of the two flows is 12 Gbps.

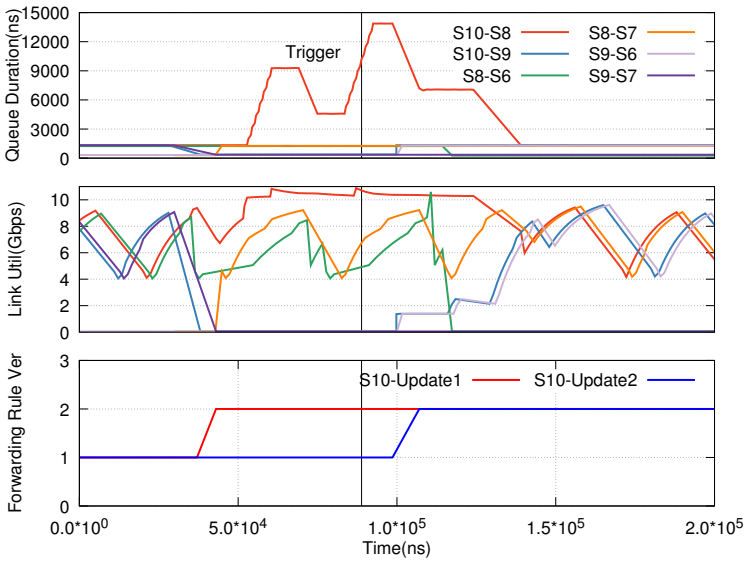


Figure 4.22: Delayed forwarding update at S10 leading to congestion at S10-S8 link

This problem may cause S8 to generate two possible triggers, one for packet drop and one for congestion. The congestion trigger is generated when the queuing duration increases beyond the threshold. *SyNDB* will start data collection on whichever trigger condition is raised first, whereas only the information about the second trigger will be sent to the collector.

Based on the packet drop and congestion trigger, the root cause can mostly be attributed to network misconfiguration or load balancing issues. To check whether a network misconfiguration occurred due to policy updates, we query the queuing duration, link utilization, and forwarding rule version, from the stored *p*-records as shown in Listing 2.

```
SELECT switch1, switch2, link_utilization*8, time_queue
    FROM (SELECT switch1, switch2 FROM links
    WHERE (switch1 IN (select switch FROM switches
    WHERE type !="tor") AND switch2 IN (SELECT switch
    FROM switches WHERE type !="tor"))) AS L
```

```
    JOIN (SELECT * FROM packetrecords) AS A
    JOIN (SELECT * FROM packetrecords) AS B
    ON (A.hash = B.hash AND A.switch = L.switch1 AND B.switch = L.switch2);

SELECT forwarding_rule_ver FROM packetrecords WHERE switch=10;
```

Listing 4.4: Query for plotting forwarding rule version of S10 along with link utilization and queuing times of core links

After plotting the output of the above queries (Figure 4.22), we notice that forwarding rule update on S10 just before queue buildup of 50 μs in S8 and increased link utilization on the S10-S8 link. Subsequently, he discovers a second update in S10. Following this update, both the queuing duration and link utilization return to levels prior to the first update. Hence one can conclude that the root cause for the transient congestion at S8 was a transient network misconfiguration due to policy updates.

Since, *SyNDB* provides network-wide packet-level visibility, it allows the operator to detect and debug the root cause of ephemeral events discussed above. While the three aforementioned examples concern specific scenarios, we believe that the capability to provide network-wide and synchronized packet-level telemetry information is a powerful tool that will allow network operators to detect and debug a wide range of complex network issues such as loops, network re-configurations, microbursts, etc.

4.8 Discussion

4.8.1 Limitations of *SyNDB*

***SyNDB* as a tool**. It is important to highlight that *SyNDB* is designed to be a tool that provides synchronized, high-resolution and network-wide telemetry information for debugging. The challenge of collecting the right set of data and the task of debugging are still very much required to be done by the human operator, probably complimented by some AI systems. These challenges are beyond the scope of this work.

Partial Deployability. *SyNDB*-enabled switches can be deployed incrementally with each new switch providing additional visibility into the network. To maximize effectiveness, deployment can be started from ToR switches as most congestion happens at the ToR layer [140]. DPTP synchronization can be achieved by adding links between adjacent ToR switches, which is not a complex undertaking [159]. To infer the core network's states, network tomography techniques [92] can be employed.

Managing Memory (SRAM) Overhead. As packet processing rate increases beyond 1 Bpps and parallel pipelines increase, *SyNDB*'s SRAM overhead increases linearly. Since, SRAM is an expensive resource, an alternative approach to scale is to use the relatively large DRAM (∼4/8GB) available in switch CPU [160]. A recent work [161] proposes using external DRAMs using RDMA channel (without CPU overhead) to store high-speed look-up tables or packet buffers. *SyNDB* could leverage such external memory to store p-records for a longer duration without having to change its design.

Reducing Broadcast Collection Overhead. On detecting a fault, *SyNDB* performs collection of p-records from all switches in the network. This may burden the collector with unnecessary data if the network is huge and the root cause is localized. We believe this could be mitigated by a simple technique. Each switch could maintain a list of links from which it received the packets that are currently stored in the ring buffer as p-records . Upon fault trigger, instead of broadcasting trigger packets, they could be selectively multicast to only the links from where packets were received in the recent recording window. This solution still provides the ability to trace every packet which appeared in the trigger switch to its source, while reducing the number of switches involved in the collection.

4.8.2 Future Directions

Timing Bugs in Distributed Systems *SyNDB* can be additionally used to debug timing bugs) [162] in distributed systems (Hadoop [4], ZooKeeper [5]. In most of these

timing bugs, a dead-lock is caused due to a missed or delayed message. *SyNDB* can help to find the reason why the message was missed or delayed by raising trigger conditions when it observes reordering or drops of certain packets.

Understanding network behaviour. *SyNDB* can also be used to find the root-cause of common issues like loops and reachability. It can additionally provide deep understanding of network behaviour in the wild and also provide answers to application-level questions like : 1) Why flows do not have fair bandwidth? 2) Why is the flow-completion time high? 3) I am seeing lots of port-scans, where are they coming from? 5) Is there a distributed TCP SYN flood attack? 6) What is the traffic pattern? 7) Is the network topology right for the traffic pattern?

Network Replay & Regression Testing. In future, we plan to develop replay tools which provide dashboards, query suggestions and an intelligent assistant [163] to network operators using AI techniques to facilitate faster debugging. We believe that *SyNDB*'s capability goes beyond debugging. The network device's configuration can be used along with network traces to create replay or simulation of the network fault. This in turn, can be used to form regression test suites. Such test suites are integral to large software development but are rare in network testing and management. For example, consider a bug in a network load-balancing implementation that causes a skewed distribution for a certain traffic combination. After the bug is fixed, *SyNDB*'s packet-level replays can be used to inject the exact same traffic combination into the network device to test the fix and also prevent regression in the future.

4.9 Summary

In this chapter, we design and implement *SyNDB*, which to the best of our knowledge, is the first packet-level synchronized monitoring framework for the entire network. *SyNDB* leverages data-plane time synchronization and the switch data-plane storage (SRAM) to

temporally store packet recordings, and upon a network fault exports the packet records to a collector for debugging. It provides the unique ability of looking back at the trace of events before the occurrence of a network fault. Additionally, we provide interfaces to configure *SyNDB* parameters at compile-time and run-time, thus giving flexilbility to the network operators. Finally, we leverage well-known relational DBMS based SQL queries to aid in debugging of complex network faults.

Chapter 5

BNV: Bare-metal Network Virtualization for Network Testing

Large data centers are challenging to run in a reliable manner due to their scale and complexity. Any changes to the network like fixing the faults, modifying topology, routing configuration, load balancing policy, etc. need to be tested in a staging environment before they are actually implemented in the production to ensure reliability and high availability. When such testing is done in a small-scale environment, they cannot emulate the complex interactions or behaviour seen in the production environment [33, 100, 164]. Hence, It is key to provide an environment to test network functionalities with high fidelity.

In this chapter, we design and implement a bare-metal network virtualization technique *BNV*, which faithfully emulates arbitrary data center network topologies. *BNV* is entirely software configurable and implemented on open source software and unmodified OpenFlow-enabled switches. The system has been deployed in a production testbed in National Cybersecurity Laboratory (NCL) and also used by several other research works [37, 38]. Our evaluations show that *BNV* can support various data center topologies with less number of switches and can facilitate building a high fidelity, repeatable

and isolated experimentation platform for data center operation and network research.

5.1 Introduction

Network experimentation is an integral part of network and systems research & development. With new paradigms like programmable networks or Software Defined Networking (SDN), networks are becoming more programmable and customizable according to application requirements. Data-center architects and network operators, constantly work on optimizing network topologies and algorithms under various scenarios and to maintain the production network in a steady state [35].

Fidelity and repeatability are two of the essential requirements for any experimentation platform. Additionally, to mimic real-life production network scenarios, flexibility and scalability in emulation of topologies are essential at both the control-plane and data-plane. Unfortunately, none of the available tools can satisfy both fidelity and flexibility in the data-plane.

In most cases, experiments rely on network emulation tools like Mininet [33], NS, etc. These tools allow experimenters to create topologies very quickly on their host machines. However, since they run on the CPU of the machines they are hosted on, the network performance depends on the deployment environment. Hence, it is hard to achieve fidelity and repeatability in the experiments. Additionally, these tools cannot scale to emulate production networks [35, 36].

Network testbeds like CloudLab [100], DeterLab [32] provide an environment for network and security experimentation. However, since the network is considered as an infrastructure [165] and managed using VLANs, experimenters cannot program their network.

In this work, we propose *BNV*, a network virtualization technique to bridge the gap between production networks and emulation tools by providing the following features:

Flexible network topology. *BNV* provides each tenant with their desired topol-

ogy in a bare-metal fashion (virtualization is done at the switch hardware) while the underlying topology is entirely different. Hence, multiple arbitrary topologies can be provisioned over the hardware switches using *BNV* as a network hypervisor. Our technique makes emulating large topologies more accessible and cheaper compared to the actual cost of building one. The experimenters can design custom algorithms or protocols to program their allocated network.

Fidelity. *BNV* provides consistent performance at hardware line-rate. Such fidelity requirement is essential for network experimentation involving applications whereby the network configurations need fine-tuning for optimal performance and regulatory verification.

Isolation and performance guarantee for multiple tenants. *BNV* allows different tenants or experimenters with different application requirements and topologies to share the same network infrastructure while guaranteeing complete isolation and repeatable performance by implementing buffer and queue management.

To implement *BNV*, we address the following challenges:

1. What is the right substrate network for *BNV* with maximum flexibility and minimum wiring changes?
2. What is the algorithm needed to find the most efficient network embedding?
3. How to implement *BNV* using existing tools?

To address (1), we propose a novel topology wiring using loop-back links in switches, to emulate multiple arbitrarily connected virtual switches. To address (2), we formulate the problem as an integer linear programming (ILP) to maximize the scalability of experimental support and fidelity. For (3), we implement *BNV* over an existing network hypervisor OpenVirteX [96] and also integrated the platform with a bare-metal provisioning system DeterLab [32]. We have deployed *BNV* in a Cyber-Security testbed called National Cybersecurity Lab (NCL) [166] for experimenters to run SDN and other experiments.

Our evaluations have shown that *BNV* can support high fidelity virtualization of network topologies, and can virtualize networks consisting of more than 100 network devices using only five top-of-rack (ToR) switches, while supporting line-rate for high-performance testing of production workloads.

This chapter is organized as following: §5.2 and §5.3 present the motivation and design of *BNV*. We describe the implementation in §5.5 and evaluate *BNV* in §5.6. Finally, we conclude in §5.8.

5.2 Motivation

In this section, we motivate the need for BNV from the perspective of fidelity, topology flexibility and scaling topologies using one to many abstraction.

5.2.1 Fidelity & Performance

There are a variety of tools like Trellis [104], Mininet [33], Maxinet [34] that enable experimenters to create arbitrary networks using container-based virtualization on a single machine or cluster of machines. These are elegant tools that allow experimenters to perform functional testing. However, these tools cannot achieve the peak line-rate for performance load testing due to various overheads, including encapsulation overheads, and unavailability of TCAM [35, 36, 167]. Hence, performance testing of large workloads for application scenarios with arbitrarily large topologies, while maintaining the characteristics of the topology is a significant challenge for researchers.

5.2.2 Topology Inflexibility

Network testbeds like DeterLab [109], CloudLab [100] or staging platforms used in the industries are usually generic Clos topologies or mini-versions of the production networks in the latter case. However, experimenters or network architects often need to vary the topology to suit a particular testing scenario, or to tune their applications on top of

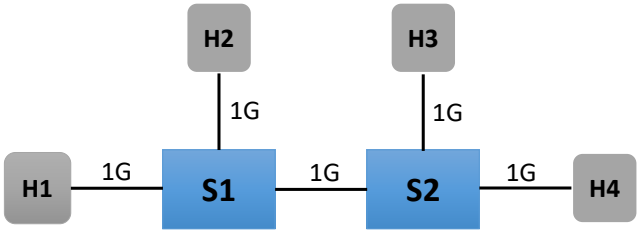


Figure 5.1: A sample topology to be emulated

a specific network. In that case, experimenters are often limited to the given testbed topology (or a subset) available. One option is that testbed providers or staging network operators modify the physical network to best suit the experimenter's need [165]. This would involve procurement of switches and configuration, wiring time and effort. A neoteric method of providing flexible network topologies to experimenters regardless of the underlying physical network would address this need.

5.2.3 Topology Scaling (One-to-Many abstraction)

To achieve topology scaling (implementing a larger virtual topology on top of a smaller substrate network) in an agile manner, we need to virtualize a single switch in to many virtual switches connected in an arbitrary fashion. CoVisor's [106] technique of flow compilation can be used to implement multiple arbitrarily connected virtual switches with a single physical switch flow table. However, this technique cannot emulate the delay and queuing characteristics between the interconnected virtual switches. To demonstrate this, let us consider a virtual network topology consisting of two switches, implemented on a single physical switch as in Figure 5.1. The hosts are mapped to separate physical servers.

We ran a continuous ping from H1 to H4 for 90 seconds. From 30 to 60 seconds, we started an iperf TCP session from H2 to H3. Now, since the TCP packets quickly fill the buffers of switch S2, we expect to see a rise in the latency of ping. The expected outcome in a real setup with two switches where the ping raises from 0.180 ms to 3-4 ms in Figure 5.2 labelled "Multi-switch". However with Covisor, there is no increase

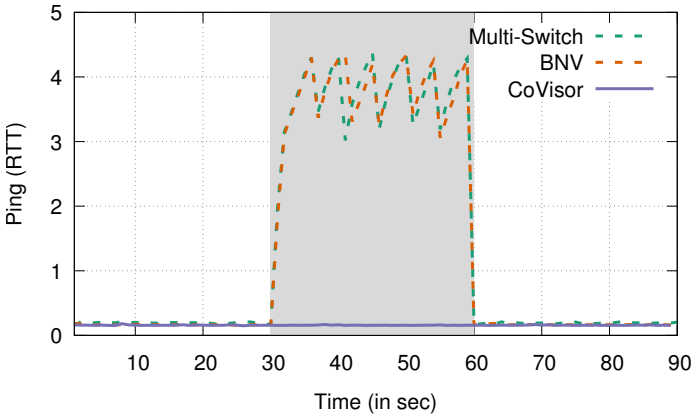


Figure 5.2: Flow Comparison with one-to-many abstractions

in ping latency, since the packet traverses only the backplane of the switch which has no contention and cannot emulate the link characteristics. Hence, we need a precise technique to perform one-to-many abstraction without losing the link characteristics for accurate experimentation.

BNV is a platform that allows researchers and practitioners to perform experiments on a virtual network that is close to the exact implementation of the actual physical network.

5.3 *BNV* Architecture

We present the architecture of *BNV*. The main component of *BNV* which differentiates it from other network hypervisors is the one-to-many abstraction which virtually slices the switches in the underlying physical infrastructure into multiple arbitrarily connected virtual switches (in form of collection of physical switch ports), without losing the queuing behavior and providing fidelity through direct virtual to physical mapping. We use the terms physical or substrate topology interchangeably.

Figure 5.3 shows the system architecture of *BNV*. *BNV* connects to the SDN switches via OpenFlow. Users submit their virtual topologies to the *BNV* Mapper, which returns an optimal mapping of the resources for the tenants with future expandability in mind.

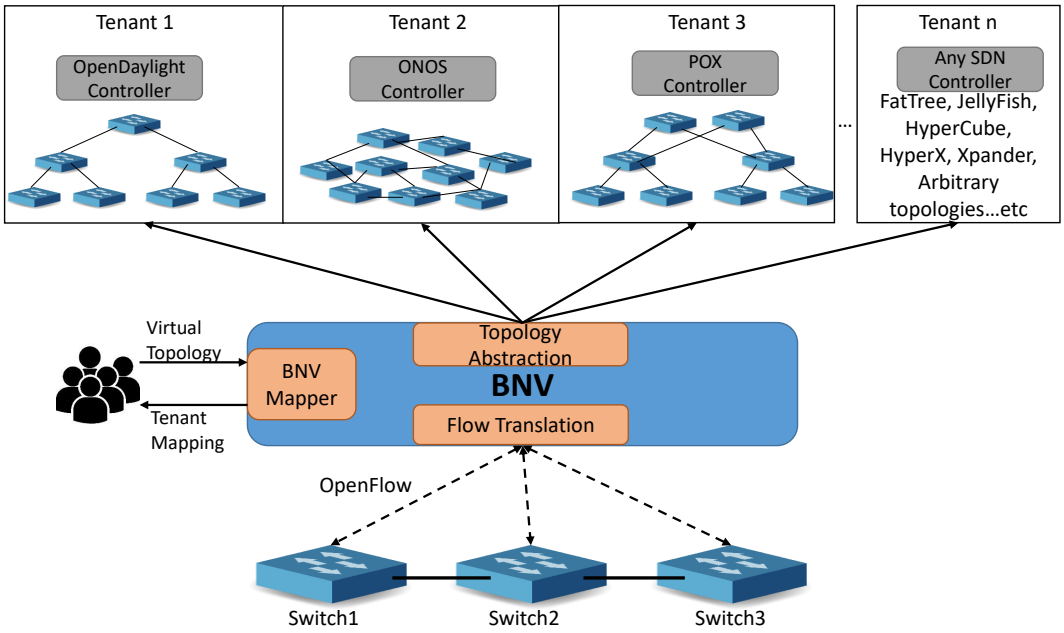


Figure 5.3: BNV System Overview

BNV takes the *BNV* Mapper input and creates the network for the tenant with the reserved physical resources.

BNV can create multiple tenants while tenants can use any SDN controllers to control their slice of the network independently, without interfering with the other tenants co-located on the same physical switch. We explain how *BNV* performs *topology abstraction* and *flow translation* in the following two sub-sections. We present the ILP based embedding model for the *BNV*Mapper module in §5.4, and the implementation details in §5.5.

5.3.1 Topology Abstraction

Consider a generic topology where switches are connected to hosts using relatively low-bandwidth links, and other switches (or core-network) using high-bandwidth links. We explain BNV's methodology of mapping arbitrarily connected switches over a given network topology using an example.

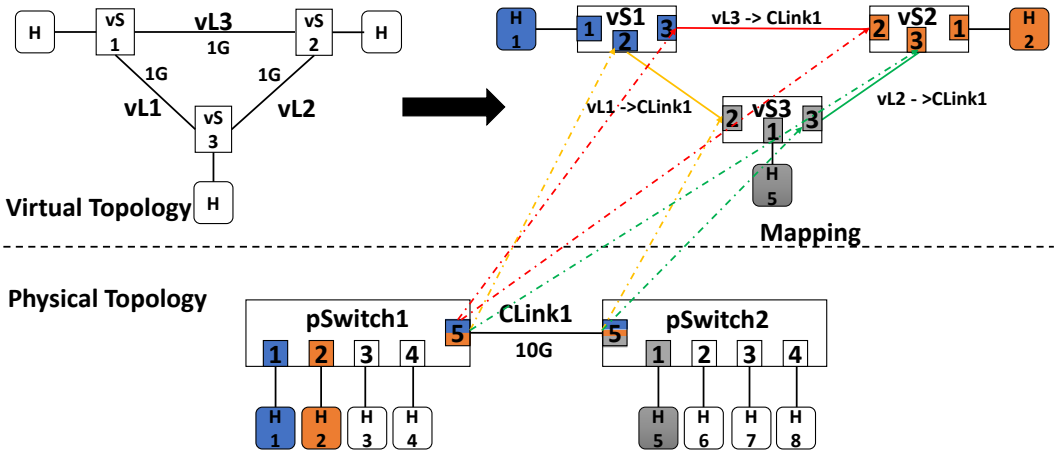


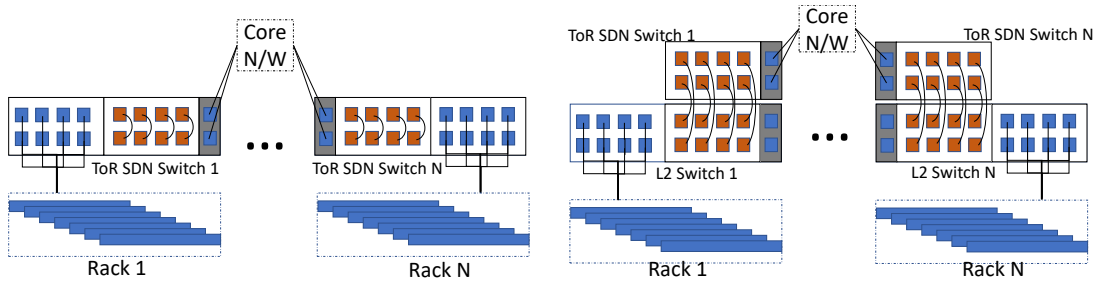
Figure 5.4: Mapping of Virtual to Physical Topology

Consider a substrate network as in Figure 5.4 with two switches of 5 ports each. The hosts are connected to the switches using 1Gbps links, and the inter-switch link has a bandwidth of 10Gbps. The virtual topology to be implemented is a triangular network (3 switches and 3 hosts) with all links of bandwidth 1Gbps.

The virtual topology mapping to the substrate is shown in Figure 5.4 where the links vL1 and vL2 are mapped to the CLink1, which is a backbone link in substrate network. However, since there are three switches in the virtual topology but only two switches in the substrate, two virtual switches (vS1 and vS2) are mapped onto a single physical switch (pSwitch1). Hence, to emulate the link vL3, the packet has to leave the port 5 of pSwitch1 and come back to the same switch. This link is called a *bounce* link.

Bounce link is a virtual link that connects two adjacent switches in the virtual topology using one or more intermediate physical switch(es) in the substrate topology. Bounce links incur additional traffic, queuing, lookup overhead due to traversal of additional links in the substrate. Hence, we avoid this scenario of using bounce links by proposing a simple wiring change in the substrate network as follows.

Loopback Link: The switches, apart from connecting to hosts or other switches, have special links which are loopback links, i.e., a link between two ports on the same



(a). Static Loopback (b). Software-Configurable Loopback

Figure 5.5: Loopback Configuration in Network

switch. If a switch has X ports, H ports are connected to the hosts, C ports are connected to other switches, and L ports are loop-ports which form $\frac{L}{2}$ loopback links. The number of loopback links can be pre-configured as shown in Figure 5.5(a) or dynamically configured by placing an L2-Switch or circuit switch between the host and the SDN Switch (ToR) as shown in Figure 5.5(b). With this configuration, we can dynamically configure the number of ports allocated to host and loopback at run-time through VLAN or circuit reconfiguration.

With the loopback links any two connected virtual switches can map to a single physical switch using the loopback link as their inter-switch link. Now, vL3 in the virtual topology can be mapped to the substrate network as shown in Figure 5.6 using loopLink1 instead of the bounce link. Loopback links provide flexibility in abstraction while maintain the fidelity of the virtual links. This means without loopback links, provisioning of various types of networks on a fixed physical infrastructure would be impractical. Referring to Figure 5.2 (§5.2), BNV using loopback observes same queuing behaviour as using multiple physically connected switches.

5.3.2 Flow/Rule Translation

BNV performs flow-table translations to provide flowspace (control-plane rules) and data-plane isolation for virtualizing multiple-links to support arbitrary topologies. Packet

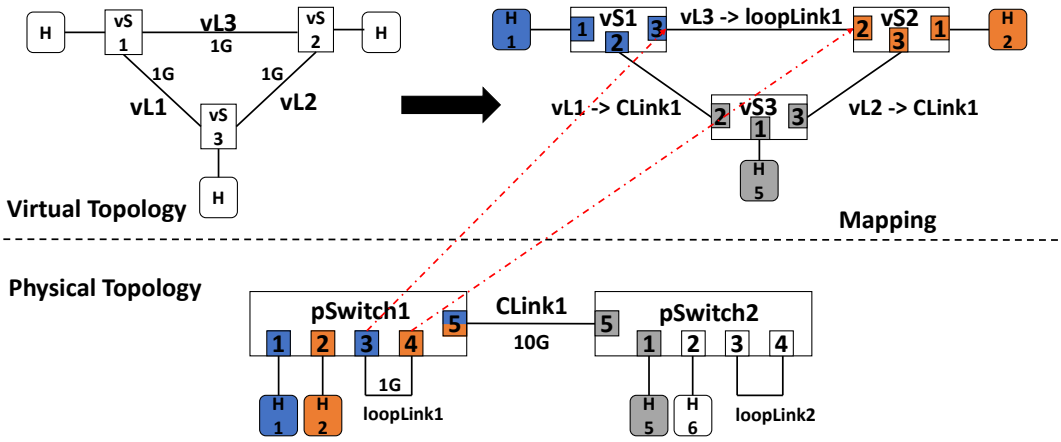


Figure 5.6: Mapping of Virtual Topology with Loopbacks

tagging and metering are used to achieve topology virtualization. We maintain the virtual link information encoded in the packet header. We call it $lTag$ for simplicity.

Referring to the virtual topology in Figure 5.6 with three links vL1, vL2 and vL3. Unique $lTags$ are generated for each virtual link. Assume the $lTags$ for vL1, vL2 and vL3 are respectively 1,2 and 3. We have two scenarios for mapping of the virtual links:

Case 1: Two connected virtual switches maps to different physical switches with a (in)direct link. (Example : vL1) In that case, any packet that travels vL1 is tagged with $lTag$ 1 on the outgoing port of CLink1. Similarly, Flowmods (control-plane rules) are translated to attach the $lTag$ in the match based on which virtual switch and virtual port the flowmod is meant for. Also, a meter m is created to rate-limit the flows to the max-rate of 1G as in the virtual topology. For instance, assume a tenant adds the below flow entry to vS1 for routing a packet via link vL1.

```
[in_port:1, ip_dest:x.x.x.x, actions=output:2]
```

The translated flowmod is shown below.

```
[in_port:1, ip_dest:x.x.x.x, actions=meter:1, lTag:1, output:2]
```

Case 2: Two connected virtual switches maps to the same physical switches (vL3).

Table 5.1: Flow Translation using loopback link

Virtual entity	Flowmod added to Switch
vS1	`pSwitch1 : [in_port=1, ip_dest : y.y.y.y, actions = meter:1, lTag:2, output:3]`
vS2	`pSwitch1 : [in_port = 4, ip_dest : y.y.y.y, lTag:2, actions = RemoveLTag, output:2]`

Consider a flowmod to enable ip-destination based forwarding from H1 to H2 are below:

```
vS1 : [in_port:1, ip_dest:y.y.y.y, actions=output:3]
vS2 : [in_port:2, ip_dest:y.y.y.y, actions=output:1]
```

Loopback link: LoopLink1 emulates vL3. The flowmods that were pushed earlier can be implemented by performing the translation of the output ports to physical ports, and adding the appropriate lTag in the physical topology as in Table 5.1. Typically Loopback links can be dedicated to a single tenant and they can naturally emulate inter-switch links.

Metering: We employ two kinds of data-plane isolation using meters:

1. Shared links (cLink1): When multiple virtual links share a single physical link, they share the same queues. We mitigate the interference by restricting a single virtual link from exhausting the port buffers by partitioning the maximum burst size of a physical link (measured using techniques in [168]) to multiple virtual links.
2. Shared Switches (pSwitch1): When a single physical switch is sliced into multiple virtual switches, it's important to slice the buffer such that the virtual switches do not run out of buffer. In this case, we apply a meter to the flows from all ports of a virtual switch. We explain the implications and observations of metering in Section 5.6.3.

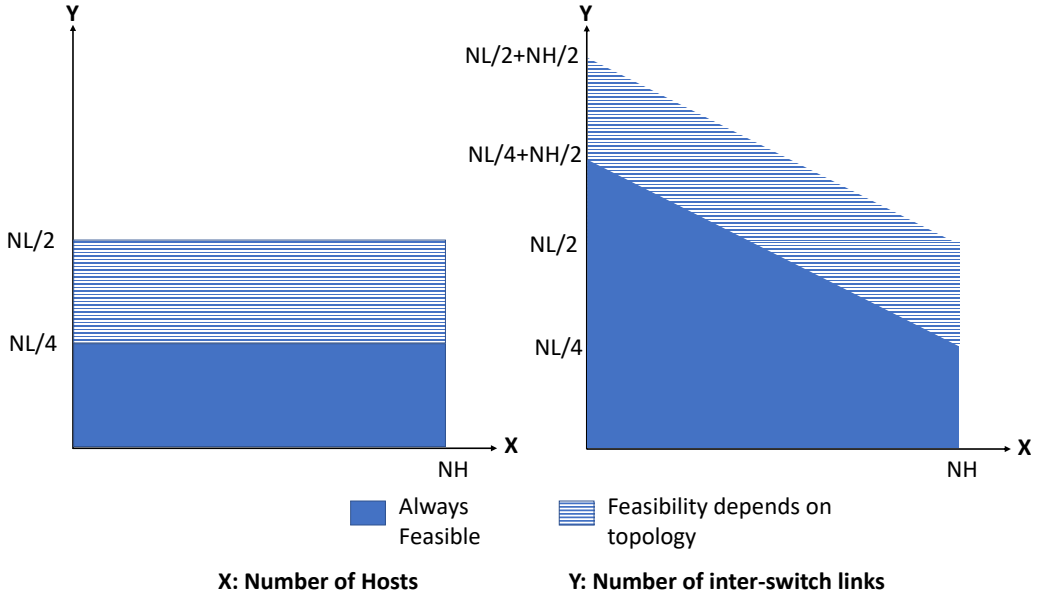


Figure 5.7: Size of Virtualized Network

5.3.3 Size of Virtualized Network

We present a discussion on the size of the network that can be virtualized using *BNV*. Consider the substrate topology with N switches. Each switch has H ports that can either be connected to the hosts or can be used as loopback links. For simplicity, assume that the capacity of each host link is 1 unit. Each switch is also connected to a backbone network with a link of capacity L units. As backbone links are generally over-subscribed, $H > L$. Finally, each backbone link with capacity L, can support up to L units of switch-to-switch links.

First, consider the case where there is no loopback link in the physical network, and NH hosts need to be provisioned in the virtual network. In the best case, up to $\frac{NL}{2}$ switch-to-switch links can be supported. In the worst case, the mapping places all switch pairs with connecting links on the same physical switches. As a result, only up to $\frac{NL}{4}$ switch-to-switch links can be supported. The feasible combinations of host and

switch-to-switch links supported are shown in the left graph in Figure 5.7.

Consider the case where loopback links are used. With NH hosts in the virtual network, up to $\frac{NL}{4}$ switch-to-switch links can always be supported. When fewer hosts are needed in the virtual network, one additional switch-to-switch link can always be added by using 2 (host) ports (on the same switch). The maximum number of switch-to-switch links that can always be supported is thus $\frac{NL}{4} + \frac{NH}{2}$. Note that $\frac{NL}{4} + \frac{NH}{2} > \frac{NL}{2}$ since we assume that $H > L$. In the cases, where bounce links are not needed, the number of switch-to-switch links that can be supported is $\frac{NL}{2} + \frac{NH}{2}$. In terms of the increase in the feasible region, the gain of using loopback links over no loopback links is $\frac{N^2H^2}{4}$.

5.4 *BNV* Mapper

In this section, we present the formulation of the network embedder (*BNV* Mapper) as an ILP to maximize the fidelity of the embedding of the virtual topology and to minimize the overheads.

5.4.1 Virtual Network Embedder

The (user) input to the *BNV* Mapper is the virtual network topology that consists of a set of : 1) hosts V, 2) (virtual) switches S and 3) (virtual) links L. Each link is bi-directional and has a required bandwidth b. A link can be associated with either (1) two switches (corelink L_c) or (2) a switch and a host (hostlink L_h).

The user input is next converted into a form where the switches are broken down into (1) Core-link. A link between two switches is represented as a link between two switch-ports. (2) Host-link. A link between a switch-port and a host. In this way, each switch s can be represented as a set of switch-links or host-links. For a link that is associated with a switch, the required TCAM size $t(s)$ can be specified.

The physical topology consists of a set of : 1) physical server machines H, 2) physical

Table 5.2: Notations used in the optimization model

S	Set of virtual switches.
L_c	Set of virtual core links.
L_h	Set of virtual host links.
L_c^m	Set of virtual core links of a virtual switch m.
L_h^m	Set of virtual host links of a virtual switch m.
b_i	The bandwidth of the link i.
c_j	CPU cores needed for host(link) j.
$t(i)$	TCAM spread for virtual link i.
N_r	TCAM Capacity of physical switch r.
H	Set of physical hosts.
R	Set of physical switches.
Q_c	Set of physical core links.
Q_h	Set of physical host links.
Y	Set of backbone links.
B_v	The capacity of link v.
C_v	The CPU core capacity of host(link) v.
$M_{p,q}^m$	Calculated intra-switch traffic (between connected physical switch p,q) for a virtual switch m .
p_v	The physical switch containing corelink v.

switches R and 3) physical links Q. Similar to the virtual topology, Q can be categorized into corelinks Q_c and hostlinks Q_h. The available bandwidth of a link i is represented as b_i. Note that, Q_c comprises of both loopback links and backbone links (links between the switches).

We define a binary decision variable x_{iv}, which is set to one if a virtual corelink i is mapped to the physical corelink v, and zero otherwise. Similarly, y_{jw}, which indicates if a virtual hostlink j is mapped to a physical hostlink w. Mapping of hostlink automatically implies mapping of virtual host to physical server. The objective function is given below by Equation (5.1).

$$\min : \sum_{\substack{i \in L_c \\ v \in Y}} x_{iv} b_i + \sum_{\substack{m \in S \\ \{p,q\} \in R}} M_{pq}^m \tag{5.1}$$

The objective function has two terms. The first term represents the amount of

resources to support mapping of corelinks ($x_{iv}b_i$) and the second term represents the resources needed to map a single virtual switch over multiple physical switches (M^m_{pq}). The overall goal is to minimize the usage of substrate backbone (core) links, since these links are (generally) under-provisioned relative to the hostlinks. This indirectly maximizes the usage of loopback links which provide higher fidelity in emulation. The constraints are:

$$\sum_{v \in Q_c} x_{iv} = 1, \forall i \in L_c \tag{5.2}$$

$$\sum_{w \in Q_h} y_{jw} = 1, \forall j \in L_h \tag{5.3}$$

$$\sum_{v \in Q_h} y_{jv} b_j \leq B_v, \forall j \in L_h \tag{5.4}$$

$$\sum_{v \in Q_h} y_{jv} c_j \leq C_v, \forall j \in L_h \tag{5.5}$$

$$\sum_{i \in L_c} \sum_{v \in Q^r_c} x_{iv} t(i) + \sum_{j \in L_h} \sum_{w \in Q^r_h} y_{jw} t(j) \leq N_r, \forall r \in R \tag{5.6}$$

$$\begin{aligned} z_{mn} &= \sum_{v \in Q_c} x_{iv} b_i + \sum_{w \in Q_h} y_{jw} b_j, \\ &\forall i \in L^m_c, j \in L^m_h, m \in S, n \in R \end{aligned} \tag{5.7}$$

$$M^m_{pq} = \min(z_{mp}.z_{mq}), \forall m \in S, \{p, q\} \in R \tag{5.8}$$

$$\sum_{v \in Q_c} x_{iv} b_i + M^m_{p_v q_v} \leq B_v, \quad \forall i \in L_c, \forall m \in S, v \in Y, \{p_v, q_v\} \in R \tag{5.9}$$

Constraint (5.2) mandates each virtual corelink to be mapped to only one substrate corelink. Constraint (5.3) makes sure that each virtual hostlink be mapped to only one physical hostlink. Constraint (5.4) ensures that each physical hostlink is provisioned within it's capacity. Constraint (5.5) ensures that the physical host is not allocated beyond it's core capacity. The notation uses hostlink instead of host, however, in the model hostlink is synonymous to host. Also, Constraint (5.4) and (5.5) are needed only for VM-based provisioning like OpenStack. They are not needed for bare-metal provisioning methods since each virtual host is allocated an entire server blade. Constraint (5.6) makes sure we do not exceed the TCAM capacity bounds of the switch. TCAM specified for every virtual switch is split equally among it's corelinks and hostlinks for simplicity in allocation.

Constraints (5.7) to (5.9) are used to model the resources (M^m_{pq}) needed to support intra-switch traffic (between physical switch p,q) for a virtual switch m, mapped onto multiple substrate switches. In order to make sure that sufficient bandwidth is provisioned for, we need to take into account the number of hosts/links mapped on each physical switch, and the amount of traffic, they can send and receive.

Big-Switch Abstraction. Constraint (5.7) defines a variable $z_{m,n}$ which depicts the total bandwidth of virtual links belong to virtual switch m that are mapped to physical switch n. Constraint (5.8) creates a R x R matrix for each virtual switch which indicates the amount of intra-switch bandwidth between any two physical switches belonging to a single virtual switch. It allocates the minimum (host/core)link bandwidth generated by each physical switch for a given pair of physical switches mapping to same big virtual

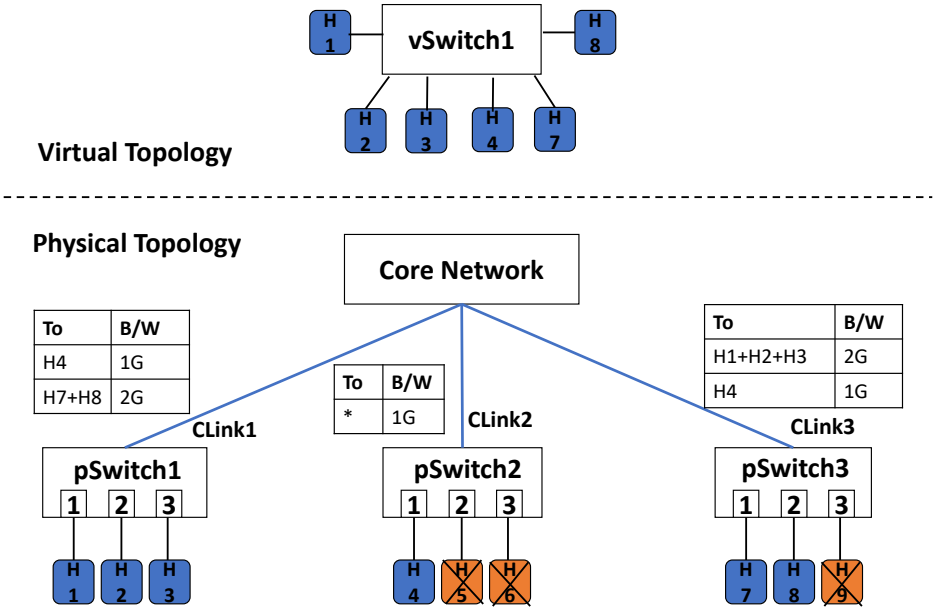


Figure 5.8: Overview of *BNV* Mapping

switch. Figure 5.8 illustrates an example of the embedding of a virtual switch with 5 hosts onto three substrate switches each mapping a different number of hosts. In Figure 5.8, each virtual switch is associated with a table containing the bandwdith to be allocated for traffic to a particular set of hosts belonging to a physical switch. Finally, Constraint (5.9) makes sure the physical corelinks are provisioned within their capacity accounting both inter-switch links and intra-switch link utilization.

While the above formulation does not take into account buffer reservation, it is easy to incorporate buffer partitioning based on the metering feature on switches that provide both rate and burst size limits.

5.4.2 Topology Convertibility

The *BNV* Mapper supports evolution of one topology to other with minimal overall migration needed by performing the following steps:

1) **Snapshot Mapping:** We take a snapshot of the current mapping of host/core links

for the tenant, and record these mappings as $x'_{i,v}$ (from $x_{i,v}$) and $y'_{j,w}$ (from $y_{j,w}$).

2) **Remapping:** Next, run the ILP with a modified objective as the below for the tenant:

$$\max : \sum_{\substack{i \in L_c \\ v \in Q_c}} x_{i,v} x'_{i,v} + \sum_{\substack{j \in L_h \\ w \in Q_h}} y_{j,w} y'_{j,w} \tag{5.10}$$

Equation (5.10) maximizes the mapping similarity between the original virtual topology, and the modified version by trying to use up the same switch-ports and hosts if possible, thus minimizing potential migration.

5.5 Implementation

We have implemented *BNV* with 1500+ lines of Java code over OpenVirtex [96] using OpenFlow 1.3. We have also integrated *BNV* (1200+ lines of python code) with a bare-metal provisioning system DeterLab [32]. We use Gurobi [169] as the ILP solver. The users submit their topology as a NS file. The SDN switches are specified by defining the switch-type as "ofswitch". For the complete virtual topology specification, refer *BNV* Usage Document [170]. For each SDN experiment, controller nodes are created so that the user can use his own SDN controllers which are connected to *BNV* using a VLAN-provisioning based out-of-band network.

The APIs for a virtual network creation are listed in Table 5.3. The API *createNetwork* initializes the tenant, and identifies the controller IP and port. *createVSwitch* creates a virtual switch. It performs two types of mapping : 1) Many-to-one abstraction, which takes several physical switches, and abstracts them as a single switch. 2) One-to-many abstraction, which can slice a physical switch into multiple virtual switches for the same tenant. This API can be called multiple times to create multiple virtual switches. *createVSwitchPort* takes as input the virtual switchID returned by *createVSwitch*, the max outgoing rate and burst (implemented using meter) and the physical switchID

Table 5.3: *BNV* Command Reference

Commands
$tenantID\ createNetwork\ (controllerIP, port)$
$vSwitchID\ createVSwitch\ (tenantID, pSwitchID...)$
$vPort\ \ createVSwitchPort\ \ (vSwitchID, pSwitchID, SwitchPort, maxRate, maxBurst)$
$void\ createCoreLink\ (vSwitchID1, vPort1, vSwitchID2,\ vPort2)$
$void\ createHostLink\ (vSwitchID1, vPort1, hostMac)$

and port number to be assigned to the virtual switch. It returns a unique *vPort* for the *vSwitch*. *createCoreLink* creates a virtual link between two virtual switch-ports. *createHostLink* creates a virtual link between a host and a virtual switch-port.

BNV virtualizes the substrate network by maintaining a mapping table {pSwitch, pSwitchPort} -> {vSwitch}. Any incoming packet, is virtualized to a particular virtual network by identifying the switch-port and the physical switch it comes from apart from the $lTag$ it carries. In order to implement the virtual link-tags ($lTag$), we leverage MAC-address encoding of OpenVirtex. An incoming packet of a virtual network goes through mac translation to encode $lTag$ (32-bit) and metering if needed to rate-limit. This isolates multiple tenants and also, multiple virtual links within the same tenant. Thus it enables to use even a single physical link as multiple virtual links for the same tenant.

We have run *BNV* extensively on the production testbed of the National Cybersecurity Lab [166] to provision SDN-based experiments with arbitrary topologies. *BNV* is also used by recent research works [37, 171] to emulate multi-switch topologies. The experimenters can completely use the functionalities supported by OpenFlow 1.3 (Meters, groups, etc.), and can develop and use custom network applications (BGP, congestion control, etc.). The network testbed connectivity is shown in Figure 5.9. The testbed currently has four HP3800 [172] SDN Switches, each connecting to a cluster of 24 Lenovo X3550 servers. The SDN switches are connected using a core-switch which is used only for L2 connectivity. The server blades are also connected to a control-switch, which is

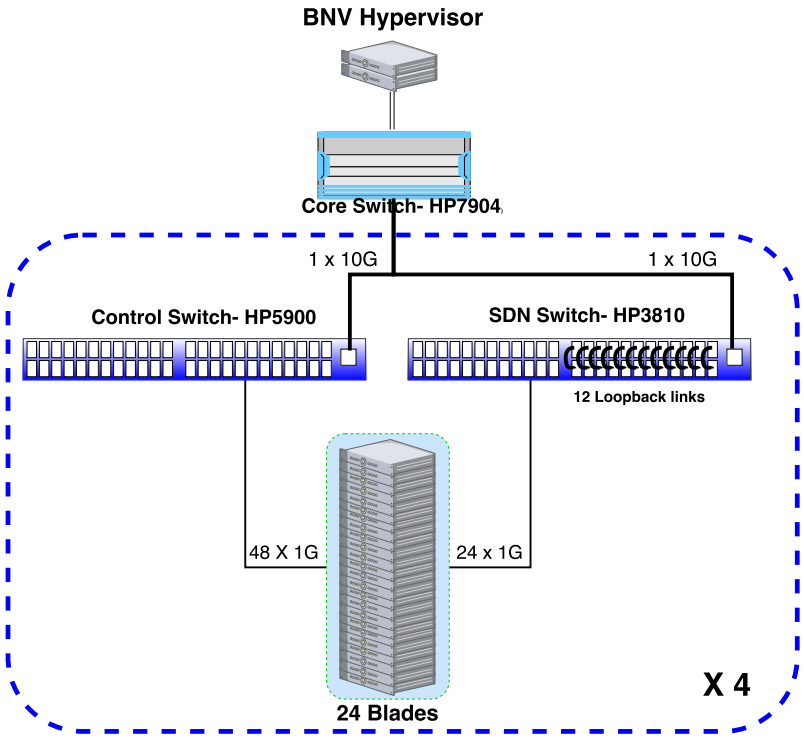


Figure 5.9: *BNV* testbed setup environment

used out-of-band management (IPMI, PXE booting). Each SDN switch has 12 loopback links and 1 10G uplink to the core-switch. Each loopback link is formed by connecting two ports using a short cable. Figure 5.10 shows the physical testbed with the static loopback wiring of 12 links per switch. We additionally perform software-configurable loopback for one cluster in our staging platform.

5.6 Evaluation

In this section, we present the evaluation of *BNV*'s ability to meet three objectives:

(1) Does the virtual network provided by BNV faithfully mimic the behavior of the actual topology?

(2) Can *BNV* mapper embed complex topologies?

(3) Does *BNV* provide performance isolation to multiple tenants?

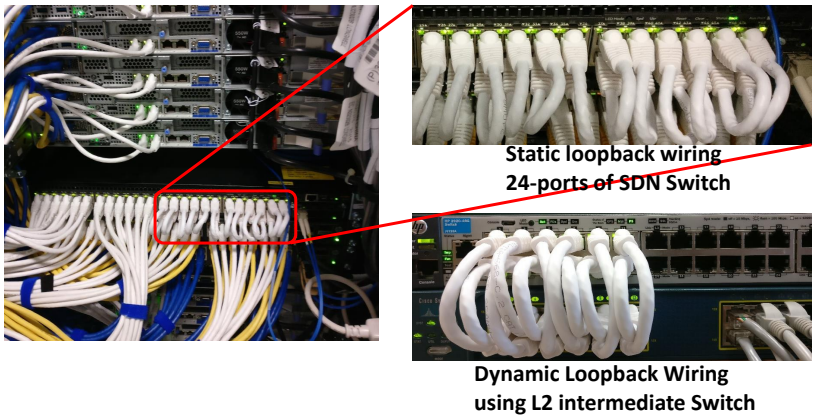


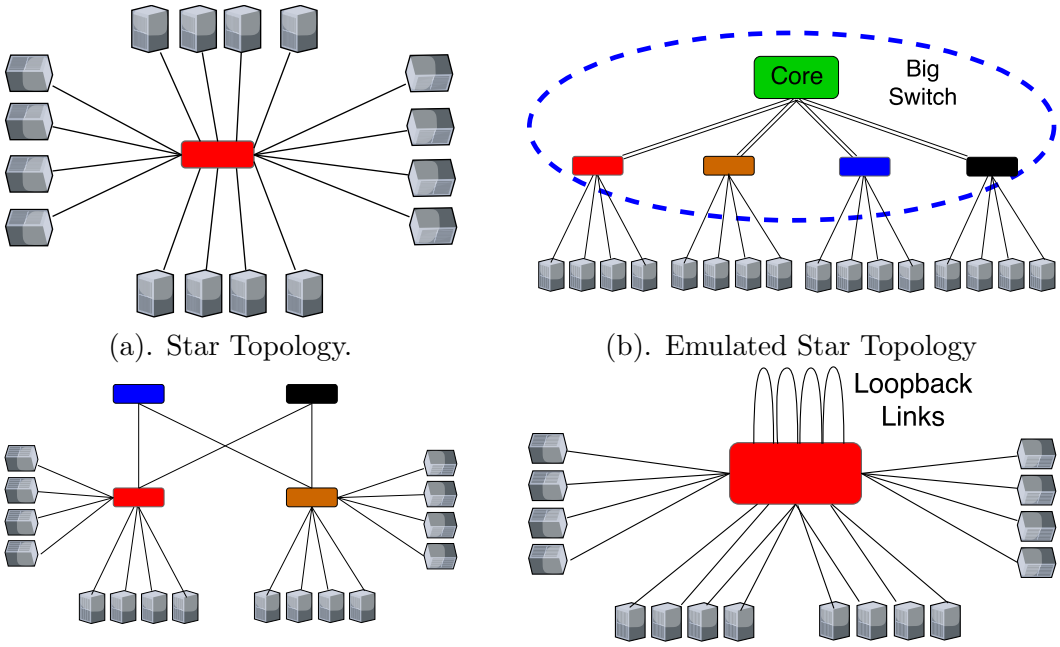
Figure 5.10: Loopbacks in Switches

5.6.1 Performance Fidelity

In this section, we evaluate the fidelity of the virtual topology created by *BNV* by making use of one-to-many abstraction and many-to-one abstraction. The evaluation considers variations of two topologies: (1) a star topology with 1 switch and 16 hosts (Figure 5.11(a) and Figure 5.11(b)) and (2) a Clos topology with 4 switches and 16 hosts (Figure 5.11(c) and Figure 5.11(d)). We change the physical wiring of the testbed to implement the physical clos topology.

We run an Apache Spark application (wordcount on a 50GB file) on all 4 topologies and we plot the CDFs of shuffle read times on the four topologies in Figure 5.12.

We observe that the CDFs of the physical and virtual networks are very similar. The use of loopback links (Figure 5.11(d) vs Figure 5.11(c)) has the same application behaviour compared to actual topology. The use of longer physical hops and multiple physical switch to support one virtual switch (Figure 5.11(b) vs Figure 5.11(a)) do not affect the performance.



(a). Star Topology.

(b). Emulated Star Topology

(c). Clos Topology

(d). Emulated Clos Topology

Figure 5.11: Topologies used for application fidelity

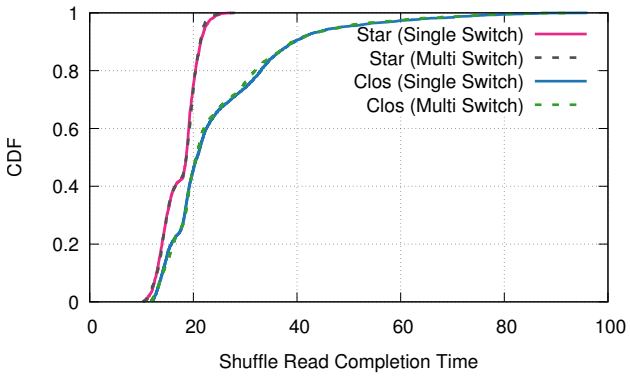


Figure 5.12: Performance comparison on emulation

5.6.2 Topology Flexibility

Recent works on topology convertibility [173] demonstrate tangible gains in network performance by changing the network topology dynamically. *BNV* can leverage its inherent

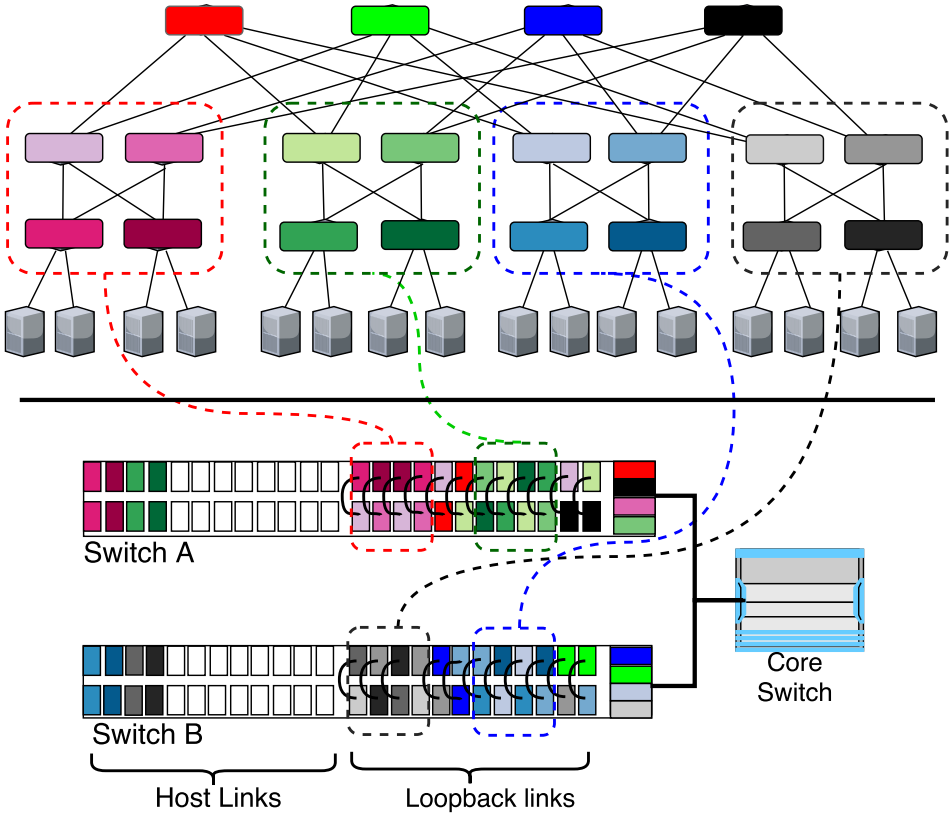


Figure 5.13: Mapping of fat-tree4 on testbed

ability to map arbitrary network topology to allow researchers to experiment with different topologies quickly. This makes it possible to perform fine-grain optimization of topology for a work-load and ability to emulate traffic patterns reliably in a topology.

BNV can virtualize the network to support various topologies like fat-tree [6], Clos, jellyfish [10], Hyper-X etc in a matter of a few seconds. We illustrate how a fat-tree is mapped on to substrate topology in Figure 5.13.

We create four experiments with various topologies: Binary Tree (16 hosts, 15 switches), Star (16 hosts, 1 switch), fat-tree with degree 4 (16 hosts, 20 switches), and jellyfish [10](16 hosts, 20 switches) and perform a wordcount application for a 50GB file in Apache Spark. We use custom partitioning in order to increase the inter-rack traffic.

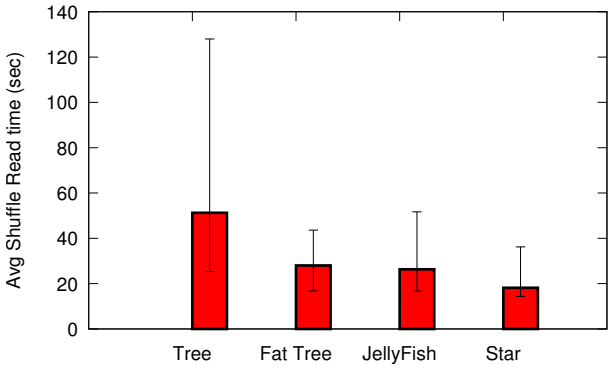


Figure 5.14: Apache Spark performance over various Topologies generated using *BNV*

We perform 10 runs and plot the average shuffle read time for the four topologies in Figure 5.14.

We observe that tree topology has the longest shuffle read time due to its very limited bisection bandwidth. The star finishes fastest since it has full bisection bandwidth. Fat-tree and jellyfish perform somewhere in between as expected. ECMP [174] was used to split the traffic equally based on link utilization calculated at the controller in the case of fat-tree and jellyfish.

Now, taking a closer look at only two topologies: fat-tree and jellyfish. Fat-tree and jellyfish observe really close shuffle time. On an average, the shuffle time in jellyfish is about 7-8% lower. This is due to high inter-rack traffic. Then we vary the data placement in Spark to increase the intra-pod locality. We plot the CDF of the shuffle read times in Figure 5.15 for fat-tree and jellyfish. Interestingly, we observe an increased shuffle time ($\sim$7.5%) in case of jellyfish compared to fat-tree. To summarize our observation, jellyfish performs much better than fat-tree when there are lot of inter-rack network traffic and fat-tree performs better than jellyfish when there are lot of intra-rack exchanges.

This performance behaviour observed is consistent with the observations made by recent work on convertible topologies [173]. Thus, *BNV* can achieve fidelity and flexibility in terms of its topology implementation.

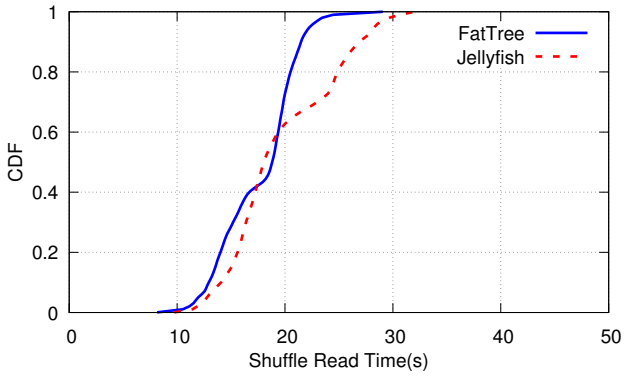


Figure 5.15: Comparison of fat-tree and jellyfish topologies for high intra-pod locality traffic

5.6.3 Isolation

We spawn three experiments on *BNV* with the following topologies: 1) Expt-1: fat-tree4 (16 hosts and 20 switches) running wordcount of 1GB file using Spark continuously with heavy inter-rack traffic, 2) Expt-2: jellyfish (random topology) topology with 16 hosts and 20 switches running iperf among all pairs and 3) Expt-3: Random topology using up the rest of the available switch-ports(8) and hosts(8) running ping between the pair of nodes with the longest hop-count. Initially, all experiments are idle with applications not running. The experimentation scenario is as follows :

1) Between 0-60 minutes, we run the application on each of the experiments for 20 minutes without interference.

2) Between 60-80 minutes (shaded region), all three applications are run in parallel.

3) We repeat step (1) between 80-140 minutes.

We present the findings in Figure 5.16, where we plot different performance metrics such as average shuffle time for expt-1(top), aggregate throughput for expt-2(middle) and ping latency for expt-3(bottom). The shaded region (comprising 20 minutes) represents the portion of time the experiments were run in parallel, and the non-shaded regions depict the performance for individual runs. Observe that the applications see no

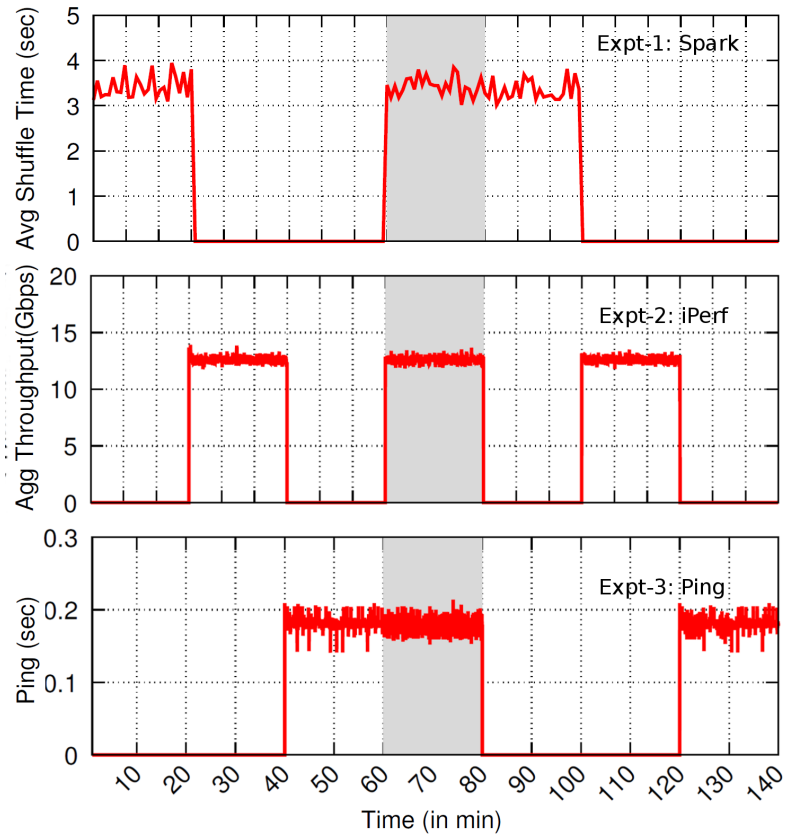


Figure 5.16: Concurrent experiments to check isolation

perceivable difference in performance when the other experiments which share the same switches and backbone links are run.

Buffer partition: When two tenants share a physical link (typically backbone links), a burst of traffic from a single tenant could fill up the buffers and TX Queue, thus impacting the other tenants. Fortunately, we can achieve an approximate buffer partition by allocating the maximum burst for a particular virtual link or set of ports (using OpenFlow meters) during provisioning by considering the switch-port buffer-size to be one of the constraints in the *BNV* mapper. In the following, we perform two experiments to measure the impact of bursty traffic.

Bursty transmission on a specific output port does not affect the traffic on

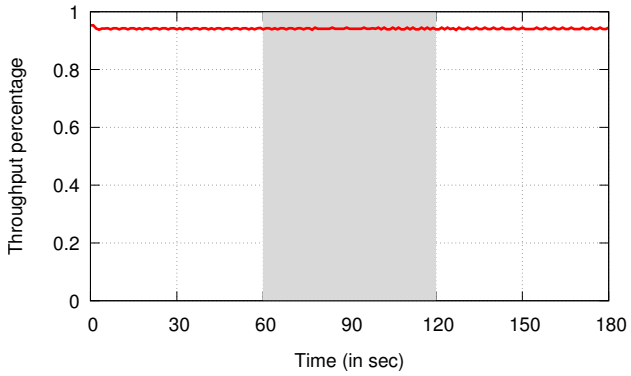


Figure 5.17: Observation of TCP traffic completely unaffected by UDP bursts in the same switch over other ports.

other switch-ports: We perform an experiment with two tenants allocated different slices of the same switch: one tenant performing UDP bursts (sending at the maximum rate) of 11 hosts to 1 host, and the other generating a TCP traffic at line-rate on hosts running over two other ports. In Figure 5.17, the shaded region is the period where the UDP bursts occur. Clearly, the UDP traffic has no impact on the TCP traffic due to the buffer isolation provided. Although switch architectures [175, 176] perform buffer isolation inherently, in order to be generic and not rely on specific switch architectures, we allocate a burst-rate for all the ports belonging to each virtual switch, and apply the same meter to all the flows belonging to the virtual switch's ports. In this way, we guarantee that the amount of buffer used by a single (or a set of) flow(s) is bounded. Although, this implies that the switch-buffer may be under-utilized in certain circumstances, we reckon that this measure is necessary to guarantee isolation, fidelity and repeatability of experiments.

Burst transmission on shared ports (and links) minimally affects the traffic: We perform an experiment, where the backbone link is shared by two virtual links (either same or different tenants). vlink1 is allocated 1/10th of the bandwidth of the physical link, and the rest of the bandwidth is allocated to virtual-link2. We transmit

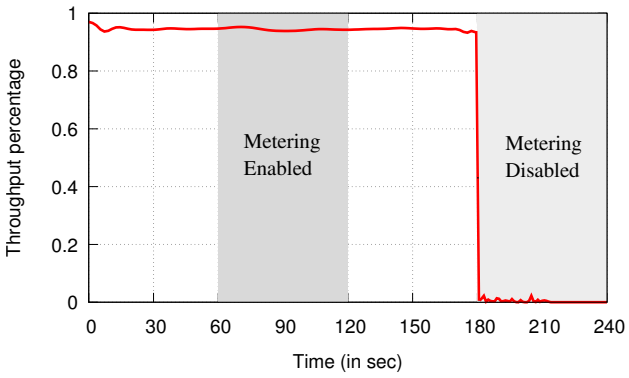


Figure 5.18: Observation of TCP traffic completely unaffected by UDP bursts in the same egress (shared)ports.

TCP flow on virtual-link1 and 10 different UDP flows on vlink2 to saturate the physical link. We observe the impact of UDP traffic-bursts of vlink2 on vlink1. The results are shown in Figure 5.18. We can observe that the TCP throughput is not affected even during the presence of burst transmission. This is primarily because we limit the maximum burst per port. At the 180^{th} second, we turn off metering-based isolation. We notice that the TCP flow on vlink1 is completely starved due to bursty UDP flows on vlink2 in the absence of burst/ rate limiting.

5.6.4 Network Mapping Simulation

In this section, we evaluate through simulation the network embedding efficiency of *BNV* by considering a network consisting of 5 switches connected using 40G Bi-directional link each as shown in Figure 5.19. The switches consist of 48 down-link ports (1G each) of which, 24 ports to be connected to physical servers, and the 24 remaining ports are used form loopback links. Thus, creating 12 loopback links per switch.We use random graph topologies and Internet Topology Zoo dataset [177] for evaluation.

1. **Fat-tree variants:** We embed increasing degrees of fat-tree topology on the underlying physical topology. We consider all the links in the fat-tree as 1Gbps links.

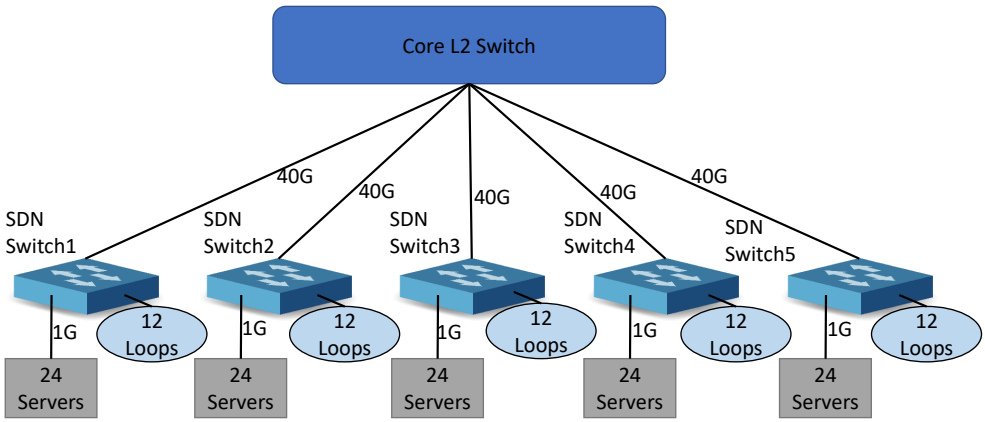


Figure 5.19: Physical Topology used for Simulation

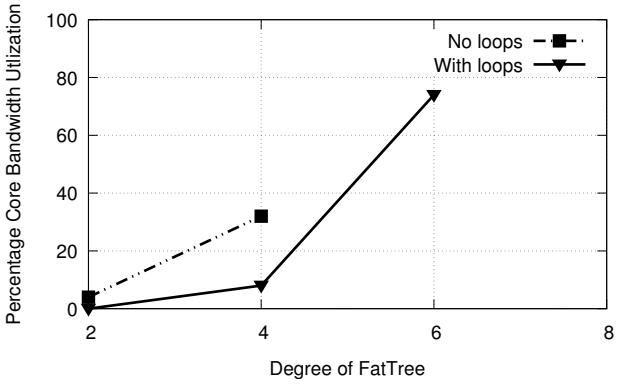


Figure 5.20: Mapping of fat-tree Topologies

For simplicity, we consider just one tenant in our evaluation. We increase the degree of fat-tree to embed until the mapping becomes infeasible on two variants of substrate topology: 1) No loopback links (no loops) and, 2) With 12 loopback links,

We plot our results in Figure 5.20 where we show the percentage of backbone link's bandwidth utilized. Each data-point is averaged over 10 runs. We observed successful (optimal) mapping consistently for all the runs. Generally, we can increase the virtual network size as long as core bandwidth is available. Hence, using up the core bandwidth slowly is a good strategy. to scale. We observe that without loopback links, the core bandwidth utilization shoots up immediately. Thus, it is unable to map fat-tree with

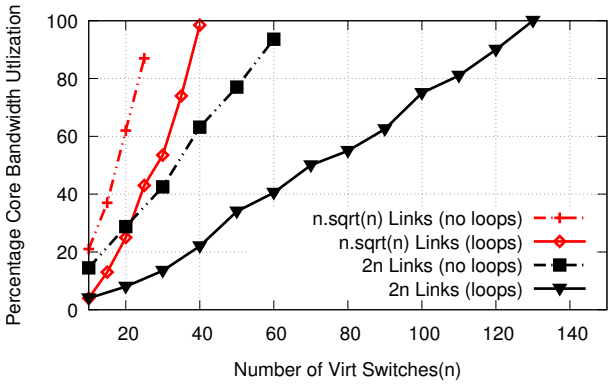


Figure 5.21: Mapping of Random Topologies

degree 6 (212 links (1Gbps each)). However, with loopback links, we are able to map fat-tree with degree 6 and able to have better utilization of the core backbone's bandwidth.

2. Random Topology: We embed random topologies with increasing number of switches and links to the physical topology. We consider two sets of random topologies with a fixed number of n switches: 1) Number of links = $2n$ and, 2) Number of links = $n\sqrt{n}$. We perform the network mapping with 10 set of topologies for each variant of random topology. We plot the percentage of backbone link's bandwidth utilization against the topology size (number of switches) in Figure 5.21. For topologies with $2n$ links, the gain is clearly evident, as we are able to fit bigger topologies (up to 130 nodes and 260 links) using loopback links compared to only 60 nodes (and 120 links) and without loopback links, leading to approximately $2\times$ gain. Similarly, for topologies with $n\sqrt{n}$ links, we are able to fit topologies with 40 switches (252 links) with loopback links, compared to 25 switches (125 links) leading to again $2\times$ gain.

3. Internet Topology Zoo: We take the entire set of topologies from Internet Topology Zoo [177] and embed them onto the physical topology with and without loopback links. We sort the physical topologies according to the number of links in the graph and plot the backbone link usage percentage with increasing size of the internet zoo topologies in Figure 5.22. We are able to map about 96.5% (252 of 261) of the

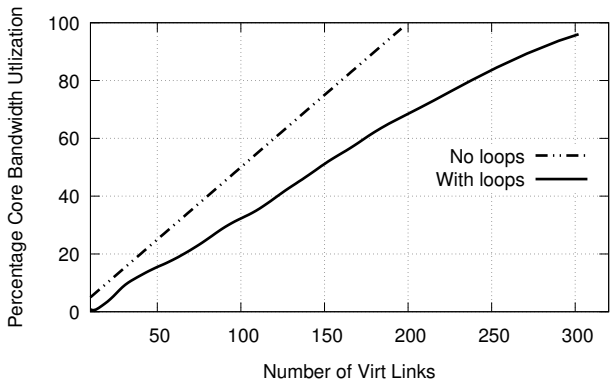


Figure 5.22: Mapping of Internet Topology Zoo

topologies using the given physical topology with loopback links. Overall, we are able to fit 50% more topologies with loopback links in the network.

Take-away: *BNV* can emulate a network topology with high fidelity and repeatability by maintaining the key characteristics of each topology while guaranteeing isolation for each experiment. *BNV* Mapper can embed a large collection of topology efficiently. The addition of loopback links increases the scalability significantly.

5.7 Limitations

While *BNV* emulates network connectivity with high fidelity and low overhead, it offers limited buffer due to the dependence on the OpenFlow switches [172] which do not offer buffer slicing capabilities. However, with recent programmable switches [58] offering better buffer slicing mechanisms per port, it is possible to achieve accurate buffer isolation.

5.8 Summary

We developed *BNV* (Bare-metal Network Virtualization), which provides high-fidelity network experimentation at data-plane and control-plane using programmable switches.

BNV can support arbitrary network topologies using a unique method of creating loop-backs in switches in order to provide high fidelity. We propose an ILP-based formulation for efficient embedding of complex topologies to the substrate. We have built *BNV* on top of OpenVirtex, and deployed on a production testbed. *BNV* accurately emulates the characteristics of the topologies implemented over the substrate network while isolating multiple experiments.

Chapter 6

Conclusion and Future Work

Data Center Networks are in a constant state of flux with complex interactions between diverse applications. To handle increasing and diverse services, the data center networks continue to grow in speeds and complexity. Precise Network diagnostics are a crucial part to keep these networks up and running. In this thesis, we propose novel techniques to enable scalable and accurate network diagnostics by developing monitoring, debugging and testing frameworks.

In this chapter, we first summarize the research contributions of this thesis. Our contributions lays the ground work for further interesting research directions like Synchronous data center networks [178] and Self Driving Networks [179] which we discuss in Future Work.

6.1 Research Contributions

In this thesis, we breakdown the overall life-cycle of network diagnostic process i.e. monitoring, debugging and testing as shown in Figure 6.1. Then, we identify problems in each step of the diagnostic process and develop solutions to address them. To tackle the need for network-wide synchronized measurements, we design *DPTP*, a precise time

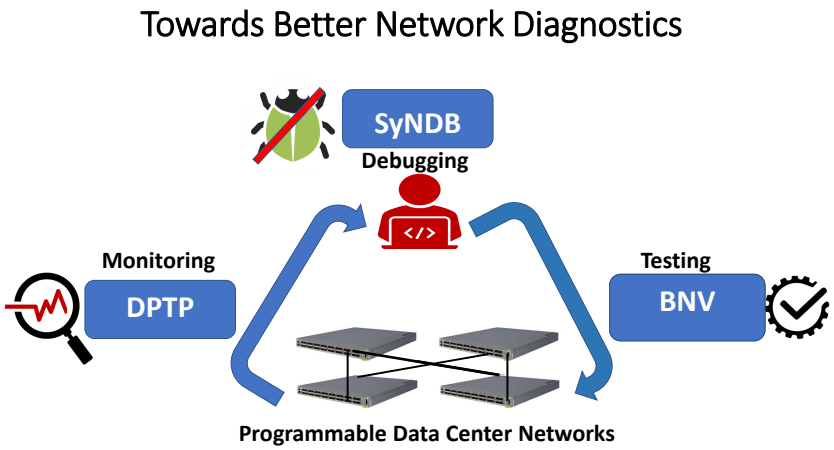


Figure 6.1: Solutions proposed in this thesis to improve network diagnostics

synchronization protocol for the network data plane. Next, to provide network-wide, correlated and fine-grained debugging, we design *SyNDB* which leverages in-switch storage and time synchronization to get recent packet-level telemetry before and after faults. Finally, to provide a hi-fidelity and scalable environment for testing the solutions, we design *BNV* which uses a novel loopback wiring technique and embedder to virtualize arbitrary network topologies.

6.1.1 *DPTP*

In the first part of this thesis, we design and implement *DPTP*, which to the best of our knowledge, is the first precise time synchronization protocol for the network data-plane. Through a measurement study, we quantify the variability in delays at every stage of data-plane processing. We incorporate these observations to offset the variable delays in the design and implementation of *DPTP*. *DPTP* performs much better than state-of-the-art techniques since it takes care of each variable delays in a component-level basis in switching. We evaluate *DPTP* on a hardware testbed and observe synchronization error of within ~50 ns between switches and hosts separated by up to 6 hops and under heavy traffic load. *DPTP* enables each packet to get the global time in the network data-

plane at line-rate. We note that this feature enables interesting applications are possible in the network data-plane (refer §6.2). *DPTP* source-code is open-sourced according to the NDA agreement with Barefoot Networks (an Intel company) and has seen significant interest from various groups working on programmable networks.

6.1.2 *SyNDB*

In the second part of this thesis, we design and implement *SyNDB*, which to the best of our knowledge, is the first packet-level synchronized record & replay framework for the entire network. *SyNDB* leverages the switch data-plane storage (SRAM) to temporally store packet recordings and exports the packet records to a collector for debugging upon a network fault. This saves the network and storage overhead for collecting the storing telemetry by a magnitude compared to previous works [29, 83]. It provides the unique ability of looking back at the trace of events before the occurrence of a network fault. In addition, *SyNDB* provide abstractions to configure *SyNDB* parameters at compile-time and run-time, which enable operators to constantly change their fault conditions and the states to be captured. Finally, *SyNDB* leverages well-known relational DBMS based SQL queries to aid in query-based debugging of complex network faults. A proposal based on *SyNDB* won Facebook research award, thus validating the need for such a framework to debug complex data center networks.

6.1.3 *BNV*

Finally, we design and implement *BNV* (Bare-metal Network Virtualization), which provides high-fidelity network experimentation at data-plane and control-plane using programmable switches. *BNV* can support arbitrary network topologies using a unique method of creating loopbacks in switches in order to provide high fidelity. We propose an ILP-based formulation for embedding of complex topologies to the substrate. We have built *BNV* on top of OpenVirtex, and also deployed on a production testbed

NCL [166]. Recently, *BNV* is also used by research works [37, 38] to emulate Fat-Trees with just two underlying switches. *BNV* accurately emulates the characteristics of the topologies implemented over the substrate network while isolating multiple experiments. *BNV*'s source-code is open-sourced and is available at https://github.com/praveingk/bnvirt.

6.2 Future Work

6.2.1 Towards Synchronous Data center Networks

With a tightly synchronized network data-plane using *DPTP*, we plan to rethink the implementation of distributed protocols like NetPaxos [124], which have strong assumptions like message ordering in the network. A recent work [180] has shown that designing network-level mechanisms to provide mostly-ordered multicast could improve the latency of Paxos by 40%. Further, the benefits of synchronized data-plane goes beyond applications to transport and scheduling protocols. Earlier works like Fastpass [181] have shown the feasibility of time-based scheduling in data centers and its benefits in reducing queuing. We believe *DPTP* enables building such global time-based scheduling in the network data-plane. We plan to explore this possibility in future.

While synchronous systems have been explored by other research communities : theory and real-time/CPS, the feasibility and benefits have not been studied in data center networking. Data center networks and distributed systems are typically asynchronous today. In future, we would be working towards identifying the building blocks and primitives to study the benefits and feasibility of building a synchronous data center. A Recent work [178] highlights the parts of the data-center that could be made synchronous. For example, faults could be handled better if the system knows it is supposed to receive a message at a particular time, but did not. Currently, asynchronous networks handle failure of an expected message using time-outs which are hard to decide and inefficient.

6.2.2 Practical Network Verification

We believe that the synchronized recording capability provided by *SyNDB* and network emulation capability provided by *BNV* allows creating network replays on staging environments that can be used as part of regression test suites. Such test suites are integral to large software development but are rare in network testing and management. For example, consider a bug in a network load-balancing implementation that causes a skewed distribution for a certain traffic combination. After the bug is fixed, *SyNDB*'s packet-level replays can be used to inject the exact same traffic combination into the network device to test the fix and also prevent regression.

6.2.3 Towards Self Driving Networks

Recent advances in machine learning and AI have spurned substantial research and deployment of self-driving vehicles. However, these technologies have not been leveraged by networking research to full extent. We believe the main challenge in applying these techniques is the availability of continuous network-wide states and telemetry which is synonymous to sensor outputs in self-driving vehicles. However, It is possible to adapt *SyNDB* to provide a consistent global snapshots of the network at a packet-level basis with parameters designed by the operator. It would then be quite practical to apply machine learning and AI techniques on this global packet-level network telemetry to infer network behavior, identify and predict network faults and re-program the network with the right intent.

Appendix A

Appendix

A.1 Network Debugging Scenario

In this section, we present a network debugging scenario to show the power of *SyNDB*. We portray the operator debugging in Section 4.7.3 in terms of step-by-step elimination based on the flow-chart shown in Figure A.1. The first step requires the network operator to identify the trigger type raised. Subsequent steps require additional information captured in the p-records. The steps needed to determine the root cause is shown as a flow-chart in Figure A.1. Each step within this flowchart corresponds to a simple SQL query which are shown in Table A.1. Note that the flow-chart is illustrative and provides an idea of how a network debugging is possible given the complete network-wide information. The custom field_list and the flow-chart can also be updated in an iterative manner, if the capture contains insufficient information, or the operator encounters a new bug not covered in the classifier.

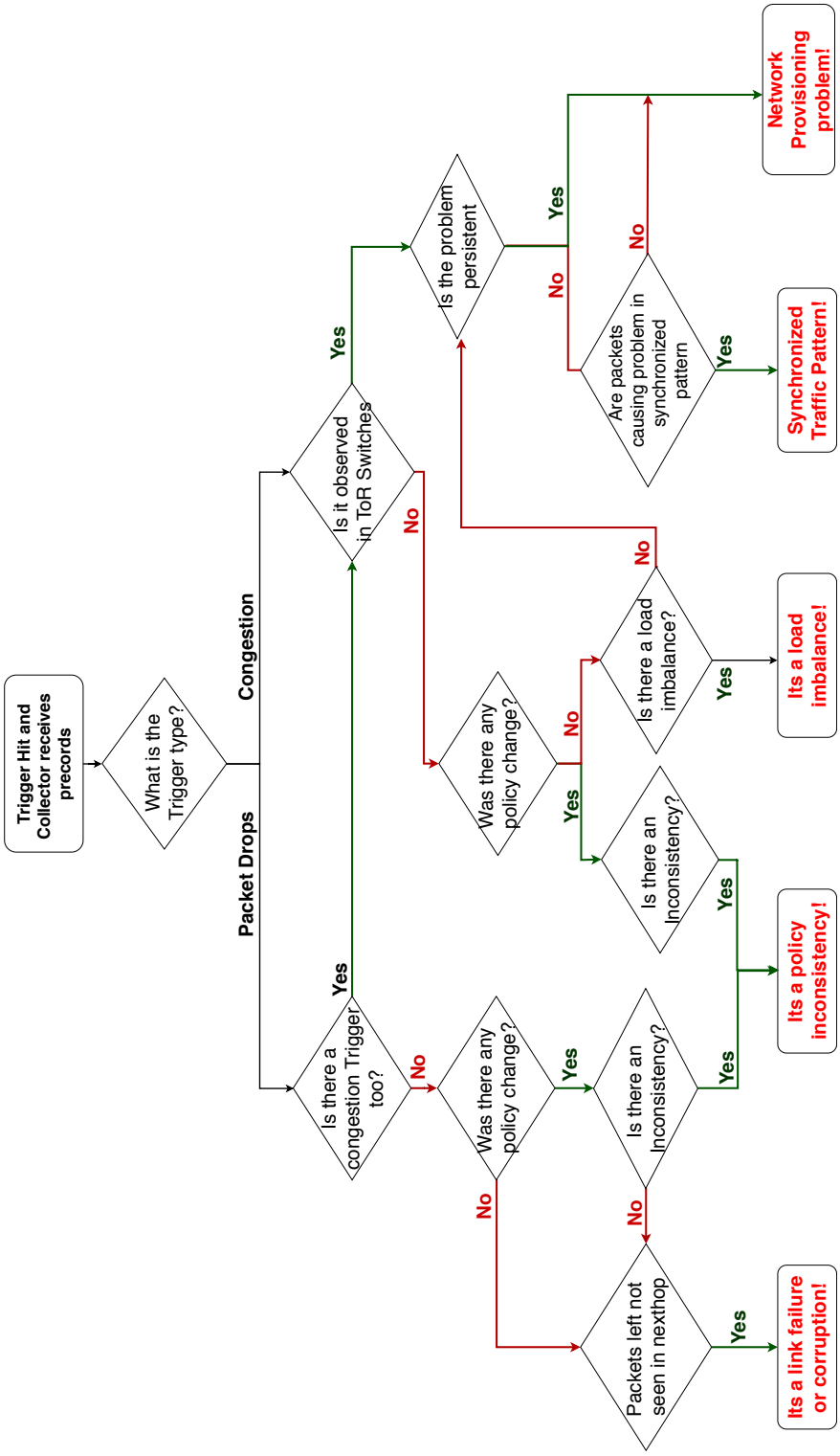


Figure A.1: Problem Debugging flow using queries

Table A.1: Queries for different questions in Figure A.1

Intent	Query
1. What is the Primary Trigger type?	`SELECT type FROM triggers WHERE time IN (select MIN(time) FROM triggers);`
2. Is there a congestion trigger?	`SELECT count(*) FROM triggers WHERE type=congestion;`
3. Is it happening in ToR switches?	`SELECT count(*) FROM triggers AS A, switches AS B WHERE (A.switch = B.switch and B.type="tor");`
4. Was there a policy change?	`SELECT A.control_plane_ver, B.control_plane_ver FROM packetrecords AS A, packetrecords AS B WHERE A.control_plane_ver != B.control_plane_ver;`
5. Is there an inconsistency?	`SELECT L.switch1, L.switch2, count(*) FROM (SELECT switch1,switch2 from links) AS L JOIN (SELECT * from packetrecords) AS A JOIN (SELECT * from packetrecords) AS B ON (A.hash = B.hash AND A.forwarding_rule_ver!=B.forwarding_rule_ver AND A.switch=L.switch1 AND B.switch=L.switch2) GROUP BY L.switch1, L.switch2;`
6. Packets left a switch by not seen in next hop?	*List the count of packets which left a switch, but not seen in any hops after the time it left the current switch (note: To weed out in-flight packets hop latency is added):* `SELECT switch,count(*) FROM packetrecords as A WHERE (A.switch IN (SELECT DISTINCT switch1 FROM links) AND hash NOT IN (SELECT hash FROM packetrecords AS A WHERE switch IN (SELECT switch2 FROM links WHERE switch1 = A.switch) AND time_in > A.time_out + HOP_LATENCY)) GROUP BY A.switch;`
7. Is there a load imbalance?	*List the queuing and link utilization of observed by packets travelling core-links:* `SELECT L.switch1, L.switch2, A.time_queue, A.link_utilization*8 FROM (select switch1,switch2 FROM links WHERE (switch1 IN (SELECT switch FROM switches WHERE type !="tor") AND switch2 IN (SELECT switch FROM switches WHERE type!="tor"))) AS L JOIN (select * from packetrecords) AS A JOIN (select * from packetrecords) AS B ON (A.hash = B.hash and A.time_in < B.time_in and A.switch = L.switch1 and B.switch = L.switch2);`
8. Is the problem persistent?	*List the number of times the same trigger condition (queue build-up) happens after the original trigger:* `SELECT count(*) FROM packetrecords AS A JOIN triggers as T ON (A.switch = T.switch and A.time_in > T.time) WHERE A.queue_duration > 10000;`
9. Are packets causing congestion in synchronized Pattern?	*List the time when packets leave the ToR switches:* `SELECT switch, time_out FROM packetrecords WHERE hash IN (SELECT hash FROM packetrecords AS A JOIN triggers as T ON (A.time_in < T.time AND A.switch = T.switch)) AND switch IN (SELECT switch FROM switches WHERE type = "tor");`

Bibliography

[1] Portable Switch Architecture. https://p4.org/p4-spec/docs/PSA-v1.0.0.pdf.

[2] Garter Forecasts Worlwide Public Cloud Revenue to Glow 17.5% in 2019. https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g.

[3] Google Datacenters. https://cloudplatform.googleblog.com/2015/06/A-Look-Inside-Googles-Data-Center-Networks.html.

[4] Hadoop distributed filesystems. http://hadoop.apache.org.

[5] Apache ZooKeeper. http://zookeeper.apache.org.

[6] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM, 2008.

[7] Liu Youyao, Han Jungang, and Du Huimin. A hypercube-based scalable interconnection network for massively parallel computing. 2008.

[8] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the*

Conference on High Performance Computing Networking, Storage and Analysis, 2009.

[9] Asaf Valadarsky, Michael Dinitz, and Michael Schapira. Xpander: Unveiling the secrets of high-performance datacenters. In *HotNets*, 2015.

[10] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

[11] Google Service Level Agreement. https://cloud.google.com/functions/sla.

[12] AWS Service Level Agreement. https://aws.amazon.com/compute/sla.

[13] U.S Cloud Computing Failure. https://www.reuters.com/article/us-cyber-cloud-disruption/u-s-cloud-computing-failure-could-spur-up-to-19-billion-in-losses-lloyds-idUSKBN1FC1UC.

[14] Microsoft to refund microsoft azure customers hit by 12 hour outage that distrupted xbox live. https://techcrunch.com/2013/02/24/microsoft-to-refund-windows-azure-customers-hit-by-12-hour-outage-that-disrupted-xbox-live/.

[15] British airways outage. https://www.forbes.com/sites/marisagarcia/2019/08/07/british-airways-flights-disrupted-by-it-failures-again/18fffa236688.

[16] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *IMC*, 2018.

[17] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, 2010.

[18] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *USITS*, 2003.

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 2014.

[20] SDNet PX Programming Language. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug1016-px-programming.pdf.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[22] P4Runtime. https://p4.org/p4-runtime.

[23] Cisco UADP. https://blogs.cisco.com/enterprise/new-frontiers-anti-aging-treatment-for-your-network.

[24] IEEE 1588 PTP and Analytics on Cisco Nexus 3548 Switch. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white-paper-c11-731501.html.

[25] Xilinx PreciseTimeBasic IEEE 1588 V2 IP Core. https://soc-e.com/products/precisetimebasic-ieee-1588-2008-v2-ptp-ip-core.

[26] Barefoot Networks. Tofino, 2018.

[27] Importance of network timesynchronization for enterprise solutions. https://www.microsemi.com/document-portal/doc_download/136220-importance-of-network-time-synchronization-for-enterprise-solutions.

[28] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, 2008.

[29] In-band Network Telemetry. https://p4.org/assets/INT-current-spec.pdf.

[30] Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. Debug determinism: The sweet spot for replay-based debugging. In *HotOS*, 2011.

[31] N. Honarmand and J. Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *ISCA*, 2014.

[32] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. DETER: Deterministic TCP replay for performance diagnosis. In *NSDI*, 2019.

[33] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[34] P. Wette, M. Drxler, and A. Schwabe. Maxinet: Distributed emulation of software-defined networks. In *IFIP Networking*, 2014.

[35] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *SOSP*, 2017.

[36] S. Y. Wang and I. Y. Lee. Minireal: A real sdn network testbed built over an sdn bare metal commodity switch. In *ICC*, 2017.

[37] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *SOSR*, 2019.

[38] Ting Qu, Raj Joshi, Mun Choon Chan, Ben Leong, Deke Guo, and Zhong Liu. SQR: In-network packet loss recovery from link failures for high-reliability datacenter networks. In *ICNP*, 2019.

[39] D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM*, 1988.

[40] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The switchware active network architecture. *IEEE Network*, 1998.

[41] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. Ants: a toolkit for building and dynamically deploying network protocols. In *IEEE Open Architectures and Network Programming*, 1998.

[42] Lim K.S. Lazar, A.A. and F. Marconcini. Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture. In *IEEE Journal of Selected Areas in Communications*, 1996.

[43] J. Biswas, A. A. Lazar, J. . Huard, Koonseng Lim, S. Mahjoub, L. . Pau, M. Suzuki, S. Torstensson, Weiguo Wang, and S. Weinstein. The IEEE P1520 standards initiative for programmable network interfaces. In *IEEE Communications Magazine*, 1998.

[44] General Switch Management Protocol (gsmp). https://datatracker.ietf.org/wg/gsmp/about/.

[45] Forwarding and Control Element Separation (forces). https://datatracker.ietf.org/wg/forces/documents/.

[46] T. V. Lakshman, T. Nandagopal, Ramachandran Ramjee, K. Sabnani , and T. Woo. The SoftRouter Architecture. In *HotNets*, 2004.

[47] N.Kang, Z.Liu, J.Rexford, and D.Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *CoNEXT*, 2013.

[48] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[49] POX SDN Controller. https://noxrepo.github.io/pox-doc/html/.

[50] Floodlight OpenFlow Controller. http://www.projectfloodlight.org/floodlight.

[51] RYU SDN Controller. https://osrg.github.io/ryu/.

[52] ONOS Project. https://onosproject.org/.

[53] OpenDayLight. https://www.opendaylight.org/.

[54] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 2014.

[55] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM*, 2013.

[56] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *SIGCOMM*, 2013.

[57] Intel. FlexPipe. https://goo.gl/PzPudG, 2018.

[58] The world's fastest and most programmable networks. https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[59] Cavium. XPliant Ethernet Switch Product Family. https://goo.gl/xzfLLo, 2018.

[60] Cisco Nexus 34180YC and 3464C Programmable Switches. Link.

[61] Broadcom Trident 3. https://packetpushers.net/broadcom-trident3-programmable-varied-volume.

[62] Innovium teralynx. https://www.innovium.com/products/teralynx.

[63] SDNet Packet Processor. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1012-sdnet-packet-processor.pdf.

[64] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991.

[65] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[66] C. D. Murta, P. R. Torres Jr., and P. Mohapatra. Qrpp1-4: Characterizing quality of time and topology in a time synchronization network. In *IEEE Globecom*, Nov 2006.

[67] Meinberg PTP Client. https://www.meinbergglobal.com/english/products/ptp-client-software-win-linux.htm.

[68] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM, 2016.

[69] R. Zarick, M. Hagen, and R. Barto. Transparent clocks vs. enterprise ethernet switches. In *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2011.

[70] FSMLabs Timekeeper Appliance. https://www.fsmlabs.com/timekeeper/enterprise-appliance.

[71] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[72] W. Lewandowski, J. Azoubib, and W. J. Klepczynski. Gps: primary tool for time transfer. *Proceedings of the IEEE*, 1999.

[73] Netflow. https://en.wikipedia.org/wiki/NetFlow.

[74] Sflow. https://en.wikipedia.org/wiki/SFlow.

[75] A simple network management protocol (snmp). https://www.ietf.org/rfc/rfc1157.txt.

[76] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *SOSR*, 2017.

[77] R.Joshi et al. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *APSys*, 2018.

[78] Xiaoqi Chen et al. Catching the microburst culprits with snappy. In *SelfDN*, 2018.

[79] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *ATC*, 2018.

[80] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *NSDI*, 2016.

[81] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM*, 2016.

[82] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. Mozart: Temporal coordination of measurement. In *SOSR*, 2016.

[83] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[84] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *SIGCOMM*, 2017.

[85] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *SIGCOMM*, 2018.

[86] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *OSDI*, 2016.

[87] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In *ATC*, 2011.

[88] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *SIGCOMM*, 2018.

[89] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM*, 2016.

[90] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *NSDI*, 2018.

[91] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *ATC*, 2015.

[92] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A simple and scalable method for sensing, inference and measurement in data center networks. In *NSDI*, 2019.

[93] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. DETER: Deterministic TCP replay for performance diagnosis. In *NSDI*, 2019.

[94] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *NSDI*, 2019.

[95] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *OSDI*, 2010.

[96] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. Openvirtex: Make your virtual sdns programmable. In *HotSDN*, 2014.

[97] R. Doriguzzi Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori. In *European Workshop on Software Defined Networking*, 2012.

[98] Flowspace Firewall. http://globalnoc.iu.edu/sdn/fsfw.htm.

[99] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments . *Computer Networks*, 2014.

[100] Robert Ricci and Eric et al. Eide. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *;login: the magazine of USENIX & SAGE*, 2014.

[101] Sivaramakrishnan S R, Jelena Mikovic, Pravein G. Kannan, Chan Mun Choon, and Keith Sklower. Enabling sdn experimentation in network testbeds. SDN-NFVSec, 2017.

[102] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[103] Network Virtualization gets Pgysical. https://octo.vmware.com/network-virtualization-gets-physical.

[104] Sapan Bhatia, Murtaza Motiwala, Wolfgang Muhlbauer, Yogesh Mundada, Vytautas Valancius, Andy Bavier, Nick Feamster, Larry Peterson, and Jennifer Rexford. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. CoNEXT, 2008.

[105] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In vini veritas: Realistic and controlled network experimentation. In *SIGCOMM*, 2006.

[106] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *NSDI*, 2015.

[107] Netronome Smart Nic. https://www.netronome.com/products/smartnic/overview.

[108] D.Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*, 2018.

[109] J. Mirkovic, T. V. Benzel, T. Faber, R. Braden, J. T. Wroclawski, and S. Schwab. The deter project: Advancing the science of cyber security experimentation and test. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

[110] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *SIGCOMM CCR*, 2003.

[111] Jelena Mirkovic, Hao Shi, and Alefiya Hussain. Reducing allocation errors in network testbeds. In *Proceedings of the 2012 Internet Measurement Conference*, IMC, 2012.

[112] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys Tutorials*, 2013.

[113] Roberto Riggio, Francesco De Pellegrini, Elio Salvadori, Matteo Gerola, and Roberto Doriguzzi Corin. Progressive virtual topology embedding in openflow networks. *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 1122–1128, 2013.

[114] Barefoot Networks. Tofino2. https://www.barefootnetworks.com/products/brief-tofino-2, 2019.

[115] Broadcom trident 4. https://www.broadcom.com/blog/trident4-and-jericho2-offer-programmability-at-scale.

[116] Broadcom StrataXGS BCM56970 Tomahawk II. https://www.broadcom.com/news/product-releases/broadcom-first-to-deliver-64-ports-of-100ge-with-tomahawk-ii-ethernet-switch.

[117] Broadcom StrataXGS BCM56980 Tomahawk III. https://www.broadcom.com/blog/at-a-glance-tomahawk-3-is-the-first-12-8-tb-s-chip-to-achieve-mass-production.

[118] Mellanox Spectrum 2. https://www.mellanox.com/page/products_dyn?product_family=277&mtag=spectrum2_ic.

[119] R.Miao et al. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *SIGCOMM*, 2017.

[120] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. SOSP, 2017.

[121] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *FAST*, 2019.

[122] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *HotNets*, 2017.

[123] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. NSDI, 2018.

[124] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. SOSR, 2015.

[125] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 2016.

[126] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. ASPLOS, 2017.

[127] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. SOSP, 2017.

[128] Amin Tootoonchian, Aurojit Panda, Aida Nematzadeh, and Scoot Shenkar. Distributed shared memory for machine learning. SysML, 2018.

[129] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. P4cep: Towards in-network complex event processing. NetCompute, 2018.

[130] Pravein Govindan Kannan, Ahmad Soltani, Mun Choon Chan, and Ee-Chien Chang. BNV: Enabling scalable network experimentation through bare-metal network virtualization. In *11th USENIX Workshop on Cyber Security Experimentation and Test (CSET 18)*. USENIX Association, 2018.

[131] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM, 2013.

[132] Junda Liu, Baohua Yan, Scott Shenker, and Michael Schapira. Data-driven network connectivity. HotNets, 2011.

[133] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC, 2015.

[134] P4 Language Consortium. 2018. Baseline switch.p4. https://github.com/p4lang/switch/blob/master/p4src/switch.p4.

[135] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

[136] Arjun Roy, Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari, Behnaz Arzani, and Alex C. Sneoren. Cloud datacenter sdn monitoring: Experiences and challenges. In *IMC*, 2018.

[137] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *SIGCOMM*, 2013.

[138] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *SIGCOMM*, 2017.

[139] K.He, E.Rozner, K.Agarwal, W.Felter, J.Carter, and A.Akella. Presto: Edge-based load balancing for fast datacenter networks. *SIGCOMM*, 2015.

[140] Q.Zhang et al. High-resolution measurement of data center microbursts. In *IMC*, 2017.

[141] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb.

[142] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter Rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *SIGCOMM*, 2015.

[143] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onu Mutlu. A Large Scale Study of Data Center Network Reliability. In *IMC*, 2018.

[144] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[145] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[146] Pyplot – matplotlib. https://matplotlib.org/api/pyplot_api.html.

[147] NetworkX Library. Link.

[148] Graphana. https://grafana.com/.

[149] P4-16 Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[150] Pf_ring. https://www.ntop.org/products/packet-capture/pf_ring.

[151] Wedge 100bf-32x. https://www.edge-core.com/productsInfo.php?cls=1cls2=180cls3=181id=335.

[152] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[153] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. Micro-burst in data centers: Observations, analysis, and mitigations. In *ICNP*, 2018.

[154] Rishi Kapoor, Alex C. Snoeren, Geoffrey M. Voelker, and George Porter. Bullet trains: A study of nic burst behavior at microsecond timescales. In *CoNEXT*, 2013.

[155] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM*, 2015.

[156] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[157] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[158] Amine Guidara et al. A study of the forwarding blackhole phenomenon during software-defined network updates. In *International Conference on Software Defined Systems (SDS)*. IEEE, 2019.

[159] V.Liu et al. Subways: A case for redundant, inexpensive data center edge links. In *CoNEXT*, 2015.

[160] EdgeCore Product Specification. https://people.ucsc.edu/ warner/Bufs/AS5900-54.pdf.

[161] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic external memory for switch data planes. In *HotNets*, 2018.

[162] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, 2016.

[163] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, and Ravi Netravali. A system-wide debugging assistant powered by natural language processing. In *SoCC*, 2019.

[164] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 2002.

[165] Cloudlab docs. http://docs.cloudlab.us/planned.html.

[166] National Cybersecurity Lab. https://ncl.sg.

[167] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.

[168] G. Jin and B. Tierney. Netest: a tool to measure the maximum burst size, available bandwidth and achievable throughput. In *ITRE*, 2003.

[169] Gurobi Optimization Framework. http://www.gurobi.com.

[170] Bnv usage. https://ncl.sg/bnvirtusage.pdf.

[171] Ting Qu et al. SQR: In-network packet loss recovery from link failures for high-reliability datacenter networks. In *ICNP*, 2019.

[172] HP 3800 DataSheet. http://h17007.www1.hp.com/docs/whatsnew/101811/4AA3-7115ENW.pdf.

[173] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Xin Sunny Wu, Dingming an Huang, and T.S. Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *SIGCOMM*, 2017.

[174] C. Hopps. Analysis of an equal-cost multi-path algorithm. *RFC 2992*.

[175] Intel Ethernet Switch Family Memory Efficiency. White Paper, 2009.

[176] Rochan Sankar Sujal Das. Broadcom smart-buffer technology in data center switches for cost-effective performance scaling of cloud applications. 2012.

[177] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 2011.

[178] Tian Yang, Robert Gifford, Andreas Haeberlen, and Linh Thi Xuan Phan. The synchronous data center. In *HotOS*, 2019.

[179] Self driving networks. https://www.juniper.net/us/en/products-services/what-is/self-driving-network.

[180] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *NSDI*, 2015.

[181] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM*, 2014.