저작자표시-비영리-변경금지 2.0 대한민국

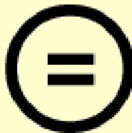
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.


변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사 학위논문

A Study on Software Defined Networking for Data Center Networks

데이터센터 네트워크에서의 소프트웨어 정의 네트워킹에 관한 연구

2015년 2월

서울대학교 대학원
전기·컴퓨터공학부
서 준 호

A Study on Software Defined Networking for Data Center Networks

지도 교수 권 태 경

이 논문을 공학박사 학위논문으로 제출함

2014년 12월

서울대학교 대학원

전기·컴퓨터공학부

서 준 호

서 준 호의 박사 학위논문을 인준함

2014년 12월

위 원 장	김 종 권	(인)
부위원장	권 태 경	(인)
위 원	엄 현 상	(인)
위 원	전 병 곤	(인)
위 원	백 상 헌	(인)

Abstract

A Study on Software Defined Networking for Data Center Networks

Junho Suh
School of Computer Science & Engineering
The Graduate School
Seoul National University

Software-Defined Data Center (SDDC) is a new paradigm of managing and operating IT infrastructure, where the resources in data center such as compute, storage, and networking are softwarized and delivered as a service to users on demand via application programming interface. Moreover, these resources are managed and controlled by software automatically–this is unprecedented in traditional IT infrastructure in which the infrastructure is typically defined by and tightly coupled with hardware and software. To realize this ideal environment, SDDC encompasses several virtualization technologies in compute, storage, and networking. In this thesis, we are more focusing on networking because the first two technologies are technologically advanced last few years, but networking evolution is still slow due to the vendor lock-in in which hardware and software of network element, which are typically proprietary, are tightly coupled. Moreover, in the network operators' perspective, configuring box-by-box manner with low-level commands results in increasing

management complexity and being error-prone.

To meet the requirements of today's users, enterprises, and carriers, Software-Defined Networking (SDN) is emerged. The main idea of SDN is to decouple control planes from data planes of network elements such as switches or routers and to replace the distributed, per-switch control plane with a (logically) centralized one on which SDN applications can control an operational network with a global network-wide view by enforcing packet forward- ing rules to the distributed data planes. This paradigm shift benefits network operators by (i) reducing the complexity of operations through automation while keeping more responsive, and (ii) optimizing the resources of operational networks with the global network-wide view to meet the dynamic nature of on-demand services in a cloud era. While SDN promises the enormous benefits as mentioned just before, it introduces new challenges: (i) increased control loop–gathers traffic and other measurements from the network and uses the gathered information to compute and install forwarding behaviours in the switches–due to the decoupling, and (ii) limitation on a distributed architecture–a (logically) centralized control plane is horizontally distributed to multiple physical servers–for large-scale production networks due to consistency overhead.

To address the first challenge, we propose, implement and evaluate OpenSample: a low-latency, sampling-based network measurement platform targeted at building faster control loops for software-defined networks. OpenSample leverages sFlow packet sampling to provide near–real-time measurements of both network load and individual flows. While OpenSample is useful in any context, it is particularly useful in an SDN environment where a network controller can quickly take action based on the data it provides. Using sampling for network monitoring allows OpenSample to

have a 100 millisecond control loop rather than the 1–5 second control loop of prior polling-based approaches. We implement OpenSample in the Floodlight OpenFlow controller and evaluate it both in simulation and on a testbed comprised of commodity switches. When used to inform traffic engineering, OpenSample provides up to a 150% throughput improvement over both static equal-cost multi-path routing and a polling-based solution with a one second control loop.

To address the second challenge, we propose FRACTAL, a framework for recursive abstraction of SDN control-plane, to address this problem. In FRACTAL, a large network is divided into multiple small networks, each of which is abstracted as a single virtual switch. This "divide-and-abstract" process is recursively iterated until a divided network can be handled by a single controller. A virtual switch is controlled by the higher level controller over OpenFlow, so that FRACTAL can coexist with other SDN mechanisms. We first carry out simulation experiments to demonstrate the issues of naive net- work partitioning. We then implement and evaluate FRACTAL with microbenchmark. Testbed-based experiments reveal that FRACTAL (i) adds small delays for non-local messages that cross divided networks, but (ii) achieves superlinearly increasing (control plane) throughput as the number of abstraction levels in the controller hierarchy grows.

Keywords : A Distributed Architecture, Network Monitoring, Software Defined Networking, Data Center Networks, Traffic Engineering

Student Number : 2008-20896

Contents

Abstract . . . . **i**

I. Introduction **1**

1.1 Data Center Networks 2

1.2 Software-Defined Networking 3

1.3 Challenges in Managing Data Center Networks through SDN 7

1.4 Thesis Structure 8

II. OpenSample: A Low-latency, Sampling-based Measurement Platform for Commodity SDN **10**

2.1 Introduction 10

2.2 Background 14

2.2.1 Data Center Network Workloads and Topologies 15

2.2.2 Non-OpenFlow Network Monitoring 16

2.3 OpenSample Design 20

2.3.1 Protocol-Aware Flow Statistics Detection 22

2.3.2 Probability of Flow Statistics Detection 23

2.3.3 Flow Detection Delay 26

2.3.4 Estimating Switch Port Utilization 27

2.3.5 Network State Snapshot Database 28

2.3.6 Traffic Engineering 28

2.4 Evaluation 30

2.4.1 Methodology 32
2.4.2 Results 36
2.4.3 Scalability 37
2.5 Related Work 38

III. FRACTAL: A Framework for Recursive Abstraction of SDN Control-Plane for Large-Scale Production Networks 41

3.1 Introduction 41
3.2 Background 44
3.2.1 A Distributed Control Plane in SDN 44
3.2.2 Impact of A Distributed Control Plane 46
3.3 FRACTAL Design 50
3.3.1 Domain Manager 53
3.3.2 "Many-to-One" Mapping 54
3.3.3 Rule Conflict Detection and Resolution 56
3.4 Evaluation 58
3.4.1 Microbenchmark 58
3.4.2 Experiments on campus network 60
3.5 Related Work 65

IV. Conclusion & Future Work 67

Bibliography 69

List of Figures

1.1 Data Center Topologies. 4

1.2 SDN Architecture Model 6

1.3 Challenges: i) Increasing Control Loop Latency and ii) Scalability in SDN control-plane. 7

2.1 An intuitive example of OpenSample-based traffic engineering running on a physical testbed. 13

2.2 The number of samples per second received at the collector as the sampling ratio increases. 19

2.3 The architecture of OpenSample providing measurement data to an SDN controller is depicted. sFlow agents running on switches provide samples to an sFlow collector which are then forwarded to flow and port analyzers. This information is aggregated into network snapshots and exposed via an open API. The SDN controller uses this API to make flow rerouting decisions. 21

2.4 The probability of getting at least two different samples from the same flow for varying flow sizes, sampling ratio and number of switches. Each line is labeled “1 in N” for the single switch case or “1 in N, k” for multiple switches where the sampling ratio is $\frac{1}{N}$ and there are k switches. 24

2.5 Aggregate throughput for ECMP, Polling, OpenSample-MLE and OpenSample-TCP traffic engineering on a k=4 fat free compared to a single non-blocking switch all with 10 Mbps links. Each bar represents the average of 5 runs with error bars showing standard deviations. Flow inter-arrival times are exponentially distributed with a 1 ms average. OpenSample uses a sampling ratio of N=50 and a 100 ms scheduling interval while polling uses a 1 s interval. 31

2.6 The percent of bytes remaining in flows at the time of detection and time of rerouting in the *Stride8* workload with 1 MB average flow size. 33

3.1 A simulation is carried out to demonstrate the performance issues in the horizontal replication of the network state among controllers in a distributed setting. Simulation results show that SSLBC performs better than LBC. However, it may affect the logic complexity of SDN applications. The distribution of flow duration affects the performance of SDN applications. 47

3.2 Microbenchmark tests reveal that the sync delays are affected by the workload to the controllers. 49

3.3 A network is illustrated as a two-level hierarchy of domain managers in FRACTAL. 51

3.4 A message translation is illustrated with two domain networks, each of which will serve as a virtual switch to the higher level controller. The message translator rewrites Openflow messages by using "*many-to-one*" mapping. 55

3.5 An example of a rule conflict is shown. Rule A already installed conflicts with new rule B, because rule B is more specific than rule A. A more specific rule has a higher priority in our design. 57

3.6 Average (per-switch) controller throughput is plotted as the level of controller hierarchy varies. The number of switches per bottom level controller exponentially decreases as the hierarchy level grows. 58

3.7 CDF of response times between switches and controllers is plotted when the hierarchy level is 4. As the level of a controller goes up (towards the top level controller), the response time increases due to processing and transmission delays. 60

3.8 A *FatTree* Data Center Topology with $k = 8$ is used for evaluation. Due to space limit, only the left half (domain A) is shown, which forms a $k = 4$ FatTree topology (domain B is omitted). Two domains are connected with 16 tunneling links. 61

3.9 For various traffic patterns, aggregate throughput for ECMP and TE on the k=8 FatTree topology (with FRACTAL) is compared to a single non-blocking switch. All links have 100 Mbps bandwidth. 64

List of Tables

2.1 The total bytes sent in 30 s and the percent of those bytes scheduled by traffic engineering for the *Stride8* workload. 35

Chapter 1

Introduction

Up until now, IT infrastructure including compute, storage, and networking resources have intentionally been kept physically and operationally separate from one another. And operators interacted with any of these resources via an operational monitoring system to configure policies, systems, and procedures. Further, these resources used to be normally possessed only to a dedicated users for local use. Of course, there is no authority to access given to users who are not member of that group. This is a custom way of managing and operating resources that IT department division has done.

As times went on, with the advent of cloud computing and virtualization, it promises a great flexibility to IT companies who carry on IT business in managing and operating IT infrastructure. Any of these resources can be softwarized and delivered as a service to users on demand. We call this kind of service as Infrastructure-as-a-Service (IaaS). With this benefit, cloud computing begins to attract attention from big three companies including Amazon, Microsoft, and Google that invest enormous fund to build data centers at global scale to provide IaaS to users.

At the same time compute and storage were evolving, however, network equipments, from switches and routers to middleboxes such as firewalls, NATs, and WAN optimizers, seemed to stand still in terms of innovations beyond feeds and speeds. The first reason to hinder innovation in network elements is that both hardware (i.e.,

data plane) and software (i.e., control plane) of network element are usually tightly integrated and they are vendor's proprietary, resulting in taking long time to adopt new protocols or technologies. The second reason is that network elements used to be complex and difficult to manage. Network administrators typically configure theses devices in box-by-box manner with low-level command line based interface. As a result, this has impeded innovation, increased complexity, and ossified ecosystem in network area.

Motivated by this observation, Software-Defined Networking (SDN) innovates the way of designing and managing networks in which it decouples control plane from data plane of network elements and replaces the distributed, per-network element control planes with a (logically) centralized control plane (i.e., controller). It then programs the forwarding behaviour of all network elements in an underlying network. Although SDN innovates networks, adopting SDN to data center networks is not straightforward.

The remainder of this chapter answers how to get toward Software-Defined Data Center, with Section 1.1 firstly describing some characteristics of current data center networks in more detail, with Section 1.2 describing Software-Defined Networking which is recently getting attention, and finally with Section 1.3 presenting how hard can it be to adopt Software-Defined Networking to data center networks.

1.1 Data Center Networks

While the purpose of this section is not to classify all data center workloads or topologies, we note certain properties in current data centers that others have ob-

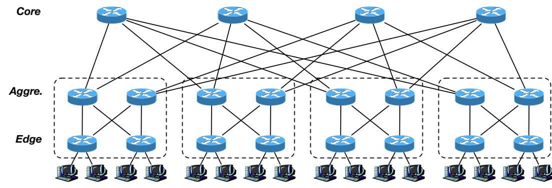
served and use these properties in this thesis. In particular, several characterizations of data center workloads [1, 2] have shown that while there are often *hot spots* in data center networks, the remainder of the networks is typically underutilized. Thus, if the topology has multiple paths for any given flow to select, it is likely that traffic experiencing congestion could be re-routed to an under-utilized path.

Fortunately, common current and future data center network topologies, e.g., Fat Tree(1(a)), HyperX(1(b)), and Jellyfish(1(c)), offer multiple paths between arbitrary endpoints. Despite this, most current data centers still use static routing configurations and/or equal-cost multi-pathing (ECMP). While these approaches attempt to minimize the occurrence of hot spots, hot spots still occur and these static approaches can do nothing in response to congestion once it occurs.

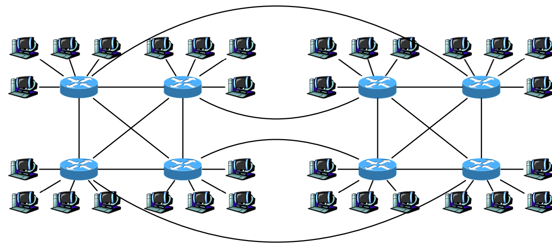
Further, we note that the traffic characterization indicates that flow sizes are approximately exponentially distributed and flow inter-arrival times are also exponentially distributed, i.e., a poisson process. As a consequence, in our emulation results, we assume exponential flow sizes and inter-arrival times.

1.2 Software-Defined Networking

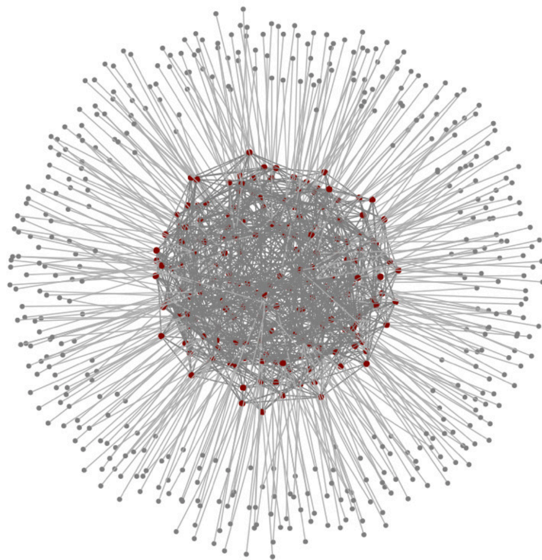
Software-Defined Networking (SDN) innovates the custom way of designing and managing traditional networks. It replaces the distributed, per-switch control planes of traditional networks with a (logically) centralized control plane (i.e., controller). It then programs the forwarding behaviour of all the switches in an underlying network. Furthermore, non-profit groups (e.g., Open Networking Foundation, ONF) well define a southbound protocol (e.g., OpenFlow) and encourage network



(a) $k = 4$ FatTree



(b) $L = 2$, $S_1 = 2$, $S_2 = 4$, $K = 1$, $T = 4$ Hyper-X



(c) Jellyfish

Fig. 1.1: Data Center Topologies.

vendors to open their functionalities of network elements supported through this protocol. A network element which complies with this protocol then performs rule matching defined in 5- or more tuples of flow against rules in flow table. If a rule is matched, it performs corresponding actions on the matched flow. Otherwise, it asks the controller for further process.

As can be seen, SDN can be characterized in three folds:

- **A (logically) centralized control-plane:** In traditional networks, control-planes are distributed to each network element. However, in SDN a centralized entity monitors state of underlying network elements and makes decision based on the state.

- **Abstraction:** No matter what the specific physical network elements exist SDN abstracts and hides the details of them. Thus, it is not required that the network applications care the details of the underlying physical network elements to configure and control, resulting in ensuring portability.

- **Programmability:** From the above attributes, network applications such as firewall, access control list, or traffic engineering can easily configure and manage the underlying network elements automatically via a combination of Application Programming Interfaces (APIs).

Moreover, ONF defines an architecture of SDN, illustrated in Figure 1.2.

- **The Infrastructure Layer:** This layer may involve either a number of physical forwarding elements or virtual switches, or both.

- **The Control Layer:** Due to the decoupling, the control planes distributed

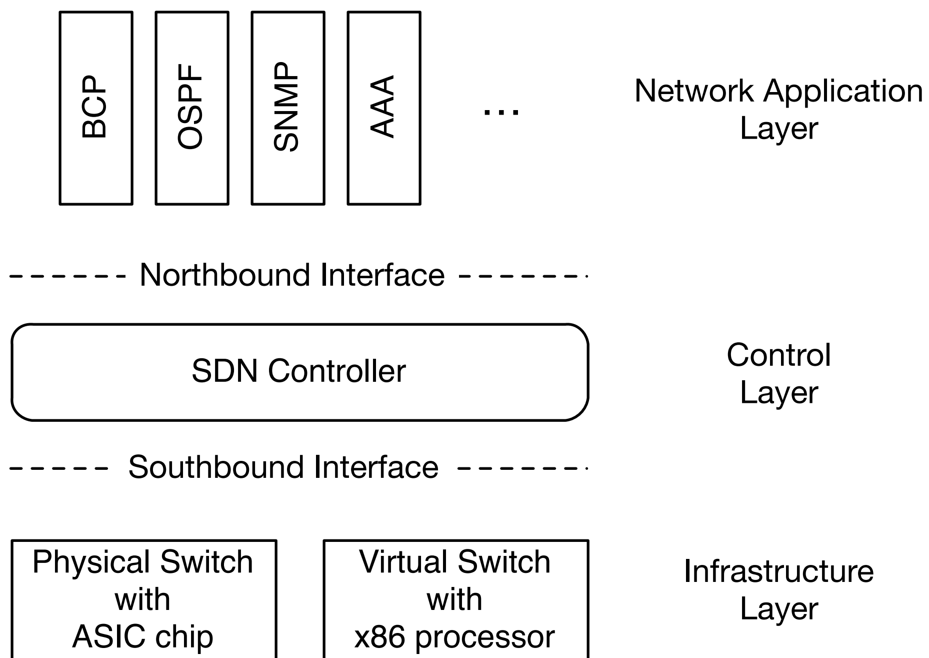


Fig. 1.2: SDN Architecture Model

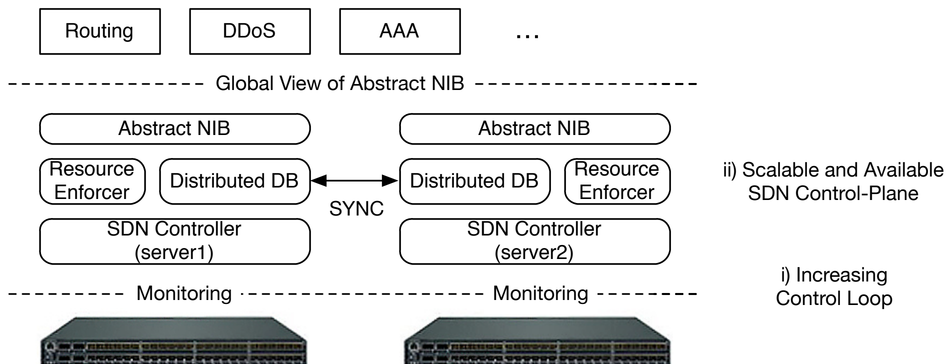


Fig. 1.3: Challenges: i) Increasing Control Loop Latency and ii) Scalability in SDN control-plane.

among each switch can now be put together at a (logically) centralized controller. It does not only control an underlying physical network elements, but also abstract the details of physical network elements. Further, it provides the basic network services, such as topology, network applications running on the application layer via the north-bound APIs.

- **The Application Layer:** In this layer, SDN users usually program their own network applications, including firewall, ACL, and traffic engineering, via combining the north-bound APIs.

1.3 Challenges in Managing Data Center Networks through SDN

Although SDN innovates the traditional ways of managing and operating networks, adopting SDN to data center networks is not straightforward. In this thesis,

we address two major challenges SDN have suffered, which arises from decoupling between control- and data-plane of the network. Moreover, it directly impact on performance (i.e., optimality) of the application layer in the SDN architecture model. The challenges are listed below:

- **Increasing a control loop:** Although the decoupling is a main benefit of applying SDN to the network, it naturally introduces increasing a control loop between them, including i) gathering traffic and other measurements from the network and ii) using the gathered information to compute forwarding rules and iii) installing forwarding behaviours in the switches. And the measurements are a bottleneck point, taking approximately 80% of the control loop shown in Figure 1.3.

- **Scalability in SDN control-plane:** Since a control-plane of SDN is a logically centralized, it must inevitably consider an architecture, where the control-plane is physically distributed due to the reason of scalability, availability, and responsiveness. It results in the two trade-offs: i) the trade-off between a consistency level (i.e., strongly consistent vs. eventually consistent) vs. performance (optimality) of the application layer in SDN architecture and ii) network application complexity vs. robustness to inconsistency in the distributed control-plane. Figure 1.3 depicts this phenomena.

1.4 Thesis Structure

The rest of this thesis is organised as follows: we first present OpenSample, a low-latency, sampling-based network measurement platform targeted at building

faster control loops for software-defined networks in Chapter 2. In Chapter 3, we present FRACTAL, a framework for recursive abstraction of SDN control plane for large scale production networks. Finally, we conclude by summarising our contributions and briefly alluding to future directions for research in Chapter 4.

Chapter 2

OpenSample: A Low-latency, Sampling-based Measurement Platform for Commodity SDN

2.1 Introduction

Software-defined networking (SDN) replaces the distributed, per-switch control planes of traditional networks with a (logically) centralized control plane that programs the forwarding behavior of all the switches in a given network. This centralized control plane, run on a controller, can act as a control loop that (i) gathers traffic and other measurements from the network and (ii) uses the gathered information to compute and install forwarding behaviors in the switches. Although there are two logical components to this control loop—measurement and control—the focus of the vast majority of SDN research has been on control. Some prior SDN research has included a measurement component [3–5], but the closest related work also notes this bias toward control [6].

OpenFlow [7], the dominant protocol used to implement SDN, provides two measurement techniques to create a global view of the network: `packet_in` messages and per-port/per-rule counters. Typically when a packet matches no switch rule, the switch sends a `packet_in` message containing the packet header (and possibly

payload) to the controller for handling, which may include installing new rules. Thus, in a typical setup, the controller receives one `packet_in` message[1] at the beginning of each flow. In addition, switches maintain counters to track the number of packets and bytes handled by each port and each OpenFlow rule.

In practice, neither of these measurement mechanisms enable a scalable, low-latency measurement system. The `packet_in` messages place a high burden on the local switch CPU and are typically limited to at most a few hundred per second [8,9]. Further, they only provide data when a new flow appears or a rule expires. Allowing a rule to expire typically causes the flow to be paused until the `packet_in` message is delivered to the controller and a new rule installed, which typically takes a few 10s of milliseconds. In the wide-area this might be tolerable, but in data centers and other local-area networks this is long enough to cause TCP timeouts and thus likely unacceptable.

On the other hand, port and flow counters are typically only updated every second or so [4], which limits the control loop of a controller to operating at a speed that is too slow to catch any but the largest flows [10]. Further, the space-granularity of counters are either per-port or per-rule. Per-port counters do not provide flow-level information, which either limits visibility into the network or requires more processing, i.e., tomography, to (probabilistically) disaggregate port-level information into flow-level tracking. Per-rule counters require that rules be installed at the granularity of the desired measurement, which couples forwarding and measurement, with both mechanisms installing rules in the switch. For example, Frenetic [5] is forced to break apart OpenFlow rules when the monitoring requests do not directly correspond to the

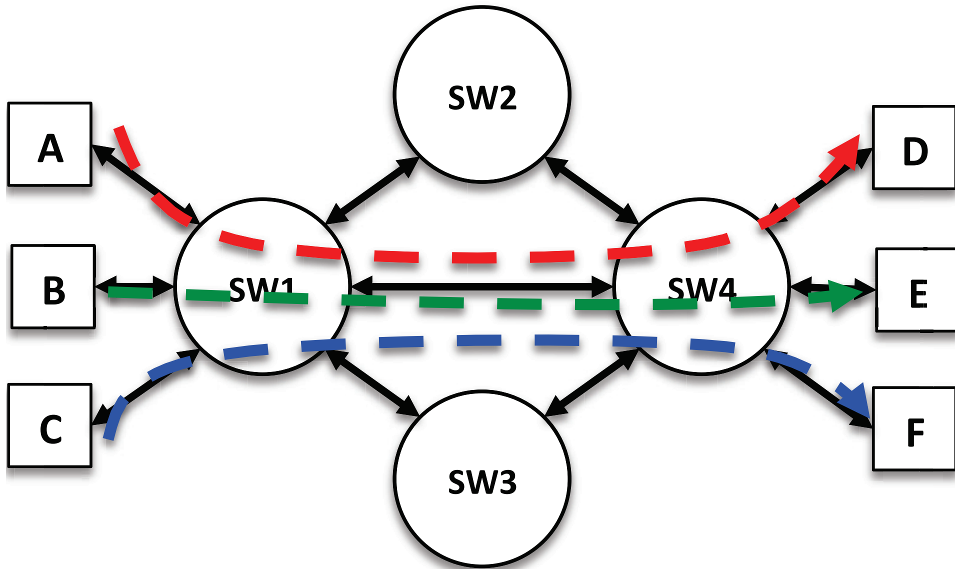
[1] It is possible to receive more than one `packet_in` message per flow especially if the flow is not connection oriented and thus sends many packets before hearing anything from the receiver.

forwarding rules, consuming even more scarce TCAM space [8].

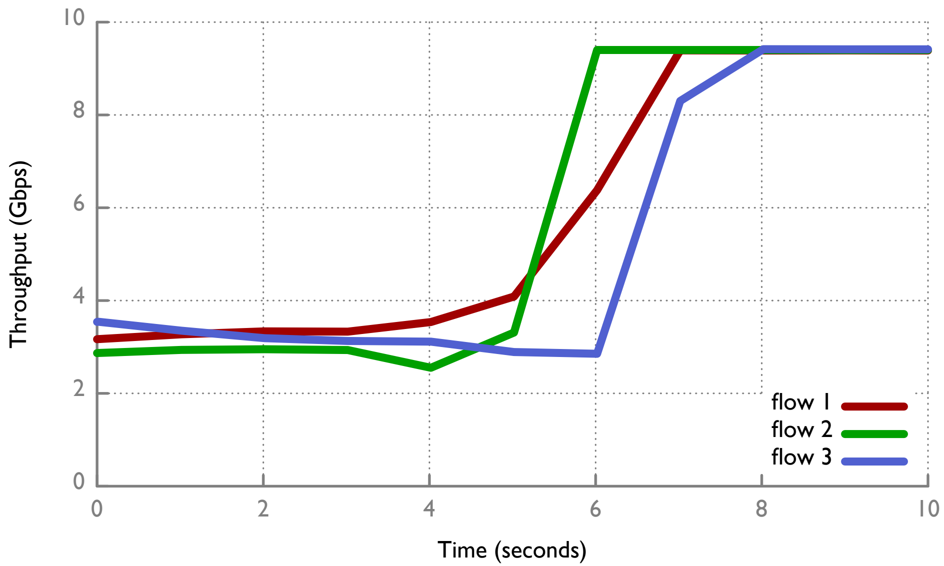
Ideally, a measurement system for software-defined networks would provide global visibility into the network and near–real-time data, i.e., latencies on the order of milliseconds. Further, it would scale to large traffic volumes without over-taxing switch control CPUs or the SDN controller.

In this paper we present the design and evaluation of a sampling-based SDN measurement system called *OpenSample* that achieves these goals. Rather than using the OpenFlow measurement mechanisms, OpenSample leverages the sFlow packet sampling functionality present in most switches. sFlow supports uniform random sampling of packets on a per-port basis. The switch forwards the header of, on average, 1 in every N packets traversing a given port to a collector. From these random samples, the collector can infer a variety of information about the network including the elephant flows present on each link and link utilization. Further, since each switch that a packet traverses can sample that packet, the effective sampling ratio of a network grows as the network grows and paths get longer.

To demonstrate the value of OpenSample's faster network monitoring, we implemented a traffic engineering application that uses OpenSample to detect congested links and the large flows using those links. In our experiments, traffic engineering informed by OpenSample provides up to 150% more aggregate throughput than both equal-cost multi-path (ECMP) routing and polling-based traffic engineering (inspired by Hedera [3]) when flow sizes are small, i.e., a flow in a slow start phase. Further, OpenSample can be implemented without the end-host modifications required by Mahout [4] and MicroTE [10], and does not require the use of expensive OpenFlow rules for fine-grained measurement like Frenetic [5].



(a) The physical OpenSample testbed with four 10 Gbps IBM G8264 switches configured with 6 hosts and 3 large flows.



(b) The throughput of three large flows both before and after enabling OpenSample-based traffic engineering.

Fig. 2.1: An intuitive example of OpenSample-based traffic engineering running on a physical testbed.

The following example, illustrated in Fig. 1(a), motivates and explains OpenSample-based traffic engineering. We configure a physical testbed with four IBM RackSwitch G8264 10GbE switches in a clique with three hosts attached to two of the switches. Hosts A, B, and C each generate long-running flows using iperf to hosts D, E, and F, respectively. Fig. 1(b) shows the throughput of each of the three flows both before and after we turn on traffic engineering. Initially, the three flows compete for bandwidth on the single shortest path (SW1-SW4) and converge to a fair share of approximately 3 Gbps each. However, there are sufficient redundant paths in the topology for each flow to follow its own disjoint path. After five seconds we enable traffic engineering, which is able to identify the three elephant flows and re-route two of them to uncongested paths. After that, each flow achieves 10 Gbps of throughput using the three different disjoint paths: 1-4, 1-2-4 and 1-3-4.

The remainder of the paper is organized as follows. Section 2.2 provides a more detailed look at sampling-based measurement as well as data center traffic characterizations. We describe the design and implementation of OpenSample in Section 2.3. Section 2.4 presents an evaluation of OpenSample using both emulation and results from a real testbed. We cover the work most closely related to OpenSample in Section 2.5.

2.2 Background

Before diving into the design, implementation and evaluation of OpenSample, we first present background material on both typical data center workloads and existing non-OpenFlow network monitoring approaches, which we use to drive the design

of our traffic engineering application.

2.2.1 Data Center Network Workloads and Topologies

While the purpose of this paper is not to characterize modern data center workloads or topologies, we note certain properties in current data centers that others have observed and use these properties in our construction and evaluation of OpenSample. In particular, several characterizations of data center workloads [1, 2] have shown that while there are often *hot spots* in data center networks, the remainder of the networks is typically underutilized. Thus, if the topology has multiple paths for any given flow to select, it is likely that traffic experiencing congestion could be re-routed to an under-utilized path.

Fortunately, common current and future data center network topologies, e.g., Fat Tree [11], HyperX [12], and Jellyfish [13], offer many diverse paths between arbitrary endpoints. Despite this, most current data centers still use static routing configurations and/or equal-cost multi-pathing (ECMP). While these approaches attempt to minimize the occurrence of hot spots, hot spots still occur and these static approaches can do nothing in response to congestion once it occurs.

Recent research efforts [3,8,10] use SDN to reroute flows in reaction to congestion. The result is a variety of good techniques for selecting alternate paths. However, these approaches rely on switch-based measurements with latencies measured in seconds, and are thus limited to detecting and rerouting only the largest flows.

Thus, all of the ingredients for substantially improved traffic engineering are present, except for the timely network measurements OpenSample provides.

Further, we note that the traffic characterization indicates that flow sizes are ap-

proximately exponentially distributed and flow inter-arrival times are also exponentially distributed, i.e., a poisson process. As a consequence, in our emulation results, we assume exponential flow sizes and inter-arrival times.

2.2.2 Non-OpenFlow Network Monitoring

While many readers are no doubt familiar with the network monitoring features of OpenFlow, i.e., per-port and per-rule byte and packet counters, they may be less familiar with other monitoring techniques such as NetFlow [14] and sFlow [15]. NetFlow produces per-flow statistics without requiring rules to be installed. sFlow provides real-time packet samples from individual switches. Mann et al. [16] recently compared NetFlow and sFlow for network monitoring at the hypervisor in virtualized data centers. Since congestion can occur anywhere in the network, we monitor both physical and virtual switches. And because the overhead of monitoring impacts its value, we are concerned with the overhead of monitoring techniques.

2.2.2.1 NetFlow

NetFlow [14] was originally developed by Cisco to provide a way to collect statistics about individual IP flows in a data network. In NetFlow, each switch (or router) maintains a flow cache that tracks flow statistics for each flow, usually identified by 5-tuple (source and destination IP address, source and destination TCP/UDP port, and IP protocol number) and type of service. As each packet arrives, its header fields are checked to see if it matches an existing entry in the flow cache. If it does, then the flow cache entry is updated appropriately, i.e., by incrementing the packet and byte counts. If the flow is not already present in the flow cache, a new entry in

the flow cache is created. NetFlow has four policies to decide when to send the flow record to a NetFlow collector: (i) when a TCP packet is seen with a FIN or RST flag indicating flow completion, (ii) when a flow idle timeout expires, (iii) when a hard timeout fires indicating that the flow has been tracked for y seconds regardless of whether it is still sending traffic, and (iv) when the flow cache is full and an entry must be evicted. When any of these four conditions hold, the switch sends a NetFlow record including flow statistics to a collector for further analysis.

Implementing NetFlow in hardware requires a dedicated CAM to track this information at line-rate. This hardware is not found in all switches and support for NetFlow is chiefly found in Cisco products and hypervisor vSwitches such as VMware ESX and Open vSwitch.

Further, NetFlow timeouts are specified at second granularity and in practice many implementations do not allow for values less than 30 seconds, so it provides little latency advantage over the low polling rates achievable using OpenFlow's per-rule counters. Since OpenSample is focused on low-latency network measurements, NetFlow is not a suitable choice for OpenSample.

It should be noted that later versions of NetFlow also include a "Sampled NetFlow" [17] mode that produces NetFlow records based on sampling 1 in N packets that traverse a switch rather than every packet. However, the samples are still applied to the records in the flow cache and records are still sent according to the same policy. Thus Sampled NetFlow incurs the same coarse-grained timeouts that make NetFlow unsuitable for low-latency monitoring.

2.2.2.2 sFlow

The sFlow [15] standard aims to provide fine-grained network measurements without requiring per-flow state at switches. Instead it relies on two forms of sampling: *packet sampling* and *port counter sampling*.

For packet sampling, the switch captures one out of every N packets on each input port[2]. It then immediately forwards the sampled packet's header encapsulated with metadata to a central collector. The metadata includes the sampling ratio of the port, the switch ID, the timestamp at the time of capture, and forwarding information such as the input and output port numbers.

The rate of samples sFlow produces is not constant; as it samples one in every N packets, the rate of samples varies based on the rate of packet arrivals. Since the packet arrival rates vary dramatically based on network load and packet size, the rate of samples also varies. Note that a packet passing through multiple switches is eligible to be sampled by every switch along the path. If a flow passes through k switches, combining the samples from those switches gives an effective factor of k increase in the sampling ratio.

From the gathered samples, the collector can probabilistically infer a number of flow statistics, e.g., it can estimate the expected number of packets and bytes in each flow by simply multiplying the number of sampled bytes and packets by the sampling ratio, N [18]. This approach statistically produces an unbiased estimator for the actual number of bytes and packets sent by the flow. In the remainder of this paper we refer to this technique for estimating the byte and packet counts of the flow as *Maximum*

[2] In actuality N is a parameter per port of a switch and need not be the same for all ports. However, in OpenSample, we assume the sampling ratio is fixed for all ports and leave exploration of per-port sampling ratio to future work.

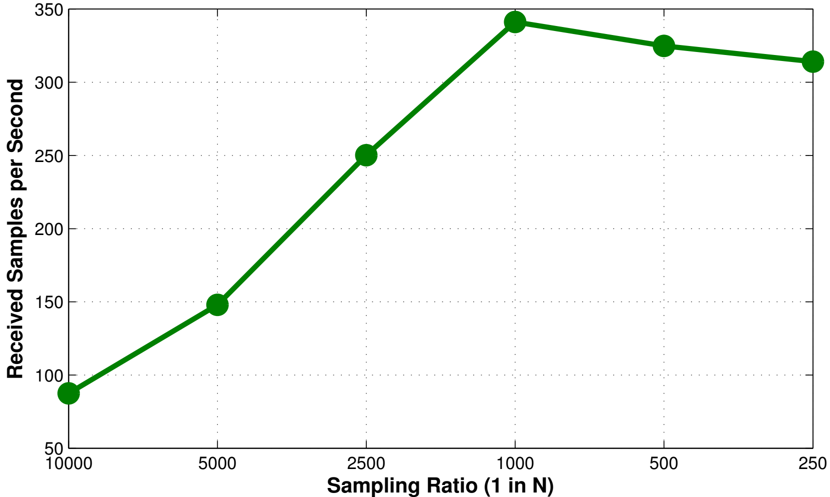


Fig. 2.2: The number of samples per second received at the collector as the sampling ratio increases.

Likelihood Estimation (MLE).

However, MLE has the limitation that it requires a large number of samples to provide accurate estimates of the true flow byte and packet counts. The expected relative error is inversely proportional to the square root of the number of samples, s, gathered from that flow. In particular, the percent error can be bounded as: % error $\leq 196 \cdot \sqrt{\frac{1}{s}}$ [18].

With an analysis of real data center workloads by Benson et al., [2] we found that an average of 60 flows totaling 3,000 packets arrive at each top-of-rack switch in any given 100 ms window. This means the average flow has 50 packets in a 100 ms window. Even if all 50 packets from a given flow are sampled, we can only estimate the flow's actual rate with approximately 30% error. In practice with realistic sampling ratio, even this is optimistic. Using MLE, there are only two ways to improve accuracy: (i) increase the sampling ratio and/or (ii) increase the sampling period.

The latter is not viable without violating OpenSample's goal of low-latency measurements.

Unfortunately, increasing the sampling ratio is difficult as well. Fig. 2.2 shows the number of samples per second our sFlow collector receives from one of our testbed switches as we increase the sampling ratio while keeping the amount of traffic going through the switch constant, i.e., 10Gbps. The number of samples per a second peaks at between 300 and 350 samples per second. We believe this limit is a consequence of the switch's control CPU being overwhelmed. With a limit of $\sim$350 samples per second, the expected number of samples for a given flow in a 100 ms time window that samples from 60 flows is less than one. While newer switches may provide faster control CPUs, it seems likely that it will be infeasible to get enough sFlow samples in a short period, i.e., 100 ms, to provide an accurate estimate of the flow throughput for the foreseeable future. Therefore, we need to do something other than MLE to estimate flow statistics accurately and in near-real-time, which is detailed in the next section.

2.3 OpenSample Design

The architecture of OpenSample is illustrated in Fig. 2.3. We use i) packet sampling, e.g., sFlow, to capture packet header samples from the network with low overhead and ii) use TCP sequence numbers from the captured headers to reconstruct nearly-exact flow statistics. Simultaneously, we use the same packet samples to estimate port utilization at sub-second time scales, described in detail below. We use a single, centralized collector that combines samples from all switches in the network

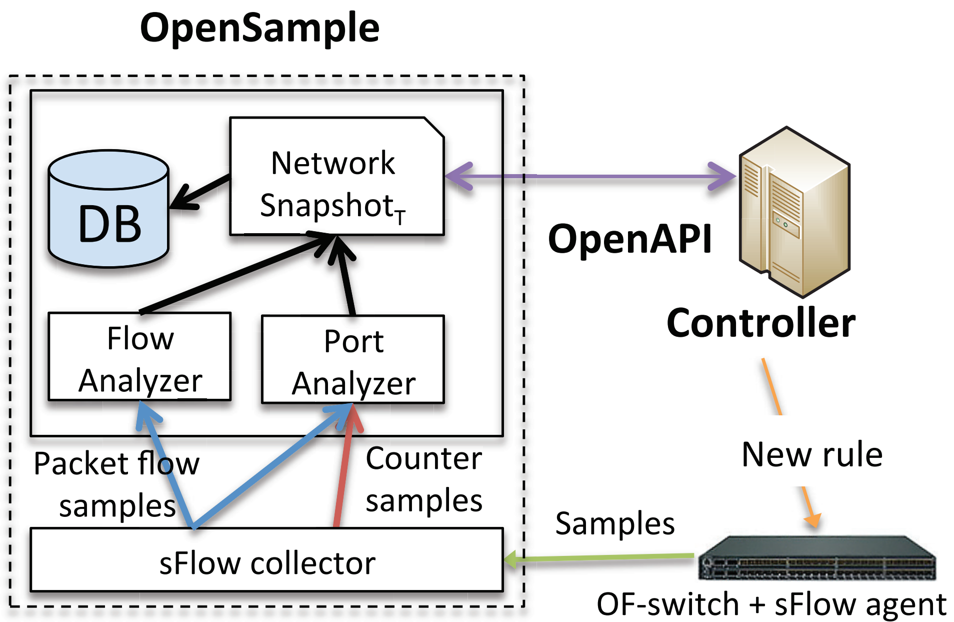


Fig. 2.3: The architecture of OpenSample providing measurement data to an SDN controller is depicted. sFlow agents running on switches provide samples to an sFlow collector which are then forwarded to flow and port analyzers. This information is aggregated into network snapshots and exposed via an open API. The SDN controller uses this API to make flow rerouting decisions.

to construct a global view of traffic in the network at both flow and link granularities.

Detailed network monitoring information has a variety of uses, including traffic engineering, resource provisioning, VM placement/migration, and intrusion detection. For illustration purposes, we focus on traffic engineering as the consumer of OpenSample's ability to very quickly detect elephant flows and estimate link utilization. In the following subsections, we describe how OpenSample extracts flow statistics from the samples, detects elephant flows, estimates link utilization of each switch's ports, and enables traffic engineering.

2.3.1 Protocol-Aware Flow Statistics Detection

As discussed earlier, accurately inferring flow statistics using MLE requires many samples per flow. To overcome the limitation of statistical inference, we exploit the fact that most traffic sent in data centers today is TCP traffic [19]. Each TCP packet carries a sequence number indicating the specific byte range the packet carries. Fortunately, when sFlow samples the header of TCP packets, this header also includes the TCP sequence numbers[3].

Thus, if we sample at least two distinct packets from a given TCP flow, we can compute an accurate measure of the flow's average rate during the sampling window by subtracting the two sequence numbers and dividing by the time between the samples.

Exploiting TCP information drastically increases estimation accuracy for any given sampling ratio. This TCP-aware sFlow analysis is the key innovation Open-

[3] The sFlow specification does not actually mention or require that samples include TCP sequence numbers, but it does specify that the preferred implementation should provide the raw packet header [20] and, in practice, both our physical switches and Open vSwitch [21] provide this information.

Sample incorporates compared to prior sFlow monitoring frameworks. In the next section, we provide analytic and simulation-based analysis of the probability of sampling two different packets from a given flow for a given number of switches, sampling ratio, and flow size. We also examine the expected time before receiving two samples from a flow for a variety or parameters.

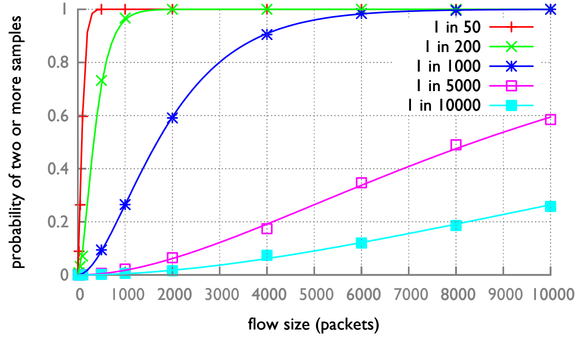
Our OpenSample-based traffic engineering mechanism considers any TCP flow for which it receives two or more samples to be an elephant flow and all elephant flows are candidates for traffic engineering. Thus, it considers far more flows to be candidates for rerouting than prior work [3, 4, 10].

Finally, we note that this approach is not limited to TCP, but can be extended to any protocol that includes sequence numbers in the header. Even if the sequence numbers represent packets and not bytes, as long as this is known a priori, the sequence numbers can be used to compute flow bandwidth rates substantially more accurately than MLE.

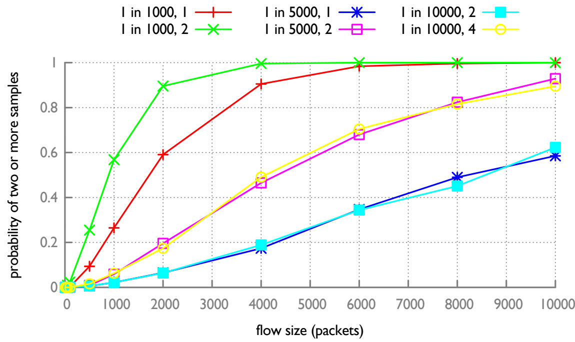
2.3.2 Probability of Flow Statistics Detection

To determine the probability of detecting a flow statistics using our enhanced *Protocol-aware flow statistics detection*, we analytically calculate the probability of getting at least two different samples from a single switch and use a simple simulator to find the same probability across one or more switches. In both cases, we evaluate the probability under a variety of different flow sizes and sampling ratio.

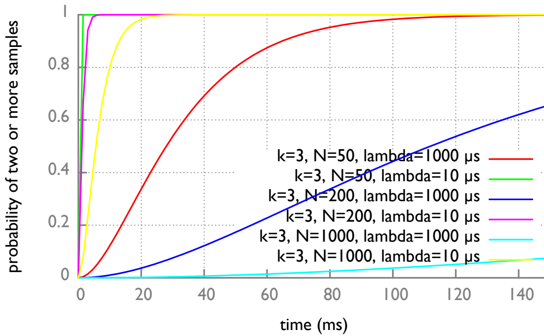
First, we develop an analytical model for a single switch. To ease exposition, we



(a) The analytical and simulated probability for one switch. The lines show the analytical values while the points show the simulated values.



(b) The simulated probability for one to four switches.



(c) The CDF of the delay to get any two consecutive samples from a flow based on average packet inter-arrival rate λ.

Fig. 2.4: The probability of getting at least two different samples from the same flow for varying flow sizes, sampling ratio and number of switches. Each line is labeled "1 in N" for the single switch case or "1 in N, k" for multiple switches where the sampling ratio is $\frac{1}{N}$ and there are k switches.

introduce the following variables:

$$\begin{aligned} n &= \text{number of packets in the flow} \\ p &= \text{probability of packet sampled } (0 \leq p \leq 1) = \frac{1}{N} \\ k &= \text{number of switches} \end{aligned}$$

Since there is no risk of sampling the same packet twice at a single switch, the probability of getting two samples from the same flow at the same switch is the probability of getting two samples from the same flow. More formally:

$$\begin{aligned} Pr\{\text{2+ samples}\} &= 1 - Pr\{\text{get zero or 1 sample}\} \\ &= 1 - Pr\{\text{get no samples}\} - \\ &\quad\; Pr\{\text{get 1 sample}\} \\ &= 1 - \binom{n}{0}(1-p)^n - \binom{n}{1}p(1-p)^{n-1} \\ &= 1 - (1-p)^n - np(1-p)^{n-1} \end{aligned}$$

When considering the case with more than one switch, the analysis becomes more complex because it must account for the possibility of sampling the same packet twice at two different switches. However, for realistic numbers of switches, k, and probability of packet sampled, p, the probability of sampling the same packet more than once at different switches is low enough that it effectively acts as the one switch model with a probability of packet sampled of kp. More formally:

$$Pr\{\text{2+ samples}\} \approx 1 - (1-kp)^n - nkp(1-kp)^{n-1}$$

To avoid analytical inaccuracy due to this simplification, we use a simple simulator to estimate the probability of getting two distinct samples from k switches. Each point is the result of 1000 simulations and the error bounds are small enough that we omit them. The results from both our analysis and simulator appear in Fig. 2.4. Fig. 4(a) shows the results of both the analysis and simulator for a single switch at various sampling ratio. The points are from the simulator and the lines are from the analysis. The two are almost identical, which provides validation for our model. Further, note that for sampling ratio greater than 1 in 200, we are nearly guaranteed to get two distinct samples from flows with more than 1000 packets, i.e., 1.5 MB or more. In contrast, for a 1000-packet flow and a 1 in 200 sampling ratio, MLE has an 87% estimated error.

Fig. 4(b) shows the simulation results for varying number of switches and sampling ratio. This shows that even for low sampling ratio, increasing the number of switches drastically improves the probability that we will get two distinct samples with low-cost. It also confirms the intuitive approximation to the one switch model. The lines with half the sampling ratio, but twice the switches closely follow each other, e.g., one switch with 1 in 5000 sampling produces the same result as two switches with 1 in 10000 sampling.

2.3.3 Flow Detection Delay

To determine how long it takes to acquire two samples from a given flow, we analytically calculate the expected delay, D. We express D as the sum of two random variables, X_1 and X_2, representing the arrival time (after the start of the flow) of the first and second sampled packets.

If packet arrivals are a Poisson process with average packet inter-arrival rate λ, then sample arrivals are a Poisson process with an average inter-arrival rate of λp. Thus, X_1 and X_2 are *i.i.d.* exponential random variables with mean $\frac{1}{\lambda p}$. Assuming a single switch, the delay to receive two samples can be stated:

$$
\begin{aligned}
E[D] &= E[X_1 + X_2] \\
&= E[X_1] + E[X_2] = \frac{1}{\lambda p} + \frac{1}{\lambda p} = \frac{2}{\lambda p}
\end{aligned}
$$

For the case with k switches, this is approximately $\frac{2}{\lambda k p}$. More accurately, D, is an Erlang-distributed [22] random variable with shape $\widehat{k} = 2$ and rate $\widehat{\lambda} = \lambda k p$. The CDF of D is shown in Fig. 4(c). We present values for $k = 3$ representing a typical 3-hop path in data center and for packet inter-arrival times of 1000 μs and 10 μs representing slow (12 Mbps) and fast (1.2 Gbps) flows (assuming 1500 byte packets). As can be seen, we easily detect all fast flows in less than 100 ms even with sampling ratio of 1 in 1000.

Note that there is nothing special about the choice of starting time and two samples. The analysis holds for the time to gather two samples after any arbitrary point in the duration of a flow.

2.3.4 Estimating Switch Port Utilization

In addition to packet samples, sFlow reports exact packet and byte counter values for each port in a switch every 5 seconds, similar to OpenFlow. This data is not useful for our goal of sub-second monitoring.

Thus we use packet samples to estimate link utilization at small timescales.

OpenSample estimates the utilization of each link during a given interval by multiplying the number of sampled packets in that interval by the average packet size. This is akin to treating all packets going through a particular link as a single "superflow" and using MLE to estimate its throughput. This estimate is accurate, despite MLE's poor error bounds, as it includes all samples from a given port.

2.3.5 Network State Snapshot Database

Every 100 ms, OpenSample generates a snapshot of the network state for consumption by other applications. This state includes the network topology (retrieved from the SDN controller), estimated utilization of every switch port, and the set of detected elephant flows. For each elephant flow, the snapshot includes the flow's five-tuple, its estimated bandwidth, and its current path. Applications such as traffic engineering can query the latest snapshot through an API, making them loosely coupled with the internals of OpenSample.

2.3.6 Traffic Engineering

Traffic engineering is a natural application of software-defined networks because the controller has a global view of the topology and it controls all of the switches' forwarding tables. An SDN controller forward a flow over a non-shortest path just as easily as over a shortest path, with no concerns about convergence time, forwarding loops, or black holes. Unfortunately, the high latency of network measurements has limited the effectiveness SDN-based traffic engineering [23].

Since we use exponentially distributed flow sizes, statistically a flow is expected to last as long as it has already lasted. Thus, if we detect an elephant flow that has

sent a significant amount of traffic, we can expect it to send that much again in the near future. By moving such flows from congested to uncongested paths, that future traffic will achieve higher throughput, as will any traffic with which it was competing. Because only the packets sent after the flow is scheduled can benefit from traffic engineering, fast detection and scheduling are crucial to the efficiency of traffic engineering. Thus, OpenSample's low-latency monitoring is an ideal candidate for traffic engineering.

While we experimented with a variety of scheduling algorithms, we found that, in general, the speed at which the scheduling control loop could operate made a more significant difference than the algorithm used. As a consequence, we use the *global first fit* algorithm presented in Hedera [3] for its simplicity. Our technical innovations focus purely on improving the speed of congestion and large flow detection rather than improved flow scheduling techniques.

While our algorithm is borrowed from Hedera, it operates on a 100 ms interval—50 times faster than Hedera's five second interval. As data center networks are upgraded from 1 Gbps to 10 and 40 Gbps, we expect flow duration to shrink, which means that traffic engineering must become proportionately faster to remain effective. Alternately, with a constant link speed faster traffic engineering can make better decisions, as shown in our evaluation.

By default, all traffic in our network follows shortest paths—we do not use Spanning Tree Protocol or equivalent. When multiple shortest paths exist, ties are broken by using equal-cost multi-path (ECMP) hashing based on the TCP/IP 5-tuple. Every 100 ms interval, the controller estimates the utilization of every link in the network and attempts to reduce congestion by moving elephant flows from highly-utilized

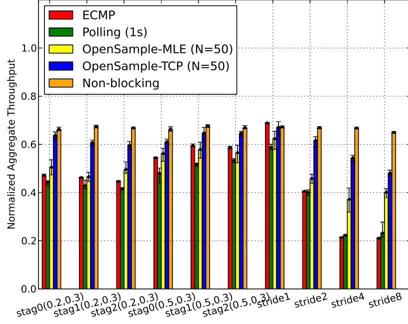
links to less utilized ones. The controller considers the set of detected elephant flows that traverse at least one congested link and uses a global first-fit algorithm to re-route them.

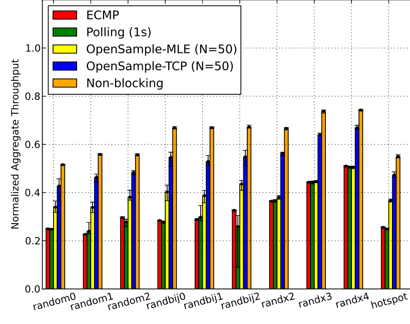
In OpenSample, both default forwarding and traffic engineering use OpenFlow. Engineered paths use high-priority OpenFlow rules and default paths use lower-priority rules. Scheduling a flow along a different path simply requires installing one new high-priority rule in each switch along the path. After an elephant flow ends, its rules time out and the switches automatically remove them. The time to install an OpenFlow rule—approximately 10 ms—is fairly small compared to OpenSample's 100 ms control interval.

2.4 Evaluation

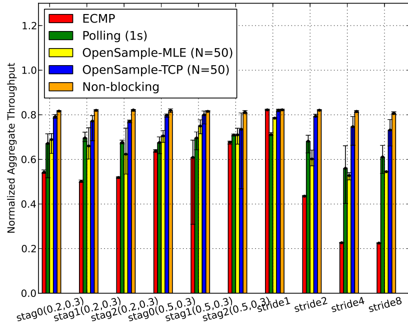
In this section, we present the results of our experimental evaluation of OpenSample. Our goal is to compare OpenSample's fast control loop against previous counter-polling–based approaches. To this end, we implemented OpenSample and a traffic-engineering application as modules for Floodlight [24], an open-source OpenFlow controller written in Java. We tested OpenSample on both the Mininet-HiFi [25] emulator and a physical network testbed of x86 servers connected with IBM RackSwitch G8264 switches.

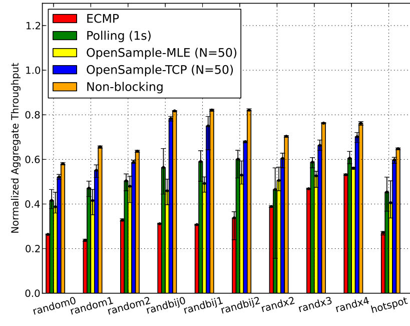
Because of our physical testbed's limited scale—only four switches—we predominantly used it to verify that our techniques work in practice, to validate our simulator framework, and to inform our design with the constraints of real-world hardware. As a consequence, we omit the testbed results except for the simple demon-





(a) Various traffic patterns with exponentially-distributed flow size with 1 MB average





(b) Various traffic patterns with exponentially-distributed flow size with 1 GB average

Fig. 2.5: Aggregate throughput for ECMP, Polling, OpenSample-MLE and OpenSample-TCP traffic engineering on a k=4 fat free compared to a single non-blocking switch all with 10 Mbps links. Each bar represents the average of 5 runs with error bars showing standard deviations. Flow inter-arrival times are exponentially distributed with a 1 ms average. OpenSample uses a sampling ratio of N=50 and a 100 ms scheduling interval while polling uses a 1 s interval.

stration of traffic engineering, shown in Fig. 2.1, and an evaluation of the sampling ratio supported on our switches, shown in Fig. 2.2.

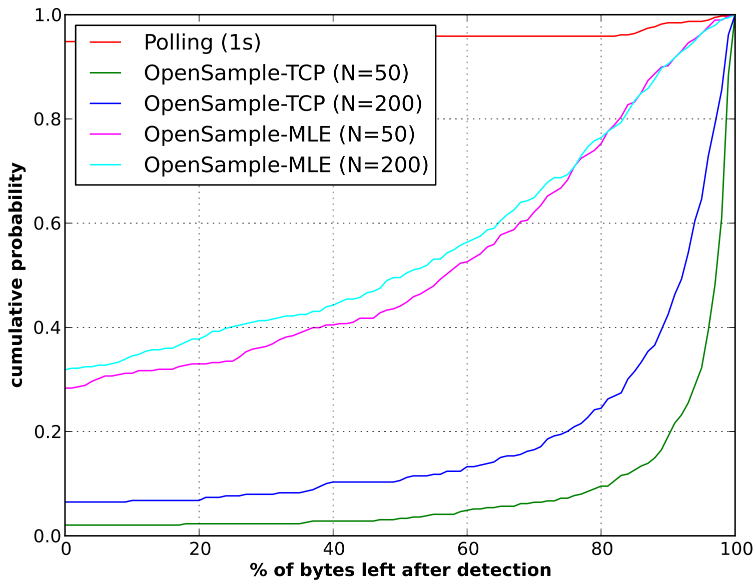
The remainder of this section focuses entirely on the emulation results, which provide an insight into OpenSample's operation at reasonable scales.

2.4.1 Methodology

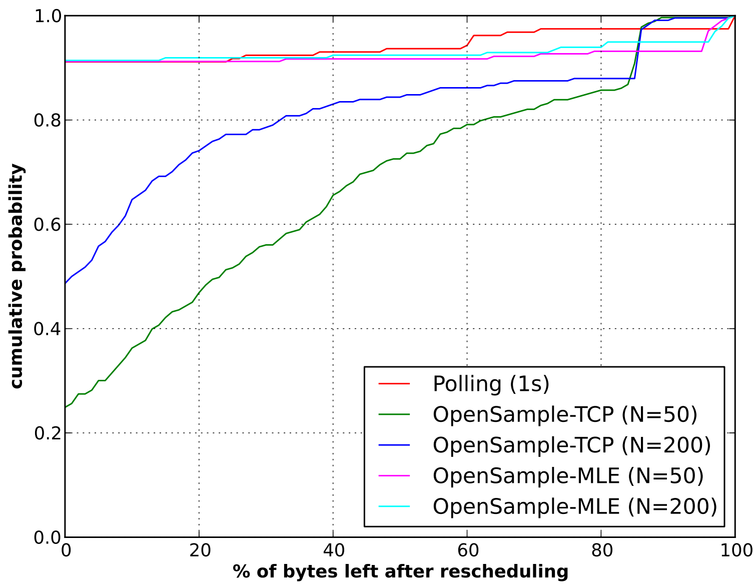
We use the Mininet-HiFi [25] network emulator to evaluate OpenSample in a controlled and repeatable environment. Mininet uses Linux containers to emulate hosts and Open vSwitch (OVS) to emulate switches, allowing a whole network to be emulated on a single computer. Mininet-HiFi uses Linux traffic shaping to emulate fixed-speed links, giving the emulated network realistic congestion and queueing delays. We set the link speed to 10 Mbps to allow for faster emulation. However, note that OpenSample can control either Mininet or a physical network with no changes.

We replicate the testbed benchmark of Hedera [3] with identical settings, including topology and workloads. We use a three-level *k=4 FatTree* as an example of a network topology with a realistic diameter and degree of multipathing (a real network would use a much larger switch radix such as k=64). We also run the workloads on an emulated single large *non-blocking* switch to determine the maximum throughput when constrained only by host NIC speeds. Note that we omit the details of FatTree and non-blocking topology and refer to Figure 2 in [3] due to the page limitation. However, we do note that because non-oversubscribed FatTree topologies are rearrangeably non-blocking, the non-blocking topology also serves as the achievable optimum performance on the FatTree.

We evaluate OpenSample with two sampling ratios: N=50 and N=200. Simple



(a) CDF of bytes left at the time of detection



(b) CDF of bytes left at the time of rerouting

Fig. 2.6: The percent of bytes remaining in flows at the time of detection and time of rerouting in the *Stride8* workload with 1 MB average flow size.

calculations indicate that to avoid exceeding the 350 sample per second limit on our physical switches, the sampling ratio should be closer to N=250,000 to handle line-rate traffic on all ports. The bulk of the discrepancy comes from the 1000x difference between link speeds in our emulations and our physical switches. The remainder stems from the fact that (i) in practice, not all ports operate at line rate simultaneously under realistic workloads and (ii) newer switches [26] have significantly faster control CPUs allowing for more than 350 samples per second.

We use a workload generator [27] originally written for Hedera. It has three different traffic communication patterns such as *Stride(s)*, *Staggered Prob (EdgeP, PodP)*, and *Random(u)*. Given a network with *N* hosts, the *Stride(s)* workload causes the host with index i to communicate with the host with index $(i+s)mod(N)$, resulting in an adversarial traffic pattern on our topology. On the contrary, in the *Random(u)* workload each host communicates with *u* different hosts that are chosen uniformly. *Staggered Prob (EdgeP, PodP)* has each flow stay within the same rack with probability *EdgeP*, stay within the same pod with probability *PodP*, and crosses pods with probability $1-EdgeP-PodP$.

Moreover, we generate two classes of flows: (i) short flows with an exponential distribution of mean 1 MB size, and (ii) long flows with the same exponential distribution but a 1 GB mean flow size. Both short and long flows follow the same inter-arrival time distribution, exponential with a mean of 1 ms.

For our performance baseline we use shortest-path ECMP forwarding with no traffic engineering. We compare three different measurement approaches: (i) Polling, (ii) OpenSample-MLE using a maximum likelihood estimator as described in Section 2.2.2.2, and (iii) OpenSample-TCP using TCP sequence numbers to infer through-

Technique	Total bytes sent	% of bytes scheduled
Polling (1s)	133 MB	25%
OpenSample-MLE (N=200)	166 MB	30%
OpenSample-MLE (N=50)	185 MB	36%
OpenSample-TCP (N=200)	226 MB	41%
OpenSample-TCP (N=50)	264 MB	62%

Table. 2.1: The total bytes sent in 30 s and the percent of those bytes scheduled by traffic engineering for the *Stride8* workload.

put as described in Section 3.3.

The Polling approach is designed to model measurement based on querying all of the flows in each switch as presented in prior work [3, 9, 28–30]. Based on their observations, we chose a polling rate of once per second as it matched the typical performance reported for the hardware switches evaluated. However we note that depending on the number of flows present in the switch and implementation details this latency varies from 75 ms to 15 s. We note that the systems which provided performance noticeably better than one second either used specialized interfaces to query counters or assumed that only a very small number of flows would be present.

To evaluate the performance of each approach, we measure the aggregate throughput achieved on the various spatial workload patterns after applying traffic engineering as described earlier. In all cases, we employ the same (Hedera) traffic engineering algorithm and change only the underlying measurement system. Meanwhile, we measure the total number of bytes sent and the fraction of those bytes that we were able to schedule on alternate routes for each case.

2.4.2 Results

Fig. 2.5 shows the throughput of a variety of workloads on our emulated configuration. Note that even the non-blocking switch does not achieve normalized throughput of 1.0 because sometimes two hosts transmit to the same destination, causing unavoidable congestion. Although a fat tree is a rearrangeably non-blocking topology, there is a significant gap between naive ECMP forwarding and the hypothetical single non-blocking switch due to collisions where multiple flows are hashed onto the same link. This gap makes the case for traffic engineering: with perfect flow scheduling it should be possible to approach the throughput of a non-blocking switch.

Polling is generally ineffective at detecting short flows (shown in Fig. 5(a)), because these flows are almost always finished before the controller can detect them. Polling is much more effective when elephant flows are longer than the polling interval (in this case one second), as in Fig. 5(b).

OpenSample-MLE outperforms polling in a few cases, but in general it suffers from the sampling bottleneck described earlier; by the time it receives enough samples to be confident that a flow is large, the flow is almost over. Thus OpenSample-MLE schedules relatively few flows.

OpenSample-TCP performs significantly better than either polling or OpenSample-MLE, because it detects and schedules elephant flows earlier. In most cases it achieves performance close to a non-blocking switch, even for fairly small (1MB) flows. Often it outperforms the alternatives by 25-50%.

Table 2.1 gives an intuition of the source of the performance gains. It shows both the total bytes transferred in the 30 s duration of the experiment and the percentage of those bytes that the traffic engineering manages to schedule for the *stride8* bench-

mark. The fraction of bytes scheduled can be considered a figure of merit for traffic engineering, since any bytes that are not scheduled are more likely to be subject to congestion. By this metric, OpenSample-TCP schedules over twice the fraction of bytes as the polling system. We can also see that the reduced congestion allowed the workload to send twice as much data in the same time, doubling throughput.

Fig. 2.6 provides deeper insight into the behavior of the measurement systems in the context of traffic engineering. Fig. 6(a) shows the fraction of bytes left in a flow at the time it is detected by each measurement system and Fig. 6(b) shows the fraction of the bytes left in a flow at the time it is actually rerouted, i.e., after the new forwarding rules have been installed. The results show that OpenSample-MLE and OpenSample-TCP dramatically outperform Polling when it comes to detecting short flows. When accounting for the scheduling interval, the time needed to compute routes and install the new routes, the advantage that OpenSample-MLE had vanishes, but OpenSample-TCP is still able to significantly outperform the alternatives.

In conclusion, OpenSample-TCP can detect elephant flows far earlier than the alternatives and, when used to drive traffic engineering, it enables the traffic engineering mechanism to schedule up to 60% of the bytes that hosts send (for *Stride8*) and to a 150% improvement in aggregate throughput (for *Stride4*) when flow sizes are small.

2.4.3 Scalability

The OpenSample collector is currently implemented with the assumption that a single machine running the collector will gather samples from all the switches in

the network[4]. Thus, the rate of samples that it can handle will limit the number of switches a single collector can monitor. Therefore, to evaluate how many samples per second a single OpenSample collector can handle, we implemented a benchmark tool by modifying Cbench [31]. Cbench is a tool intended to measure the performance of OpenFlow controllers by sending large numbers of `packet_in` messages as if they were from a collection of switches. Our modified version sends sFlow datagrams rather than sending `packet_in` messages.

While we omit a full presentation of these results for varying imposed loads, we found the OpenSample collector was able to process more than 100,000 samples per second. This means our current OpenSample implementation can handle samples from at least 285 switches assuming each switch sends 350 samples per second. That implies a single OpenSample is able to handle production data centers servicing 4K or 8K hosts with 1:2 or 1:5 oversubscription ratio, respectively [8]. Further, recent SDN controller efforts [32] have shown the ability to process and respond to as many as 10 million events per second. As a consequence, we believe that with more careful engineering, we could handle as many as 28,500 switches with a single collector, but we leave this to future work.

2.5 Related Work

There has been a significant amount of work on WAN traffic engineering, but much less work on traffic engineering on data center networks. Traffic engineering in data centers has only become a topic of interest in recent years due to the adoption of

[4]While we believe that it is possible to build a hierarchical version of OpenSample which uses multiple collectors to monitor more switches than a single collector can handle and aggregates their different network views, we leave this as a topic of future work.

multipath topologies; without multiple paths there is nothing to engineer.

Miura et al. [33] describe cases where parallel workloads can generate traffic patterns that cause congestion in data center networks that use single-path routing. They show that it is possible to increase network utilization by hand-optimizing routing tables, but do not provide any algorithmic solution.

The first practical data center traffic engineering work we know of is Hedera [3]. They found that congestion can occur even in full-bisection-bandwidth networks due to routing collisions—ECMP reduces collisions compared to single-path routing but does not eliminate them. They provide algorithms to estimate demand of network-limited flows and to schedule flows in a Clos network. They poll switch counters every five seconds to detect congestion and elephant flows and they use OpenFlow to reroute flows. Our work is significantly influenced by Hedera, while also taking into consideration the realities and limits of existing switches.

DevoFlow [9] addresses many inefficiencies of OpenFlow by "devolving" control of some things, such as microflow creation and multipath, to switches. They also propose using sFlow sampling but do not implement it using real hardware.

Helios [29] discusses building a fast control loop for hybrid optical/electrical data center networks and is able to complete a full control loop in approximately 100 ms, but does so by minimizing the number of installed flows to read and they make use of a proprietary RPC mechanism rather than a standard measurement mechanism like sFlow.

MicroTE [10] characterizes several data center workloads, finding similar collision-induced congestion as Hedera and DevoFlow. They modify servers to perform network measurement and report data to a central controller every second, which infers

congestion based on server-level information. They also describe optimizations such as server-based aggregation to reduce the overhead of network monitoring. Like Hedera and DevoFlow, MicroTE uses OpenFlow to perform rerouting.

Mahout [4] has similar goals to OpenSample, but seeks to provide low-latency elephant flow detection by using queue depth at end-hosts rather than using network-based measurements. When using switches with support for packet sampling, OpenSample offers a more resource-efficient measurement platform and is potentially more deployable, since changing switch configuration is likely easier than installing new software on all end-hosts.

Multipath TCP [34] (MPTCP) allows a single TCP connection to be split into multiple *subflows* that take different paths. Each subflow uses TCP congestion control to monitor and respond to congestion and MPTCP dynamically shunts data to less-congested paths. This can be viewed as a monitoring and traffic engineering system that is implemented entirely on end hosts using local knowledge.

OpenSketch [6] proposes configurable network measurement hardware that can calculate approximate "sketches" of common statistics, such as heavy hitters and flow size distribution. If implemented in switches, OpenSketch could enable even faster traffic engineering by efficiently and rapidly detecting elephant flows directly in switches. Although OpenSketch enables software-defined measurement in the same way that OpenFlow enabled software-defined forwarding, it is based on a clean-slate redesign of portions of the switch hardware.

InMon sFlow-RT [35] is similar to the network analyzer component of OpenSample. Although it is designed to integrate with an OpenFlow controller, its traffic engineering capabilities are not clearly documented.

Chapter 3

FRACTAL: A Framework for Recursive Abstraction of SDN Control-Plane for Large-Scale Production Networks

3.1 Introduction

Software Defined Networking (SDN) decouples the control planes from data planes of switches; it replaces a distributed, per-switch control plane with a (logically) centralized one on which SDN applications can control an operational network with a global network-wide view by enforcing packet forwarding rules to the distributed data planes. This paradigm shift benefits network operators by (i) reducing the complexity of operations through automation while keeping more responsive, and (ii) optimizing the resources of operational networks with the global network-wide view to meet the dynamic nature of on-demand services in a cloud era.

From an architecture perspective, the scalability, resilience, and responsiveness is required by designing a (logically) centralized SDN control-plane to cover a large sized network for future carrier-grade cloud or cloud 2.0 services. In such large scale networking environments, it is inevitable that a (logically) centralized SDN control plane should be distributed to multiple controllers, each of which is in charge of its own partition. Many research studies have been done to address the scalability

issue [36–39], most of which leverage *replicating* and *partitioning* a network state in the control plane horizontally.

However, some of recent investigations pointed out that the horizontal (or flat) distribution of network state is not so effective for large-scale production networks, e.g., inter-datacenter networks, multi-site enterprise networks, and wide area networks, due to the latency taken by querying or replicating the network state among controllers. [40] focused on the SDN controller placement problem – how many controllers are needed and where they should be located to cover the whole network. Further, [41] illustrated how the level of consistency impacts the performance and complexity of network control logics. In case of low complexity, the inconsistent network state information is likely to lead to suboptimal decisions. If any tolerance against inconsistency is to be allowed, highly complex control is required.

Meanwhile, [42, 43] took a novel approach for scalability, where the SDN control plane is hierarchically organized and hence the locality is exploited for local events. It effectively limits the propagation scope of events at the control plane, reducing the amount of the network state information transmitted or replicated. However, it requires (i) the redesign of the existing SDN control plane, e.g., services and network control logics, and (ii) the development of a new protocol between parent-child SDN controllers. To the best of our knowledge, there is still no solution to take into consideration both the locality and the distributed control plane at the same time.

Inspired by the literature, in this paper, we present FRACTAL: a **F**ramework for **R**ecursive **A**bstraction of SDN **C**on**T**rol plane for large-sc**AL**e production networks. FRACTAL leverages *recursion* to achieve scalability. The key idea behind FRACTAL is that a large network is partitioned into multiple small networks; a small network

can be further divided into smaller networks. This process can be recursively repeated depending on the whole network size, and the capacity of a controller. A partitioned network at any hierarchical level is called a *domain network*, which is controlled by a *domain controller*. Thus, the central idea of FRACTAL is to abstract a domain network to its parent controller as a single big virtual switch over OpenFlow.

The gain of FRACTAL is threefold. First, since FRACTAL partitions a single large network into multiple small networks, and utilizes a locality by organizing domain controllers hierarchically, it effectively eliminates the overhead of disseminating or replicating the information of events that are locally significant. Second, by hierarchically organizing the control plane, FRACTAL transparently abstracts a domain network as a single virtual switch that establishes a connection to its own SDN controller through the same southbound protocol like OpenFlow. Hence, there is no need to develop a new protocol between the controller and its switches. Third, FRACTAL no longer introduces the modification to existing services such as a host tracker, a topology manager, a statistics manager, a switch manager, and other SDN applications due to the properties of FRACTAL, e.g., transparency.

The contributions of this paper can be summarized as follows:

- We investigate the impact of a distributed control plane on SDN applications by simulation- and prototype-based experiments.
- We provide a scalable architecture of control plane with no modification in the existing systems and protocols.
- We implement FRACTAL with an open source SDN controller. Since its implementation has a form of software application, it can be ported to other plat-

forms easily.

- We deploy and evaluate the FRACTAL solution on the operational networks.

The remainder of the paper is organized as follows. Section 3.2 provides the performance issues in the distributed controllers by benchmark tests. We then describe the FRACTAL design in Section 3.3. Section 3.4 evaluates the FRACTAL's scalability and performance overhead. The deployment experience with our operational campus network is also discussed in this section. We cover the related work in Section 3.5.

3.2 Background

Before elaborating on FRACTAL, we provide background materials on the issues of a distributed control plane. We then perform experiments to show its limitations on simple topologies.

3.2.1 A Distributed Control Plane in SDN

The main feature of an SDN is a (logically) centralized control plane that provides SDN applications with a global network view. That is, it provides the information of network elements such as switches, routers, and middleboxes to SDN applications by abstracting the network state. On behalf of SDN applications, it collects the configurational and operational information from the network elements, and maintains a structured data model (say, a data schema). Further, it helps application developers building a wide variety of SDN applications easily through APIs.

There are many implementations of the data model of the control plane in both

research and industry communities. ONIX [36] is the first paper that defines the data model, called Network Information Base (NIB). The NIB holds a collection of information about network entities. ONIX further provides multiple methods for concurrently running applications to access and to modify the NIB consistently, and supports registration for notifications on state changes or the addition/deletion of an entity. And the recently published paper, ONOS [37], abstracts network elements by a graph data structure, and hence relies on the open source projects such as the Titan graph database and Cassandra key-value store for distribution and persistence, and the Blueprints graph API to expose the network state to applications.

In the industrial world, the OpenDaylight (ODL) project [39] is an open source project sponsored by Linux Foundation for building a production level of an SDN controller platform with which various user demands are satisfied. The ODL controller provides the model-driven service abstraction layer (MD-SAL) for the SDN control plane, aiming at providing a common and generic support to applications and plugins of network elements. Thus, MD-SAL allows developers of applications and plugins to develop through the APIs derived from a single model. MD-SAL adopts the IETF standards to describe the data model of network elements in the Yang [44] language and to configure the model through NETCONF [45]. Yang defines both configuration data and state data of network elements, as well as the format of event notifications from network elements. It thus allows network elements to define and trigger the remote procedure calls via NETCONF.

In order to improve performance, reliability and responsiveness, the SDN control plane must disseminate the network state among controllers (or servers). In the principle of distributed systems, the *CAP theorem* is proved that it is impossible to si-

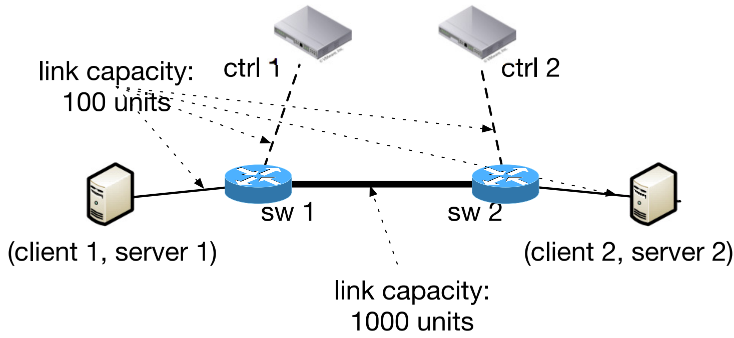
multaneously satisfy all three of the following guarantees: Consistency, Availability, and Partition tolerance. In other words, only two attributes out of three can be guaranteed; a distributed system typically seeks either Consistency and Partition tolerance (CP), or Availability and Partition tolerance (AP). In the large networking settings (which is our primary focus), AP is often the objectives due to delay of network state propagation.

3.2.2 Impact of A Distributed Control Plane

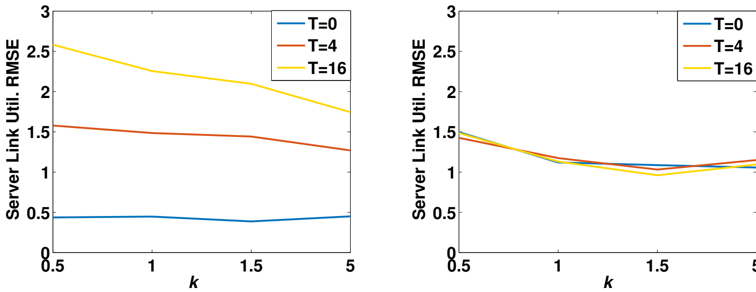
We now investigate a distributed SDN control plane to answer such a question–*how does a distributed SDN control plane impact the performance of SDN applications.*–which is a main motivation of this paper.

Simulation: To answer this question, we first carry out simulation experiments by leveraging a custom flow-level simulator with identical settings to [41], including the topology (as shown in Figure 1(a)), the sync overhead, and the controllers. When a flow request (to any of two servers) arrives at switch i (i is 1 or 2), the corresponding controller i decides to which server the flow is set up–the objective is to minimize the maximum link utilization in our network. If the controller has a global network-wide view by combining both the physical network state from within its domain as well as the link utilization of the other domain and it chooses the path with the lowest maximum link utilization, it is a simple link balancing controller (LBC). Note that it is likely that the global network-wide view is potentially stale. Whereas, if a controller is aware of the (potentially stale) global network-wide view and it has a logic to tolerate [41], this is a separate state link balancing controller (SSLBC).

We vary the workload (i.e., arrivals of flow requests) using exponentially dis-



(a) A linear topology is used in simulation setting; note that clients and servers are colocated. When a switch i (i is 1 or 2) receives a flow request from a client i, it will be forwarded to controller i.



(b) The imbalance between the two server links is worsened as the sync interval T increases.

(c) SSLBC is not affected by the sync interval since its logic is independent of the staleness of network state.

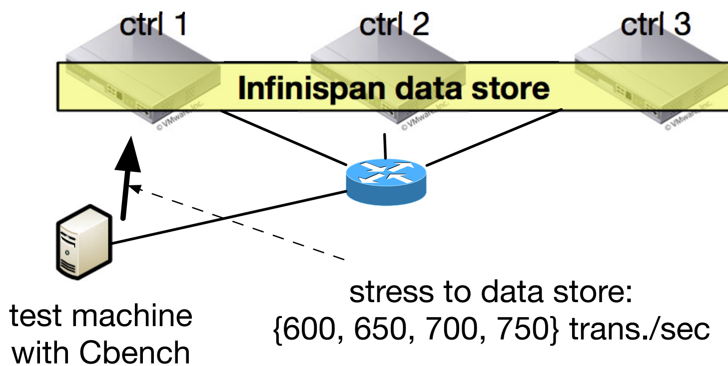
Fig. 3.1: A simulation is carried out to demonstrate the performance issues in the horizontal replication of the network state among controllers in a distributed setting. Simulation results show that SSLBC performs better than LBC. However, it may affect the logic complexity of SDN applications. The distribution of flow duration affects the performance of SDN applications.

tributed flow inter-arrival times (average is 10 unit time) and Weibull distributed flow durations. We also vary the sync overhead ($T = \{0, 4, 16\}$ in simulation unit time); $T = 0$ means the network state is instantly shared between the controllers. To change the distribution of flow durations, the shape parameter is varied ($k = \{0.5, 1, 1.5, 5\}$) with the fixed scale parameter ($\lambda = 10$ in simulation unit time). Note that λ is the average flow duration. Thus, as k decreases, the frequency of flows whose duration is less than the average increases. We measure a root mean squared error (RMSE) of the utilization of the links to the servers to compare the performance of LBC and SSLBC. For the details of the simulation experiments, refer to [41].

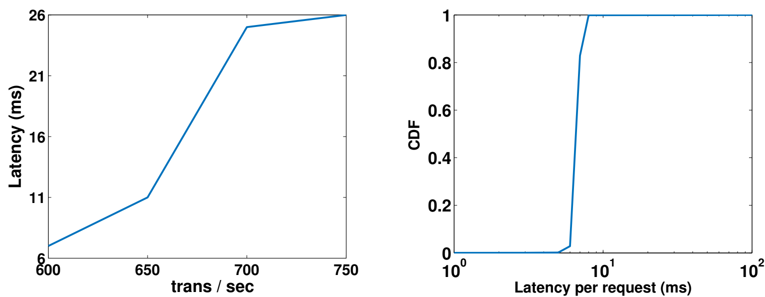
Figure 1(b) shows that the server link RMSE is increased significantly as we increase the sync interval of a distributed SDN control plane. This means that LBC is limited since it receives the state of the links (of the other controller) after T unit time, which may be stale. Consequently, SDN applications requires a logic to tolerate the staleness, resulting in increasing the complexity of SDN applications' logic. Figure 1(c) shows that SSLBC (which seeks to maximize the local link utilization) is much more robust against staleness than LBC due to its logic to tolerate.

Moreover, Figure 1(b) also shows that as k increases, the performance of LBC and SSLBC are more robust against the staleness of a distributed SDN control plane. This implies that we should utilize the locality of events for latency-sensitive SDN applications by limiting the propagation scope of short flows.

Microbenchmark: Since the simulation settings are not so realistic, we further carry out experiments on a real testbed. To this end, we build a cluster of three physical servers (Figure 2(a)), each of which is running a production-level SDN controller (i.e., OpenDaylight) on the platform of Intel Xeon 6 core CPUs at 2.1 GHz and 16 GB



(a) A test machine that generates events and three controllers are connected by a switch. When controller 1 receives the event, the updated state is replicated to the other two controllers.



(b) As the number of transactions per second increases, the synchronization (sync) delay increases due to computational overhead.

(c) CDF of the computation latencies of transactions is plotted when the transactions per second is 750.

Fig. 3.2: Microbenchmark tests reveal that the sync delays are affected by the workload to the controllers.

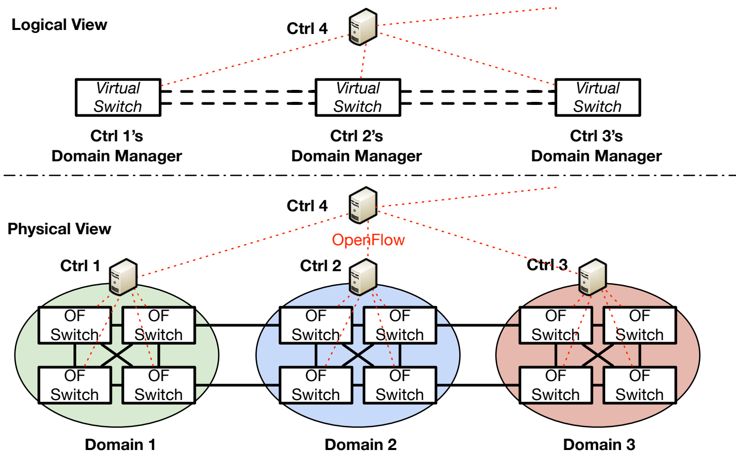
RAM. Note that OpenDaylight leverages Infinispan [46], which is a distributed in-memory key/value data store to implement a distributed SDN control plane.

To stress this data store, we use Cbench [47] on a test machine, which emulates a bunch of switches and sends `packet_in` messages, and receives `flow_mod` messages (from controller 1) to modify the flowtable. The workload to the Infinispan can be determined by varying the number of switches emulated and the rate of new flows generated at each switch. We vary the number of transactions per second (trans/sec= $\{600, 650, 700, 750\}$) to the data store.

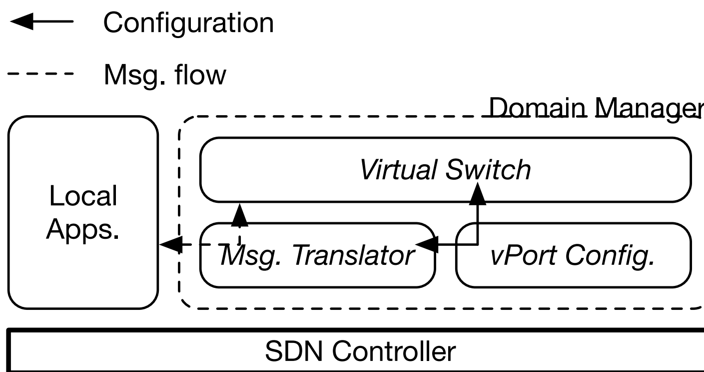
Figure 2(a) shows the testbed consisting of the three controllers and the test machine. In Figure 2(b), the sync delay (from the moment of a `packet_in` arrival at Infinispan to the moment of finish of synchronization among the controllers) increases as we increase transactions per second. At 750 transactions per second, the sync delay to synchronize the distributed SDN control plane is approximately 25 ms. As the average flow duration in production data center networks is around 100 ms [48], it is 25% overhead. However, Figure 2(c) is the cdf of the computation latencies (which is sync delay - queueing delay) at 750 transactions per second, showing that almost 90% of transactions take a few milliseconds of computation. This indicates 25 ms overall delay in synchronization is significant compared with the latency due to computation only.

3.3 FRACTAL Design

FRACTAL aims at building a scalable framework for an SDN control plane by partitioning a large-scale network into multiple small networks recursively. A par-



(a) A large-scale network is partitioned into three domain networks, which form a two-level hierarchy in FRACTAL. Red dotted lines indicate Openflow connections.



(b) Flow of operations in a domain manager is shown. Virtual ports are defined by vPort configuration, and Openflow messages forwarded to or arriving at virtual ports are transparently translated. The domain network is then abstracted as a single virtual switch.

Fig. 3.3: A network is illustrated as a two-level hierarchy of domain managers in FRACTAL.

titioned network is abstracted as a *virtual* switch, which is controlled by a single controller. Note that the higher level controller (of the whole network) controls small networks as virtual switches through the same southbound protocol (e.g., OpenFlow). Figure 3(a) illustrates how a network is hierarchically abstracted by FRACTAL.

FRACTAL can be characterized in the following for scalability.

- **Recursively partitioning a whole network:** FRACTAL exploits a divide-and-conquer approach to achieve scalability. It recursively breaks down a large-scale network into multiple small networks (a small network can be again partitioned into smaller networks) until a single controller can handle a bottom level network. At every level, a partitioned network is called a domain network, and its controller is called a domain controller. All messages meaningful only in a domain network are handled by its domain controller only. Otherwise, the messages will go to the next higher level controller. By limiting the scope of locally significant messages (of the SDN control plane), FRACTAL substantially reduces the overhead of synchronization among controllers in the whole network. Note that local messages are usually more frequent than non-local ones.

- **Transparently abstracting a domain network as a *virtual* switch:** To apply the recursive partitioning mechanism to the SDN control plane, a partitioned network should be modeled as a single network element. To this end, FRACTAL transparently abstracts a domain network as a *virtual* switch. A higher level controller then establishes connections to its lower level "virtual switches" with the same southbound protocol(e.g., OpenFlow). The virtual

switch of a domain network is actually the corresponding domain controller. In the abstraction process, a “many-to-one” mapping between multiple switches in its domain network to a single virtual switch happens.

- **Providing a co-existing solution to other distributed SDN mechanisms:** FRACTAL is not an exclusive solution to other distributed mechanisms which may horizontally replicate or partition the network state information to achieve availability and responsiveness. Rather FRACTAL coexists with other solutions for the local control plane to achieve the same goal.

Now we describe the details of a domain manager which is a main component of FRACTAL.

3.3.1 Domain Manager

To transparently abstract a domain network as a *virtual switch*, we exploit a control plane of software switches (e.g., [21, 49]) so that a domain controller of a domain network can serve as an OpenFlow switch. The domain manager realizes a *virtual switch*, and consists of a connection manager, a state manager, and a configuration manager. The connection manager manages OpenFlow connection instances with other controllers, checks the status of the connections by keep-alive messages, and processes OpenFlow messages. The state manager maintains the internal state (of a *virtual switch*) such as flow tables in which rules to be enforced for the domain network, ports from which virtual ports are built, and statistics of flows and virtual ports of its domain network.

Moreover, the domain manager provides a CLI-like functionality for querying

and configuring the *virtual switch* through the configuration manager. That is, the configuration manager provides the interfaces for adding (deleting) virtual ports to the virtual switch, installing (retrieving) rules, and so on. For example, to activate (deactivate) a *virtual switch*, a network operator inputs commands as follows:

```
fractal-util init <vsw-id>
fractal-util del vsw-id
```

and the configuration manager will return the `vsw-id` of the switch. To add (delete) a virtual port (vport) and specify it with a (physical) port in its domain network for transparent abstraction, the commands would be as follows:

```
fractal-util add-port vsw-id <vport#> sw-id port#
fractal-util del-port vport#
```

3.3.2 "Many-to-One" Mapping

When a domain controller provides transparent abstraction (for its next higher level controller), a "many-to-one" *mapping* between the switches in its domain network and a virtual switch is required; in particular, there are two types of translations: (i) topology translation and (ii) message translation.

Topology Translation: The first step for the topology translation is to specify the relations between the switches of a domain network and a *virtual switch*. For this, a domain manager maintains a bidirectional hash map consisting of a pair of `(vsw-id, vport#)` and `(sw-id, port#)` which is specified by a `fractal-util`

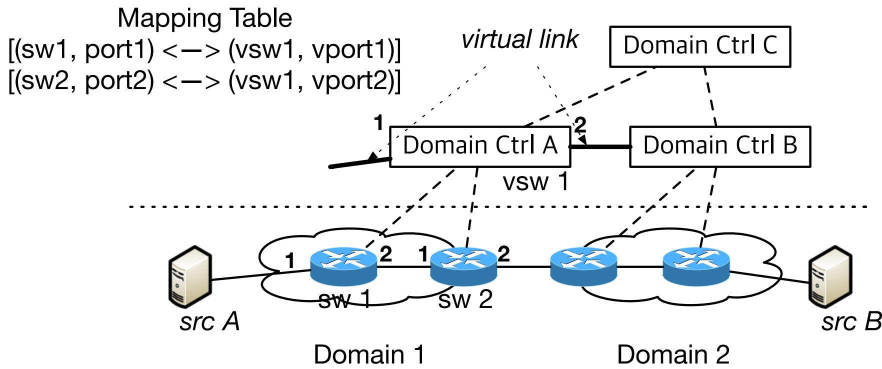


Fig. 3.4: A message translation is illustrated with two domain networks, each of which will serve as a virtual switch to the higher level controller. The message translator rewrites Openflow messages by using "*many-to-one*" mapping.

in the above. Further, the map provides a straightforward translation for the topology.

Message Translation: Since a domain network is seen as a *virtual switch* to other (domain) controller, all non-local messages triggered on a switch in the domain network should look like the messages triggered on the *virtual switch*. In OpenFlow, two types of messages are believed to be influenced by this message translation: *switch-to-controller* messages such as `packet_in` and *controller-to-switch* messages such as `packet_out` and `flow_mod`.

Figure 3.4 illustrates a "*many-to-one*" mapping between two switches in a domain network A and a FRACTAL's *virtual switch* in a domain controller A. In this example, a mapping table contains two entries: [($switch_1$, $port_1$)↔($vswitch_1$, $port_1$)] and [($switch_2$, $port_2$)↔($vswitch_1$, $port_2$)]. When a `packet_in` message is triggered from $port_1$ of $switch_1$, it is to be exported to the domain controller C, the message

translator in the domain manager rewrites the `in_port` field in the `packet_in` message with *vport*$_1$ of the *vswitch*$_1$ by looking up the mapping table. In the opposite direction, it rewrites the `out_port` field in the `packet_out` message with *port*$_1$ of *switch*$_1$. However, the message translation of `flow_mod` message is not straightforward, since it is to be interpreted in the *virtual switch* rather than a domain network. For this, the message translator interacts with local SDN applications such as the topology manager to transform the rules written for the *virtual switch* into ones for the domain network.

Moreover, there is one special message for the link layer discovery protocol (LLDP). It is used for advertising the neighbor link status between *virtual switches*, but real LLDP packets are in a domain network. We call this mechanism *virtualized* LLDP dissemination. In FRACTAL, only the links between the virtual ports have to be exposed to the higher level controller. Thus, the controller of a lower level domain network has to redirect the LLDP packets incoming at virtual ports. The translator takes over this role, it virtualizes and redirects these LLDP packets to its higher level controller by the above process. Besides, in case of LLDP forwarding, the translator also changes the port information in the LLDP packets so that they are interpreted by the higher level controller.

3.3.3 Rule Conflict Detection and Resolution

When translating `flow_mod` from a rule written for a *virtual switch* to the one for the corresponding domain network, it is possible that the translated rules may conflict with the ones already installed in the domain network. Specifically, it happens when the translated rules are more specific than the ones already installed. For

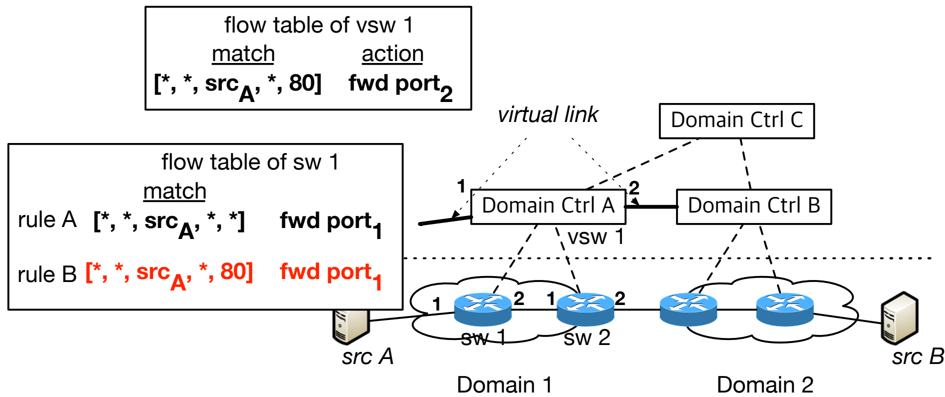


Fig. 3.5: An example of a rule conflict is shown. Rule A already installed conflicts with new rule B, because rule B is more specific than rule A. A more specific rule has a higher priority in our design.

example, Figure 3.5 illustrates an example of a rule conflict that rule A and rule B conflict with each other because rule B has one more attribute (i.e., `ip_port=HTTP`) than rule A.

The domain manger maintains an *n-dimensional flowspace* to keep track of what low-level OpenFlow rules are already installed in its domain network. Here n is the number of tuples in rule specification. When a new rule is installed in the *virtual switch*, it checks the flowspace. If no conflict is detected, it installs the rules with the higher priority than the ones installed in the domain network. Otherwise, when a lower level controller receives a rule that conflicts with local rules, it explicitly denies the rule installation request, `flow_mod`, by replying an error message, `flow_mod_failed`. Then, the network application receiving the error message may perform path re-calculation.

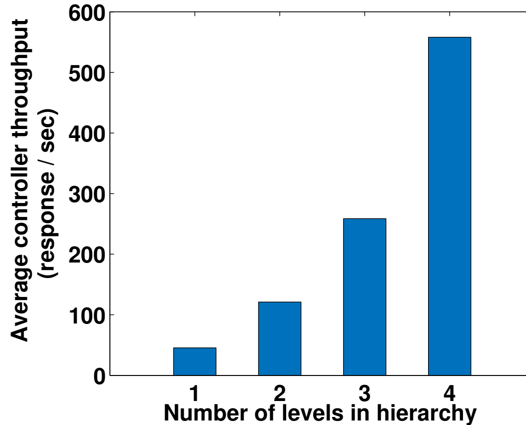


Fig. 3.6: Average (per-switch) controller throughput is plotted as the level of controller hierarchy varies. The number of switches per bottom level controller exponentially decreases as the hierarchy level grows.

3.4 Evaluation

In this section, we explain a microbenchmark used to evaluate the scalability and the performance overhead of FRACTAL. This microbenchmark is performed on an emulated network with Cbench [47] to stress FRACTAL. We then describe how FRACTAL performs on a real network considering an SDN application for traffic engineering on data center network.

3.4.1 Microbenchmark

Scalability: To evaluate the scalability, we use Cbench which stresses FRACTAL by varying the number of switches that send a number of requests, and measures the per-switch responsiveness (i.e., the number of `flow_mod` messages corresponding to `packet_in` messages per second per switch) as we vary the number of con-

trollers that comprise FRACTAL.

Figure 3.6 shows the controller throughput as the level of the controller hierarchy increases (from 1 to 4). The controller throughput means the number of `packet_in` messages processed by a bottom level controller divided by the number of switches that the controller manages. Assuming 80 switches in the whole network, we partition the network by adopting *binary tree* as a hierarchical structure. For instance, if the hierarchy level is 1, a single controller handles all the 80 switches. If the level is 4, each bottom level domain controller handles only 10 switches (there are 8 bottom level controllers). The per-switch controller throughput superlinearly increases as the hierarchy level grows since the FRACTAL framework divides the network, in which a single controller can in turn process relatively much more messages in the per-switch perspective.

Performance Overhead: Partitioning a network incurs some overhead to the FRACTAL framework. In abstracting a domain network, its domain manager looks up the mapping table for the topology and rewrites the messages for the context of the *virtual switch*. Notice that there is no additional overhead to the data plane; packets are forwarded at line rate. FRACTAL adds no additional traffic in the control plane either. FRACTAL only adds the processing overhead to the messages that requires mapping and resolution (in case of rule conflicts).

To quantify the overhead due to partitioning, we measure the response time between `packet_in` and the corresponding `flow_mod` when the level of controller hierarchy is four. Note that level-1 and level-4 indicate the bottom level and top level controllers in the hierarchy, respectively. If there is no rule to forward the packet, a switch triggers the `packet_in` message. The SDN controller then processes this

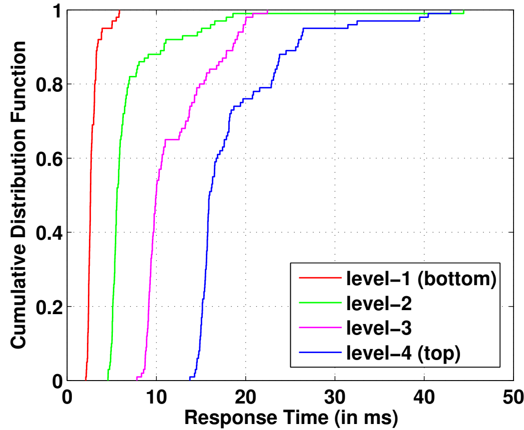


Fig. 3.7: CDF of response times between switches and controllers is plotted when the hierarchy level is 4. As the level of a controller goes up (towards the top level controller), the response time increases due to processing and transmission delays.

message and generates the `flow_mod` message to the switch to install the forwarding rule.

Figure 3.7 evaluates the performance overhead that partitioning incurs. The response time increases by 5 ms on average for crossing each level in the hierarchy. Thus, there is a tradeoff between the per-switch controller throughput and the response time of a non-local message in determining the level of controller hierarchy.

3.4.2 Experiments on campus network

A data center topology: We now build a data center network on our campus network; two island networks are connected on our campus network as an overlay. That is, one network contains a ($k = 4$, where k is the number of pods per core switch and a pod consists of two layers of $k/2$ switches) FatTree topology connected with the other through tunnels, both of which forms a $k = 8$ FatTree topology with 16

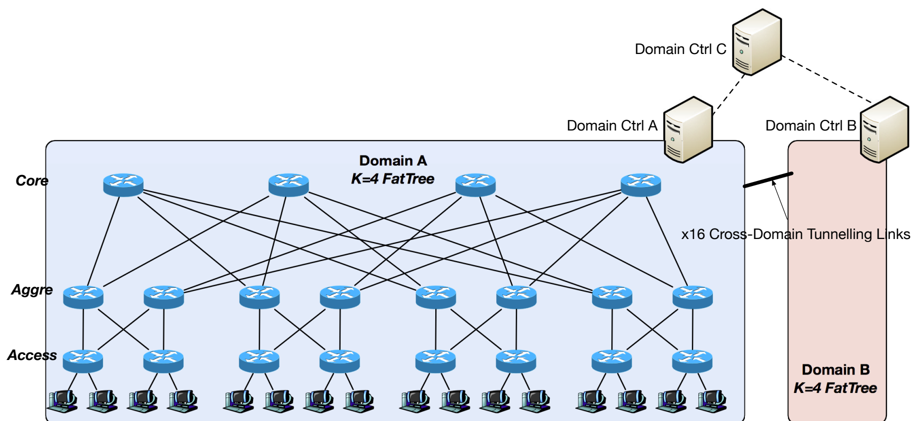


Fig. 3.8: A *FatTree* Data Center Topology with $k = 8$ is used for evaluation. Due to space limit, only the left half (domain A) is shown, which forms a $k = 4$ FatTree topology (domain B is omitted). Two domains are connected with 16 tunneling links.

tunneling links between the networks. There are three domain controllers: one for each network for local events and the 3rd one on the campus network for non local events. Figure 3.8 illustrates the overall topology.

For the purpose of comparison, we consider another network setting, which is a single large *non-blocking* switch. This setting is configured to find out the maximum throughput since traffic is constrained only by link speeds. Because non-oversubscribed FatTree topologies are rearrangeably non-blocking [50], a non-blocking FatTree topology can also achieve the optimum performance.

A traffic engineering on data center topology: A traffic engineering (TE) can highlight the SDN's benefit because the controller has a global network state such as a topology, flow statistics and link utilization. The controller can thus arbitrarily choose best paths even if they are not shortest, with no concerns about convergence time, forwarding loops, or black holes.

Since the traffic pattern in data center networks turns out to follow exponentially distributed flow sizes [48], an existing flow is expected to last regardless of its duration so far. Thus, if we detect an elephant flow that has sent a significant amount of traffic, we can expect it to continue sending traffic. Hence, the TE application shunts flows from congested paths to uncongested ones.

However, In TE (or SDN in general), fast detection and scheduling are crucial to the efficient network utilization. The fast detection of small size flows is well studied in [51, 52]. Further, fast scheduling of flows on large-scale networks is well studied in Hedera [50]. In this paper, we use a polling mechanism to gather the network state and run TE every 1 second. For implementation simplicity, we use the *global first fit* algorithm [50] for rerouting.

By default, all traffic in our network follows shortest paths—we do not use Spanning Tree Protocol or equivalent. When multiple shortest paths exist, ties are broken by using equal-cost multi-path (ECMP) hashing based on the TCP/IP 5-tuple. At every second, TE gathers the utilization of every link in the network and detects congested links and the corresponding flows. TE then reschedules all flows routed to the congested links to less utilized ones.

In our experiments, both default forwarding and TE rely on OpenFlow. Paths selected by TE use high-priority OpenFlow rules and default paths use lower-priority rules. Scheduling a flow along a different path simply requires installing one new high-priority rule in each switch along the path. After an elephant flow ends, its rules time out and the switches automatically remove them. The time to install an OpenFlow rule—approximately 10 ms with the setting in Figure 3.8.

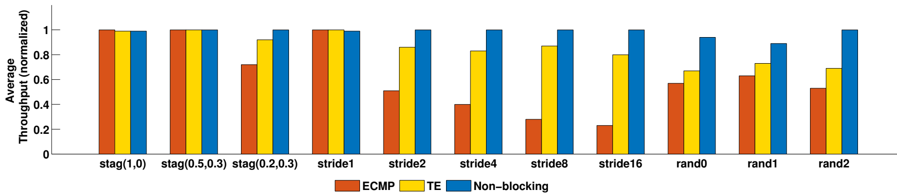
We measure the aggregate data throughput as we vary spatial workload patterns

with the TE mechanism above. In all cases, we employ the same TE algorithm (Hedera). Meanwhile, we measure the total number of bytes sent (data traffic) and the fraction of those bytes that we can schedule on alternate routes in each case.

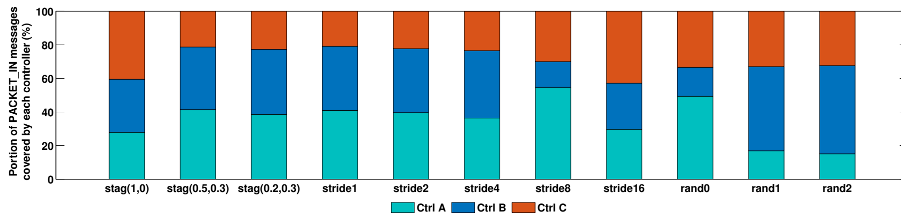
Workloads: We use a workload generator originally written for Hedera. It has three different traffic patterns such as *Stride(s)*, *Staggered Prob (EdgeP, PodP)*, and *Random(u).* Given a network with N hosts, the *Stride(s)* workload causes host i to communicate with host $(i+s)modN$, resulting in adversarial traffic patterns on the topology depending on s. On the contrary, with the *Random(u)* workload, each host communicates with u different hosts that are chosen uniformly. *Staggered Prob (EdgeP, PodP)* has each flow stay within the same rack with probability *EdgeP*, stay within the same pod with probability *PodP*, and cross a pod with probability $1 - EdgeP - PodP$. For our performance baseline we use shortest-path ECMP forwarding with no TE. Flow sizes are exponentially-distributed with 1 GB average. Flow inter-arrival times are exponentially distributed with 1 ms average.

Results: Figure 9(a) shows the (data traffic) throughput of various flow patterns on the topology in Figure 3.8 and on the ideal single non-blocking switch. Note that even the non-blocking switch does not achieve maximum throughput (1.0) since sometimes two hosts transmit packets to the same destination, causing congestions. Although a FatTree topology is rearrangeably non-blocking, there is a significant gap between ECMP forwarding and the single non-blocking switch due to collisions where multiple flows are (hashed and) forwarded onto the same link. This degradation is mitigated by TE due to its flow scheduling.

We also counts the number of `packet_in` messages sent to each domain controller in Figure 3.8, to see whether the overall control overhead is efficiently dis-



(a) Normalized aggregate throughput on the data center topology in Figure 3.8 shows that TE outperforms ECMP for various traffic patterns.



(b) The number of `packet_in` messages handled by each controller: *Ctrl A* and *Ctrl B* mean the lower level controllers which take on each *k=4 FatTree* network, *Ctrl C* means the higher level controller which takes on the partitioned domains.

Fig. 3.9: For various traffic patterns, aggregate throughput for ECMP and TE on the k=8 FatTree topology (with FRACTAL) is compared to a single non-blocking switch. All links have 100 Mbps bandwidth.

tributed among controllers. Figure 9(b) shows the number of `packet_in` messages processed by each controller with the same scenario for Figure 3.9. *Ctrl A* and *Ctrl B* mean the results of the lower level controllers, and *Ctrl C* means the result of the higher level controller.

The total number of `packet_in` messages means the entire control overhead to be handled by any controllers Figure 9(b) demonstrates that the three controllers split the entire control overhead moderately. Especially, in most scenarios, the overhead portion of the higher level controller is much less than those of the others. It means that control messages sent to the higher level controller are effectively reduced by the FRACTAL abstraction.

3.5 Related Work

Recently, there has been a significant amount of work on distributing SDN control-plane to implement a (logically) centralized SDN controller, achieving scalability, availability, and responsiveness. In this section, we introduce some works published chronologically.

The first research on distributing SDN control-plane to multiple physical servers was studied in ONIX [36]. The main idea was providing a middleware on top of each distributed control-plane, and abstracting control-planes shown as a global view of the network provided to the network application layer. However, they didn't evaluate the architecture in terms of an overhead of a state distribution. In addition, since the source is not publicly available, there is no way to evaluate the architecture.

HyperFlow [53] proposed a distributed event-based control plane that passively

synchronizing local network views of SDN controllers. To this end, the authors leverage a wide-area distributed storage system, WheelFS [54]. In addition, HyperFlow localizes to make decision for local network events to reduce the synchronization overhead.

ONOS [37] is the first open source SDN controller supporting a distributed control-plane. The authors evaluate the various open sources implementing a distributed architecture, including Titan [55] for graph database, Cassandra [56], and RAMCloud [57] for distributed key-value store.

Up until now, we survey the works on distributing control-plane of SDN to achieve scalability. Along with these efforts, there are several researches on scaling the performance of a SDN controller itself. The beacon controller [32] improves the throughput of message IO by using multiple threads. McNettle [58] proposes a scaling mechanism using multicore server; it increases an event processing throughput scaling with the number of system CPU cores.

Recently, there are some efforts to reduce a control loop for responsiveness. OpenSample [51] proposes a streaming algorithm to process packet samples. It achieves a 100ms control loop rather than the 1-5s control loop of prior polling approaches, resulting in a 150% throughput improvement of a network application, in specifically traffic engineering requiring a low control loop. Together with this streaming algorithm, Planck [52] proposes a novel network monitoring tool leveraging oversubscribed port mirroring supported by commodity switches. It achieves a control loop at 280us-7ms timescales on a 1 Gbps switch and 275us-4ms timescales on a 10 Gbps switch.

Chapter 4

Conclusion & Future Work

This thesis investigates the challenges of *Software Defined Data Center*, specifically networking. Due to the decoupling of control and data plane, and the scalability issue of a (logically) centralized controller, adopting SDN to data center networks inherently introduces i) a control loop is significantly increased as network size grows and ii) a horizontal distributed architecture is limited on scalability for large-scale inter-data center networks.

For the first challenge, we have presented OpenSample, a working prototype of a low-latency, sampling-based measurement platform, and a data center traffic engineering application based on OpenSample. Our primary contribution is lowering the latency to gather accurate measurements of network load and elephant flows from 1–5 seconds to 100 milliseconds. Faster detection of elephant flows allows traffic engineering to better schedule the network, yielding increased throughput—up to 150% in some cases. In general, workloads that cause more congestion benefit more from our work and our improvement over prior efforts is more significant for smaller flows. OpenSample works with unmodified Ethernet switches, making it deployable without waiting for new hardware or modifying end-host software.

For the second challenge, we address one of the well-known issues in SDN—scalability. We first show that the current approach of distributing the network state into multiple controllers horizontally has the limitation like the convergence latency.

Motivated from this, we present FRACTAL, a framework for scalable dissemination of the network state over the control plane. The key idea of FRACTAL is to "divide-and-abstract" a large network recursively until a divided network can be fully handled by a controller. FRACTAL presents a tradeoff between the message processing delay over the controller hierarchy and the control plane throughput. We demonstrate the benefit of FRACTAL by building a testbed emulating a data center with the open source SDN controller, OpenDaylight.

We believe that there is significant opportunity for future work in this space as well. We hope to explore dynamically adapting per-port sampling ratio based on observations and combining both maximum likelihood and sequence-number–based estimations for improved accuracy. Lastly, merchant silicon vendors may introduce ASICs that support sampling entirely in the data plane allowing for very high sampling ratio, i.e., N=64, even with 10 Gbps links. This opens the possibility of control loops that operate as fast as 100 μs. Moreover, we first seek to find out the optimal level of the controller hierarchy given network environments and SDN requirements. Also, current FRACTAL implementation requires manually configuration. Thus, we plan to implement FRACTAL for automatic installation and configuration by leveraging the open source distributed configuration service, ZooKeeper.

Bibliography

[1] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting data center networks with multi-gigabit wireless links," in *SIGCOMM*, 2011.

[2] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.

[3] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *NSDI*, 2010.

[4] A. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM*, 2011.

[5] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ICFP*, 2011.

[6] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *NSDI*, 2013.

[7] "Openflow-switch," https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[8] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: scalable ethernet for data centers," in *CoNEXT*, 2012.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, and P. Yalagandula, "DevoFlow: Scaling flow management for high-performance networks," in *SIGCOMM*, 2011.

[10] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: fine grained traffic engineering for data centers," in *CoNEXT*, 2011.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[12] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, "HyperX: topology, routing, and packaging of efficient large-scale networks," *SC Conference*, 2009.

[13] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *NSDI*, 2012.

[14] B. Claise, Ed., "Cisco Systems NetFlow Services Export Version 9," RFC 3954. http://www.ietf.org/rfc/rfc3954.txt, October 2004.

[15] "sFlow," http://sflow.org/about/index.php.

[16] V. Mann, A. Vishnoi, and S. Bidkar, "Living on the edge: Monitoring network flows at the edge in cloud data centers," in *COMSNETS*, 2013.

[17] "Sampled NetFlow [Cisco IOS Software Releases 12.0 S]," http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/12s_sanf.html.

[18] P. Phaal and S. Panchen, "Packet sampling basics," http://www.sflow.org/packetSamplingBasics/index.htm.

[19] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *SIGCOMM*, 2010.

[20] P. Phaal and M. Lavine, "sFlow Version 5," http://www.sflow.org/sflow_version_5.txt.

[21] "Open vSwitch," http://openvswitch.org.

[22] "Erlang distribution," http://en.wikipedia.org/wiki/Erlang_distribution.

[23] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, implementation and evaluation of congestion control for multipath TCP," in *NSDI*, 2011.

[24] "Floodlight openflow controller," http://www.projectfloodlight.org/floodlight/.

[25] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *CoNEXT*, 2012.

[26] R. Ozdag, "Intel Ethernet Switch FM6000 Series - Software Defined Networking," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[27] Sivasankar Radhakrishnan, "cluster_loadgen," https://bitbucket.org/nikhilh/mininet_tests/src/9f051450a32a8411f03b7f6bbb99cb436c5a4a73/hedera/hedera.

[28] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for openflow switch evaluation," in *PAM*, 2012.

[29] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," in *SIGCOMM*, 2010.

[30] A. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM*, 2011.

[31] "Oflops," http://www.openflow.org/wk/index.php/Oflops.

[32] D. Erickson, "The beacon openflow controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491189

[33] S. Miura, T. Boku, T. Okamoto, and T. Hanawa, "A dynamic routing control system for high-performance PC cluster with multi-path Ethernet connection," in *IPDPS*, 2008.

[34] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *SIGCOMM*, 2011.

[35] "sFlow-RT," http://inmon.com/products/sFlow-RT.php.

[36] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968

[37] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "Onos: Towards an open, distributed sdn os," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620744

[38] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491193

[39] "Open daylight project," http://www.opendaylight.org/.

[40] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342444

[41] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: State distribution trade-offs in software defined networks," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342443

[42] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342446

[43] S. Schmid and J. Suomela, "Exploiting locality in distributed sdn control," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 121–126. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491198

[44] "Rfc6020, yang-a data modeling language for the network configuration protocol (netconf)," https://tools.ietf.org/html/rfc6020.

[45] "Rfc6241-network configuration protocol (netconf)," https://tools.ietf.org/html/rfc6241.

[46] "Infinispan: distributed in-memory key/value data grid and cache," http://infinispan.org.

[47] "Cbench," http://archive.openflow.org/wk/index.php/Oflops.

[48] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879175

[49] "Indigo project," http://www.projectfloodlight.org/indigo/.

[50] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855730

[51] J. Suh, T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, June 2014, pp. 228–237.

[52] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 407–418. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626310

[53] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, ser. INM/WREN'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863133.1863136

[54] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, "Flexible, wide-area storage for distributed systems with wheelfs," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 43–58. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558981

[55] "Titan: Distributed graph database," http://thinkaurelius.github.io/titan/.

[56] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[57] "Ramcloud," https://ramcloud.stanford.edu.

[58] A. Voellmy and J. Wang, "Scalable software defined network controllers," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser.

SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 289–290. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342414