Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
Lehrstuhl für Intelligente Netze
und Management Verteilter Systeme

Toward Principled Enterprise Network Management

vorgelegt von
Daniel Levin (MSc.)
geb. in Massachusetts, USA

Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Jean-Pierre Seifert, TU Berlin, Germany
Gutachterin:	Prof. Anja Feldmann, Ph. D., TU Berlin, Germany
Gutachter:	Prof. Marco Canini, Ph. D., Université catholique de Louvain, Belgium
Gutachter:	Rob Sherwood, Ph. D., Big Switch Networks, USA

Tag der wissenschaftlichen Aussprache: 07 July 2014

Berlin 2014
D 83

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum Daniel Levin (MSc.)

Abstract

Compared to the contemporary system and software landscapes, traditional computer network operations have notably lagged behind with regard to their ability to incorporate abstractions and interfaces for management automation and orchestration. Networks expose unnecessary low-level complexity to the operator. One prevailing observation is that historically, traditional network infrastructure has exhibited limited extensibility due to its very often vertically-integrated nature. This, in turn, limits the ability of network practitioners and researchers to develop and introduce such automation and abstraction into network operations.

There has been a growing realization not only within the networking research community but also in industry that current solutions to ongoing network challenges cannot be sustained and that new solutions must be applied to the underlying sources of network management complexity. Over the past several years, Software Defined Networking (SDN) has emerged as a paradigm in which the fundamentally needed, yet missing abstractions of traditional network management can be realized.

Leveraging the recent emergence of such software defined networks, this thesis makes contributions that address ongoing challenges related to network design, planning, management, and troubleshooting. In this thesis, we investigate and characterize key state distribution design trade-offs arising from the "logically centralized network control plane", a fundamental architectural hallmark of software defined networks. We then explore the problem of consistent, distributed policy composition in the logically centralized control plane.

Based on the "logically centralized" control plane abstraction, we propose an approach, Software Transactional Networking, to enable distributed, concurrent, multi-authorship of network-policy subject to key consistency and liveness properties. We propose a novel troubleshooting tool, OFRewind, which leverages the logically centralized control plane to realize an abstraction for network-wide record and replay debugging in networks. We devise an architecture and methodology, Panopticon, to enable the incremental deployment of the logically centralized network control plane into existing enterprise networks, exposing the abstraction of a logical Software Defined Network. Finally, we identify open research problems and promising future directions for work and conclude.

Zusammenfassung

Im Vergleich zur heutigen System- und Softwarelandschaft, sind traditionelle Netzwerkparadigmen weitgehend zurückgeblieben, insbesondere im Bezug auf die Möglichkeit, Abstraktionsmechanismen und Schnittstellen zur Automatisierung von Netzwerkverwaltung und -orchestrierung bereitzustellen.

Netzwerke exponieren dem Betreiber eine systemnahe, nicht notwendige Komplexität. Es ist allgemein bekannt, dass traditionelle Netzwerkinfrastruktur nur begrenzt erweiterbar ist, was wiederum auf ihren weitgehend vertikal integrierten Charakter zurückzuführen ist. Dies beschränkt die Möglichkeiten von Forschern und Netzwerkbetreibern, Systeme zur Automatisierung und zur Abstraktion von Netzwerken zu entwickeln und schließlich im operativen Umfeld anzuwenden.

Sowohl in der Industrie als auch in der Forschungsgemeinschaft wird zunehmend realisiert, dass aktuelle Lösungsansätze den weiterhin bestehenden Herausforderungen nicht mehr standhalten. Damit einhergehend wird die zunehmende Notwendigkeit erkannt, neuartige Lösungen zur Netzwerkverwaltung zu entwickeln und anzuwenden. In den letzten Jahren wurde Softwarde Defined Networking (SDN) bekannt als ein neues Paradigma, mit welchem die fundamental notwendigen, aber noch immer fehlenden, Abstraktionsfähigkeiten traditioneller Netzwerkverwaltung verwirklicht werden können. Durch das neuerliche Aufkommen von SDN leistet diese Arbeit einen Beitrag zu mehreren Herausforderungen in Bezug auf Netzwerkdesign, -planung, -verwaltung, sowie -fehlerbehebung.

In dieser Arbeit werden zentrale Abwägungen im Bezug auf die Verteilung von Zustand auf der Steurungsebene analysiert. Diese Abwägungen entstehen durch das Konzept von einer "logisch zentralisierten Netzwerkkontrollschicht", einem fundamentalen Konzept von SDNs. Zunächst wird das Problem der konsistenten, verteilten Komposition von Richtlinien in der logisch zentralisierten Kontrollschicht angegangen. Basierend auf dieser Abstraktion wird ein neuer Ansatz, "Software Transactional Networking", vorgestellt. Dieser Ansatz erlaubt verteilte und nebenläufige Regelkompositionen von multiplen Akteuren unter Einhaltung von Konsistenz- und Livenesseigentschaften. Des Weiteren wird ein neuartiges Werkzeug zur Fehlerbehebung, OFRewind, vorgestellt, welches die logisch zentralisierte Steuerungsschicht nutzt um eine Abstraktion für netzwerkweites "Aufzeichnen und Wiedergeben" zu ermöglichen.

Darüber hinaus wird eine Architektur und Methodik, Panopticon, vorgestellt, welche die inkrementelle Einführung von logisch zentralisierter Nertzwerkkontrolle in existierenden Firmennetzwerke ermöglicht und hierbei lediglich die Abstraktion eines logischen SDNs exponiert. Schließlich werden offene Forschungsfragen diskutiert und vielversprechende Zukunftsausrichtungen identifiziert.

Pre-published Papers

Parts of this thesis are based on pre-published papers co-authored with other researchers. I wish to extend a huge "Thank You" to each and every one of my co-authors for their valuable contributions! All co-authors have been acknowledged as scientific collaborators of this work.

Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A. **OFRewind: Enabling Record and Replay Troubleshooting for Networks**. In Proceedings of *USENIX ATC 2011*, Portland, Oregon.

Levin, D., Wundsam, A., Heller, B. Handigol, N., and Feldmann, A. **Logically Centralized? State Distribution Trade-offs in Software Defined Networks**, In Proceedings of *HotSDN 2012*, Helsinki.

Canini, M., Kuznetsov, P., Levin, D., and Schmid, Stefan. **Software Transactional Networking: Concurrent and Consistent Policy Composition**, In Proceedings of *HotSDN 2013*, Hong Kong.

Canini, M., Kuznetsov, P., Levin, D., and Schmid, S. **Towards Distributed and Reliable Software Defined Networking**, *Brief Announcement* In Proceedings of *DISC 2013*, Jerusalem.

Extended version: Canini, M., Kuznetsov, P., Levin, D., and Schmid, S. **A Distributed SDN Control Plane for Consistent Policy Updates**, Research Report, `http://arxiv.org/abs/1305.7429`.

Levin, D., Canini, M., Schmid, S., and Feldmann, A. **Toward Transitional SDN Deployment in Enterprise Networks**, Extended Abstract, In Proceedings of *ONS 2013*, Santa Clara, California.

Levin, Dan, Canini, Marco, Schmid, Stefan., and Feldmann, Anja. **Incremental SDN Deployment in Enterprise Networks**, Demo, In Proceedings of *Sigcomm 2013*, Hong Kong.

Extended version: Levin, D., Canini, M., Schmid, S., Schaffert, F., and Feldmann, A. **Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks**, In Proceedings of *USENIX ATC 2014*, Philadelphia, Pennsylvania.

Canini, M., De Cicco, D., Kuznetsov, P., Levin, D., Schmid, S., and Vissicchio, S. **STN: A Robust and Distributed SDN Control Plane**, Extended Abstract, In Proceedings of *ONS 2014*, Santa Clara, California.

Contents

1

Introduction

Over the past few years, numerous corporations [22,76], universities [21,50,135], government agencies [16, 72], and institutions large and small have begun out-tasking some of their computing software, infrastructure, and systems administration to dedicated applications- and cloud-hosting providers. Such providers which offer software and computing *Infrastructure as a Service* (abbreviated IaaS) have enjoyed significant success and growth, as they shift the burden and costs of managing complex technical systems away from the enterprise to more specialized service providers. These service providers benefit greatly from economies of scale and maintain core competencies in systems and software development and operations. For customers of IaaS providers, the ability to count on predictable, typically fee-based service costs while improving operational efficiency and reliability is a huge benefit of out-tasking. Ultimately, the shift toward IaaS adoption allows IaaS customers to focus more on their core competencies than IT infrastructure maintenance.

The growth and success of IaaS can be largely credited to the emergence of powerful management abstractions including system virtualization (e.g. Xen, KVM, VMWare) [46, 163], software automation and orchestration frameworks (e.g., OpenStack, libvirt) [36, 39] and deployment and continuous-integration tools (Cfengine, Nightbird) [101, 105] with well-defined open programming interfaces. These technologies enable IaaS providers to efficiently operate and manage their complex and dynamic software systems and scalably grow their service offerings. Indeed, for IaaS providers, these abstractions have proven critical for evolving the computation platform underlying IaaS away from the individual server or cluster, toward "The datacenter as a computer" [35].

Compared to the contemporary system and software landscapes, traditional computer network operations have notably lagged behind with regard to their ability to incorporate abstractions and interfaces for management automation and orchestration [145]. Networks thus, expose unnecessary low-level complexity to the operator. The underlying explanations for this problem, which we explore in more detail in Chapter 2, are not just technical in nature, but also stem from economic and business factors which apply to network hardware and software vendors.

One prevailing observation is that traditional network infrastructure exhibits limited extensibility due to its very often vertically-integrated nature [113]. This, in turn, limits the ability of network practitioners and researchers to develop and introduce such automation and abstraction into network operations. As the tasks inherent to network operation and management entail coping with this exposed complexity, we turn our attention to network management.

1.1 Network Management

The term **Network management**, at face value, concerns the governing of state in- and behavior of the network. The specific meaning of the term, and the objectives it entails however, depends largely upon the environment and community in which it is invoked. In the context of an internet service provider, the term reflects the activities common to the *Network Operations Center* or NOC: Namely, monitoring the state of devices and links in the network and responding to outages, failures, and "trouble tickets" [48]. The IETF standardization body first refers to the term in *RFC 1052* [83], a document which describes the goal of standardizing the protocols for setting and reading configuration variable state from network devices over the network itself, and whose context lays the foundation for future network management protocols such as SNMP [84]. In this thesis, we apply a broad definition to the term "network management" to encompass the monitoring and governing of state within the network.

The nature of network state is vast and varied and encompasses device-level configuration directives and parameters, physical and logical device connectivity, packet look-up tables and content-addressable data structures, and the computational state of hundreds to even thousands of different distributed routing, traffic-engineering, and fault-recovery protocols. Nevertheless, a network's state can be conceptually decomposed into two entities: The state of the **data plane** [1], e.g., the forwarding table entries used to process data traffic arriving on an in-bound interface of a network device, and the **control plane**, which encompasses the state and logic from which the data plane state is derived.

[1] In this thesis, we use the term "data plane" interchangeably with "forwarding plane"

It is the role of the network operator to **manage the network** – that is – to define and implement the **network policy** as the collective inputs to the network control and data planes, and to furthermore monitor behavior and adjust control inputs in an on-going fashion to ensure that the resulting network behavior adheres to said policy. To this end, operators must specify and implement proper connectivity and reachability, enforce network access control policies, provision quality of service (QoS) resources, filter malicious traffic, detect and prevent network intrusions, and accommodate a wide and growing range of mobile devices into the network. Notably, the network policy is not a static specification, and the operator must ensure each network behavior remains in compliance as the policy evolves. When behavior deviates from policy, the operator must detect the anomaly and troubleshoot the network. This job is made all the more complex by the distributed and often *black-box* nature of network devices.

Despite the complexity inherent to network management, the human operator remains largely entangled in the "update-monitor-adjust" control loops of network management [145]. Multiple studies have determined that the majority of failures in carrier and datacenter IP networks arise from human error during normal operation. [19, 111]. In the carrier network context, it has been shown that 20% of all network policy compliance failures occur during scheduled maintenance [110]. In corporate and enterprise environments, studies of human-error induced access-control and firewall policy violations demonstrate positive correlations with the complexity of the policies (i.e., number of rules, objects, and interfaces) [164, 165]. These observations support that human interaction with the running network remains an on-going operational challenge.

1.2 Challenges for Network Management

The operational challenges for network management are manyfold and are related to fundamental systems and distributed computing problems:

Distributed State: Computer networks are fundamentally distributed systems, made up of physically distributed devices, with distributed configuration and forwarding state. Virtually every device in the network furthermore participates in one or more distributed protocols, whose purpose is to share sufficient state among the devices to enable the successful transmission of data across the network. In the context of monitoring and troubleshooting, observations made from any single vantage point can only inform the operator of the state local to that point. Obtaining a consistent snapshot of the data plane and control plane is therefore challenging, and becomes impractical for networks at scale when the rate of any state change eclipses the time required to collect the state at every network device. In traditional networks, the degree to which, and fashion in which state can be exchanged for

control purposes is largely defined through long-term standardization processes and fixed or "baked into" the specifications of the control protocols themselves.

Black Box Components: Computer networks are made up of switches, routers, and middle-boxes which can include firewalls (packet-level, application-level, and otherwise), load-balancers, intrusion detection systems (IDS), and traffic optimizers, any of which may modify, duplicate, de-duplicate, compress, re-order, and otherwise manipulate traffic. These devices can largely be regarded as *black-box* components: The internal logic of such devices which defines what actions are applied to traffic is generally not accessible to the network operator. Black box components can increase the difficulty of troubleshooting network anomalies, as the causal relationship between configuration inputs and resulting traffic behavior can not be understood by analytical means alone [166].

Heterogeneity: The term "single pane of glass" [7, 38] has been coined in the systems and network operations community, to describe the ideal, operational "holy grail" where every input to a complex systems (e.g. a computer network) is expressed through a single, unified, common interface. Despite the profound desire for such interfaces, in practice, networks increasingly comprise more and more heterogeneous devices [147], whose lifecycles in many cases, evolve organically over time. As a consequence, network management necessarily involves an on-going race to encapsulate and abstract the ever-widening control interface complexity from the human operator.

Resource Contention: The network's capacity to transport data is fundamentally constrained by resource availability: Data transmission capacities of links, look-up table and memory capacities of the data plane, and increasingly, power consumption define key resource constraints which networks must observe. In the presence of multiple parties vying for these shared, finite resources, the problem of how to allocate such resources is a fundamentally difficult problem.

Concurrency: Networks exhibit and must deal with concurrency at many levels. At the highest level, all but the very smallest of networks are managed by teams of humans, whose policy specifications over the distributed network infrastructure can, by no means, be guaranteed to be given and implemented in a strictly sequential fashion. In terms of the distributed routing protocols in use today [86, 87], the execution and interaction of the control plane with the data plane is a fundamentally concurrent process when viewed over the collective network state.

Missing Abstractions: Good abstractions hide system complexities behind interfaces that expose ideally, only the minimum inputs and outputs required to efficiently control the system. Designing good interfaces is an inherently difficult trade-off problem, where designers must choose between exposing (or leaking) too much detail and unduly restricting system functionality. In the ideal network, data plane functionality is organized according to the OSI layer model abstraction: Reliable transport is implemented on best-effort end-to-end delivery, which is implemented on best-effort local delivery, over some physical medium. Despite these abstractions, real networks exhibit frequent layer abstraction violations [58, 150]. In contrast to the limited abstractions of the data plane, the traditional network control plane is a jumbled "bag of protocols" lacking abstraction virtually altogether [146]. As we illustrate further in Section 2.1.2, network operators today express *network-wide* objectives in terms of the various configuration parameters exposed by the distributed control plane – from which each network device updates its own data plane forwarding based on the device's *local* view of the network. In other words, to express a network-wide objective, an operator must translate a global solution into parameters that govern a distributed algorithm which then makes decisions based on local information. The operator must then compensate for or correct the outcome accordingly. Network operators have consequently adapted to such problems of weak or altogether lacking abstractions in networks to become "Masters of Complexity." [145]

1.3 Opportunities for Network Management

There has been a growing realization not only within the networking research community [5, 27, 64, 114, 168] but also in industry [20, 23] that current solutions to ongoing network challenges cannot be sustained and that new solutions must be applied to the underlying sources of network management complexity. Over the past several years, Software Defined Networking (SDN) has emerged as a paradigm in which the fundamentally needed, yet missing abstractions of traditional network management can be realized.

An SDN exhibits the key property that the control plane is decoupled from the data plane. This decoupling introduces the possibility to make much of the data plane forwarding behavior directly programmable via a protocol such as the OpenFlow standard [114]. Control plane state and logic can then be implemented as a "logically centralized" software program, running on commodity servers, which maintain a global view of the network. Crucially, this decoupling enables the network control plane data to evolve independently from the network data plane forwarding functionality.

Similar to the frameworks and tools used to realize IaaS, network operators can leverage this centralized network control abstraction to decouple networks into more modular, independently evolving components. The "logically centralized" abstraction is

an appealing alternative to reasoning about distributed protocols and keeping track of tens or hundreds of thousands of lines of configuration scattered among thousands of devices. Ultimately, by leveraging the centralized control vantage point, SDN promises operators new opportunities to orchestrate network-wide behavior, in a predictable, testable, principled manner.

1.4 Our Contributions

The contributions of this thesis address a spectrum of network management challenges by introducing principled approaches to network design, planning, management, and troubleshooting. Among our contributions, we:

1. Investigate and characterize key state distribution design trade-offs arising from the logically centralized network control plane.
2. Identify the problem of consistent, distributed policy composition in the logically centralized control plane. We propose an approach, **Software Transactional Networking**, to enable distributed, concurrent, multi-authorship of network-policy subject to key consistency and liveness properties.
3. Propose a novel troubleshooting tool, **OFRewind**, which leverages the logically centralized control plane to enable systematic, replay debugging in networks.
4. Propose an architecture and methodology, **Panopticon**, to enable the incremental deployment of the logically centralized network control plane into existing enterprise networks, exposing the abstraction of a logical Software Defined Network.

1.5 Outline

In the remainder of this thesis, we review in **Chapter 2** the background and state of the art in network management and network troubleshooting. We then discuss the recent emergence of Software Defined Networks, and in particular, the opportunities it presents to rethink traditional approaches to network management and troubleshooting operations.

In **Chapter 3**, we investigate design trade-offs of a key abstraction for principled network management: the "logically-centralized" control plane. Software Defined Networking (SDN) gives network designers freedom to refactor the network control plane, enabling the network control logic to be designed and operated on a global network view, as though it were a centralized application, rather than a distributed system – logically centralized. Regardless of this abstraction, control plane state and

logic must inevitably be physically distributed to achieve responsiveness, reliability, and scalability goals. Consequently, we ask: "How does distributed SDN state impact the performance of a *logically centralized* control application?"

Motivated by this question, we characterize the state exchange points in a distributed SDN control plane and identify two key state distribution trade-offs. We simulate these exchange points in the context of an existing SDN load balancer application. We evaluate the impact of inconsistent global network view on load balancer performance and compare different state management approaches. Our results suggest that SDN control state inconsistency significantly degrades performance of logically centralized control applications agnostic to the underlying state distribution.

We next turn our attention in **Chapter 4** to the problem of distributed policy composition in the logically centralized control plane. We characterize the problem of consistent policy composition: namely the need to maintain certain safety properties in the presence of multi-authored, network policy. As network policy specification and control is rarely specified and implemented in a purely sequential and monolithic fashion, but rather by teams of operators, responsible for different aspects of the network, we focus on concurrency issues which can arise. Indeed, conflicts among concurrent policy updates may result in packet-forwarding inconsistencies on the data plane, even when each update is installed with "per-packet consistent" update semantics [138]. We introduce the problem of *consistent* composition of concurrent policy updates. Intuitively, consistent concurrent policy composition must *appear* as though there is no concurrency neither between any policy updates, nor between a policy update and in-flight packets on the data plane.

We propose an elegant policy composition abstraction based on a *transactional interface* with *all-or-nothing* semantics: a policy update is either *committed*, in which case the policy is guaranteed to compose consistently over the entire network and the update is installed in its entirety, or *aborted*, in which case, no packet is affected by it. Consequently, the control application logic is relieved from the cumbersome and potentially error-prone synchronization and locking tasks, and control applications are kept light-weight. We sketch a simple implementation of the transactional synchronization: our approach is based on fine-grained locking on network components and avoids complex state machine replication.

Leveraging the abstraction of the logically centralized control plane, in **Chapter 5** we introduce OFRewind, a novel application of the SDN programming interface to realize the abstraction of network-wide record and replay troubleshooting in networks. Network debugging can be a daunting task, due to their size, distributed state, and the presence of *black box* components such as commercial routers and switches, which are often poorly instrumentable and coarsely configurable. The debugging tool set available to administrators has also remained limited, providing aggregated statistics (SNMP), sampled data (NetFlow/sFlow), or local measurements on single hosts (tcpdump). *Record and replay debugging* – a powerful approach for general systems has been however, a lacking approach for networks. We present the

design of OFRewind, which enables *scalable*, *multi-granularity*, *temporally consistent recording* and *coordinated replay* in a network, with fine-grained, dynamic, centrally orchestrated control over both record and replay. OFRewind helps operators to reproduce software errors, identify data-path limitations, or locate configuration errors.

In **Chapter 6**, we then turn our attention to the challenge of introducing the SDN programming interface into existing networks. Despite the promise of SDN to enabled principled approaches to long-standing network operations problems, with the exception of a few notable deployments in the wild, e.g., Google's B4 [94], it remains largely an experimental technology for most organizations. As the transition of existing networks to SDN will not happen instantaneously, we consider hybrid networks that combine SDN deployments alongside legacy networking gear an important avenue for transitioning to SDN; yet research focusing in these environments has so far been modest. Hybrid networks posses practical importance, are likely to be a problem that will span several years, and present multiple interesting challenges, including the need for radically different networking paradigms to co-exist. We present the design and implementation of Panopticon, an architecture for operating networks that combine legacy and incrementally deployable SDN switches. Panopticon exposes an abstraction of a logical SDN in a partially upgraded legacy network, where SDN benefits can extend over the entire network.

Finally in **Chapter 7** we summarize our contributions and provide an outlook on future research directions and open problems.

2
Background

In this chapter we present the background and state of the art of network management and its relationship to the traditional, distributed network control plane. We then review practices and concepts pertaining to troubleshooting networks. Finally, we introduce the recent emergence and evolution of software defined networking which underlies and enables our contributions in network management and troubleshooting.

2.1 Network Management

In the most general sense, **network management**, concerns the monitoring and governing of the state and logic of a network. In essence, it encompasses the processes and mechanisms that run between the human operator and the network devices and involves the high-level specification for the network behavior. Figure 2.1 illustrates the organizational relationships between the operator and a very simple representation of a network consisting of common network devices (e.g., Ethernet switches, IP routers, and middleboxes such as firewalls).

The processes and mechanisms employed in managing a network can be conceptually organized into a structure called the network **management plane** illustrated in Figure 2.1. The management plane provides the mechanisms for the human operators to figure out what is happening in the network, and express how to configure network behavior. It can be though of as an interface between the operator and the network devices. The network devices themselves can be conceptually broken

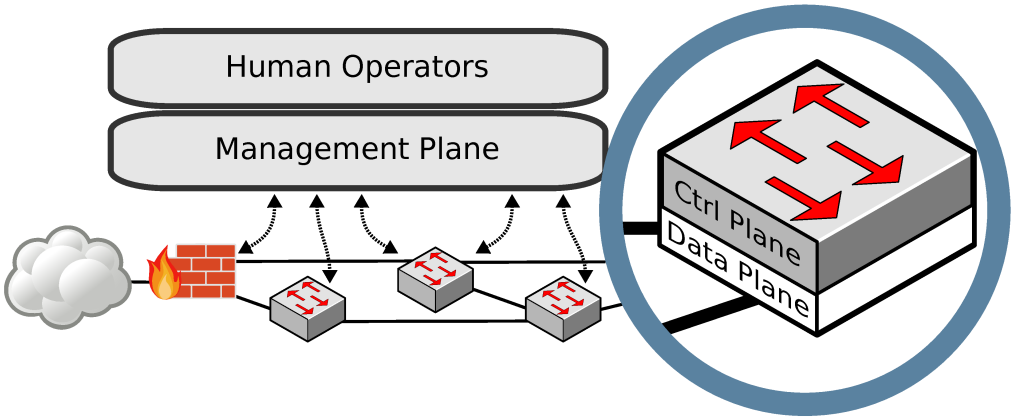


Figure 2.1: Conceptual organization of the management, control, and data plane of a very simple network.

down into two components: (*i*) the **data plane** which is responsible for processing incoming packets (i.e., forwarding, dropping, or modifying them) at packet-arrival timescales (i.e., microseconds to nanoseconds), and (*ii*) the **control plane** which encompasses the state and logic from which the data plane is derived, often realized as distributed algorithms (e.g., IP routing protocols) running on the network devices over longer timescales (milliseconds to seconds). It is via the management plane that the operator provides inputs at human timescales (hours to days) to the network control plane, which then influences how the data plane forwards traffic in the network.

It is important to note that already, one can observe that there exist layers of indirection between the human operator who specifies the network policy and the data plane which processes packets. As we soon observe in Section 2.1.2, however, this indirection does not necessarily provide helpful abstractions for network management.

To more concretely illustrate network management and how it relates to the control and data plane, we present a bottom-up overview. We begin with the device-centric view of the data plane and how it relates to the control plane. We then move up to the network-centric view of the control plane and how it relates to the management plane. We finally illustrate the challenges introduced by the way these layers are organized in traditional networks.

2.1.1 Device-centric Perspective: Data- and Control Plane

From the device perspective, we look at the data plane state and how the control plane affects it. Data plane state generally entails: (*i*) the **forwarding information base** which is used to direct incoming packets arriving on a port at a device

(*ii*) packet and byte counters on a per-port and per-device basis, (*iii*) port and link medium activation or inactivation, (*iv*) end-host or adjacent device arrival and departure events, (*v*) other traffic-related metrics and events which may depend on the physical medium type.

To illustrate the relationship between the data plane state and control plane from the device perspective, let us first consider the very simple example of the Ethernet [80] switch which began to see adoption and standardization in the early 1980s. As we look back, we can also observe the historic progression of new functionalities which brings along new data plane state and control plane complexity.

The data plane of a traditional Ethernet switch is characterized by a lookup table which translates a destination MAC address of an incoming packet[1] to an egress switch port. The control plane of the Ethernet switch learns the location of hosts in the network by observing the ports on which data packets arrive, and associating the sender of the packet as being reachable via that switchport. When a packet arrives at an Ethernet switch, the destination address is consulted in the lookup table, and the packet is sent out the corresponding egress port. If a destination address is not contained in the lookup table, the packet is simply flooded. Thus, the control plane encompasses the "MAC address learning" logic which discovers the location of MAC addresses, and maps those addresses to switch ports, thus populating the lookup table.

The design choices underlying Ethernet were made to favor simplicity of the forwarding logic in hardware. The simplicity-oriented design choices of Ethernet however introduced a number of limitations: in particular, such networks were originally susceptible to partition in the event of a link failure, as redundant links could not be supported due to the forwarding loops they would introduce. A solution was needed to enable redundant links, and in 1985, the distributed spanning tree protocol (STP) [129] was proposed to address the problem. Conceptually, STP can be viewed as a complementary control plane mechanism to MAC learning, which leads to the establishment of additional data plane forwarding state – namely, indicating which ports must not forward traffic to avoid loops.

Following our simple Ethernet switch through time to today's networks, one observes that more and more complexity has been added: Virtual LANs [77] to abstract broadcast-domain boundaries from physical wiring, faster and more complex STP variants [78, 90] to accommodate virtual LANs, port security [79] to enable network access control over end-hosts, link aggregation [81], link-level flow control [82], and more.

While the flat address space and flooding-based approaches of Ethernet are simple, they pose scalability challenges for larger networks. By contrast, both the data-plane and control-plane of IPv4 and IPv6 routed networks are far more complex. The mechanism of longest-destination-prefix-based matching of the IP data plane

[1] Ethernet datagrams are commonly known as "frames" , but we call them "packets" for simplicity

introduces more complexity into the forwarding hardware compared to the simple Ethernet MAC lookup table. Additionally, the control plane of the networking layer of the Internet involves far more complex, distributed routing protocols such as OSPF [87], ISIS [85], and BGP [89]. We return to look at the distributed control plane again from the network-centric perspective in Section 2.1.2.

From the device-centric perspective, the distributed control plane requires additional configuration state. Common control plane configuration state includes: link metrics to inform the path selection algorithms of the routing protocols, routing policies and path preferences and costs, access control lists, and a myriad of other protocol-specific parameters. In modern switches, routers, and middleboxes, there can be thousands of such standardized (and proprietary) control plane functionalities. Each of these introduces configuration requirements which may or may not be exposed to the management plane.

Now that we have illustrated the relationship between the data plane and the control plane from the device perspective, we look at device management:

Interactive Device Control: Early network devices (and indeed many devices today, still) provide a serial console, e.g., an RS-232 compliant port to which a terminal could be used to type commands and read out device state. The serial console however scales poorly to the management of large networks, as it requires physical proximity to the device. In 1983, the Telnet protocol was standardized [92] to enable the interactive management of IP-enabled devices and was adopted by many network devices to enable interactive console access. Due to the need for authentication and confidentiality, the first version of the secure shell protocol (**SSH**) was introduced in 1995 and later standardized [88] in 2006.

Non-Interactive Device Control: In order to facilitate the management of larger networks without the need for persistent on-site administrator presence or human-interactive console sessions, numerous device management standards began to emerge in the 1980s. The Host Monitoring Protocol [93] **(HMP)** is one of the first standardized protocols to emerge in 1983 for the collection of network device state. The standard describes the protocol operation and packet formats which allow a monitoring client to query network devices (at the time, referred to as Interface Message Parsers or IMP) for throughput, performance, and other device meta-state such as administrative log-in and device configuration modification events and fault and crash reports.

More recently in 1990, the Simple Network Management Protocol [84] **(SNMP)** was developed and standardized within the IETF. The goal of SNMP is to provide an extensible, easy to use application-layer network protocol for collecting device statistics and triggered events and setting values for configurable, operational parameters. To remain extensible, SNMP itself does not specify what statistics and control can

be manipulated within a device. Rather, it uses a standardized Management Information Base **(MIB)**, a hierarchical data structure which contains device-specific object identifiers which can be read or set. The purpose of the MIB is to separate the mechanism for controlling device sate from the way that state is structured. An important observation here is that the MIB itself is free to evolve and indeed, differ across various devices – making a homogeneous control abstraction difficult to realize across different MIBs.

The earliest versions of SNMP did not provide any measures to ensure confidentiality or authenticity of payloads. As a consequence, SNMPv1 had no practically effective authentication or authorization mechanisms. Nevertheless, the protocol found widespread adoption in local area networks and evolved to adopt these functionalities in later versions. As the protocol evolved, more focus was given to improving the security properties of the protocol. The current standard version, SNMPv3 uses the Transport Layer Security (TLS) standard to ensure payload confidentiality and authenticity.

One observation stemming from the evolution of the above, device-centric management protocols is that over time, these protocols have not led to control state or logic migrating away from the distributed network devices themselves. Rather, as these protocols have been designed with extensibility in mind, they have accommodated an increasing level of complexity of device state and logic over time. As for the evolution of the above management protocols themselves, the strongest observable trend has been toward introducing and improving security properties.

2.1.2 Network-centric Perspective: Control- and Management Plane

We now move from the device-centric perspective to the network-centric perspective of management. From this perspective, the network operator is concerned with specifying the configuration of the control plane, determining what is happening in the network, and determining how to modify control plane configuration to achieve the desired network behavior. The challenge for the operator is to express the desired network packet-forwarding behavior of the data plane in terms of configuration inputs to the control plane.

To illustrate why expressing network-level behavior in terms of distributed control plane configuration is a non-trivial task, let us consider the problem of network traffic engineering. Given a network and a traffic matrix which describes the observed traffic volumes between sources and destinations in the network, traffic engineering at a high level, is the process of assigning traffic to paths in the network, usually with the goal of optimizing an objective such as minimizing the most utilized link in the network. Traffic engineering is an activity which requires an up-to-date view of the entire network routes and traffic demands in order to compute an optimal solution.

Instead of directly expressing the results of the traffic engineering optimization problem in terms of new data plane forwarding rules (i.e., the path to be taken at each router toward any given destination), the output of the traffic engineering problem must be translated into control plane configuration (e.g. OSPF weights [87]) and installed at each router in the network. Each router then participates in a distributed computation of FIB state based on its own view of the network, ultimately leading to updates to the data plane forwarding behavior. In this context, it has been argued that the "current division of functionality between the data, control, and management plane has resulted in uncoordinated, decentralized state updates that are antithetical to the goals of network-wide decision-making" [140].

Furthermore, it is not uncommon for this process to lead to transient forwarding loops and unreachability during the convergence process of the distributed routing protocol [60, 159]. A significant body of research has been devoted to addressing the challenges of coping with the complexities of translating network-wide management objectives into device-level configuration:

Franois, et. al. [60] propose an approach to eliminate transient forwarding loops and unreachability by incrementally adjusting OSPF and ISIS path metrics throughout the update. This approach takes advantage of the existing interfaces to these specific routing protocols and as such, does not require modification of the routing protocols themselves. Vanbever, et. al. [159] propose a methodology to address the problem of modifying the configuration of link-state Interior Gateway Protocols (IGP) subject to the safety property of eliminating IP service outages. The authors consider scenarios such as the addition or the removal of routing hierarchy in an existing IGP and the replacement of one IGP with another. The methodology proposed by the authors can guarantee certain safety properties by imposing a strict ordering on the modification of the running routing protocols. Consensus Routing [95] presents a consistency-first approach to adopting forwarding updates in the context of inter-domain routing. Using distributed snapshots and a consensus protocol, each router ensures that every other router along the path toward a destination agrees on each routing update. The protocol separately addresses safety and liveness properties to achieve correctness guarantees on certain forwarding behavior, e.g., loop-free packet-forwarding.

Ultimately, the fundamental problem of reconciling network-wide objectives with the locally-computed forwarding derived from distributed control protocols has led some in the research community to propose a re-factoring of the network control plane. Proposals such as ForCES [5], RCP [55], and 4D [64] along with others propose decomposing the traditional IP control plane functionalities into separate "decision" and "disseminiation" planes. These proposals can be seen as early calls [56] for what is now referred to as a software defined networking, which we revisit in Section 2.3.

2.2 Network Troubleshooting

Having looked at the spectrum of challenges inherent to network management, one observes that the distributed state and logic and complex processes of network operation introduces opportunities for failures and undesired, anomalous behaviors in the network. When problems do occur, operators must **troubleshoot**, or systematically localize the undesired behavior, find the components responsible, and correct the behavior of those components. While the principles of network troubleshooting share much in common with general and distributed systems troubleshooting, the techniques and tools are more specialized, due to the challenges of networks. These include the often black-box nature of network devices, as well as the high data and event rates which can occur at packet-forwarding timescales.

The first step toward troubleshooting any problem involves observing the problem symptoms. Observing problems in networks is itself a non-trivial challenge as it combines the challenges of network monitoring with the problem of deciding whether a particular observed behavior is itself symptomatic of a problem.

Device-level Monitoring: At roughly the same time that network switches, routers and other devices were beginning to support standardized management interfaces and protocols (e.g., SNMP), software tools started to emerge which could visualize the device statistics gathered via such protocols. The Multi Router Traffic Grapher [119] emerge in the early 1990s along with other similar tools such as Cacti [1] to enable visualization of network traffic trends and events. These tools built upon a popular time-series data storage tool called the round-robin database tool or RRD [13]. Today, both commercial and open-source monitoring software built upon these foundations continues to be used widely [121].

Passive Network Monitoring: To understand the behavior of traffic in the network, information from devices such as port-based packet counters is often combined with the network topology to identify "hot spots" or heavily utilized links. Packet counters alone however, are insufficient to identify the sources and destinations of traffic contributing to such hot spots as they say nothing about the origin of the traffic.

Packet-level monitoring of the network is a passive monitoring approach that can reveal more information about the source and destination of traffic. Tcpdump [14] is one such example of a passive packet-level monitoring tool, which can record both packet header and payload contents arriving on a network interface for later analysis. Recording the full contents or even just the packet headers of every packet at line-rate on high-speed links is a significant technical challenge in many networks. Traditionally, networks often employ specialized solutions [2, 4] for packet-level monitoring of high-speed links. Another challenge related to packet-level monitoring approaches is

the problem of choosing the vantage point from which to monitor traffic, as pervasive deployment of dedicated high-speed link monitors may be cost-prohibitive.

To address these challenges, many network switches and routers integrate IP flow monitoring capabilities such as Netflow [122], SFlow [131], and IPFIX [91]. These approaches address two main challenges with packet-level monitoring, namely, the challenge of capturing detailed traffic statistics on high-speed links as well as the need to pervasively monitor traffic over the entire network. To meet these challenges, such flow monitoring techniques often keep statistics over source-destination pair traffic flows (e.g, the common 5-tuple: src_ip, dst_ip, src_port, dst_port, transport protocol). To deal with high speed links, they may additionally sample traffic by randomly selecting packets for which flow statistics will be recorded. As flow-monitoring may be performed at the existing network devices without the need to deploy specialized monitoring equipment, it potentially enables more vantage points over which traffic statistics can be collected, at the expense of losing visibility into the higher-layer protocol header and payload information.

Active Network Monitoring: A complementary approach to passive monitoring is to actively probe or benchmark the network. Two of the commonly used tools used to check for basic network reachability are *ping* and *traceroute*. These tools are often used in an ad-hoc fashion to check whether two hosts can send and receive packets, as well as to obtain a representation of the path over which the packets travel. Popular tools for measuring and estimating achievable data throughputs include *iperf* [6] and *Nping* [10].

It is critical to note that these approaches and tools experience significant limitations in terms of their ability to benchmark performance and explain underlying causes of network packet forwarding behavior. For example, reachability demonstrated via ICMP does not reveal anything about protocol-specific firewalling [98]. Alternately, traceroute is notoriously problematic in the presence of load-balancers, MPLS, and other middleboxes [151]. While it is beyond the scope of this background section to cover all the ways in which such tools can misrepresent internal network behavior, operators face many pitfalls when relying exclusively upon traditional monitoring approaches.

Once problem symptoms have been observed through monitoring, the troubleshooter can attempt to reproduce the problem to better establish the cause and effect relationship between a system input or event and the symptom. To leverage this approach, the system under test must have a **deterministic** behavior that does not depend on timing or randomness. **Race conditions**, in which a problematic behavior emerges due to the non-deterministic timings of interacting components are an example of problems which can not be deterministically reproduced.

There are multiple challenges related to reproducing symptomatic behavior in networks. The distributed nature of network devices introduces challenges recording

a temporally-consistent history of the relevant data plane and control plane events in the network. Replaying the events in a coordinated fashion that is "sufficiently faithful" to the original timing and sequence is also necessary for symptoms to be reproduced. These challenges in part motivate the approach we later introduce in Chapter 5 for a network-wide record and replay troubleshooting tool.

2.3 Software Defined Networks

The operational challenges of correctly navigating the complex interactions between the network management plane, control plane, and data plane have led many in the networking community to consider the need to refactor these functionalities. Over the last ten years, there has been a growing interest not only within the networking research community [5, 27, 64, 114, 168] but also in industry [8, 17, 20, 23, 37] to address the complexities of network operations and at the same time, introduce new flexibility and a fundamental infrastructure evolvability into networks.

Software Defined Networking (**SDN**) has emerged as a paradigm in which certain badly needed, yet largely missing abstractions for traditional network management can be realized. A software defined network, illustrated in Figure 2.2 exhibits the key property that the control plane is decoupled from the data plane. While conceptually simple, this architectural decoupling introduces the possibility to make much of the data plane forwarding behavior directly programmable via an API such as the OpenFlow protocol standard [114].

The OpenFlow protocol realizes a switch programming interface between packet-forwarding hardware and a software program known as a *controller*. This interface provides the foundation for the *software defined network* by enabling the *controller* to read and modify the low-level packet-forwarding behavior of OpenFlow switches, inserting or removing individual packet-forwarding rules at the switch.

In the OpenFlow protocol, a rule is defined in terms of a *match-action* primitive: Namely, an expression which is matched against the various header fields of incoming packets, and a corresponding forwarding action to be applied to all matching packets. The sequence of packets which matches a specific fule is commonly referred to as a *flow*, the specific nature of which is defined by the level of granularity at which rules are defined. The matching pattern of a rule may include Layer 1 (physical switch ports), through Layer 2 and 3 (MAC and IP addresses), to Layer 4 (TCP and UDP ports) headers. The set of matching rules, and the actions associated with and performed on each match are held in the switch and known as a *flow table*. The OpenFlow protocol has evolved since its original specification, to increase the flexibility and scope of what specific data plane protocols may be used to determine packet-forwarding decisions and what actions may be applied to network traffic.

In a software defined network, control plane state and logic can be implemented as a "logically centralized" software program, running on commodity servers, which maintain and operate over a *global view of the network*. Global objectives such as those illustrated in our traffic engineering scenario from Section 2.1.2 can be expressed as a function over a graph representation of the network. Forwarding rules can be directly derived from these computations and then installed directly and more importantly, *systematically*, into the data plane of the network devices via the switch-programming API [114]. By architecturaly decoupling the network forwarding and control planes, both are better suited to evolve independently. In the networking research community, the possibility to enable the control plane to evolve independently from the forwarding plane is seen as an opportunity to address certain long standing challenges of network ossification [114].

The contemporary notion of a software defined network – in which the control plane and data plane are separated, is not entirely new and in historical context, we can point to examples of earlier attempts to realize similar network design patterns and achieve similar management objectives.

In the mid 1990s, for example, as the Internet began to see early adoption by the general public for commercial applications, network researchers realized that the standardization process for introducing new network control protocols and interfaces, required to introduce innovative new services, was slow and inefficient. Networking researchers proposed that networks could expose programming interfaces to realize more flexible control. Research into the area of Active Networks [52, 162] sought to alleviate this problem by making network devices expose APIs to control traffic and allocate resources, to realize a network programming model similar to that of general purpose computing systems. Key architectural challenges arose however, to realizing a general programming model for the network. Notably, the lack of a rigorous security/trust model for the programming of network elements as well as performance limitations of generally programmable network devices, played a significant role in limiting the general acceptance and widespread deployment of Active Networks.

The motivation to separate the control and data planes for faster network evolution and improved control led to a number of proposals for alternative architectures, as we describe in 2.1.2. One of the early, successful attempts at realizing such an architecture was in the context of solving the challenge of dynamic policy enforcement in enterprise networks. Conventionally, enterprise networks rely largely on Ethernet switches and IP routers whose ability to express complex and dynamic access control is limited in the presence of device and network address space mobility. Ethane [42] emerged as a practical approach to realizing dynamic policy enforcement and is widely regarded as the modern precursor to the contemporary approach to SDN as realized via a general *match-action* switch programming abstraction.

Regardless of the historical twists and turns encountered along the way to our contemporary notion of SDN, the explicit control over the data plane forwarding

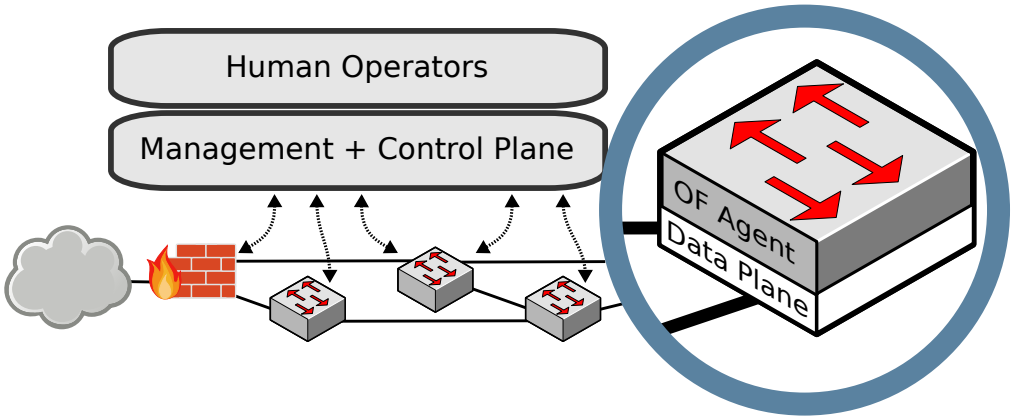


Figure 2.2: Conceptual organization of the management, control, and data plane of a Software Defined Network. Note that the softward-defined switches in such a network do not integrate control plane logic, but rather an agent which exposes a packet-forwarding API to a separate control program.

rules and control plane distribution model affords flexibility beyond that granted by the traditional network control plane. This flexibility has enabled innovation complementary to this thesis, in numerous areas of network management and operation [126]: verifying data plane and control plane invariants [41, 98, 99], network policy and programming language abstractions [59, 116], network virtualization [9] and network function virtualization [132] to name just a few.

Over the remainder of this thesis, we leverage the "logically centralized" control abstraction toward improved network management and troubleshooting. In the next chapter, we begin by examining certain fundamental design trade-offs inherent to this particular control abstraction.

3

The Logically Centralized Control Plane

The emergence of Software Defined Networking (SDN) [113] has sparked significant interest in rethinking classical approaches to network architecture and design. SDN enables the network control plane logic to be decoupled from the network forwarding hardware, and moves the control logic and state to a programmable software component, the *controller*. One of the key features enabled through this decoupling is the ability to design and reason about the network control plane as a *centrally* controlled application operating on a global network view (GNV) as its input. In essence, SDN gives network designers freedom to refactor the network control plane, allowing network control logic to be designed and operated as though it were a centralized application, rather than a distributed system – logically centralized.

Thus, SDN designers now face new choices; in particular, how centralized or distributed should the network control plane be? Fully *physically* centralized control is inadequate because it limits *(i)* responsiveness, *(ii)* reliability, and *(iii)* scalability. Thus, designers resort to a physically distributed control plane, on which a logically centralized control plane operates. In doing so, they face trade-offs between different consistency models and associated liveness properties.

Strongly consistent control designs always operate on a consistent world view, and thus help to ensure coordinated, correct behavior through consensus. This process imposes overhead and delay however, and thus limits responsiveness which can lead to suboptimal decisions.

Eventually consistent designs integrate information as it becomes available, and reconcile updates as each domain learns about them. Thus, they react faster and can cope with higher update rates, but potentially present a temporarily *inconsistent*

world view and thus may cause incorrect behavior. For instance, an inconsistent world view can cause routing loops or black holes.

Consequently, it is important to understand how physically distributed control plane state will impact the performance and correctness of a control application logic designed to operate as though it were centralized. Specifically, when the underlying distributed control plane state leads to inconsistency or staleness in the global network view, how much does the network performance suffer?

We approach this problem by systematically characterizing the state exchange points in a distributed SDN control plane. We then identify two key state distribution trade-offs that arise: *(i)* The trade-off between control application performance (optimality) and state distribution overhead and *(ii)* Application logic complexity vs. robustness to inconsistency in the underlying distributed SDN state.

We simulate these trade-offs in the context of an existing SDN application for flow-based load balancing, in order to evaluate the impact of an inconsistent global network view on the performance of the "logically centralized" control application. We compare two different control application approaches which operate on distributed SDN state: A simple approach that is ignorant to potential inconsistency in the global network view, and a more complex approach that considers the potential inconsistency in its network view when making a load balancing decision. In our simulation scenario, initial results demonstrate that global network view (GNV) inconsistency significantly degrades the performance of our network load balancing application which is naïve to the underlying distributed SDN state. The more complex application state management approach is more robust to GNV inconsistency.

We place our sensitivity study in the context of related work in Section 3.1. We present the inherent state exchange points and trade-offs in SDN design in Section 3.2 and discuss their impact on our example application in Section 3.3. We quantitatively explore the trade-offs using a simulation approach in Section 3.4, present our experiment setup and preliminary results, then conclude in Section 3.5.

3.1 Context: Distributed Network State

To put our work into the context of the more general problem of reasoning about distributed network state, we focus on three main categories: (1) Distributed control applications that motivate our study, (2) Control frameworks that provide a logically centralized view to SDN applications, and (3) Previous studies on routing state distribution trade-offs.

Our work is inspired by SDN applications such as in-network load-balancing [70,160] that require a distributed control plane implementation for scalability, but at the same time require an up-to-date view of the network to optimize their objective function. Our study explores this specific trade-off while being agnostic to control plane

implementation specifics – proactive vs reactive, micro vs macro-flow management, or short vs long timescale switch-controller interactions.

Onix [103] is a control plane platform designed to enable scalable control applications. Its main contribution is to abstract away the task of network state distribution from applications and provide them with a logical view of the network state. Onix provides a general API for control applications, while allowing them to make their own trade-offs among consistency, durability, and scalability. The paper does not evaluate the impact of these trade-offs on control application objectives; our study aims to kick-start investigation into this area. Similarly, Hyperflow [156] is a distributed event-based control plane for OpenFlow that allows control applications to make decisions locally by passively synchronizing network-wide views of the individual controller instances. The paper evaluates the limits of how fast the individual controllers can synchronize and the resultant inconsistency, but does not evaluate the impact of this inconsistency on the application objective. Consistent Updates [139] focuses on state management between the physical network and the network information base (NIB) to enforce consistent forwarding state at different levels (per-packet, per-flow). It does not explore the implications of distributed SDN control plane state consistency on network objective performance.

Correctness vs. liveness trade-offs emerge in the context of current and historical intra- and inter-domain routing protocol design. Consensus Routing [95] presents a consistency-first approach to adopting forwarding updates in the context of inter-domain routing. Using distributed snapshots and a consensus protocol, each router ensures that every other router along the path toward a destination agrees on each routing update. The protocol separately addresses safety and liveness properties to achieve correctness guarantees on forwarding behavior, e.g. loop-free packet-forwarding. Various studies have examined the impact of stale and inaccurate intra-domain link state on quality-of-service oriented path selection objectives [65, 143]. Both measurement and analytical studies have characterized the effects of different link-state collection, aggregation, and update announcement patterns in terms of the resulting QoS path selection objectives.

Probabilistically Bounded Staleness [33] is a set of models that predicts the expected consistency of an eventually-consistent data store – the underpinning of an eventually-consistent distributed SDN control plane. It provides a useful platform for exploring design choice consequences and performance trade-offs for realizing replicated, distributed SDN control state.

3.2 SDN State Distribution and Trade-offs

Before we dive into characterizing the state distribution and management trade-offs of SDN, we first describe the setting in which we define our problem. Figure 3.1 illustrates the key state exchange points in an SDN, organized into three logical

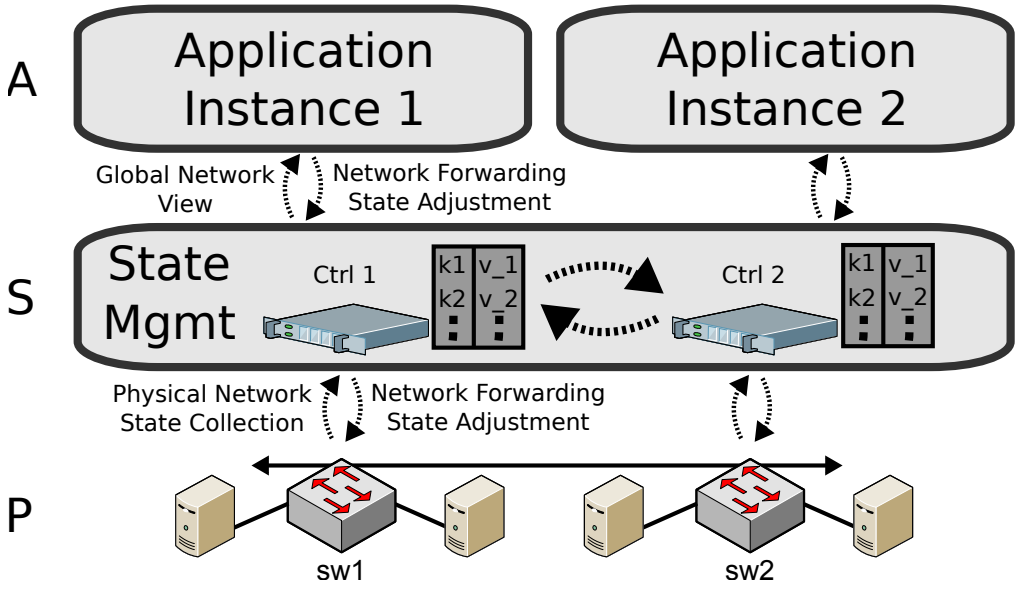


Figure 3.1: SDN state distribution and management conceptualized in layers: (A)pplication, (S)tate Management, (P)hysical Network

layers. This representation is similar to SDN control platforms found in Onix [103]. Each dashed arrow in the figure indicates a state exchange point in the SDN.

At the bottom (layer P), the *physical network* consists of the hardware forwarding devices which store the *forwarding information base* (FIB) state of the network data plane (e.g., TCAM Entries and configured port speeds), as well as associated meta-data including packet, flow, and port counters. The devices of the physical network are grouped into one or more separate *controller domains*, where each domain has at least one physical *controller*. Figure 3.1 depicts two domains: "Ctrl 1" governs "sw 1", and "Ctrl 2" governs "sw2". The devices of each domain expose read and write interfaces for the meta-data and FIB state to the domain controller in the state management layer, indicated by layer S.

The state management (Layer S) is the core of the SDN, and is realized by the controllers of each domain, which collect the physical network state distributed across every control domain. This component is sometimes called the "Network Operating System" (NOS), as it enables the SDN to present an abstraction of the physical network state to an instance of the control application (Layer A), in the form of a global network view. The control logic for each application instance may be run as a separate process directly on the controller hardware within each domain.

Each controller maintains a *Network Information Base* (NIB) data structure, which is a view of the global network state presented to an application. For instance, the NIB contents presented to a network load-balancing SDN control application would include at least link capacity and utilization state. The NIB at each controller

is periodically and independently updated with state collected from the physical network (e.g., through port counters or flow-level statistics gathering). Additionally, controllers synchronize their NIB state among themselves in order to disseminate their domain state to other controller domains.

Different manners of distributed, replicated storage models may be used to realize the NOS state distribution and management, including transactional databases, distributed hash tables, and partial-quorum mechanisms [142]. One key property of any NOS state distribution approach is the degree of state consistency achieved – strong (e.g., via transactional storage) vs. eventual consistency. A strongly consistent NOS will never present inconsistent NIB state to an application (i.e., each reader of the NIB will always see the value of the last completed write operation made to the NIB). However, the state distribution imposes overhead and thus limits the rate at which NOS state can be updated. While an update is being processed, applications continue to operate on a *stale* (but consistent) world view, even though more current information may be locally available. Eventually consistent approaches react faster, but temporarily introduce inconsistency – different global network views being presented to the individual physical controller instances.

Consequently, **trade-off #1** arises between the consistency model underlying NOS state distribution and the control application objective optimality. The performance of the network in relation to the control application's objective can suffer in the presence of inconsistent or stale global network view. Uncoordinated changes to the physical network state may result in routing loops, sub-optimal load-balancing, and other undesired application-specific behavior. The cost to achieving consistent state in the global network view entails higher rates of control synchronization and communication overhead, thus also imposing a penalty on responsiveness.

The degree to which the control application logic is more or less *aware* of the distributed nature of the underlying global network view constitutes **trade-off #2** between application logic complexity and robustness to stale NOS state. A "logically centralized" application that is unaware of the potential staleness of its input is simpler to design. An application which is aware of underlying distributed NOS state can take measures to separate and compare the inter-domain global network view with its own local domain view, and avoid taking action based solely on stale input.

3.3 Example Application: Network Load Balancer

To investigate these trade-offs, we choose a well-known arrival-based network load balancer control application. The load balancer objective is to minimize the maximum link utilization in our network. We present and compare two implementations featuring different state (and staleness) awareness and management approaches.

Link Balancer Controller (LBC): The simpler of our two application approaches is inspired by Aster*x [70] and "Load Balancing Gone Wild" [160]. Within a specific domain, upon a dataplane-triggered event (e.g. reaction to a new flow arrival or proactive notification of link imbalance within the domain), a global network view (table of links and utilizations) is presented by the NOS to the domain application instance. This view combines both the physical network state from within the domain as well as any inter-domain link utilization updates from other controllers. A list of paths (with utilizations) is generated from each ingress switch in the domain to every server which can respond to incoming requests (reachability information is provided by the NOS). From this list, the path with the lowest max link utilization is chosen on which to assign the next arriving flow and the appropriate forwarding state is installed in the physical network.

Separate State Link Balancer Controller (SSLBC): This control application keeps fresh intra-domain physical network state separate from updates learned through inter-domain controller synchronization events. The arrival-based path selection incorporates logic to ensure convergence properties on load distribution.

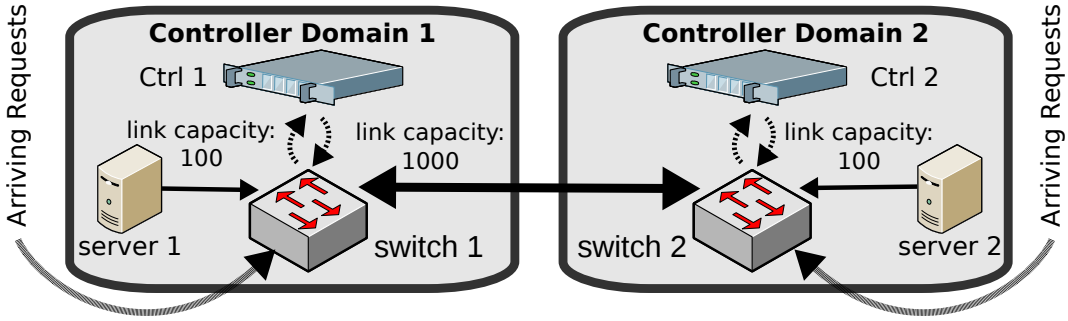
Using the (potentially stale) global network view, it determines the ingress-to-server path, with the maximum link utilization along the path (we call this path P_{max}). It calculates what fraction F of traffic *would* need to be redistributed off of P_{max}, (onto the other links) to balance all the paths, however no active flows are actually migrated. Next, using only the fresh link utilization state from within its own domain, it calculates new link metrics within the domain. Using a convergence parameter $\alpha \in (0, 1]$, it scales each link utilization value by $F \times \alpha$ (a fraction of the domain's contributed load from path P_{max} across the links of other available paths). From this set of new link metrics, the path with the minimum max link utilization is then chosen for the next arriving flow. Effectively, the global network view guides each application instance to redistribute a scaled (α) fraction of its local link imbalance on a flow-by-flow arrival basis.

3.4 Experiments

We now discuss our investigation of the trade-offs described in the last sections. We describe the custom simulator we use, discuss our experiment setup, and describe and discuss our findings.

3.4.1 Simulation

To explore our state distribution trade-offs in a controlled and deterministic manner, we develop a custom simulator to implement the key state-exchange interfaces of an SDN, as introduced in Section 3.2. We opt for a custom flow-level simulation in this study, as we are first and foremost interested in having explicit control over the



(a) Simulated topology

Link Name	Type	Capacity (Bandwidth)
s1-sw1	server-switch	100 units
s2-sw2	server-switch	100 units
sw1-sw2	inter-switch	1000 units
sw2-sw1	inter-switch	1000 units

(b) Simulated link capacities

Figure 3.2: Simulated topology and link capacities

specific aforementioned state-exchange interfaces. We release our simulation as an open-source tool [108].

Our simulation is designed to capture interactions between three SDN layers from Figure 3.1. First, a graph data-structure [68] representing the physical network is instantiated with a topology structure, and link capacity and utilization annotations. Next, the NOS is instantiated from individual controller instances and controller domains are mapped to the network graph. Each controller is given its own copy of the graph data-structure as its NIB to update and distribute among the other controllers. Finally a flow workload definition is created, which is list of 4-tuples: Arrival time (in simulation time), ingress switch (where it enters our simulated network), duration, and average link utilization. The workload, controllers, and physical topology are provided to the simulation, and the simulation then begins iterating through the workload.

As the simulation iterates through the workload, simulation time is updated to the time of the flow arrival. Any existing flows which have ended prior to this time are terminated, and their consumed link utilization is freed back into the available link capacity along the path it used. Each controller then records the link utilization values from the physical network within its domain. Next, an all-to-all synchronization may be triggered depending on whether the chosen synchronization period for the simulation run has elapsed.

The controller application of the domain in which the flow arrives is given the opportunity to assign the new flow to a path to any of the servers of which it is aware.

Each controller application is able to assign flows entering from its domain's ingress switch to a server replica in the other controller domain. Based on its view of the network and the specific state management approach of the control application, the flow will be allocated to a path with the objective of minimizing the maximum link utilization in the network. The process is illustrated using 6 example steps in Figure 3.3.

3.4.2 Experiment Setup

For our first experiments, we choose our topology to be as simple as possible, yet still involve distributed state – two cooperating controller domains, as illustrated in Figure 3.2(a). Each domain consist of a single switch and a single server. For simulation purposes, we consider upstream traffic (e.g., http requests toward servers) negligible compared to downstream (e.g., streaming download content toward the switches), and therefore only simulate the link capacities and utilizations of four links in total, as given by the table in Figure 3.2(b). We consider both servers as identical replicas, able to serve content for all requests, constrained only by the available downstream bandwidth.

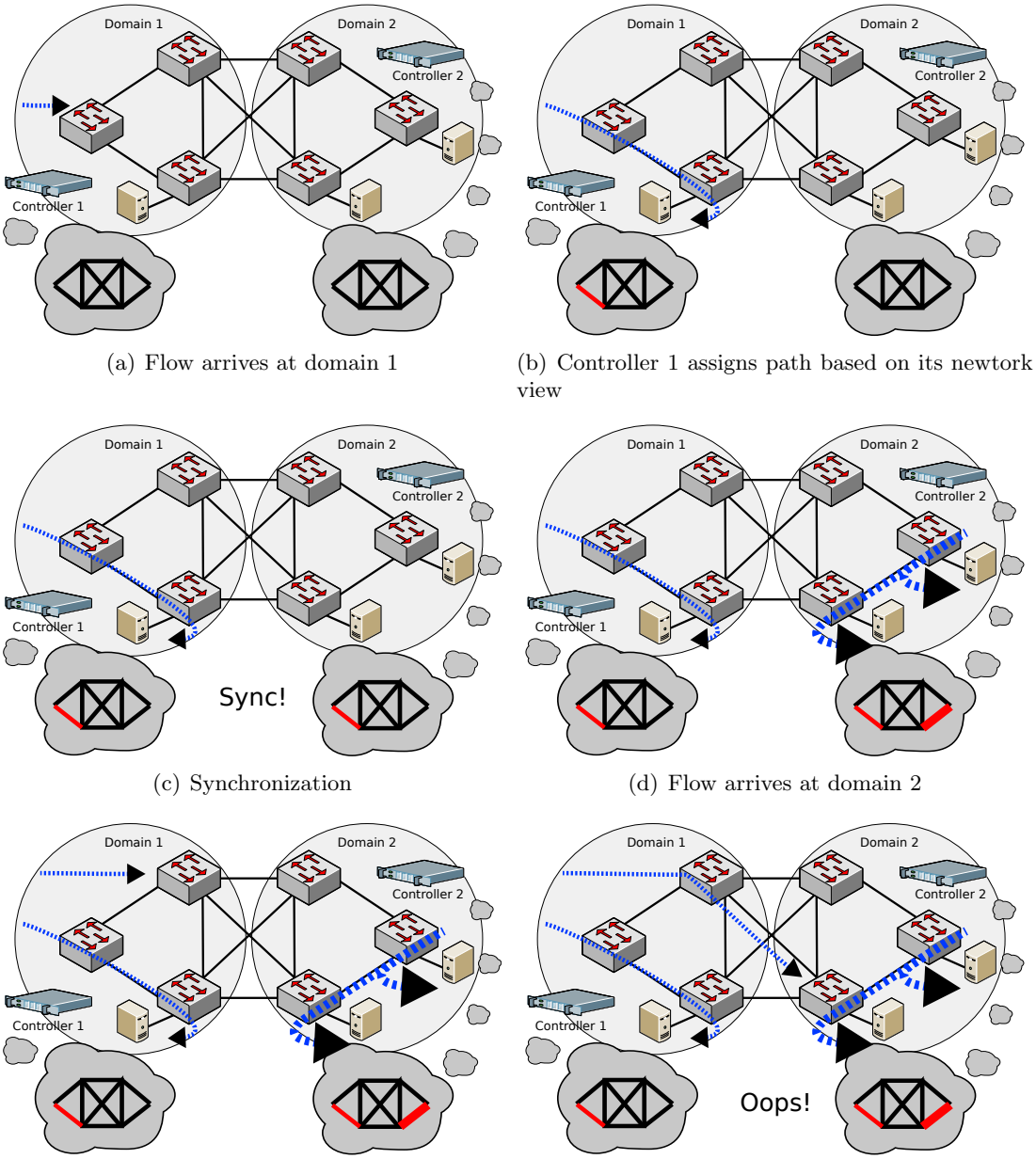
We choose fat inter-domain links, as they enable both domains to better cooperate in serving incoming requests. Alternately, a bottleneck between domains incentiveizes each domain to keep flows within its domain – limiting the value of an inter-domain cooperative load-balancing application.

The load balancer objective is to minimize the difference between all link utilizations, we choose RMSE – root mean squared error (the Euclidean distance) – of the maximum link utilization along each server-switch path in the network. Thus, if the maximum link utilization over every server-to-switch path is equal, our RMSE metric is 0.

In our experiments, we present results using an α value of 0.3, however our results are not sensitive to this parameter in the simple scenario because of 1) the binary choice of path presented to each controller and 2) flows are unsplittable. We present results for this simple topology, however simulations run on 3-domain chain and ring topologies exhibit very similar behavior. It is important to note that our results for this simple model are laregely influenced by the topology and in general, the topology may have an influence on the results even though we do not see it see it for 3-domain chain or ring topologies.

3.4.3 Workload

We use two different workloads to drive our simulations and explore the impact of inconsistent NOS state on network load balance. We first utilize a deterministic, controlled workload to impart a link utilization imbalance. This workload oscillates



(a) Flow arrives at domain 1

(b) Controller 1 assigns path based on its newtork view

(c) Synchronization

(d) Flow arrives at domain 2

(e) Controller 2 assigns path based on its newtork view

(f) Controller 1 assigns path to new flow based on its newtork view

Figure 3.3: Illustration of a Simulation Execution

ingress load between the two switches, keeping constant the total load ingress into the network. Second, we apply a more realistic workload using exponentially distributed flow inter-arrival times and Weibull distributed flow durations.

We realize the first of our two workloads, by choosing a flow arrival rate that is driven by a sin-function of a discrete time parameter t. More specifically, the flow arrival rate at each switch over a simulation time step $(t, t+1)$, is defined by $sin(t/T) + 1$ and $sin(t/T - T/2) + 1$ respectively, where T is the period of the oscillation. We choose a wave period of $T = 64$ simulation timesteps and run the simulation for 256 time steps, which leads to oscillating link utilization dynamics. We consider different workloads consisting of $32, 64, 128$ flows arriving within each time period to understand the impact of medium, heavy, and over-subscribed workloads respectively. We present results using 32 arrivals per timestep with fixed flow duration of 2 timesteps and a fixed average link utilization of 1 link capacity-unit per timestep. This gives a total ingress of 64 load units into the network at any point in time, keeping in mind that each server-to-switch link can carry a maximum capacity of 100 units.

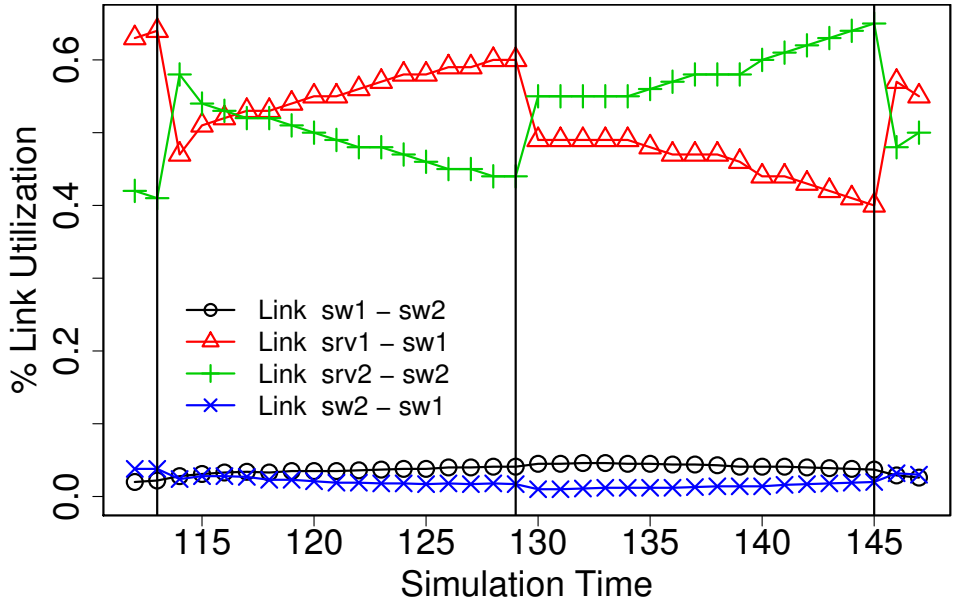
3.4.4 Results

We now use the load balancer based upon the LBC state management approach with different synchronization intervals: $0, 1, 2, 4, 8$ and 16 timesteps. Here, 0 implies that we synchronize the state between any two changes in NOS state.

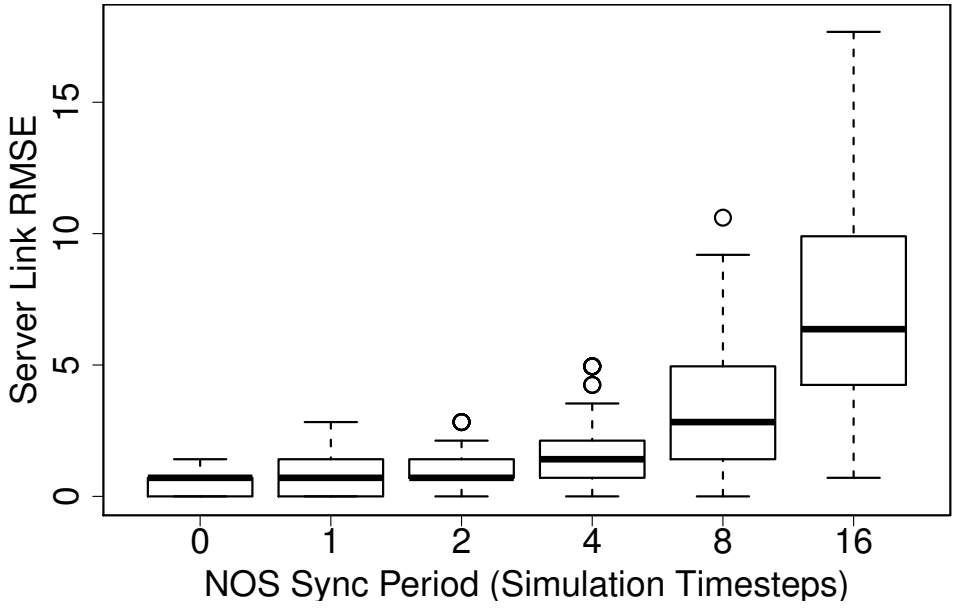
Controlled workload—LBC

For the simulation with synchronization interval 16 Figure 3.4(a) shows an excerpt of the link utilization time series progression for the time period of 112 to 147. From the figure we confirm that the two server-switch links are indeed the "bottlenecks" with a utilization in the range of 40% to 60%. We also see the impact of synchronization at time steps 113, 129, and 145 (see support lines). After each synchronization step the load balancer reassigns a significant fraction of newly arriving flows to the other server and therefore to the other path — thus significantly changing the link utilizations. This results in an improved link balance for about 4-6 time steps later. Beyond that time, however, diverging link utilization state leads to inconsistent global network view as seen by each controller. The load balancer makes increasingly poorer decisions and the link imbalance grows. Also due to the oscillations in the workload, the switch with the greater ingress load changes over time. Similar patterns apply for shorter synchronization intervals and higher workload flow arrival rates.

To evaluate the load balancer performance, we consider the RMSE values over each simulation run, and exclude the first 16 timesteps. In Figure 3.4(b), we present the RMSE values for simulation run of increasing synchronization period in the form of a box-plot. Each box-plot shows the center half of the data (the box) with the median marked. The whiskers show the 95 percentiles and outliers are drawn separately. Recall, an RMSE of zero corresponds to balanced loads. We see that the RMSE



(a) Link Utilization Timeseries (Sync Period = 16)

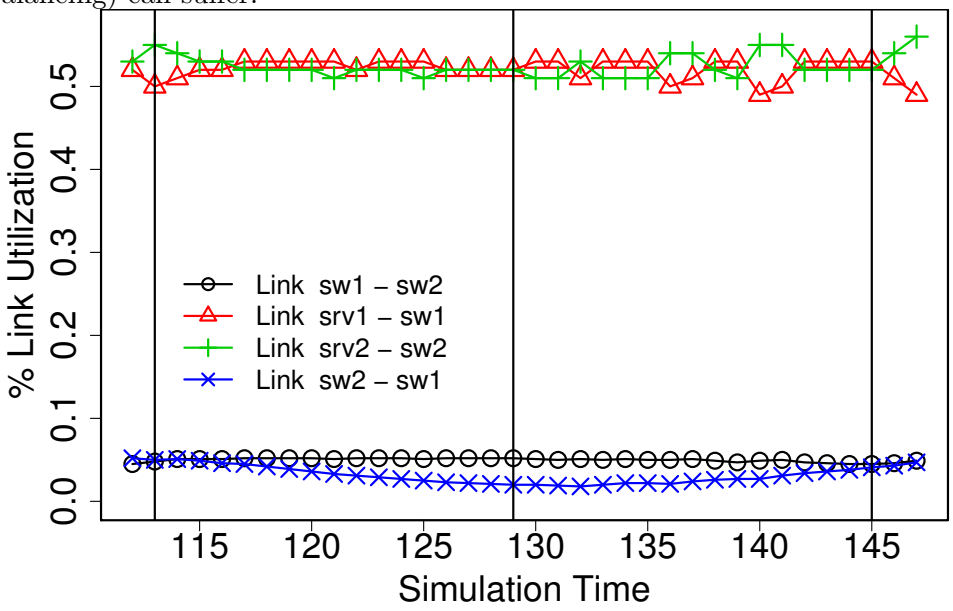


(b) Load Balancer Performance vs. Sync Period

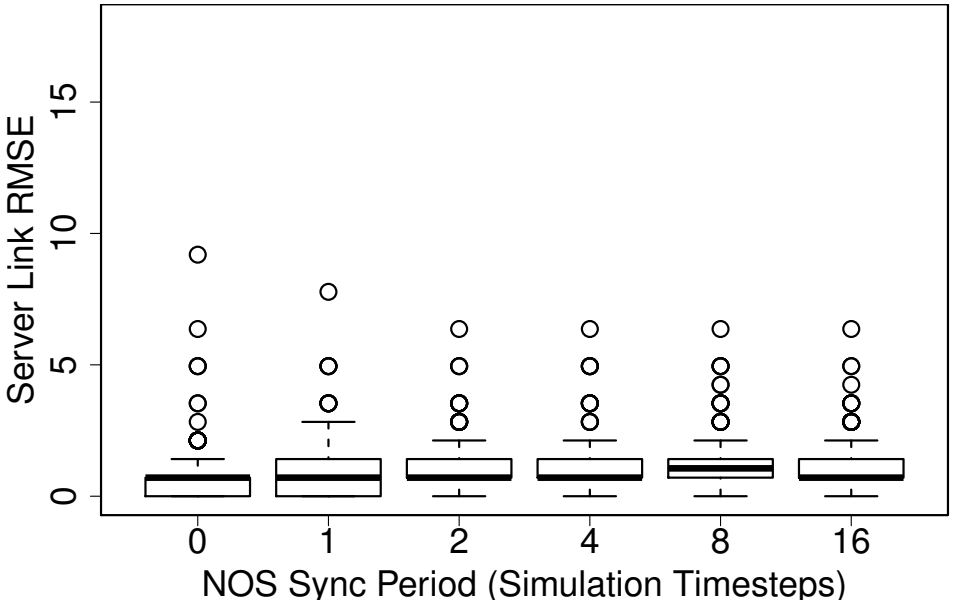
Figure 3.4: LBC: global network view inconsistency vs. control application performance

value range increases as the synchronization periods increases. This is due to the effects highlighted by Figure 3.4(a). The median RMSE at sync period 16 is over 8× the imbalance at sync period 1. This underlines our first trade-off – as the global

network view becomes inconsistent, the application performance (in this case load balancing) can suffer.



(a) Link Utilization Timeseries (Sync Period = 16)



(b) Load Balancer Performance vs. Sync Period

Figure 3.5: SSLBC: global network view inconsistency vs. control application performance

Controlled workload—SSLBC

Next, we examine the second trade-off, namely, how the SSLBC state management approach is able to handle the above workload. Figure 3.5(b) shows the box-plot of the RMSE metric for each synchronization interval using the SSLBC state management approach. Comparing Figure 3.4(b) and 3.5(b) we observe that at high NOS synchronization rates, SSLBC achieves a mean RMSE comparable to LBC. As NOS staleness increases, the performance of SSLBC degrades far less as compared to LBC. However, for high rates of NOS synchronization, SSLBC may not perform as well because it is more conservative with respect to load balancing. Alternately, this improves SSLBC robustness to NOS staleness.

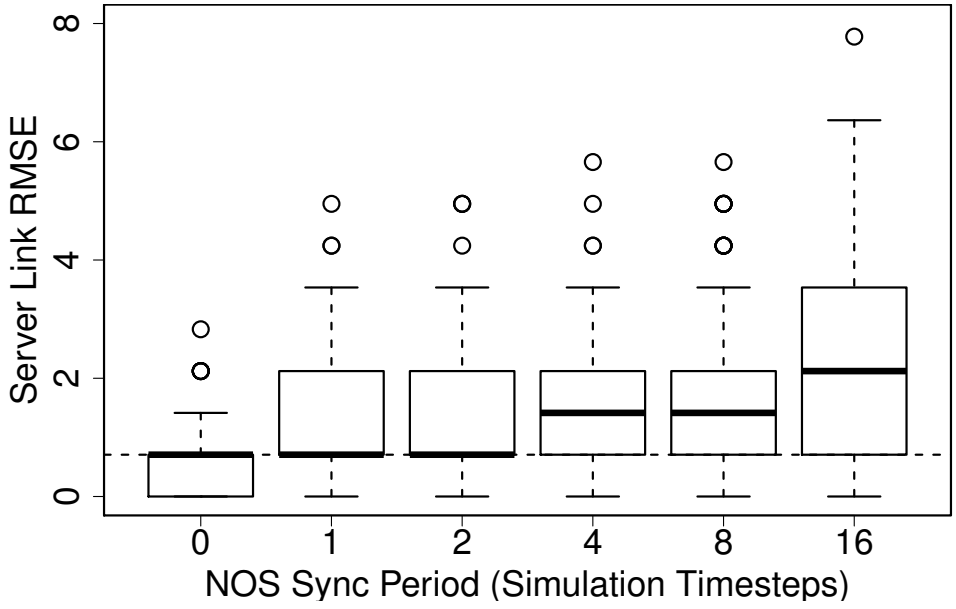
For example, Figure 3.5(a) shows the link utilizations for the same period as Figure 3.4(a). We see again see the effect of synchronization as well as the effect of the oscillations in the imposed load. However, the effects are much smaller, leading to lower RMSE values. Thus, we see the effects of the second trade-off – a more conservative control application design which is aware of underlying distributed state leads to less sensitivity to NOS staleness.

Toward a more realistic workload

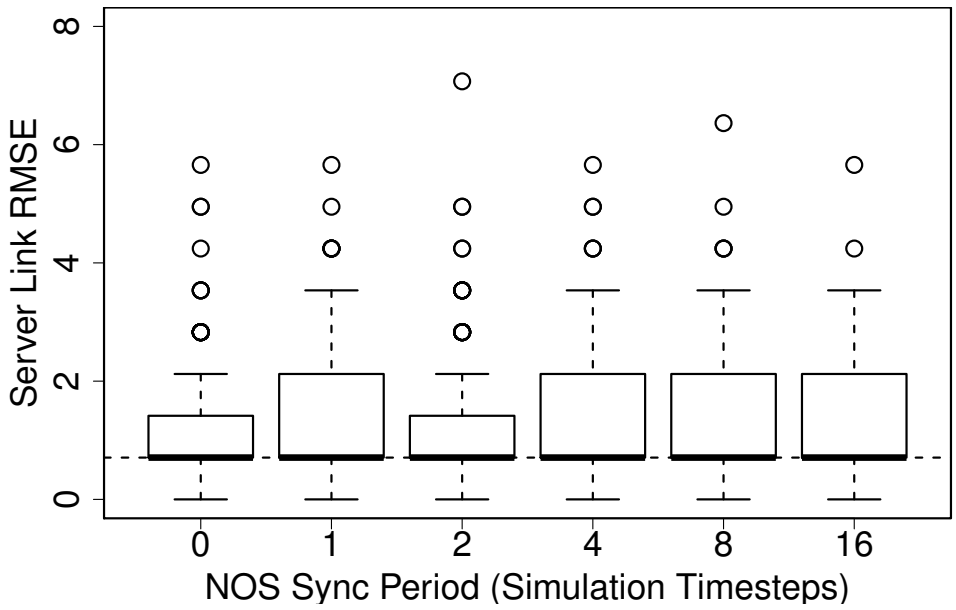
We now evaluate the two previously illustrated trade-offs in the context of a more realistic workload. The simplistic *sin* function workload from earlier effectively illustrates the cost of NOS inconsistency to the control application performance. A more realistic flow arrival process and duration distribution is necessary, however, to better estimate how these expected trade-off may behave in practice.

This more realistic workload uses exponentially distributed flow inter-arrival times to define the ingress load on a per-switch basis. We modulate the mean of the exponential distribution by the wave function from earlier to achieve ingress load oscillations between switch 1 and switch 2. Flow durations are obtained from Weibull distribution with mean 10 (timesteps) and shape 0.5 to achieve a network ingress load over the simulation that is comparable to the earlier presented workload.

In this evaluation, as before, we vary the time interval between NOS synchronization events, and compare the load balancing performance of our LBC and SSLBC state management approaches. We reuse the same performance metric, RMSE over the server links and for each synchronization interval, present a box-plot of these values over an entire simulation run. We see in Figure 3.6(a) that as the sync period increases, the LBC again exhibits a noticeable increase in median RMSE. By comparison, Figure 3.6(b) shows the SSLBC performance remains almost unchanged for increasing sync periods. Additionally, at each synchronization period, the median RMSE of the SSLBC is equal to or strictly less (better) than that of the LBC application state management approach. The SSLBC shows less improvement over the



(a) LBC Performance vs. Sync Period



(b) SSLBC Performance vs. Sync Period

Figure 3.6: Load balancer performance comparison under more realistic workload

LBC at lower synchronization periods, as a heavy-tailed flow duration distribution can not be so easily accommodated by an arrival-based load balancing approach. These results support our earlier conclusions on the impact of stale NOS state on the control application performance.

3.5 Summary

In this chapter, we investigate the details behind the *Logically Centralized* control plane enabled through Software Defined Networking. Logically centralized world views, as presented by controller frameworks such as Onix [103] enable simplified programming models. However, as the logically centralized world view is mapped to a physically distributed system, fundamental trade-offs emerge that affect application performance, liveness, robustness, and correctness. These trade-offs, while well studied in a different context in the distributed systems community, are relevant to the networking community as well. It is our position that they should be revisited in the context of design choices exposed by software defined networks.

This chapter characterizes the key state distribution points in the distributed SDN control plane. We identify two concrete trade-offs, between *staleness* and *optimality*, and between *application logic complexity* and *robustness to inconsistency*. We further discuss a simulation based approach to experimentally investigate these trade-offs. For a well-known SDN application (load balancing) and a simple topology, we find that *(i)* view staleness significantly impacts optimality and *(ii)* application robustness to inconsistency increases when the application logic is *aware* of distribution.

Although our current work focusses on a load-balancing control application scenario, note that similar trade-offs arise in other SDN control applications, e.g, in distributed firewalls, intrusion detection, admission enforcement (policy enforcement correctness vs. liveness), routing, and middle-box applications [136] (path choice optimality vs. liveness).

Certainly, more work is required to make our results quantitatively meaningful in the general SDN context. To this end, in the following chapters, we

- model and analyze *concurrency* in distributed SDN control in **Chapter 4**.
- extend our *simulation* approach and introduce emulation and test-bed approaches to more realistically model traffic characteristics, and resource constraints, under different SDN deployment scenarios in **Chapter 6**
- and apply our tools to larger and more complex *topologies* in **Chapter 6**

Ultimatley, we intend for this sensititivy study to spark a discussion about the details and trade-offs of logical centralization. Building on our early model and prototype, we aim to develop a tool-set that facilitates practical exploration of these state-distribution trade-offs for further SDN applications and deployment scenarios, their concrete algorithms, workloads, and topologies.

4

Software Transactional Networking: Consistent Distributed Policy Composition

In Chapter 3 we illustrated through sensitivity-study, a landscape of the SDN control plane design trade-offs. In this chapter, we pick up our investigation into the distributed SDN control plane with a specific focus on the problem of ensuring safety and liveness properties in the presence of concurrency.

The raison d'être of Software Defined Networking (SDN) is to enable the specification of the network-wide policy, that is, the desired high-level behavior of the network. While SDN makes this possible, there are several other requirements that are not immediately accommodated by this model. First, the network-wide policy unlikely comes from a single, monolithic specification, but rather, it is composed of *policy updates* submitted by different functional modules (*e.g.*, access control, routing, *etc.*). It may ultimately come from multiple human operators, each of which is only responsible for a part of it. Second, while control in SDN is logically centralized, important considerations such as high availability, responsiveness, and scalability dictate that a robust control platform be realized as a distributed system.

To support modular network control logic and multi-authorship of network policy, we face the issues of policy *composition* and *conflict resolution*. In previous work, Foster *et al.* [59] and Ferguson *et al.* [57] have addressed these two issues to a good extent in centralized (or *sequential*) settings—namely, in which there exists a central point for resolving policy conflicts and serializing policy composition. In this work, we bring existing approaches one step further, and present a solution for

distributed composition of network policies that supports *concurrency* among the network control modules.

To satisfy commercial deployment requirements, Onix [103] realizes a control platform that relies on existing distributed systems techniques and builds upon the notion of a Network Information Base (NIB), *i.e.*, a data structure that maintains a copy of the network state. Different control modules can operate concurrently by reading from and writing to the NIB. While Onix handles the replication and distribution of the NIB, it does not provide policy synchronization semantics. Instead, Onix expects developers to provide the logic that is necessary to detect and resolve conflicts of policy specification. In this chapter, we argue that *synchronized policy composition* must be an indispensable element of the distributed control platform.

We illustrate in Section 4.1 that a concurrent execution of *overlapping* policy updates (*i.e.*, affecting overlapping sets of traffic flows), may result in serious inconsistencies on the data plane (*e.g.*, in violation to the policy, traffic not being forwarded to an IDS middlebox). To fix this, we need to ensure that concurrent overlapping policy updates are *consistently* synchronized. Informally, we propose to define consistency by reducing it to the equivalence to a sequential policy composition, in the spirit of how correctness is defined for concurrent data structures [74]. We distinguish between strong and weak consistency levels for policy composition. Informally, we say that strongly consistent policy composition results in all data plane traffic experiencing the same globally ordered sequence of composed policy updates. Weakly consistent policy composition allows different traffic flows to experience differently ordered sequences of composed policy updates. In the absence of permanently pending policy updates, both strongly and weakly consistent concurrent policy compositions eventually lead to the same network-wide forwarding state.

Of course, we envision that some policy updates cannot be installed either because their composition is not defined [57, 59] (*e.g.*, mutually exclusive forwarding actions) or due to unforeseen network conditions (*e.g.*, failures or scarce network resources). Therefore, we stipulate that the synchronization interface should be *transactional*. A policy update submitted by a control module either *commits*, *i.e.*, the update is successfully installed, or *aborts*, *i.e.*, it does not affect the data plane. We term our approach "Software Transactional Networking", inspired by the software transactional memory (STM) abstraction for optimistic concurrency control [144].

Implementing strongly consistent transactional synchronization can be achieved by centrally serializing the requests at a single master controller or using state-machine replication. However, we believe such heavy-weight solutions are not required, since weakly consistent policy composition is good enough for many cases. We describe a surprisingly simple light-weight implementation of it that only requires low-level, per-switch port conflict resolution, which can be implemented by, *e.g.*, maintaining one lock per hardware switch. Our implementation assumes that rules for composition be specified using the earlier work [57, 59], and ensures that the data plane is never affected by partially installed policies using a variation of the *two-phase*

update protocol of Reitblatt *et al.* [138]. Consequently, we argue that policy synchronization protocols must be first-class citizens in software-defined networking by presenting counter examples and describing solutions.

The remainder of this chapter is organized as follows. In Section 4.1, we demonstrate scenarios in which naïve NIB-based approaches to implementing concurrent policy updates result in inconsistencies in the data plane. In Section 4.2, we briefly describe the details of our model and state the problem of consistent policy composition. In Section 4.3, we sketch our implementation of weakly consistent policy composition. We overview the related work in Section 4.4. We conclude by discussing limitations of our results and speculating on future work in Section 4.5.

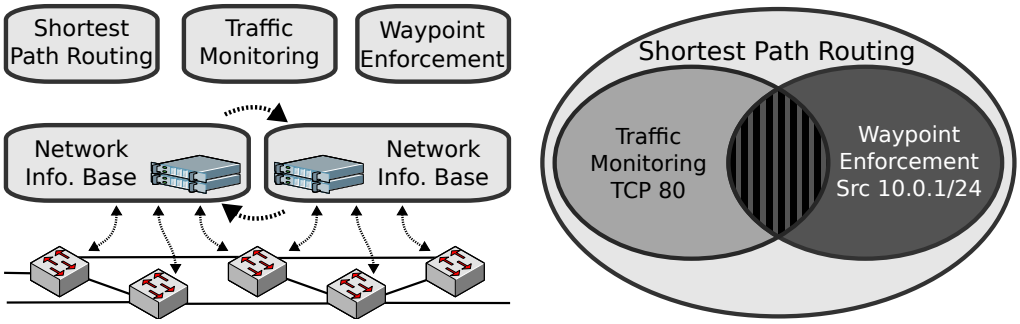
4.1 Concurrent Composition

We now discuss two simple examples that illustrate the need to extend previous work on SDN policy composition by addressing the concurrency issues in the distributed controller platform setting. Figure 4.1(a), logically illustrates our setting, an Onix-like SDN architecture in which two distributed controller instances run three functional modules—a packet repeater (hereafter called shortest path routing), a web traffic monitor, and a waypoint enforcement module coupled to an Intrusion Detection System (IDS).

In the first example, we begin with the policy composition scenario introduced in *Frenetic* [59], a network programming language and run-time system to automatically and correctly "compose" policies from separate control modules, and compile them to low-level forwarding rules. Assume the initial network policy comes from just the shortest path routing and web traffic monitor. Because these modules generate overlapping policy updates, they must compose their forwarding rules before any rule may be installed. The crux of the composition is that the correct forwarding rule priorities must be chosen such that the more specific web-monitoring rule matches packets with higher priority than the overlapping shortest path forwarding rule. As we illustrate in the flow-space diagram in Figure 4.1(b), the policy whose action applies to the striped region of flow-space, for example, must be defined consistently over the entire network.

Let us first assume that the monitoring and forwarding policies are generated sequentially in time. Even in the absence of concurrency, some synchronization step would be necessary for the latter controller instance to detect and resolve which policy should apply to any overlapping region of flow-space. For example, the composition of a monitor update request to count HTTP packets (`tcp_port=80` $\rightarrow$ `count`)[1] with a forwarding update request to forward packets arriving from address `10.0/16` to

[1] Although in OpenFlow each rule is implicitly associated with packet and byte counters, for presentation sake we represent "count" as an explicit action.



(a) 3 control modules in an Onix-like architecture.

(b) Per-policy flow-space.

Figure 4.1: Three composed policies and their respective flow-space overlaps.

Policy Source	Priority	Match Pattern	Actions
Monitoring	0	`tcp_port=80`	`count`
Waypoint Enforcement	0	`src=10.0.*`	`fwd`(2)
Monitor & Waypoint	1	`src=10.0.*` $\wedge$ `tcp_port=80`	`count`; `fwd`(2)

Table 4.1: Policies and the resulting forwarding rules

port 2 (`src=10.0.*` $\rightarrow$ `fwd`(2)) should result in the forwarding rules as given by table 4.1 regardless of the order in which the requests are processed.

In the absence of such an agreement, conflicting rule priorities may be selected and different paths through the network would yield different policies, by virtue of the order in which the respective controller instance completed its update. For this specific case, the resulting data-plane inconsistency at the end of both updates would be that some paths through the network monitor web traffic volume while some do not. Note that even when both controllers utilize per-packet consistent network update semantics [138], the matching rule priority issue is not guaranteed to be resolved.

Next, we demonstrate how an inconsistent policy composition arises when multiple controller instances execute asynchronously and concurrently. Consider again two separate controller instances executing the three functional modules as depicted in Figure 4.1(a) over some time period based on their view of the network. Each controller instance executes with the goal of installing its resulting composed policy, in the form of new or modified flow-table entries into the underlying switches, using per-packet consistent updates.

Note that whenever policy updates compose trivially, *i.e.*, they affect disjoint network-wide slices of the flow-space, concurrency issues resolve automatically (as in the case of the FlowVisor [148]). In contrast, when composition is non-trivial, given concurrent execution, the previous policy composition problem becomes more difficult to

detect and resolve. Without synchronization it is impossible at any single point in time for a particular controller instance to ensure that its policy specification does not conflict with that of any other's (see the companion technical report [40]).

We illustrate an asynchronous policy composition example using two controller instances A and B running concurrently in Figure 4.2. At time t_0, controller A begins a per-packet consistent network update to begin collecting web traffic statistics. From the perspective of controller A, it begins a transition along path i. Soon afterward, at time t_1, an influx of scanning traffic from source network 10.0.1/24 triggers the IDS module of controller instance B to execute a waypoint forwarding policy update to capture the scanning traffic for analysis. As the consistent update from controller A is still in progress, controller B computes an update as though it were transitioning along edge j. By time t_2, once the update execution of A and B have completed, at least three rules with different matching priorities will have been installed at each switch for the overlapping region of flow-space depicted in stripes in Figure 4.1(b). In the absence of a policy synchronization semantic, any of the three illustrated end-states may be reachable in the network. Each crossed-out flow-space diagram violates either the monitoring or waypoint enforcement policy, respectively. The goal of concurrent consistent policy composition is to ensure that only one such final state is reached eventually, everywhere in the network, and that the final state is not in violation of the policies which lead to its installation.

4.2 The STN Abstraction

This section gives an overview of the *Software Transactional Networking (STN)* architecture we propose. This architecture relies on a middleware offering a simple transactional interface where policy updates can be applied; the middleware implements the updates correctly and efficiently. We define what *correctly* means below. The discussion of efficient implementations is postponed to the subsequent section.

4.2.1 Interface

Concurrent policy updates may *conflict* in the sense that, intuitively, they may apply mutually exclusive actions to overlapping sets of packets. The goal of the synchronization layer of the network control platform is to make sure that all requests are composed in a "meaningful" manner, so that every packet in the network does not experience an inconsistent policy composition. On the other hand, if some conflicting policies cannot be automatically composed,[2] the STN layer may reject some requests. In that case, the corresponding control modules receive a *nack* response equipped with some meta-information explaining what kind of conflict was witnessed.

[2]Note that certain conflicts can be arbitrated, *e.g.*, by requiring a priori knowledge of a strict priority order across modules as in [57].

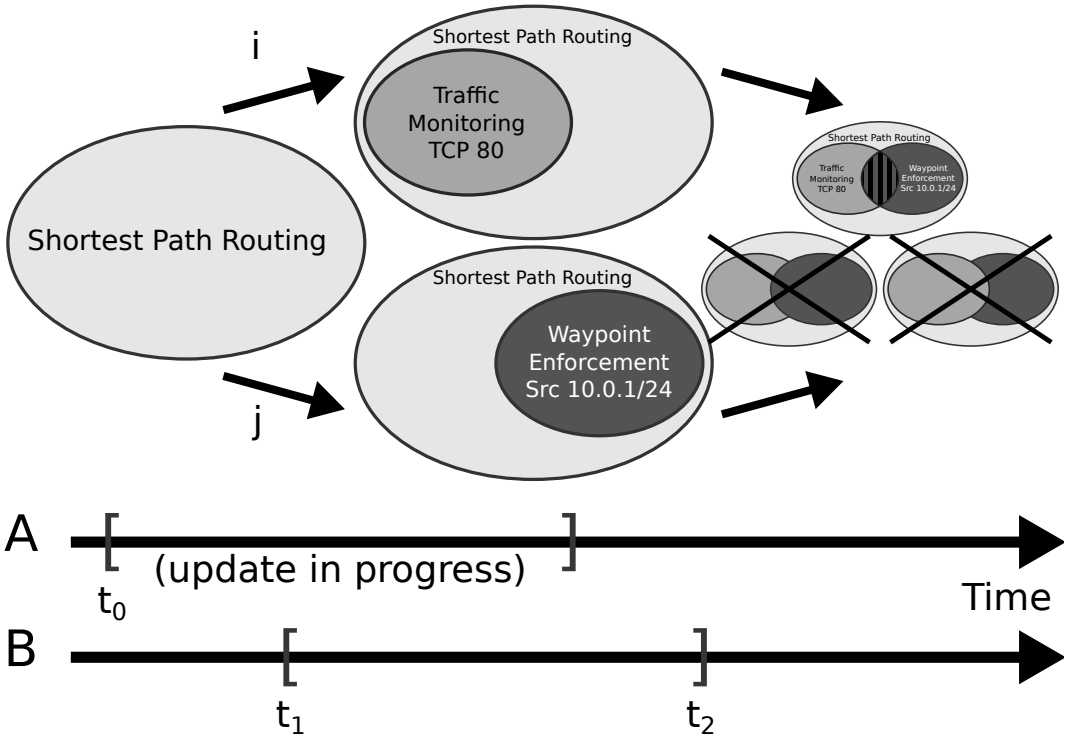


Figure 4.2: Possible policy composition outcomes from two concurrent policy updates. Conflicting compositions (crossed out) must be avoided.

Formally, a controller module tries to install a new policy p (a collection of rules to be installed at different switches) by invoking *apply*(p). The invocation may return *ack* meaning that the policy has been installed or *nack*(e) meaning that the policy has been rejected, where e is the explanation why.

4.2.2 Preliminaries

Before we can introduce our notions of update consistency, some formalism is needed. We call the sequence of invocations and responses of policy-update requests plus all data-plane events (such as data packet arrivals and departures at the ports of the switches), a *history* H. We say that request r_1 *precedes* request r_2 in H, and we write $r_1 \prec_H r_2$, if the response of r_1 precedes the invocation of r_2. Note that, in a concurrent execution, $\prec_H$ is a partial order on the set of requests in H. A request r is *complete* in H if H contains a response of r. A complete request is said to be *committed* in H if its response is *ack*, or *aborted* otherwise.

For a history H, let $P_c(H)$ denote the set of policies resulting from sequential composition of all committed policies in H in the order respecting $\prec_H$, *i.e.*, no request r is (sequentially) processed before any request r' such that $r' \prec_H r$. Let $P_i(H)$ be

the set of policies that result from a sequential composition of a *subset of* policies submitted by incomplete requests in H. Thus, a policy $p \circ p'$ such that $p \in P_c(H)$ and $p' \in P_i(H)$, is a composition of all already installed policies and a subset of policies that are being installed in H.

Following [138], we stipulate that every packet joining the network incurs a set of *traces*: sequences of causally related *located packets*, *i.e.*, $(packet, port)$ pairs, indicating the order in which the packets traverse the network. Informally, a trace is *consistent with a policy* p [138] if all the events of the trace are triggered respecting p. For a trace π in a history H, let H_π denote the prefix of H up to the first event of π (*i.e.*, up to the moment when the corresponding packet enters the network).

4.2.3 Consistency

Intuitively, we want to ensure that every trace in H respects a policy resulting from a *consistent* composition of previously and concurrently installed policies. We distinguish between *strong* and *weak* consistent composition. In the strong version, policy-update requests are composed in the same linear order, and in the weak one incomplete policies can be composed arbitrarily.

For example, weakly consistent composition allows the scenario depicted in Figure 4.2, where packets can be affected by any policy update that is not yet completed (corresponding to the policies along paths i and j in the figure). In contrast, strongly consistent policy may only take one of the paths, *i.e.*, before both updates complete, all the traffic is processed either according to the one of the concurrent policy updates or the other one, and then it is processed by the composition of the two.

Formally, we say that a history H provides *weakly consistent composition* if for every trace π in H, there exist policies $p \in P_c(H_\pi)$ and $p' \in P_i(H_\pi)$, such that π respects $p \circ p'$ (the composition of p and p').

Respectively, a history H provides *strongly consistent composition* if there exist a total order $S = (p_1, p_2, \ldots)$ of policies installed by requests in H such that (1) S respects $\prec_H$, and (2) every trace π in H respects a policy $p_1 \circ p_2 \cdots p_{i-1} \circ p_i$ that consists of all policies submitted by committed requests in H_π and a subset of policies submitted by a subset of incomplete requests in H_π. In other words, we require that there exists some global order in which policies evolve.

Note that we deliberately leave specifying the details of how exactly the *sequential* composition is done to the control logic since this is in general application-specific (and realizable, *e.g.*, through previous work in [59]). In particular, for our weaker notion of consistency, concurrently committed requests may be required to *commute*, *i.e.*, their sequential composition results in the same policy regardless of the order in which they are applied. Indeed, since we allow concurrently installed policies to be

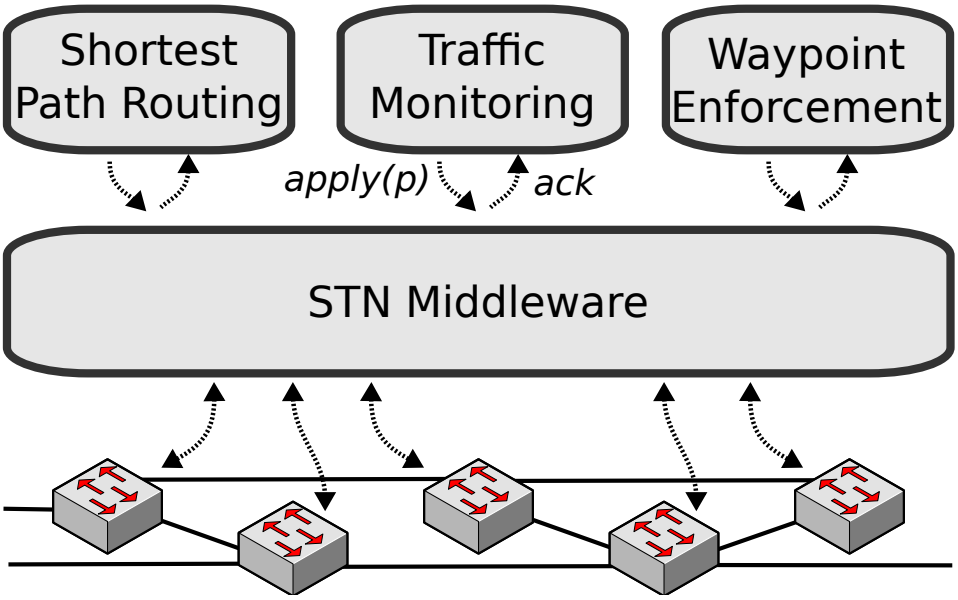


Figure 4.3: A logical representation of the STN middleware.

witnessed by different traces to be composed in arbitrary order, we may want this order to be effectively the same (*e.g.*, recall the composition of the monitor update and the shortest path routing update in Section 4.1).

4.3 The STN Middleware

In this section, we sketch a simple implementation of weakly consistent policy composition. We investigate a proposal for strongly consistent policy composition in a separate technical report [40]. On the progress side, we guarantee that every request eventually completes by returning *ack* or *nack*. To filter out a trivial solution that always returns *nack*, we require that no policy-update request aborts, unless it faces a *conflicting* policy with which it cannot be composed. The exact definition of a conflict is left to the control application; a practical definition comprises regarding mutually exclusive actions to overlapping sets of packets as conflicts.

The basic building block of our solution is the *two-phase update* algorithm [138]. In the first phase, the algorithm first installs the new policy on the internal ports[3] of the network (in arbitrary order) for packets with a unique tag number. In the second phase, the algorithm updates the ingress ports of the network (in arbitrary order) to equip all incoming packets with the new tag number. This way, while the

[3]Internal and ingress ports are defined as per [138].

policy update is in progress, every packet is processed either only by the old policy or only by the new one, but never a mixture of both.

Several implementations are possible based on different granularity of *locking*. A trivial solution that provides consistency could maintain a global lock on the entire network: To install a new policy, a controller grabs the global lock and then runs the two-phase update algorithm in the absence of contention. Assuming that the global lock is implemented in the starvation-free manner (*e.g.*, by a fair central server), we obtain a solution providing strongly consistent composition.

Below we describe a more intelligent algorithm that only requires a weak form of local (*per-switch*) synchronization. Our algorithm assumes that each switch exports an atomic *read-modify-write* operation $rmw(x, m, f)$, where x is a switch, $m \subseteq States$ is a "mask" matching a subset of the forwarding rules in x's state, and $f : 2^{States} \rightarrow 2^{States}$ is a function that updates the part of the state of x matching m based on the read value. However, note that we do not necessarily require the switches to implement this atomic read-modify-write operation; in a NIB-based implementation of STN, this operation may be incorporated into the NIB itself.

To illustrate by example, when a control module wants to collect statistics on all packets with `tcp_port=80` arriving to switch x, it should check if x currently maintains rules that affect packets to TCP port 80. If this is the case, *e.g.*, x forwards all packets with `src=10.0.*` to port 2 (see the example in Section 4.1), then function f stipulates that the new rules to be added to the configuration of x are:

Priority	Match	Actions
0	`tcp_port=80`	`count`
1	`src=10.0.*` $\wedge$ `tcp_port=80`	`count`; `fwd`(2)

The pseudocode of our algorithm is sketched in Algorithm 1. A controller first chooses a unique tag number, and then goes through every switch in the network. In the first phase of the algorithm, the controller tries to atomically install the new policy using the read-modify-write (confined in the *atomic* block) at every switch. If, at a given switch, the currently installed policy p cannot be composed with existing policies, all previous updates are rolled back and *nack* is returned, together with the information about conflicting policies.

In the second phase, the controller updates the ingress ports of every switch to tag the incoming traffic with the new number, computed based on the tags of previously installed policies. Note that we hide some extra complexity here: we need to make sure that all policies that may coexist in the network are represented at every affected switch. Therefore, we may need to concurrently maintain a separate tag number for each subset of previously and concurrently installed policies. This brings new interesting challenges related to consistent tag generation and garbage collection of obsolete policies (see [138] and the discussion in Section 4.5).

```
procedure apply(p)
    tag := choose a unique id for the new policy;
    foreach switch x do
        atomic{
            if x can be composed with p (wrt f) then
                foreach subset s of tagged policies at x do
                    tag' := the composition of tags of s and tag;
                    add the composed set of rules marked with tag' to x;
                end
            else
                remove all previously installed rules;
                return nack(e), where e is the reason why;
            end
        }
    end
    foreach switch x do
        foreach ingress port i of x do
            atomic{
                tag' := the composition of tags at i and tag;
                add the rule to tag all traffic to i with tag';
            }
        end
    end
    return ack;
```

Algorithm 1: Concurrent policy update algorithm: code for *apply*(p).

The correctness of the algorithm stems from the fact that when a packet that arrives at an ingress port is processed, it obtains a tag number of a policy that has been previously installed at all internal ports of the network. Moreover, from the moment when an update request for policy p is complete, every packet to arrive is processed according to a composed policy that contains p. Also, the composed policy trivially preserves the real-time order of non-overlapping requests. Thus, we implement weakly consistent policy composition. Finally, assuming atomic read-modify-write primitives, we observe every policy-update request commits, unless it witnesses a policy it cannot be sequentially composed with.

4.4 Related Work

The need for a distributed control plane has already been motivated in different contexts, *e.g.*, to reduce the latency of reactive control [73], to place local functionality closer to the data plane [71], to improve Internet routing [104] by outsourcing the route selection, and so on. The focus of our work is on how to design

a distributed control plane which supports concurrency subject to consistent policy composition.

Our work builds upon two recent threads of research: (1) *consistent network updates* [138] and (2) abstractions for horizontal and vertical *composition* [59, 117].

In [138], Reitblatt *et al.* introduce the notion of consistent network updates that guarantee that during transition from an initial configuration to a new one every packet (*per-packet* consistency) or every flow (*per-flow* consistency) is processed either by the initial configuration or by the new one, but never by a mixture of the two. In a centralized setting, they proposed a two-phase per-packet consistent update algorithm that first installs the new configuration on the internal ports and then updates the ingress ones.

Foster *et al.* [59] present the *Frenetic* language, which includes a run-time system to correctly compose policy modules to low-level rules on switches. Compared to the alternative OpenFlow interface, the language simplifies network programming as it circumvents the coupling of different tasks (like routing, access control, traffic monitoring, *etc.*). As we have shown, the modular composition concepts in [59] are a very useful abstraction also for the design of a distributed control plane.

We are not the first to explore SDN architecture as a distributed system. A study of SDN from a distributed systems perspective is given in Onix [103], a control plane platform designed to enable scalable control applications. Its contribution is to abstract away the task of network state distribution from control logic, allowing application developers to make their own trade-offs among consistency, durability, and scalability. However, Onix expects developers to provide the logic that is necessary to detect and resolve conflicts of network state due to concurrent control. In contrast, we study concurrent policy composition mechanisms that can be leveraged by any application in a general fashion. In the previous chapter, we studied the implications of the logically centralized abstraction provided by SDN and the resulting design choices inherent to the strong or eventual network view consistency. We demonstrate certain consequences of this design choice on the performance of a distributed load-balancing control application.

4.5 Discussion and future work

Our sketch of an STN implementation hid many technical implementation details under the rug. Of course, we plan to address those in future work and below we speculate how.

The support of concurrent and consistent network updates features several additional challenges which have not been addressed so far. For example, while tags in the packet headers can be used to isolate different flows from each other and ensure that only one policy is applied throughout the packet paths, the number of required tags

grows exponentially in the number of policies: given k policies, in the worst case, we may have to maintain the order of 2^k policy compositions. Adding these tags ensures consistent per-packet forwarding, but the solution is not scalable. Automatically reducing the number of used tags within the STN is however complicated. We envision some new and interesting trade-offs between the complexity of the state maintained at network components and the amount of synchronization required to ensure consistency across them.

Also the size of the Forwarding Information Base (FIB), *i.e.*, number of flow table entries that can be stored, may be a limiting factor for policy composition. While this problem is not specific to the distributed controller but general, we envision that an efficient STN middleware may be able to optimize the policy implementation by automatically aggregating multiple forwarding rules into a single one, and/or by distributing ("load-balancing") rules across multiple switches where possible.

The current definition of consistency of composition works on *per-packet* basis, but seems to be straightforward to extend it to the *per-flow* level (along the lines of [138]), as long as we know how to define flow delimiters.

There are many ways to implement the *read-modify-write* primitive giving access to the network switches. At a high level, it depends on how the rights of accessing a switch are distributed across the controllers. If there is a single controller that is allowed to modify the configuration of the switch, then we can implement the *rmw* primitive at the controller, so that it is responsible for the physical installation of the composed policy. An alternative solution is to provide a lock abstraction by the switch itself. For example, a weak form of locking can be achieved by maintaining a single *test-and-set* bit at a switch. Also, we believe that even more fine-grained locking mechanisms are possible, where, *e.g.*, instead of acquiring locks on entire switches, transactions synchronize on *ports* only. We leave the discussion of implementation details and further optimizations for future work.

5

OFRewind: Enabling Record and Replay Troubleshooting for Networks

Having investigated key SDN control plane design opportunities and challenges, we now turn our attention to leveraging the *logically centralized* control plane to enable new, powerful approaches to troubleshooting network anomalies. In spite of many efforts to the contrary, problem localization and troubleshooting in operational networks still remain largely unsolved problems today. Consider the following anecdotal evidence:

Towards the end of October 2009, the administrators of the Stanford OpenFlow network deployment began observing strange CPU usage patterns in their switches. The CPU utilization oscillated between 25% and 100% roughly every 30 minutes and led to prolonged flow setup times, which where unacceptable for many users. The network operators began debugging the problem using standard tools and data sets, including SNMP statistics, however the cause for the oscillation of the switch CPU remained inexplicable. Even an analysis of the entire control channel data could not shed light on the cause of the problem, as no observed parameter (number of: packets in, packets out, flow modifications, flow expirations, statistics requests, and statistics replies) seemed to correlate with the CPU utilization. This left the network operator puzzled regarding the cause of the problem.

This anecdote (further discussion in Section 5.3.2) hints at some of the challenges encountered when debugging problems in networks. Networks typically contain *black box* devices, e.g., commercial routers, switches, and middleboxes, that can be only coarsely configured and instrumented, via command-line or simple protocols such

as SNMP. Often, the behavior of black box components in the network cannot be understood by analytical means alone – controlled replay and experimentation is needed.

Furthermore, network operators remain stuck with a fairly simplistic arsenal of tools. Many operators record statistics via NetFlow or sFlow [131]. These tools are valuable for observing general traffic trends, but often too coarse to pinpoint the origin problems. Collecting full packet traces, e.g., by tcpdump or specialized hardware, is often unscalable due to high volume data-plane traffic. Even when there is a packet trace available, it typically only contains the traffic of a single VLAN or switch port. It is thus difficult to infer temporal or causal relationships between messages exchanged between multiple ports or devices.

Previous attempts have not significantly improved the situation. Tracing frameworks such as XTrace [134] and Netreplay [31] enhance debugging capabilities by pervasively instrumenting the entire network ecosystem, but face serious deployment hurdles due to the scale of changes involved. There are powerful tools available in the context of distributed applications that enable fully deterministic recording and replay, oriented toward end hosts [51, 102]. However, overhead for the fully-deterministic recording of a large network with high data rates can be prohibitive and the instrumentation of 'black' middleboxes and closed source software often remains out of reach.

In this chapter, we present a new approach to enable practical network recording and replay. We leverage an emerging class of programmable networks called **software defined networks**, such as those based on OpenFlow [113], Tesseract [168], or Forces [5]. These architectures *split* control-plane decision-making off from data-plane forwarding. In doing so, they enable custom programmability and central coordination of the control-plane, while allowing for commodity high-throughput, high-fanout data-plane forwarding elements.

We discuss, in Section 5.1, the design of **OFRewind**, a tool that takes advantages of these properties to significantly improve the state-of-the-art for recording and replaying network domains. **OFRewind** enables *scalable*, *temporally consistent*, *centrally controlled* network recording and *coordinated* replay of traffic in an OpenFlow controller domain. It takes advantage of the flexibility afforded by the programmable control-plane, to dynamically *select* data-plane traffic for recording. This improves data-path component scalability and enables *always-on* recording of critical, low-volume traffic, e.g., routing control messages. Indeed, a recent study has shown that the control plane traffic accounts for less than 1% of the data volume, but $95-99\%$ of the observed bugs [30]. Data-plane traffic can be *load-balanced* across multiple *data-plane recorders*. This enables recording even in environments with high data rates. Finally, thanks to the centralized perspective of the controller, **OFRewind** can record a *temporally consistent* trace of the controller domain. This facilitates investigation of the temporal and causal interdependencies of the messages exchanged between devices in the controller domain.

During replay, **OFRewind** enables the operator to select which parts of the traces are to be replayed and how they should be mapped to the replay environment. By partitioning (or *bisecting*) the trace and automatically repeating the experiment, our tool helps to narrow down and isolate the problem causing component or traffic. A concrete implementation of the tool based on OpenFlow is presented in Section 5.2 and is released as free and open source software [11].

Our work is primarily motivated by operational issues in the OpenFlow-enabled production network at Stanford University. Accordingly, we discuss several case studies where our system has proven useful, including: switch CPU inflation, broadcast storms, anomalous forwarding, NOX packet parsing errors, and other invalid controller actions (Section 5.3). We in addition present a case study in which **OFRewind** successfully pinpoints faulty behavior in the Quagga RIP software routing daemon. This indicates that **OFRewind** is not limited to locating OpenFlow-specific bugs alone, but can also be used to reproduce other network anomalies.

Our evaluation (Section 5.4) shows *(a)* that the tool scales at least as well as current OpenFlow hardware implementations, *(b)* that recording does not impose an undue performance penalty on the throughput achieved, and *(c)* that the messaging overhead for synchronization in our production network is limited to 1.13% of all data-plane traffic.

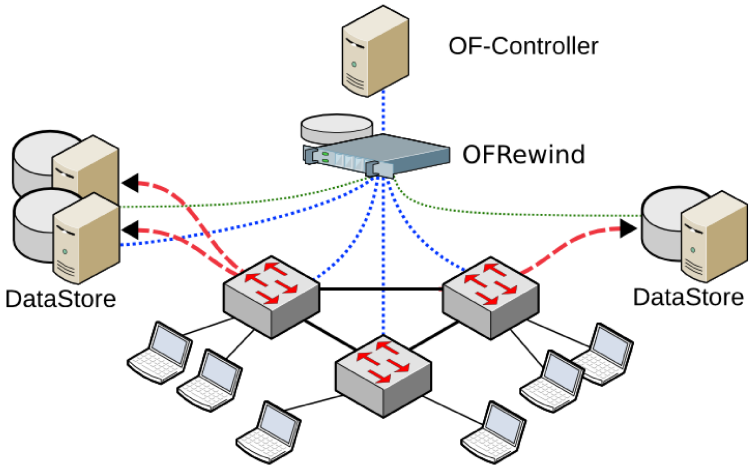
While using our tool, we have made and incorporated the following key observations:

(1) A full recording of all events in an entire production network is infeasible, due to the data volumes involved and their asynchronous nature. However, one usually needs not record all information to be able to reproduce or pinpoint a failure. It suffices to focus on *relevant subparts*, e.g., control messages or packet headers. By selectively recording critical traffic subsets, we can afford to turn *recording on by default* and thus reproduce many unforeseen problems *post facto*.

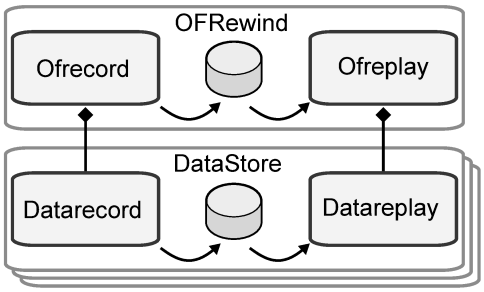
(2) Partial recordings, while missing some data necessary for fully deterministic replay, can be used to reproduce symptomatic network behavior, useful for gaining insights in many debugging situations. With careful initialization, the behavior of many network devices turns out to be *deterministic with respect to the network input*.

(3) By replaying *subsets of traffic* at a *controlled pace*, we can, in many cases, rapidly repeat experiments with different settings (parameters/code hooks) while still reproducing the error. We can, for instance, *bisect* the traffic and thus localize the sequence of messages leading to an error.

In summary, **OFRewind** is, to the best of our knowledge, the first tool which leverages the properties of software defined networking to enable practical and economically feasible recording and replay debugging of network domains. It has proven useful in a variety of practical case studies, and our evaluation shows **OFRewind**



(a) System components



(b) Software modules

Figure 5.1: Overview of **OFRewind**

does not significantly affect the scalability of OpenFlow controller domains and does not introduce undue overhead.

5.1 OFRewind System Design

In this section, we discuss the expected operational environment of **OFRewind**, its design goals, and the components and their interaction. We then focus on the need to synchronize specific system components during operation.

5.1.1 Environment / Abstractions

Our system design leverages programmable *software defined networks*, for instance, those based upon OpenFlow [113], Tesseract [168], or Forces [5], in which *standardized data-plane elements* (switches) perform fast and efficient packet forwarding,

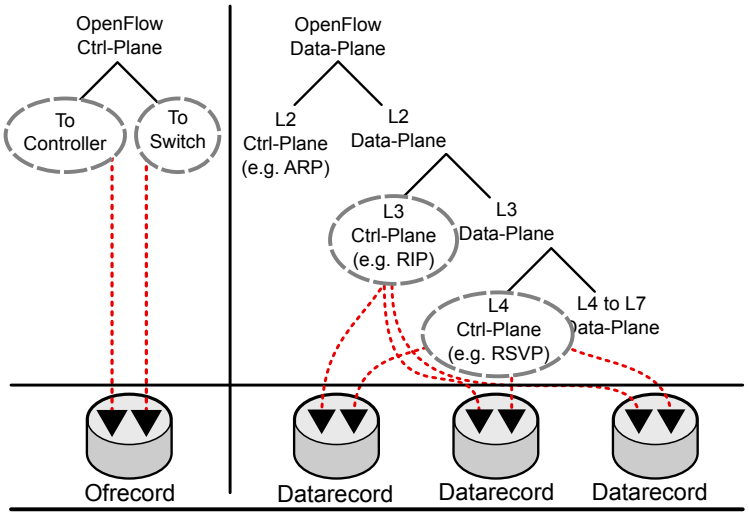


Figure 5.2: **OFRewind** Traffic strata

and the control-plane is *programmable* via an external controlling entity, known as the *controller*. Programmability is achieved through *forwarding rules* that *match* incoming traffic and associate them with *actions*. We call this layer of the network the *substrate*, and the higher-layer *superstrate* network running on top of it *guest*. We call the traffic exchanged between the switches and the controller the substrate *control-plane*. The higher-layer control-plane traffic running inside of the substrate *data-plane* (e.g., IP routing messages) is called the guest *control-plane*. The relationship between these layers and traffic strata is shown in Figure 5.2.

Even though not strictly mandated almost all software defined network deployments group several switches to be centrally managed by one controller, creating a *controller domain*. We envision one instance of **OFRewind** to run in one such controller domain. Imagine, e.g., a campus building infrastructure with 5-10 switches, 200 hosts attached on Gigabit links, a few routers, and an uplink of 10GB/s.

5.1.2 Design Goals and Non-Goals

As previously stated, recording and replay functionalities are not usually available in networks. We aim to build a tool that leverages the flexibility afforded by software defined networks to realize such a system. We do not envision **OFRewind** to do automated *root-cause analysis*. We do intend it to help *localize* problem causes. Additionally, we do not envision it to *automatically tune recording* parameters. This is left to an informed administrator who knows what scenario is being debugged.

Recording goals: We want a *scalable* system that can be used in a realistic-sized controller domain. We want to be able to record critical traffic, e.g., routing mes-

sages, in an *always-on* fashion. What is monitored should be specified in *centralized configuration*, and we want to be able to attain a *temporally consistent* view of the recorded events in the controller domain.

Replay goals: We want to be able to replay traffic in a *coordinated* fashion across the controller domain. For replaying into a different environment or topology (e.g., in a lab environment) we want to *sub-select* traffic and potentially *map* traffic to other devices. We include *time dilation* to help investigate timer issues, create stress tests, and allow "fast forwarding" to skip over irrelevant portions of the traffic. *Bisection* of the traffic between replays can assist problem localization whereby the user repeatedly partitions and sub-selects traffic to be replayed based on user-defined criteria (e.g., by message types), performs a test run, then continues the bisection based on whether a problem was reproducible.

(Absence of) determinism guarantees: As opposed to host-oriented replay debugging systems which strive for determinism guarantees, **OFRewind** does not – and cannot – provide strict determinism guarantees, as *black boxes* do not lend themselves to the necessary instrumentation. Instead, we leverage the insight that network device behavior is *largely deterministic* on control plane events (messages, ordering, timing). In some cases, when devices deliberately behave non-deterministically, protocol specific approaches must be taken.

5.1.3 OFRewind System Components

As seen in Figure 5.1(a), the main component of our system, **OFRewind**, runs as a proxy on the substrate control channel, i.e., between the switches and the original controller. It can thus intercept and modify substrate control-plane messages to control recording and replay. It delegates recording and replay of *guest* traffic to *DataStore* components that are locally attached at regular switch ports. The number of *DataStores* attached at each switch can be chosen freely, subject to the availability of ports.

Both components can be broken down further into two modules each, as depicted in Figure 5.1(b): They consist of a recording and a replay module with a shared local storage, labeled *Ofrecord* and *Ofreplay*, and *Datarecord* and *Datareplay* respectively.

Record: *Ofrecord* captures *substrate* control-plane traffic directly. When *guest* network traffic recording is desired, *Ofrecord* translates control messages to instruct the relevant switches to selectively mirror *guest* traffic to the *Datarecord* modules. **OFRewind** supports dynamic selection of the *substrate* or *guest* network traffic to record. In addition, flow-based-sampling can be used to record only a fraction of the data-plane flows.

Name	Target	Traffic type
ctrl	Ctrl	OF msgs
switch	Switch	OF msgs
datahdr	Switch	OF msgs/ data-plane traffic from headers
datafull	Switch	OF msgs/full data-plane traffic

Table 5.1: *Ofreplay* operation modes

Replay: *Ofreplay* re-injects the traces captured by *Ofrecord* into the network, and thus enables domain-wide *replay debugging*. It emulates a controller towards switches or a set of switches towards a controller, and directly replays *substrate* control-plane traffic. *Guest* traffic is replayed by the *Datareplay* modules, which are orchestrated by *Ofreplay*.

5.1.4 Ofrecord Traffic Selection

While it is, in principle, possible to collect full data recordings for every packet in the network, this introduces a substantial overhead both in terms of storage as well as in terms of performance. *Ofrecord*, however, allows selective traffic recording to reduce the overhead.

Selection: Flows can be classified and selected for recording. We refer to traffic *selection* whenever we make a conscious decision on what subset of traffic to record. Possible categories include: *substrate* control traffic, *guest* network control traffic, or *guest* data traffic, possibly sub-selected by arbitrary match expressions. We illustrate an example selection from these categories in Figure 5.2.

Sampling: If selection is unable to reduce the traffic sufficiently, one may apply either packet or flow sampling on either type of traffic as a reduction strategy.

Cut-offs: Another data reduction approach is to record only the first X bytes of each packet or flow. This often suffices to capture the critical meta-data and has been used in the context of intrusion detection [109].

5.1.5 Ofreplay Operation Modes

To support testing of the different entities involved (switches, controller, end hosts) and to enable different playback scenarios, *Ofreplay* supports several different operation modes:

ctrl: In this operation mode, replay is directed towards the controller. *Ofreplay* plays the recorded substrate control messages from the local *storage*. This mode enables debugging of a controller application on a single developer host, without

need for switches, end-hosts, or even a network. Recorded data-plane traffic is not required.

switch: This operation mode replays the recorded *substrate* control messages toward the switches, reconstructing each switch's *flow table* in real time. No controller is needed, nor is *guest* traffic replayed.

datahdr: This mode uses packet headers captured by the *Datarecord* modules to re-generate the exact flows encountered at recording time, with dummy packet payloads. This enables full testing of the switch network, independent of any end hosts.

datafull: In this mode, data traffic recorded by the *DataStores* is replayed with complete payload, allowing for selective inclusion of end host traffic into the tests.

In addition to these operation modes, *Ofreplay* enables the user to further tweak the recorded traffic to match the replay scenario. Replayed messages can be *sub-selected* based on source or destination host, port, or message type. Further, message destinations can be *re-mapped* on a per-host or per-port basis. These two complementary features allow traffic to be re-targeted toward a particular host, or restricted such that only relevant messages are replayed. They enable *Ofreplay* to play recorded traffic either toward the original sources or to alternative devices, which may run a different firmware version, have a different hardware configuration, or even be of a different vendor. These features enable **OFRewind** to be used for *regression testing*. Alternately, it can be useful to map traffic of multiple devices to a single device, to perform stress tests.

The *pace* of the replay is also adjustable within *Ofreplay*, enabling investigation of pace-dependent performance problems. Adjusting replay can also be used to "fast-forward" over portions of a trace, e.g., memory leaks in a switch, which typically develop over long time periods may be reproduced in an expedited manner.

5.1.6 Event Ordering and Synchronization

For some debugging scenarios, it is necessary to preserve the exact message order or mapping between *guest* and *substrate* flow data to be able to reproduce the problem. In concrete terms, the guest (data) traffic should not be replayed until the substrate (control) traffic (containing the corresponding substrate actions) has been replayed. Otherwise, *guest* messages might be incorrectly forwarded or simply dropped by the switch, as the necessary flow table state would be invalid or missing.

We do not assume tightly synchronized clocks or low latency communication channels between our **OFRewind** and the *DataStores* components. Accordingly, we cannot assume that synchronization between recorded *substrate* and *guest* flow traces, or order between flows recorded by different *DataStores* is guaranteed per se. Our

design does rely on the following assumptions: (1) The *substrate* control-plane channel is reliable and order-preserving. (2) The *control channel* between **OFRewind** and each individual *DataStore* is reliable and order-preserving, and has a reasonable mean latency l_c (e.g., 5 ms in a LAN setup.) (3) The *data-plane* channel from **OFRewind** to the *DataStores* via the switch is not necessarily fully, but sufficiently reliable (e.g., 99.9% of messages arrive). It is not required to be order-preserving in general, but there should be some means of explicitly guaranteeing order between two messages. We define the data-plane channel mean latency as l_d.

Record: Based on these assumptions, we define a logical clock C [106] on *Ofrecord*, incremented for each *substrate* control message as they arrive at *Ofrecord*. *Ofrecord* logs the value of C with each *substrate* control message. It also broadcasts the value of C to the *DataStores* in two kinds of synchronization markers: *time binning markers* and *flow creation markers*.

Time binning markers are sent out at regular time intervals i_t, e.g., every 100ms. They group flows into bins and thus constrain the search space for matching flows during replay and help reconstruct traffic characteristics within flows. Note that they do not impose a strict order on the flows within a time bin.

Flow creation markers are optionally sent out whenever a new flow is created. Based on the previous assumptions, they induce a total ordering on all flows whose creation markers have been successfully recorded. However, their usage limits the scalability of the system, as they must be recorded by all *DataStores*.

Replay: For synchronization during replay, *Ofreplay* assumes the role of a synchronization master, reading the value of C logged with the *substrate* messages. When a *DataStore* hits a synchronization marker while replaying, it synchronizes with *Ofreplay* before continuing. This assures that in the presence of *time binning markers*, the replay stays loosely synchronized between the markers (within an interval $I = i_t + l_d + l_c$). In the presence of *flow creation markers*, it guarantees that the order between the marked flows will be preserved.

5.1.7 Typical Operation

We envision that users of **OFRewind** run *Ofrecord* in an always-on fashion, always recording selected *substrate* control-plane traffic (e.g., OpenFlow messages) onto a ring storage. If necessary, selected *guest* traffic can also be continuously recorded on *Datarecord*. To preserve space, low-rate control-plane traffic, e.g., routing announcements, may be selected, sampling may be used, and/or the ring storage may be shrunk. When the operator (or an automated analytics tool) detects an anomaly, a replay is launched onto a separate set of hardware, or onto the production network during off-peak times. Recording settings are adapted as necessary until the anomaly can be reproduced during replay.

During replay, one typically uses some kind of debugging by elimination, either by performing binary search along the time axis or by eliminating one kind of message at a time. Hereby, it is important to choose orthogonal subsets of messages for effective problem localization.

5.2 Implementation

In this section, we describe the implementation of **OFRewind** based on OpenFlow, selected for currently being the most widely used switch programming API for *software defined networks*. OpenFlow is currently in rapid adoption by testbeds [61], university campuses [3], and commercial vendors [8].

OpenFlow realizes a *software defined network* by providing an open protocol between packet-forwarding hardware and a commodity PC (the *controller*). The protocol allows the *controller* to exercise flexible and dynamic control over the forwarding behavior of OpenFlow enabled Ethernet switches at a per-flow level. The definition of a flow can be tailored to the specific application case—OpenFlow supports an 11-tuple of packet header parts, against which incoming packets can be matched, and flows classified. These range from Layer 1 (switch ports), to Layer 2 and 3 (MAC and IP addresses), to Layer 4 (TCP and UDP ports). The set of matching rules, and the actions associated with and performed on each match are held in the switch and known as the *flow table*.

We next discuss the implementation of **OFRewind**, the synchronization among the components and discuss the benefits, limitations, and best-practices of using OpenFlow to implement our system. The implementation, which is an OpenFlow controller in itself, and based on the source code of FlowVisor [148] is available under a free and open source license at [11].

5.2.1 Software Modules

To capture both the *substrate* control traffic and *guest* network traffic we use a hybrid strategy for implementing **OFRewind**. Reconsider the example shown in Figure 5.1(a) from an OpenFlow perspective. We deploy a proxy server in the OpenFlow protocol path (labeled **OFRewind**) and attach local *DataStore* nodes to the switches. The **OFRewind** node runs the *Ofrecord* and *Ofreplay* modules, and the *DataStore* nodes run *Datarecord* and *Datareplay*, respectively. We now discuss the implementation of the four software components *Ofrecord*, *Datarecord*, *Ofreplay* and *Datareplay*.

***Ofrecord*:** *Ofrecord* intercepts all messages passing between the switches and controller and applies the selection rules. It then stores the selected OpenFlow control (*substrate*) messages to locally attached data storage. Optionally, the entire flow

table of the switch can be dumped on record startup. If recording of the *guest* network control and/or data traffic is performed, *Ofrecord* transforms the `FLOW-MOD` and `PACKET-OUT` commands sent from the controller to the switch to *duplicate* the packets of selected flows to a *DataStore* attached to a switch along flow path. Multiple *DataStores* can be attached to each switch, .e.g., for load-balancing. The order of flows on the different *DataStores* in the system is retained with the help of synchronization markers. Any match rule supported by OpenFlow can be used for packet selection. Additionally, flow-based-sampling can be used to only record a fraction of the flows.

***Datarecord*:** The *Datarecord* components located on the *DataStores* record the selected guest traffic, as well as synchronization and control metadata. They are spawned and controlled by *Ofrecord*. Their implementation is based on `tcpdump`, modified to be controlled by *Ofrecord* via a TCP socket connection. Data reduction strategies that cannot be implemented with OpenFlow rules (e.g., packet sampling, cut-offs) are executed by *Datarecord* before writing the data to disk.

***Ofreplay*:** *Ofreplay* re-injects OpenFlow control-plane messages as recorded by *Ofrecord* into the network and orchestrates the guest traffic replay by the *Datareplay* components on the *DataStores*. It supports replay towards the *controller* and *switches*, and different levels of data-plane involvement (*switch*, *datahdr*, *datafull*, see Section 5.1.5.) Optionally, a flow table dump created by *Ofrecord* can be installed into the switches prior to replay. It supports traffic *sub-selection* and *mapping* towards different hardware and *time dilation*.

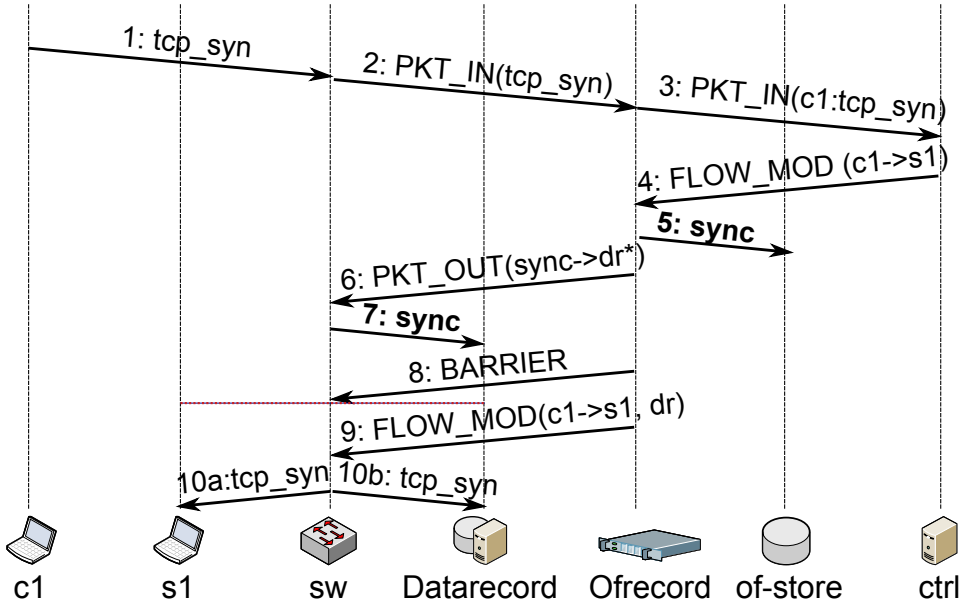
***Datareplay*:** The *Datareplay* components are responsible for re-injecting guest traffic into the network. They interact with and are controlled by *Ofreplay* for timing and synchronization. The implementation is based on `tcpreplay`. Depending on the record and replay mode, they reconstruct or synthesize missing data before replay, e.g., dummy packet payloads, when only packet headers have been recorded.

5.2.2 Synchronization

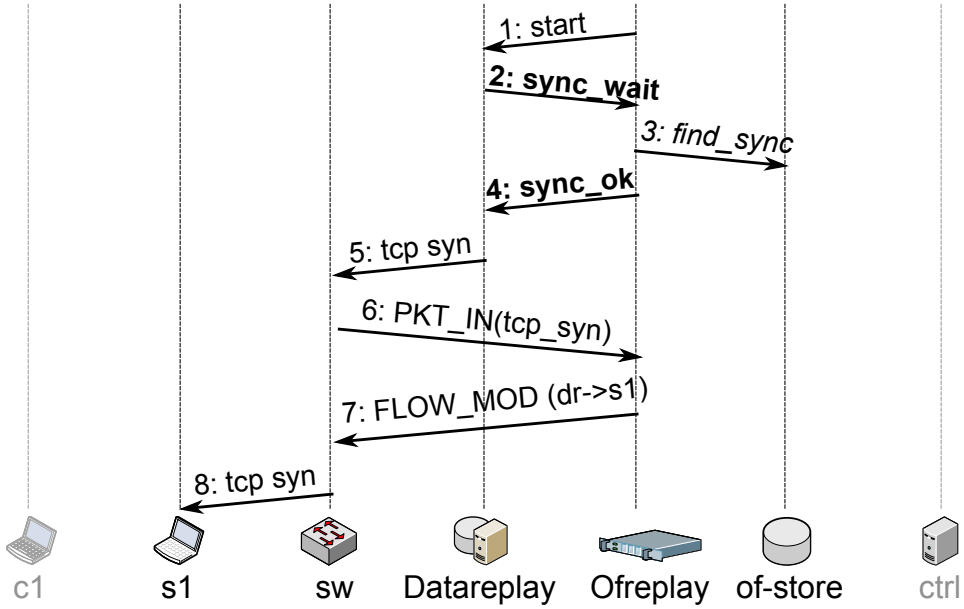
As we do not assume precise time synchronization between *Ofrecord* and the *DataStores*, the implementation uses *time binning markers* and *flow creation markers*, as discussed in Section 5.1.6. These are packets with unique ids flooded to all *DataStores* and logged by *Ofrecord*. The ordering of these markers relative to the recorded traffic is ensured by OpenFlow `BARRIER` messages[1]. We now discuss by example how the markers are used.

Record synchronization: Figure 5.3(a) illustrates the use of *flow creation markers* during recording. Consider a simple *Ofrecord* setup with two hosts *c1* and *s1*

[1] A BARRIER message ensures that all prior OpenFlow messages are processed before subsequent messages are handled. In its absence, messages may be reordered.



(a) *Ofrecord*



(b) *Ofreplay*

Figure 5.3: *DataStore* synchronization mechanism in **OFRewind**

connected to switch *sw*. The switch is controlled by an instance of *Ofrecord*, which in turn acts as a client to the network controller *ctrl*. *Ofrecord* records to the local storage *of-store*, and controls an instance of *Datarecord* running on a *DataStore*. Assume that a new TCP connection is initiated at *c1* toward *s1*, generating a *tcp syn* packet (Step 1). As no matching flow table entry exists, *sw* sends *msg1*, an OpenFlow `PACKET-IN` to *Ofrecord* (Step 2), which in turn relays it to *ctrl* (step 3). *Ctrl* may respond with *msg2*, a `FLOW-MOD` message (step 4). To enable synchronized replay and reassembly of the control and data records, *Ofrecord* now creates a flow creation marker (*sync*), containing a unique id, the current time, and the matching rule of *msg1* and *msg2*. Both *msg1* and *msg2* are then annotated with the id of *sync* and saved to *of-store* (step 5). *Ofrecord* then sends out 3 messages to *sw1*: first, a `PACKET-OUT` message containing the *flow creation marker* sent to *all DataStores* in step 6. This prompts the switch to send out *sync* to all its attached *DataStores* (step 7). The second message sent from *Ofrecord* is a `BARRIER` message (step 8), which ensures that the message from step 7 is handled before any subsequent messages. In step 9, *Ofrecord* sends a modified `FLOW-MOD` message directing the flow to both the original receiver, as well as *one DataStore* attached to the switch. This prompts the switch to output the flow both to *s1* (step 10a) and *DataStore* (step 10b).

Replay synchronization: For synchronizing replay, *Ofreplay* matches data-plane events to control plane events with the help of the flow creation markers recorded by *Ofrecord*. Consider the example in Figure 5.3(b). Based on the previous example, we replay the recording in *data-plane mode* towards the switch *sw* and host *s1*. To begin, *Ofreplay* starts *Datareplay* playback on the *DataStore* in step 1. *Datareplay* hits the flow creation marker *sync*, then sends a `sync_wait` message to the controller, and goes to sleep (step 2). *Ofrecord* continues replay operation, until it hits the corresponding flow creation marker *sync* on the *of-store* (step 3). Then, it signals *Datareplay* to continue with a `sync_ok` message (step 4). *Datareplay* outputs the packet to the switch (step 5), generating a `PACKET-IN` event (step 6). *Ofreplay* responds with the matching `FLOW-MOD` event from the OpenFlow log (step 7). This installs a matching flow rule in the switch and causes the flow to be forwarded as recorded (step 8).

5.2.3 Discussion

We now discuss the limitations imposed by OpenFlow on our implementation, and best practices for avoiding replay inaccuracies.

OpenFlow limitations: While OpenFlow provides a useful basis for implementing **OFRewind**, it does not support all the features required to realize all operation modes. OpenFlow does not currently support **sampling** of either flows or packets. Thus, *flow sampling* is performed by *Ofrecord*, and packet sampling is performed by *Datarecord*. This imposes additional load on the channel between the switches and the *DataStores* for data that is not subsequently recorded. Similarly, the OpenFlow

data-plane does not support forwarding of **partial packets**[2]. Consequently, full packets are forwarded to the *DataStore* and only their headers may be recorded. OpenFlow also does not support automatic **flow cut-offs** after a specified amount of *traffic*[3]. The cut-off can be performed in the *DataStore*. Further optimizations are possible, e.g., regularly removing flows that have surpassed the threshold.

Avoiding replay inaccuracies: To reliably reproduce failures during replay in a controlled environment, one must ensure that the environment is properly initialized. We suggest therefore, to use the flow table dump feature and, preferably, reset (whenever possible) the switches and controller state before starting the replay operation. This reduces any unforeseen interference from previously installed bad state.

When bisecting during replay, one must consider the interdependencies among message types. `FLOW-MOD` messages are for example, responsible for creating the flow table entries and their arbitrary bisection may lead to incomplete or nonsense forwarding state on the switch.

Generally speaking, replay inaccuracies can occur when: (a) the chain of causally correlated messages is recorded incompletely, (b) synchronization between causally correlated messages is insufficient, (c) timers influence system behavior, and (d) network communication is partially non-deterministic. For (a) and (b), it is necessary to adapt the recording settings to include more or better-synchronized data. For (c) a possible approach is to *reduce* the traffic being replayed via sub-selection to reduce stress on the devices and increase accuracy. We have not witnessed this problem in our practical case studies. Case (d) requires the replayed traffic to be modified. If the non-determinism stems from the transport layer (e.g., TCP random initial sequence numbers), a custom transport-layer handler in *Datareplay* can shift sequence numbers accordingly for replay. For application non-determinism (e.g., cryptographic nonces), application-specific handlers must be used.

When the failure observed in the production network does not appear during replay, we call this a **false negative** problem. When the precautions outlined above have been taken, a repeated *false negative* indicates that the failure is likely not triggered by network traffic, but other events. In a **false positive** case, a failure is observed during replay which does not stem from the same root cause. Such inaccuracies can often be avoided by careful comparison of the symptoms and automated repetition of the replay.

[2] It *does* support a cut-off for packets forwarded to the *controller*.
[3] Expiration after a specified amount of *time* is supported.

Case study	Class	OF-specific
Switch CPU Inflation	black box (switch)	no
Anomalous Forwarding	black box (switch)	yes
Invalid Port Translation	OF controller	yes
NOX Parsing Error	OF controller	yes
Faulty Route Advertisements	software router	no

Table 5.2: Overview of the case studies

5.3 Case Studies

In this section, we demonstrate the use of **OFRewind** for localizing problems in *black box network devices*, *controllers*, and *other software components*, as summarized in Table 5.2. These case studies also demonstrate the benefits of *bisecting* the control-plane traffic (5.3.2), of *mapping* replays onto different pieces of hardware (5.3.3), from a production network onto a developer machine (5.3.5), and the benefit of a *temporally consistent* recording of multiple switches (5.3.6).

5.3.1 Experimental Setup

For our case studies we use a network with switches from three vendors: **Vendor A**, **Vendor B**, **Vendor C**. Each switch has two PCs connected to it. Figure 5.4 illustrates the connectivity. All switches in the network have a control-channel to *Ofrecord*. *DataStores* running *Datarecord* and *Datareplay* are attached to the switches as necessary. We use NOX [125], unless specified otherwise, as the high level controller performing the actual routing decisions. It includes the *routing* module, which performs shortest path routing by learning the destination MAC address, and the $spanning - tree$ module, which prevents broadcast storms in the network by using Link Layer Discovery Protocol (LLDP) to identify if there is a loop in the network. All OpenFlow applications, viz. NOX, FlowVisor, *Ofreplay*, and *Ofrecord*, are run on the same server.

5.3.2 Switch CPU Inflation

Figure 5.5 shows our attempt at reproducing the CPU oscillation we reported in Section 5. As stated earlier, there is no apparent correlation between the ingress traffic and the CPU utilization. We record all control traffic in the production network, as well as the traffic entering/exiting the switch, while the CPU oscillation is happening. Figure 5.5(a) shows the original switch performance during recording. We, then, iteratively replay the corresponding control traffic over a similar switch in our isolated experimental setup. We replay the recorded data traffic to 1 port of the switch and connect hosts that send `ICMP` datagrams to the other ports. In

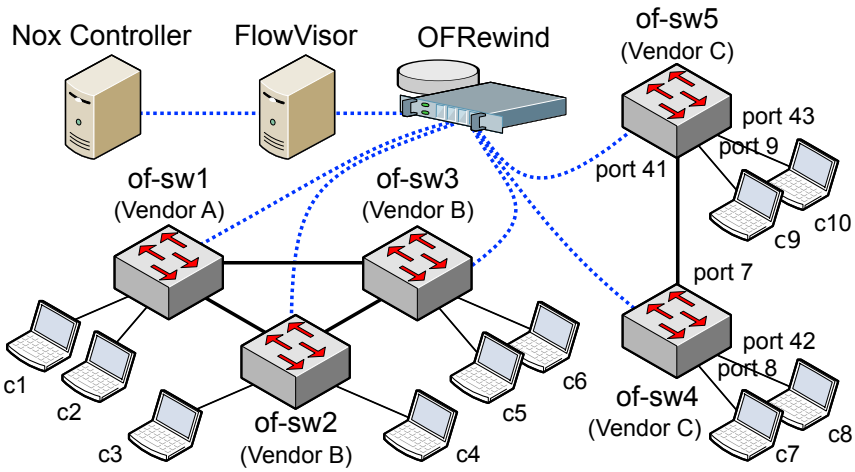


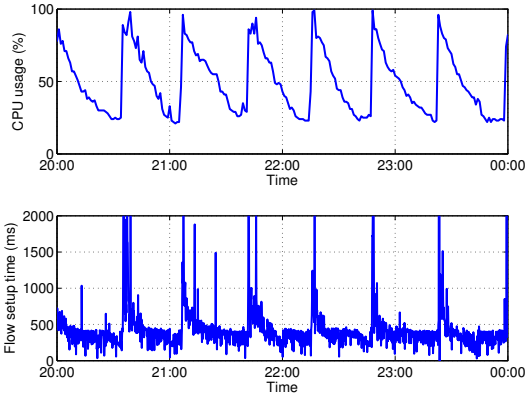
Figure 5.4: Lab environment for case studies

Counts	Match
duration=181s	in_port=8
n_packets=0	dl_type=arp
n_bytes=3968	dl_src=00:15:17:d1:fa:92
idle_timeout=60	dl_dst=ff:ff:ff:ff:ff:ff
hard_timeout=0	actions=`FLOOD`

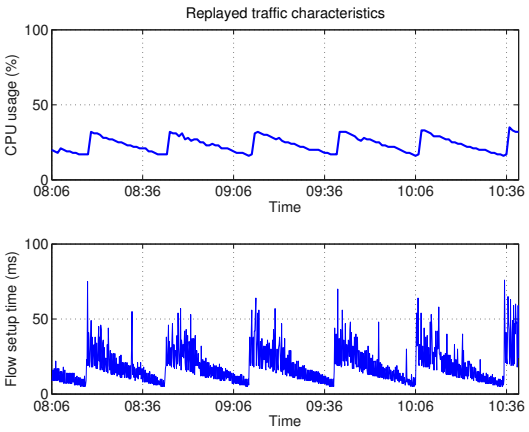
Table 5.3: **Vendor C** switch flow table entry, during replay.

each iteration, we have *Ofreplay bisect* the trace by OpenFlow message type, and check whether the symptom persists. When replaying the port and table statistic requests, we observe the behavior as shown in Figure 5.5(b). Since the data traffic is synthetically generated, the amplitude of the CPU oscillation and the flow setup time variation is different from that in the original system. Still, the sawtooth pattern is clearly visible. This successful reproduction of the symptom helps us identify the issue to be related to port and table statistics requests. Note that these messages have been causing the anomaly, even though their arrival rate (24 messages per minute) is not in any way temporally correlated with the perceived symptoms (30-minute CPU sawtooth pattern). We reported this issue to the vendor, since at this point we have no more visibility into the switch software implementation.

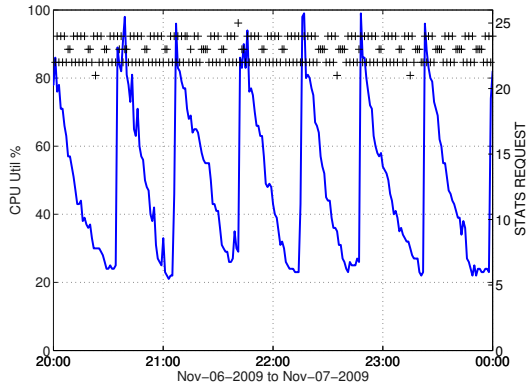
OFRewind thus, has proved useful in localizing the cause for the anomalous behavior of a black box component that would otherwise have been difficult to debug. Even though the bug in this case is related to a prototype OpenFlow device, the scenario as such (misbehaving black box component) and approach (debugging by replay and bisection of control-channel traffic) are arguably applicable to non-OpenFlow cases as well.



(a) Switch performance with original traffic



(b) Switch performance with replayed traffic



(c) `STATS-REQUEST` vs. CPU

Figure 5.5: Sawtooth CPU pattern reproduced during replay of port and table `STATS-REQUEST` messages. Figure (c) shows no observable temporal correlation to message arrivals.

5.3.3 Anomalous Forwarding

To investigate the performance of devices from a new vendor, **Vendor C**, we record the substrate and guest traffic for a set of flows, sending 10 second delayed `ping` between a pair of hosts attached to the switch from **Vendor B** (`of-sw3` in Figure 5.4). We then use the device/port mapping feature of *Ofreplay* to play back traffic from $c7$ to $c8$ over port 8 and port 42 belonging to the switch from **Vendor C**, in Figure 5.4.

Upon replay, we observe an interesting limitation of the switch from **Vendor C**. The `ping` flow stalls at the ARP resolution phase. The ARP packets transmitted from host $c7$ are received by host $c10$, but not by $c8$ nor $c9$. The flow table entry created in `of-sw4` during replay, is shown in Table 5.3, similar to that created during the original run. We conclude that the `FLOOD` action is not being properly applied by the switch from **Vendor C**.

Careful observation reveals that traffic received on a "low port" (one of the first 24 ports) to be flooded to any "high ports" (last 24 ports) and vice-versa is not flooded correctly. Effectively, the flood is restricted within a 24 port group within the switch (lower or higher). This fact has been affirmed by the vendor, confirming the successful debugging.

We additionally perform the replay after adding static ARP entries to the host $c7$. In this case, we observe that flow setup time for the subsequent unicast `ping` traffic on **Vendor C** is consistently higher than that observed for **Vendor B** and **Vendor A** switches. This indicates that **OFRewind** has further potential in profiling switches and controllers.

5.3.4 Invalid Port Translation

In this case study, we operate *Ofreplay* in the *ctrl* mode in order to debug a controller issue. The controller we focus on is the publicly available FlowVisor [148].

FlowVisor (FV) is a special purpose OpenFlow controller that acts as a proxy between multiple OpenFlow switches and controllers (*guests*), and thus assumes the role of a hypervisor for the OpenFlow control-plane (see Figure 5.4). To this end, the overall flow space is partitioned by FV into distinct classes, e.g., based on IP addresses and ports, and each guest is given control of a subset. Messages between switches and controllers are then filtered and translated accordingly.

We investigate an issue in where the switch from **Vendor C** works fine with the NOX controller, but not through the FV. We record the OpenFlow control-plane traffic from the switch to FV in our production setup, as seen on the left side of Figure 5.6. We then replay the trace on a developer system, running *Ofreplay*, FV

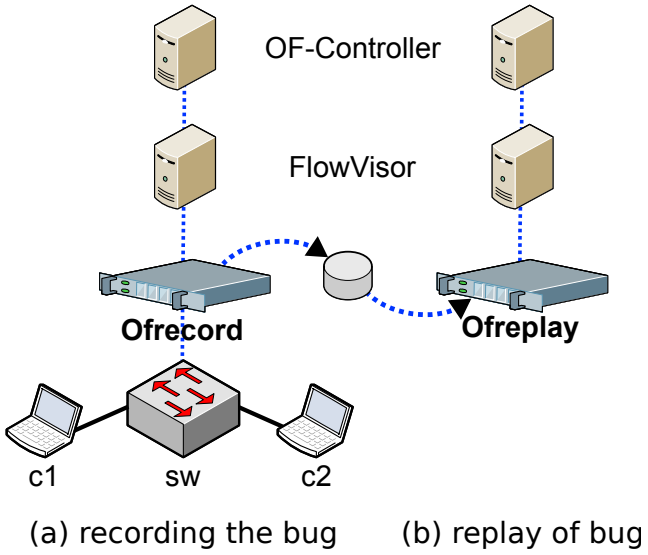


Figure 5.6: Debugging of FlowVisor bug #68

and the upstream controller on a single host for debugging. *Ofreplay* thus assumes the role of the switch.

Through repeated automated replay of the trace on the development host, we track down the source of the problem: It is triggered by a switch announcing a non-contiguous range of port numbers (e.g., 1, 3, 5). When FV translates a `FLOOD` action sent from the upstream controller to such a switch, it incorrectly expands the port range to a contiguous range, including ports that are not announced by the switch (e.g., 1, **2**, 3, **4**, 5). The switch then drops the invalid action.

Here, **OFRewind** proves useful in localizing the root cause for the failure. Replaying the problem in the development environment enables much faster turnaround times, and thus reduces debugging time. Moreover, it can be used to verify the software patch that fixes the defect.

5.3.5 NOX `PACKET-IN` Parsing Error

We now investigate a problem, reported on the NOX [125] development mailing list, where the NOX controller consistently drops the ARP reply packet from a specific host. The controller is running the `pyswitch` module.

The bug reporter provides a `tcpdump` of the traffic between their switch and the controller. We verify the existence of the bug by replaying the control traffic to our instance of the NOX. We then gradually increase the debug output from NOX as we play back the recorded OpenFlow messages to NOX.

Repeating this processes reveals the root cause of the problem: NOX deems the destination MAC address `00:26:55:da:3a:40` to be invalid. This is because the MAC address contains the byte `0x3a`, which happens to be the binary equivalent of the character ':' in ASCII. This "fake" ASCII character causes the MAC address parser to interpret the MAC address as ASCII, leading to a parsing error and the dropped packet. Here, *Ofreplay* provides the necessary debugging context to faithfully reproduce a bug encountered in a different deployment, and leads us to the erroneous line of code.

5.3.6 Faulty Routing Advertisements

In a departure from OpenFlow network troubleshooting, we examine how **OFRewind** can be used to troubleshoot more general, event-driven network problems. We consider the common problem of a network suffering from a mis-configured or faulty router. In this case, we demonstrate how **OFRewind** can be advantageously used to identify the faulty component.

We apply **OFRewind** to troubleshoot a documented bug (Quagga Bugzilla #235) detected in a version of the RIPv1 implementation of the *Quagga* [133] software routing daemon. In the network topology given by Figure 5.7, a network operator notices that shortly after upgrading *Quagga* on software router B, router C subsequently loses connectivity to Network 1. As routing control-plane messages are a good example of low-volume guest control-plane traffic, they can be recorded by *Ofrecord* always-on or, alternatively, as a precautionary measure during upgrades. Enabling *flow creation sync markers* for the low-volume routing control-plane messages ensures the global ordering is preserved.

The observation that router C loses its route to Network 1 while router B maintains its route, keys the operator to record traffic arriving at and departing from B. An analysis of the *Ofrecord* flow summaries reveals that although RIPv1 advertisements arrive at B from A, no advertisements leave B toward C. Host-based debugging of the RIPd process can then be used on router B in conjunction with *Ofreplay* to replay the trigger sequence and inspect the RIPd execution state. This reveals the root cause of the bug – routes toward Network 1 are not announced by router B due to this (0.99.9) version's handling of classful vs. CIDR IP network advertisements – an issue inherent to RIPv1 on classless networks.

5.3.7 Discussion

Without making any claims regarding the representativeness of the workload or switch behavior, in this limited space, we highlight in these case studies, the principle power and flexibility of **OFRewind**. We observe that **OFRewind** is capable of replaying subpopulations of control or data traffic, over a select network topology

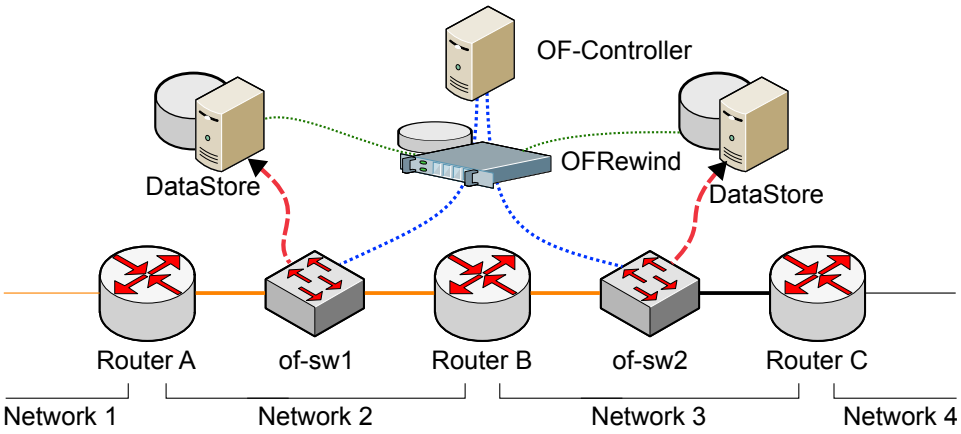


Figure 5.7: Quagga RIPv1 bug #235

(switches and ports) or to select controllers, in a sandbox or production environment.

We further note that **OFRewind** has potential in switch (or controller) benchmarking. By creating a sandbox for experimentation that can be exported to a standard replay format, a network operator can concretely specify the desired behavior to switch (or controller) design engineers. The sandbox can then be run within the premises of the switch (or controller software) vendor on a completely new set of devices and ports. On receiving the device (or software), the network operator can conduct further benchmarking to compare performance of different solutions in a fair manner.

Comparison with traditional recording Based on the case presented in the last section, we compare the effort of recording and instrumenting the network with and without **OFRewind**. Note that while the specific traffic responsible for the failure is small (RIP control plane messages), the total traffic volume on the involved links may be substantial. To attain a synchronized recording of this setup without **OFRewind**, and in the absence of host-based instrumentation, one has to (1) deploy monitoring servers that can handle the *entire* traffic on each link of interest, e.g., [115], and redeploy as link interests change. Then, one must either (2a) reconfigure both switches to enable span ports (often limited to 2 on mid-range hardware) or (2b) introduce a tap into the physical wiring of the networks. Manually changing switch configurations runs a risk of operator error and introducing a tap induces downtime and is considered even riskier. (3) Additionally, the monitoring nodes may have to be synced to microsecond level to keep the flows globally ordered, requiring dedicated, expensive hardware. With **OFRewind**, one requires only a few commodity PCs acting as *DataStores*, and a single, central configuration change to record a *consistent* traffic trace.

of-simple	reference controller emulating a learning switch
nox-pyswitch	NOX controller running Python `pyswitch` module
nox-switch	NOX controller running C-language `switch` module
flowvisor	Flowvisor controller, running a simple allow-all policy for a single guest controller
of-record	*Ofrecord* with substrate-mode recording
ofrecord-data	*Ofrecord* with guest-mode recording, with one data port and sync beacons and barriers enabled

Table 5.4: Notation of controllers used in evaluation

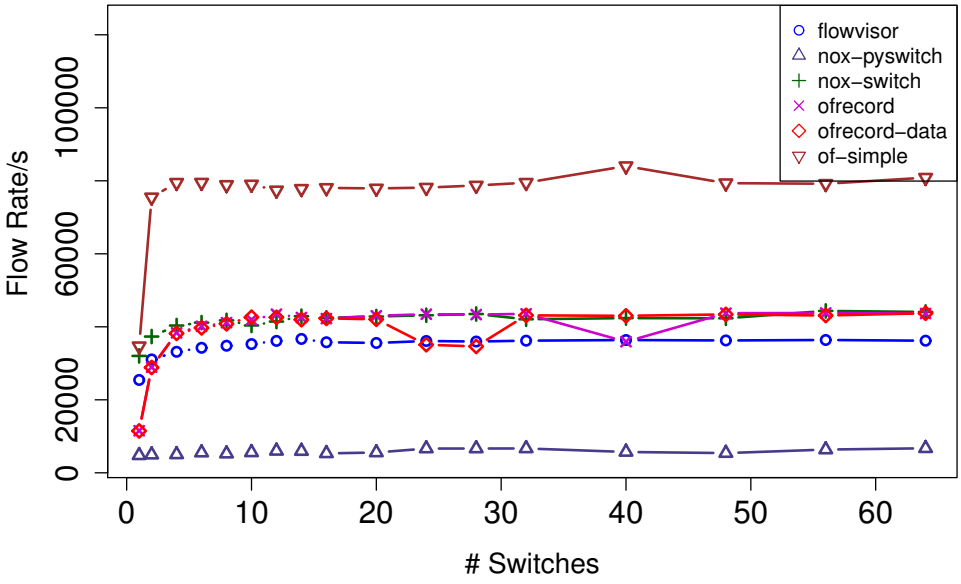


Figure 5.8: # Switches vs. median flow rate throughputs for different controllers using cbench.

5.4 Evaluation

When deploying **OFRewind** in a live production environment, we need to pay attention to its scalability, overhead and load on the switches. This section quantifies the general impact of deploying *Ofrecord* in a production network, and analyzes the replay accuracy of *Ofreplay* at higher flow rates.

5.4.1 Ofrecord Controller Performance

A key requirement for practical deployment of *Ofrecord* in a production environment is recording performance. It must record fast enough to prevent a performance bottleneck in the controller chain.

Using `cbench` [45], we compare the controller performance exhibited by several well known controllers, listed in Table 5.4. *Of-simple* and NOX are stand-alone controllers, while *flowvisor* and *ofrecord* act as a proxy to other controllers. *Ofrecord*

is run twice: in substrate mode (*ofrecord*), recording the OpenFlow substrate traffic, and in guest-mode (*ofrecord-data*), additionally performing OpenFlow message translations and outputting sync marker messages. Note that no actual guest traffic is involved in this experiment.

The experiment is performed on a single commodity server (Intel Xeon L5420, 2.5 GHz, 8 cores, 16 GB RAM, 4xSeagate SAS HDDs in a RAID 0 setup). We simulate, using `Cbench`, 1-64 switches connecting to the controller under test, and send back-to-back `PACKET-IN` messages to measure the maximum flow rate the controller can handle. `Cbench` reports the current flow rate once per second. We let each experiment run for 50 seconds, and remove the first and last 4 results for warmup and cool-down.

Figure 5.8 presents the results. We first compare the flow rates of the stand-alone controllers. *Nox-pyswitch* exhibits a median flow rate of *5,629* flows/s over all switches, *nox-switch* reports *42,233* flows/s, and *of-simple 78,908* flows/s. Consequently, we choose *of-simple* as the client controller for the proxy controllers. We next compare *flowvisor* and *ofrecord*. *Flowvisor* exhibits a median flow throughput of *35,595* flows/s. *Ofrecord* reports *42,380* flows/s, and *ofrecord-data* reports *41,743*. There is a slight variation in the performance of *ofrecord*, introduced by the I/O overhead. The minimum observed flow rates are *28,737* and *22,248*. We note that all controllers exhibit worse maximum throughput when only connected to a single switch, but show similar performance for $2 - 64$ switches.

We conclude that *Ofrecord*, while outperformed by *of-simple* in control-plane performance, is unlikely to create a bottleneck in a typical OpenFlow network, which often includes a FlowVisor instance and guest domain controllers running more complex policies on top of NOX. Note that all controllers except *nox-pyswitch* perform an order of magnitude better than the maximum flow rates supported by current prototype OpenFlow hardware implementations (max. *1,000* flows/s).

5.4.2 Switch Performance During Record

When *Ofrecord* runs in *guest mode*, switches must handle an increase in OpenFlow control-plane messages due to the sync markers. Additionally, `FLOW-MOD` and `PACKET-OUT` messages contain additional actions for mirroring data to the *DataStores*. This may influence the *flow arrival rate* that can be handled by a switch.

To investigate the impact of *Ofrecord* deployment on switch behavior, we use a test setup with two 8-core servers with 8 interfaces each, wired to two prototype OpenFlow hardware switches from **Vendor A** and **Vendor B**. We measure the supported flow arrival rates by generating minimum sized UDP packets with increasing destination port numbers in regular time intervals. Each packet thus creates a new flow entry. We record and count the packets at the sending and the receiving interfaces. Each run lasts for 60 seconds, then the UDP packet generation rate is increased.

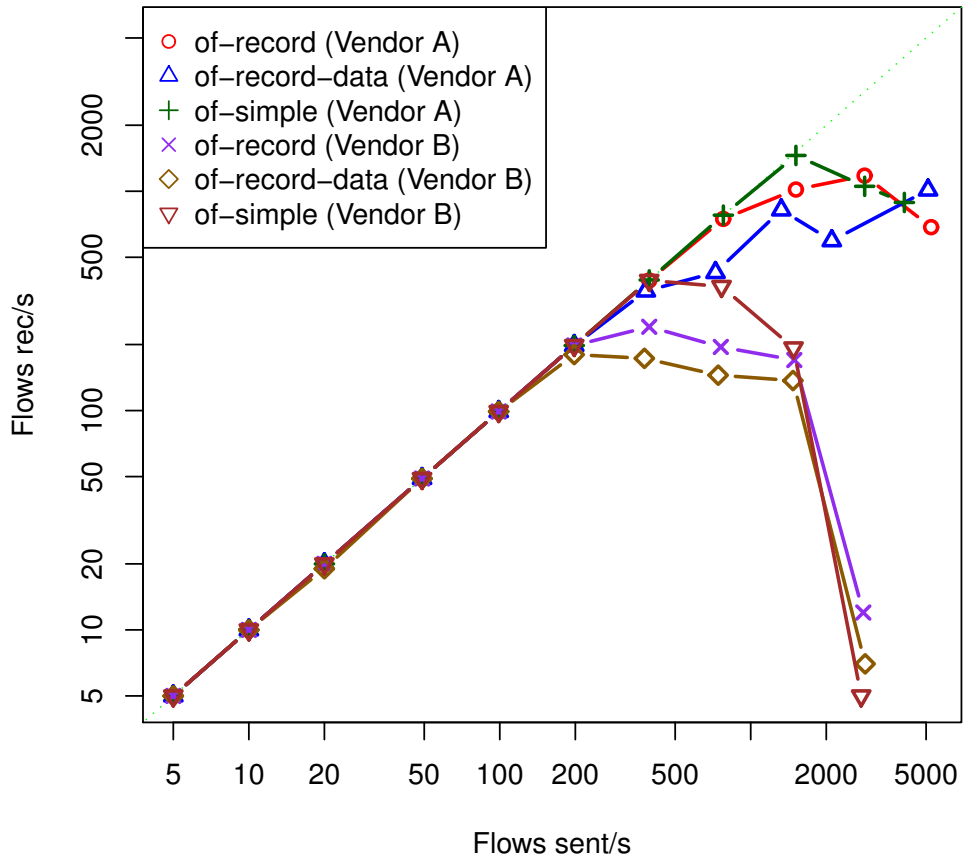


Figure 5.9: Mean rate of flows sent vs. successfully received with controllers *ofrecord*, *ofrecord-data*, and *of-simple* and switches from **Vendor A** and **B**.

Figure 5.9 presents the flow rates supported by the switches when controlled by *ofrecord*, *ofrecord-data*, and *of-simple*. We observe that all combinations of controllers and switches handle flow arrival rates of at least 200 flows/s. For higher flow rates, the **Vendor B** switch is CPU limited and the additional messages created by *Ofrecord* result in reduced flow rates (*of-record*: 247 flows/s, *of-record-data*: 187) when compared to *of-simple* (393 flows/s). **Vendor A** switch does not drop flows up to an ingress rate of 400 flows/s. However, it peaks at 872 flows/s for *ofrecord-data*, 972 flows/s for *ofrecord* and 996 flows/s for *of-simple*. This indicates that introducing *Ofrecord* imposes an acceptable performance penalty on the switches.

5.4.3 DataStore Scalability

Next we analyze the scalability of the *DataStores*. Note that *Ofrecord* is not limited to using a single *DataStore*. Indeed, the aggregate data-plane traffic (T_s bytes in c_F flows) can be distributed onto as many *DataStores* as necessary, subject to the number of available switch ports. We denote the number of *DataStores* with n and

enumerate each *DataStore* D_i subject to $0 \leq i < n$. The traffic volume assigned to each *DataStore* is T_i, such that $T_s = \sum T_i$. The flow count on each *DataStore* is c_i.

The main overhead when using *Ofrecord* is caused by the *sync markers* that are flooded to *all DataStores* at the same time. Thus, their number limits the scalability of the system. *Flow-sync markers* are minimum-sized Ethernet frames that add constant overhead ($\theta = 64B$) per new flow. Accordingly, the absolute overhead for each *DataStore* is: $\Theta_i = \theta \cdot c_i$. The absolute overhead for the entire system is $\Theta = \sum \Theta_i = \theta \cdot c_F$, the relative overhead is: $\Theta_{rel} = \frac{\Theta}{T_s}$.

In the Stanford production network, of four switches with one *DataStore* each, a 9 hour day period on a workday in July 2010 generated $c_F = 3,891,899$ OpenFlow messages that required synchronization. During that period, we observed 87.977 GB of data-plane traffic. Thus, the overall relative overhead is $\Theta_{rel} = 1.13\%$, small enough to not impact the capacity, and allow scaling up the deployment to a larger number of *DataStores*.

5.4.4 End-to-End Reliability And Timing

We now investigate the end-to-end reliability and timing precision of **OFRewind** by combining *Ofrecord* and *Ofreplay*. We use minimum size flows consisting of single UDP packets sent out at a uniform rate to obtain a worst-case bound. We vary the flow rate to investigate scalability. For each run, we first record the traffic with *Ofrecord* in guest-mode with flow sync markers enabled. Then, we play back the trace and analyze the end-to-end drop rate and timing accuracy. We use a two-server setup connected by a single switch of **Vendor B**. Table 5.5 summarizes the results. Flow rates up to 200 Flows/s are handled without drops. Due to the flow sync markers, no packet reorderings occur and all flows are replayed in the correct order. The exact inter-flow timings vary though, upwards from 50 Flows/s.

To investigate the timing accuracy further, we analyze the relative deviation from the expected inter-flow delay. Figure 5.10 presents the deviations experienced by the flows during the different phases of the experiment. Note that while there are certainly outliers for which the timing is far off, the *median* inter-flow delay remains close to the optimum for up to 100 Flows/s. Higher rates show room for improvement.

5.4.5 Scaling Further

We now discuss from a theoretical standpoint the limits of scalability intrinsic to the design of **OFRewind** when scaling beyond currently available production networks or testbeds. As with other OpenFlow-based systems, the performance of **OFRewind** is limited by the switch flow table size and the switch performance when

Rate (Flows/s)	Drop %	sd(timing, in ms)
5	0 %	4.5
10	0 %	15.6
20	0 %	21.1
50	0 %	23.4
100	0 %	10.9
200	0 %	13.9
400	19%	15.8
800	41 %	21.5

Table 5.5: **OFRewind**—end-to-end measurements with uniformly spaced flows consisting of 1 UDP packet

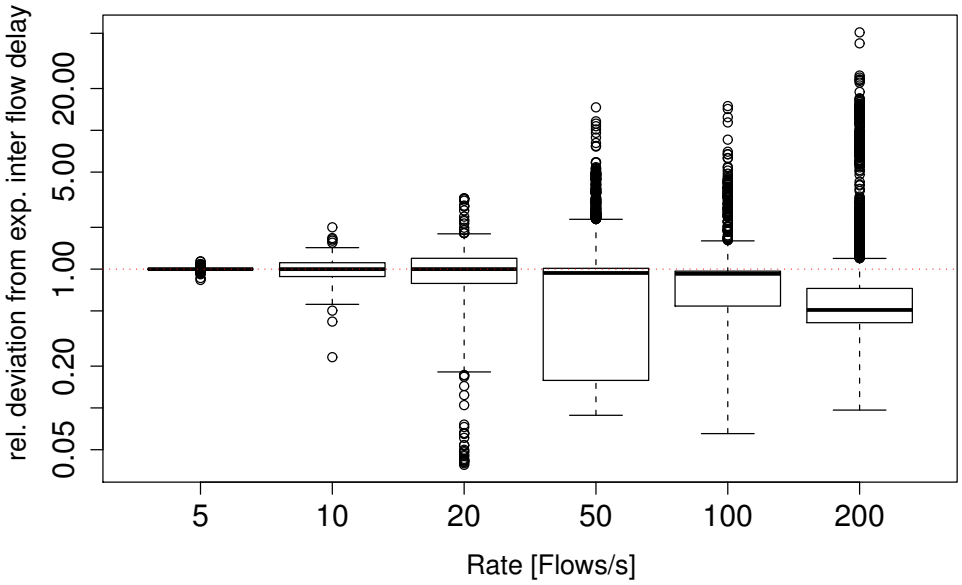


Figure 5.10: End-to-end flow time accuracy as a boxplot of the relative deviation from expected inter-flow delay.

updating and querying the flow table. We observe these to be the most common performance bottlenecks in OpenFlow setups today. Controller domain scalability is limited by the capacity of the link that carries the OpenFlow control channel, and the network and CPU performance of the controller. Specific to **OFRewind**, the control-plane components require sufficient I/O performance to record the selected OpenFlow messages – not a problem in typical developments. When recording data-plane network traffic, *DataStore* network and storage I/O capacity must be sufficient to handle the aggregate throughput of the selected flows. As load-balancing is performed over *DataStores* at flow granularity, **OFRewind** cannot fully record individual flows that surpass the network or storage I/O capacity of a single *DataStore*. When *flow creation markers* are used for synchronization, the overhead grows linearly with the number of *DataStores* and the number of flows. Thus, when the average flow size in a network is small, and synchronization is required, this may limit the scalability of a controller domain. For scaling further, **OFRewind** may in the future be extended to a *distributed controller domain*. While a quantitative evaluation is left for future study, we note that the lock-step approach taken to coordinate the replay of multiple instances of *Datarecord* and *Datareplay* (see Section 5.1.6) can be extended to synchronize multiple instances of **OFRewind** running as proxies to instances of a distributed controller. The same trade-offs between accuracy and performance apply here as well.

5.5 Related Work

Our work builds on a wealth of related work in the areas of recording and summarizing network traffic, replay debugging based on networks and on host primitives, automated problem diagnosis, pervasive tracing, and testing large-scale systems.

Recording/summarizing network traffic: Apart from the classical tool *tcpdump* [14], different approaches have been suggested in the literature to record high-volume network traffic, by performance optimization [32, 53], by recording abstractions of the network traffic [49, 54, 112], or omitting parts of the traffic [62, 109]. Our selection strategies borrow many of their ideas, more can be incorporated for improved performance. Note that these systems do not target network replay, and that all integrated control- and data-plane monitoring systems face scalability challenges when monitoring high-throughput links as the monitor has to consider all data-plane traffic, even if only a subset is to be recorded. Similar to our approach of recording in a software defined network, *OpenSafe* [34] leverages OpenFlow for flexible network monitoring but does not target replay or provide temporal consistency among multiple monitors. Complementary to our work, *OpenTM* [157] uses OpenFlow statistics to estimate the traffic matrix in a controller domain. *MeasuRouting* [136] enables flexible and optimized placement of traffic monitors with the help of OpenFlow, and could facilitate non-local *DataStores* in **OFRewind**.

Network replay debugging: *Tcpdump* and *tcpreplay* [15] are the closest siblings to our work that target network replay debugging. In fact, **OFRewind** uses these tools internally for data-plane recording and replay, but significantly adds to their scope, scalability, and coherence: It records from a controller domain instead of a single network interface, can *select* traffic on the control-plane and *load-balance* multiple *DataStores* for scalability, and can record a *temporally consistent* trace of the controller domain.

Replay debugging based on host primitives: Complementary to our network based replay, there exists a wealth of approaches that enable replay debugging for distributed systems on end-hosts [28, 75, 118]. DCR [29], a recent approach, emphasizes the importance of the control-plane for debugging. These provide fully deterministic replay capabilities important for debugging complex end-host systems. They typically cannot be used for *black box* network components.

Automated problem diagnosis: A deployment of **OFRewind** can be complemented by a system that focuses on automated problem diagnosis. Sherlock diagnoses network problems based on passive monitoring [127], and other systems infer causality based on collected message traces [24, 141]. They target the debugging and profiling of individual applications while our purpose is to support debugging of networks.

Pervasive tracing: Some proposals integrate improved in-band diagnosis and tracing support directly into the Internet, e.g., by pervasively adding a trace ID to correlated requests [134] or by marking and remembering recently seen packets throughout the Internet [31]. We focus on the more controllable environment of a single administrative domain, providing replay support directly in the substrate, with no changes required to the network.

Testing large-scale networks: Many approaches experience scalability issues when dealing with large networks. The authors of [66] suggest to scale down large networks and map them to smaller virtualized testbeds, combining *time dilation* [67] and disk I/O simulation to enable accurate behavior. This idea may aid scaling replay testbeds for **OFRewind**.

5.6 Summary

Our work presented in this chapter addresses an important void in debugging operational networks – scalable, economically feasible recording and replay capabilities. We present the design, implementation, and usage of **OFRewind**, a system capable of recording and replaying network events, motivated by our experiences troubleshooting network device and control-plane anomalies. **OFRewind** provides control over the topology (choice of devices and their ports), timeline, and selection of traffic to be collected and then replayed in a particular debugging run. Using simple case studies, we highlight the potential of **OFRewind** for not only reproducing

operational problems encountered in a production deployment but also localizing the network events that trigger the error. According to our evaluation, the framework is lightweight enough to be enabled per default in production networks.

Some challenges associated with network replay are still under investigation, including *improved timing accuracy*, *multi-instance synchronization*, and *online replay*. **OFRewind** can preserve flow order, and its timing is accurate enough for many use cases. However, further improvements would widen its applicability. Furthermore, synchronization among multiple *Ofrecord* and *Ofreplay* instances is desirable, but nontrivial, and might require hardware support for accurate time-stamping [115].

In a possible extension of this work, *Ofrecord* and *Ofreplay* are combined to form an *online replay* mode. Recorded messages are directly replayed upon arrival, e.g., to a different set of hardware or to a different *substrate* slice. This allows for online investigation and troubleshooting of failures in the sense of a *Mirror VNet* [167].

Our next steps involve gaining further experience with more complex use cases. We plan to collect and maintain a standard set of traces that serve as input for automated regression tests, as well as benchmarks, for testing new network components. Thus, we expect **OFRewind** to play a major role in helping ongoing OpenFlow deployment projects[4] resolve production problems.

[4]There are ongoing production deployments of OpenFlow-enabled networks in Asia, Europe, as well as the US.

6

Panopticon: Incremental deployment for SDN

6.1 Introduction

Finally, we turn our attention to the problem of facilitating the adoption of the software-defined network programming interface into existing network deployments. With the exception of a few notable deployments in the wild, e.g., Google's B4 [94], SDN remains largely an experimental technology for most organizations. As the transition of existing networks to SDN will not be instantaneous, we consider hybrid networks that combine SDN and legacy networking devices an important avenue of research; yet research focusing on these environments has so far been modest. Hybrid networks posses practical importance, are likely to be a problem that will span years, and present a host of interesting challenges, including the need for radically different networking paradigms to not only co-exist, but also offer clear, immediate benefits.

An appealing approach to hybrid networking can be achieved by introducing SDN devices into existing networks and abstracting the legacy network devices away as "expensive wires" to expose a programmatic "logical SDN" interface — conceptually, a representation of the network limited to just the SDN devices. Motivated by the need to better understand the potential and limits for the logical SDN abstraction for hybrid networks, in this work, we showcase the power and utility of the logical SDN by reasoning through and implementing use-case control applications built on this abstraction. Realizing this abstraction comes at the cost however, of diverting traffic from legacy switches through SDN devices. We thus explore the impact of

the logical SDN on the network traffic flow performance through experiments in a high-performance emulation environment.

While commercial SDN deployment started within data-centers [17] and the WAN [94], the roots of today's SDN arguably go back to the policy management needs of enterprise networks [42, 43]. In this paper, we focus on mid to large enterprise networks, *i.e.*, those serving hundreds to thousands of users, whose infrastructure is physically located at a locally-confined site. We choose this environment due to its complexity as well as the practical benefits that SDN network orchestration promises.

Enterprises stand to *benefit from SDN on many different levels*, including: (*i*) network policy can be declared over high-level names and enforced dynamically at fine levels of granularity [42, 59, 116], (*ii*) policy can dictate the paths over which traffic is directed, facilitating middlebox enforcement [132] and enabling greater network visibility, (*iii*) policy properties can be verified for correctness [98, 99], and (*iv*) policy changes can be accomplished with strong consistency properties, eliminating the chances of transient policy violations [138].

Existing enterprises that wish to leverage SDN however, face the problem of how to deploy it. SDN is not a "drop-in" replacement for the existing network: SDN redefines the traditional, device-centric management interface and requires the presence of programmable switches in the data plane. Consequently, the migration to SDN creates new opportunities as well as notable challenges:

Realizing the benefits. In the enterprise, the benefits of SDN should be realized as of the first deployed switch. Consider the example of Google's software-defined WAN [94], which required years to fully deploy, only to achieve benefits after a complete overhaul of their switching hardware. For enterprises, it is undesirable, and we argue, unnecessary to completely overhaul the network infrastructure before realizing benefits from SDN. An earlier return on investment makes SDN more appealing for adoption.

Eliminating disruption while building confidence. Network operators must be able to incrementally deploy SDN technology in order to build confidence in its reliability and familiarity with its operation. Without such confidence, it is risky and undesirable to replace all production control protocols with an SDN control plane as a single "flag-day" event, even if existing deployed switches already support SDN programmability. To increase its chances for successful adoption, any network control technology, including SDN, should allow for a small initial investment in a deployment that can be gradually widened to encompass more and more of the network infrastructure and traffic.

Respecting budget and constraints. Rather than a green field, network upgrade starts with the existing deployment and is typically a staged process—budgets are constrained, and only a part of the network can be upgraded at a time.

To address these challenges, we present **Panopticon**, a novel architecture for realizing an SDN control plane in a network that combines legacy switches and routers with SDN switches that can be incrementally deployed. We call such networks *transitional networks*. Panopticon abstracts the transitional network into a logical SDN, extending SDN capabilities potentially over the entire network. As an abstraction layer, Panopticon is responsible for hiding the legacy devices and acting as a "network hypervisor" that maps the logical SDN abstraction to the underlying hardware. In doing so, Panopticon overcomes key limitations of current approaches for transitional networks, which we now briefly review.

6.1.1 Current Transitional Networks

We begin with the "dual-stack" approach to transitional or "hybrid" SDN, shown in Figure 6.1a, where the flow-space is partitioned into several disjoint slices and traffic is assigned to either SDN or legacy processing [113]. To guarantee that an SDN policy applies to any arbitrary traffic source or destination in the network, the source or destination must reside at an SDN switch. Traffic within a flow-space not handled by SDN forwarding and traffic that never traverses an SDN switch may evade policy enforcement, making a single SDN policy difficult to realize over the entire network.

In summary, this mode's prime limitation is that it does not rigorously address how to realize the SDN control plane in a partial SDN deployment scenario, nor how to operate the resulting mixture of legacy and SDN devices as an SDN. It thus requires a contiguous deployment of hybrid programmable switches to ensure SDN policy compliance when arbitrary sources and destinations must be policy-enforced.

The second approach (Figure 6.1b) involves deploying SDN at the network access edge [44]. This mode has the benefit of enabling full control over the access policy and the introduction of new network functionality at the edge, *e.g.*, network virtualization [17]. Unlike a data-center environment where the network edge may terminate at the VM hypervisor, the enterprise network edge terminates at an access switch. At the edge of an enterprise network, to introduce new functionalities not accommodated by existing hardware involves replacing thousands of access switches. This mode of SDN deployment also limits the ability to apply policy to forwarding decisions within the network core (*e.g.*, load balancing, waypoint routing).

6.1.2 Panopticon

Panopticon realizes an SDN control plane for incrementally deployable software-defined networks. Our main insight is that *the benefits of SDN to enterprise networks can be realized for every source-destination path that includes at least one SDN switch.* Thus, we do not mandate a full SDN switch deployment—a small subset of

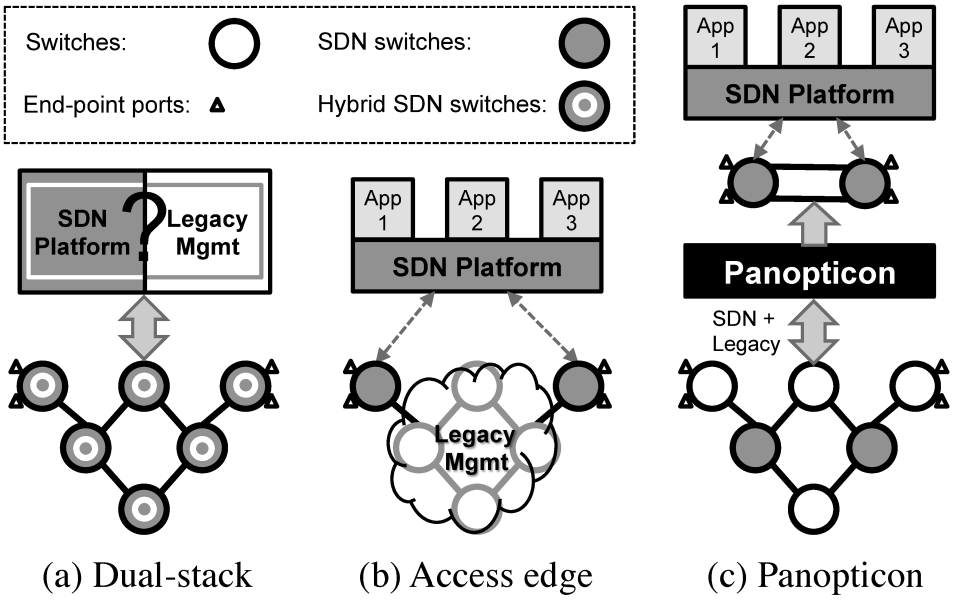


Figure 6.1: Current transitional network approaches vs. Panopticon: (*a*) Dual-stack ignores legacy and SDN integration. (*b*) Full edge SDN deployment enables end-to-end control. (*c*) Panopticon partially-deployed SDN yields an interface that acts like a full SDN deployment.

all switches may suffice. Conceptually, a single SDN switch traversed by each path is sufficient to enforce end-to-end network policy (*e.g.*, access control). Moreover, traffic which traverses two or more SDN switches may be controlled at finer levels of granularity enabling further customized forwarding (*e.g.*, traffic load-balancing).

Based on this insight, we devise a mechanism called the Solitary Confinement Tree (SCT), which uses VLANs to ensure that traffic destined to operator-selected switchports on legacy devices passes through at least one SDN switch. Combining mechanisms readily available in legacy switches, SCTs correspond to a spanning tree connecting each of these switchports to SDN switches, overcoming VLAN scalability limitations.

Just as many enterprise networks regularly divert traffic to traverse a VLAN gateway or a middlebox, a natural consequence of redirecting traffic to SDN switches is an increase in certain path lengths and link utilizations. As we discuss later (§6.4), deployment planning requires careful consideration to mind forwarding state capacities and to avoid introducing performance bottlenecks. Consequently, Panopticon presents operators with various resource-performance trade-offs, *e.g.*, between the size and fashion of the partial SDN deployment, and the consequences for the traffic.

As opposed to the dual-stack approach, Panopticon (Figure 6.1c) abstracts away the partial and heterogeneous deployment to yield a logical SDN. As we reason later (§ 6.2.4), many SDN control paradigms can be achieved in a logical SDN. Panopticon enables the expression of any end-to-end policy, as though the network were one big, virtual switch. Routing and path-level policy, *e.g.*, traffic engineering can be expressed too [25], however the abstract network view is reduced to just the deployed SDN switches. As more of the switches are upgraded to support SDN, more fine-grained path-level policies can be expressed.

In summary, we make the following contributions:

1. We design a network architecture for realizing an SDN control plane in a transitional network (§ 6.2), including a scalable mechanism for extending SDN capabilities to legacy devices.
2. We demonstrate the system-level feasibility of our approach with a prototype (§ 6.3).
3. We conduct a simulation-driven feasibility study and a traffic performance emulation study using real enterprise network topologies (with over 1500 switches) and traffic traces (§ 6.4).

6.2 Panopticon SDN Architecture

This section presents the Panopticon architecture, which abstracts a transitional network, where not every switch supports SDN, into a logical SDN. The goal is to enable an SDN programming interface, for defining network policy, which can be extended beyond the SDN switches to ports on legacy switches as well.

Our architecture relies on certain assumptions underlying the operational objectives within enterprise networks. To verify these, we conducted five in-person interviews with operators from both large ($\geq$10,000 users) and medium ($\geq$500 users) enterprise networks and later, solicited 60 responses to open-answer survey questions from a wider audience of network operators [107].

Based on our discussions with network operators, and in conjunction with several design guidelines (*e.g.*, see [18,47]), we make the following assumptions about mid to large enterprise networks and hardware capabilities. Enterprise network hardware consists primarily of Ethernet bridges, namely, switches that implement standard L2 mechanisms (*i.e.*, MAC-based learning and forwarding, and STP) and support VLAN (specifically, 802.1Q and per-VLAN STP). Routers or L3 switches are used as gateways to route between VLAN-isolated IP subnets. For our purposes, we assume a L3 switch is also capable of operating as a L2 switch. In addition, we assume that enterprises no longer intentionally operate "flood-only" hub devices for general packet forwarding.

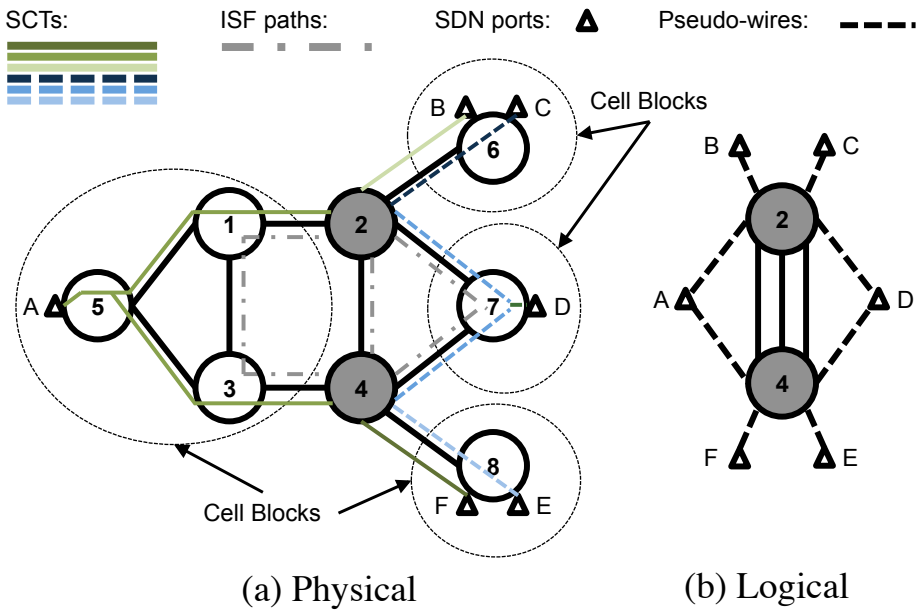


Figure 6.2: Transitional network of 8 switches (SDN switches are shaded): (*a*) The SCTs (Solitary Confinement Trees) of every SDNc (SDN-controlled) port overlaid on the physical topology. (*b*) Corresponding logical view of all SDNc ports, connected to SDN switches via pseudo-wires.

Under these assumptions about legacy enterprise networks, Panopticon can realize a broad spectrum of logical SDNs: Panopticon can extend SDN capabilities to potentially every switchport in the network, however not every port need be included in the logical SDN. We envision an operator may conservatively choose to deploy Panopticon only in part of the network at first, to build confidence and reduce up-front capital expenditure, and then iteratively expand the deployment.

To accommodate iterative expansion of the logical SDN, we divide the set of switchports in the network into *SDN-controlled (SDNc) ports*, that is, those that need to be exposed to and controlled through the logical SDN and *legacy ports*, those that are not. Note that while an SDNc port is conceptually an access port to the logical SDN network, it is not necessarily physically located on an SDN switch (see port *A* in Figure 6.2): It may be connected to an end-host or a legacy access switch.

We extend SDN capabilities to legacy switches by ensuring that all traffic to or from an SDNc port is always restricted to a *safe end-to-end path*, that is, a path that traverses at least one SDN switch. We call this key property of our architecture *Waypoint Enforcement*. The challenge to guaranteeing Waypoint Enforcement is that we may rely *only on existing mechanisms and features readily available* on legacy switches.

6.2.1 Realizing Waypoint Enforcement

Panopticon uses VLANs to restrict forwarding and guarantee Waypoint Enforcement, as these are ubiquitously available on legacy enterprise switches. To conceptually illustrate how, we first consider a straightforward, yet impractical scheme: For each pair of ports which includes at least one SDNc port, choose one SDN switch as the waypoint, and compute the (shortest) end-to-end path that includes the waypoint. Next, assign a unique VLAN ID to every end-to-end path and configure the legacy switches accordingly. This ensures that all forwarding decisions made by every legacy switch only send packets along safe paths. However, such a solution is infeasible, as VLAN ID space is limited to 4096 values, and often fewer are supported in hardware for simultaneous use. Such a rigid solution furthermore limits path diversity to the destination according and cripples fault tolerance.

Solitary Confinement Trees. To realize guaranteed Waypoint Enforcement in Panopticon, we introduce the concept of a *Solitary Confinement Tree* (SCT): a scalable Waypoint Enforcement mechanism that provides end-to-end path diversity. We first introduce the concepts of cell block and frontier. Intuitively, the role of a cell block is to divide the network into isolated islands where VLAN IDs can be reused. The border of a cell block consists of SDN switches and is henceforth called the *frontier*.

Definition 1 (Cell Blocks) *Given a transitional network G,* Cell Blocks $CB(G)$ *is defined as the* set of connected components *of the network obtained after removing from G the SDN switches and their incident links.*

Definition 2 (Frontier) *Given a cell block $c \in CB(G)$, we define the* Frontier $\mathcal{F}(c)$ *as the subset of SDN switches that are adjacent in G to a switch in c.*

Intuitively, the solitary confinement tree is a spanning tree within a cell block, *plus* its frontier. Each SCT provides a safe path from an SDNc port π to every SDN switch in its frontier—or if VLAN resources are scarce, a *subset* of its frontier, which we call the *active frontier*. A single VLAN ID can then be assigned to each SCT, which ensures traffic isolation, provides per-destination path diversity, and allows VLAN ID reuse across cell blocks. Formally, we define *SCT*s as:

Definition 3 (Solitary Confinement Tree) *Let $c(\pi)$ be the cell block to which an SDNc port π belongs. And let* $\mathrm{ST}(c(\pi))$ *denote a spanning tree on $c(\pi)$. Then, the* Solitary Confinement Tree $\mathrm{SCT}(\pi)$ *is the network obtained by augmenting* $\mathrm{ST}(c(\pi))$ *with the (active) frontier $\mathcal{F}(c(\pi))$, together with all links in $c(\pi)$ connecting a switch $u \in \mathcal{F}(c(\pi))$ with a switch in* $\mathrm{SCT}(\pi)$.

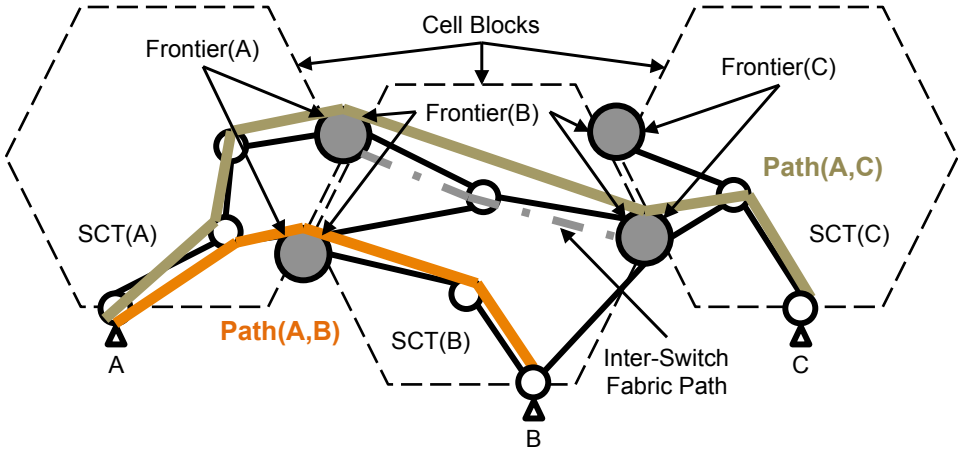


Figure 6.3: The forwarding path between A and B goes via the frontier shared by SCT (A) and SCT (B); the path between A and C goes via an Inter-Switch Fabric path connecting SCT (A) and SCT (C).

Example. Let us consider the example transitional network of eight switches in Figure 6.2a. In this example, SCT (A) is the tree that consists of the paths $5 \to 1 \to 2$ and $5 \to 3 \to 4$. Instead note that SCT (B), which corresponds to the path $6 \to 2$, includes a single SDN switch because switch 2 is the only SDN switch adjacent to cell block $c(B)$. Figure 6.2b shows the corresponding logical view of the transitional network enabled by having SCTs. In this logical view, every SDNc port is connected to at least one frontier SDN switch via a pseudo-wire (realized by the SCT).

6.2.2 Packet Forwarding in Panopticon

We now illustrate Panopticon's basic forwarding behavior (Figure 6.3). As in any SDN, the control application is responsible for installing the necessary forwarding state at the SDN switches (*e.g.*, in accordance with the access policy) and for reacting to topology changes (fault tolerance is discussed in § 6.2.3).

Let us first consider traffic between a pair of SDNc ports s and t. When a packet from s enters SCT(s), the legacy switches forward the packet to the frontier based on MAC-learning, which establishes a symmetric path. Note that a packet from s may use a different path within SCT(s) to the frontier for each distinct destination. Once traffic toward t reaches its designated SDN switch $u \in \mathcal{F}(c(s))$, one of two cases arises:

SDN switches act as VLAN gateways. This is the case when the destination SDNc port t belongs to a cell block whose frontier $\mathcal{F}(c(t))$ shares at least one switch u with $\mathcal{F}(c(s))$. Switch u acts as the designated gateway between SCT(s) and

SCT(t), that is, u rewrites the VLAN tag and places the traffic within SCT(t). For instance, in the example of Figure 6.2a, switch 2 acts as the gateway between ports A, B and C.

Inter-Switch Fabric (ISF). When no SDN switch is shared, we use an *Inter-Switch Fabric (ISF) path*: point-to-point tunnels between SDN switches which can be realized *e.g.*, with VLANs or GRE. In this case, the switch u chooses one of the available paths to forward the packet to an SDN switch $w \in \mathcal{F}(c(t))$, where w is the designated switch for the end-to-end path $p(s,t)$. In our example of Figure 6.2a, ISF paths are shown in gray and are used *e.g.*, for traffic from B or C to E or F, and vice versa.

We next turn to the forwarding behavior of legacy ports. Again, we distinguish two cases. First, when the path between two legacy ports only traverses the legacy network, forwarding is performed according to the traditional mechanisms and is unaffected by the partial SDN deployment. Policy enforcement and other operational objectives must be implemented through traditional means, *e.g.*, ACLs. In the second case, anytime a path between two legacy ports necessarily encounters an SDN switch, the programmatic forwarding rules at the switch can be leveraged to police the traffic. This is also the case for all traffic between any pair of an SDNc and a legacy port. In other words, Panopticon *always guarantees safe paths for packets from or to every SDNc port,* which we formally prove in the technical report [107].

6.2.3 Architecture Discussion

Having described all components of the architecture, we now discuss certain key properties.

Key SCT properties. Recall that one VLAN ID is used per SCT and that VLAN IDs can be reused across Cell Blocks. Limiting the size of the active frontier allows further VLAN ID reuse across fully-disjoint SCTs within the same cell block. A different path may be used within the SCT for each distinct destination. SCTs can be precomputed and automatically installed onto legacy switches (*e.g.*via SNMP) however, re-computation is required when the physical topology changes.

ISF path diversity trade-offs. Within the ISF, there may be multiple paths between any given pair of SDN switches. We expect that some applications may require a minimum number of paths. A minimum of two disjoint paths is necessary, to tolerate single link failures. If the ISF is realized using a VLAN-based approach, each path consumes a VLAN ID from every cell block it traverses. Alternative mechanisms, *e.g.*, IP encapsulation or network address translation can be used to implement the ISF depending on SDN and legacy hardware capabilities.

Coping with broadcast traffic. Broadcast traffic can be a scalability concern. We take advantage of the fact that each SCT limits the broadcast domain size, and

we rely on SDN capabilities to enable in-network ARP and DHCP proxies as shown in [100]. We focus on these important bootstrapping protocols as it was empirically observed that broadcast traffic in enterprise networks is primarily contributed by ARP and DHCP [100, 128]. Last, we note that in the general case, if broadcast traffic must be supported, the overhead that Panopticon introduces is proportional to the number of SCTs in a cell block, which, at worst, grows linearly with the number of SDNc ports of a cell block.

Tolerating failures. We decompose fault tolerance into three orthogonal aspects. First, within an SCT, Panopticon relies on standard STP mechanisms to survive link failures, although to do so, there must exist sufficient physical link redundancy in the SCT. The greater the physical connectivity underlying the SCT, the higher the fault tolerance. Additionally, the coordination between SDN controller and legacy STP mechanisms allows for more flexible fail-over behavior than STP alone. When an SDN switch at the frontier $\mathcal{F}$ of an SCT notices an STP re-convergence, we can adapt the forwarding decisions at $\mathcal{F}$'s SDN switches to restore connectivity. A similar scheme can address link failures within the ISF.

Second, when SDN switches or their incident links fail, the SDN controller recomputes the forwarding state and installs the necessary flow table entries. Furthermore, precomputed fail-over behavior can be leveraged as of OpenFlow version 1.1 [137].

Third, the SDN control platform must be robust and available. In this respect, previous work [103] demonstrates that well-known distributed systems techniques can effectively achieve this goal.

6.2.4 Realizing SDN Benefits

By now, we have described how Panopticon shifts the active network management burden away from the legacy devices and onto the SDN control plane. This conceptually reduces the network to a logical SDN as presented in Figure 6.2b. Consequently, we want to be able to reason about what types of policy can be specified and which applications can be realized in such a transitional network.

Panopticon exposes an SDN abstraction of the underlying partial SDN deployment. In principle, any control application that runs on a full SDN can be supported in Panopticon since, from the perspective of the application, the network appears as though it is a full SDN deployment consisting of just the SDN switches. In practice, there are a small number of caveats.

SDNc ports in the logical SDN. An SDNc port in Panopticon is not necessarily physically located at an SDN switch, and it may be attached to multiple SDN switches. Accordingly, the SDN controller must take into account that each SDNc port may be reached from its frontier via multiple paths. Furthermore, visibility into how resources are shared on legacy links can not be guaranteed.

Logical SDN vs. full SDN. As an abstraction layer, Panopticon is responsible for hiding the legacy devices and acts as a "network hypervisor" that maps the logical SDN abstraction to the underlying hardware (similar to the concept of network objects in Pyretic [116]). However, because the global network view is reduced to the set of SDN switches, applications are limited to control the forwarding behavior based on the logical SDN. This should not be viewed strictly as a limitation, as it may be desirable to further abstract the entire network as a single virtual switch over which to define high-level policies (*e.g.*, access policy) and have the controller platform manage the placement of rules on physical switches [97]. Nevertheless, the transitional network stands to benefit in terms of management simplification and enhanced flexibility as we next illustrate.

More manageable networks. Arguably, as control over isolation and connectivity is crucial in the enterprise context we consider, the primary application of SDN is *policy enforcement*. As in Ethane [42], Panopticon enables operators to define a single network-wide policy, and the controller enforces it dynamically by allowing or preventing communication upon seeing the first packet of a flow as it tries to cross an SDN switch.

The big switch [97] abstraction enables the network to support Ethernet's plug-and-play semantics of flat addressing and, as such, simplifies the handling of host mobility. This can be observed from the fact that our architecture is an instance of the fabric abstraction [44]. The ISF represents the network core and SCTs realize the edge. At the boundary between the SCTs and ISF, the SDN switches enable the decoupling of the respective network layers, while ensuring scalability through efficient routing in the ISF.

More flexible networks. The controller maintains the global network view and performs route computation for permitted flows. This provides the opportunity to efficiently enforce middlebox-specific traffic steering within the SDN-based policy enforcement layer, as in SIMPLE [132]. Integrating middleboxes in Panopticon requires that middleboxes are connected to SDNc ports.

A logical SDN also enables the realization of strong consistency semantics for policy updates [138]. Although legacy switches do not participate in the consistent network update, at the same time, they do not themselves express network policy—as that forwarding state resides exclusively on SDN switches.

Putting it all together, Panopticon is the first architecture to realize an approach for operating a transitional network as though it were a fully deployed SDN, yielding benefits for the entire network, not just the devices that support SDN programmability.

6.3 Panopticon Prototype

To cross-check certain assumptions on which Panopticon is built, this section describes our implementation and experimental evaluation of a Panopticon prototype. The primary goal of our prototype is to demonstrate feasibility for legacy switch interaction–namely, the ability to leverage path diversity within each SCT, and respond to failure events and other behaviors within the SCT.

Our prototype is implemented upon the POX OpenFlow controller platform [12] and comprises two modules: path computation and legacy switch interaction.

Path computation. At the level of the logical SDN, our path computation module is straightforward: it reacts to the first packet of every flow and, if the flow is permitted, it uses the global network view to determine the shortest path to the destination. Consequently, it installs the corresponding forwarding rules. Our implementation supports two flow definitions: (1) the aggregate of packets between a pair of MAC addresses, and (2) the micro-flow, *i.e.*, IP 5-tuple. As each SDNc port may be reached over multiple paths from the SDN switches on its frontier, our prototype takes into account the behavior of STP within the SCT (monitored by the component below) to select the least-cost path based on source-destination MAC pair.

Legacy switch interaction. The Spanning Tree Protocol (STP) or a variant such as Rapid STP, is commonly used to achieve loop freedom within L2 domains and we interact with STP in two ways. First, within each SCT, we configure a per-VLAN spanning tree protocol (*e.g.*, Multiple STP) rooted at the switch hosting the SCT's SDNc port. We install forwarding rules at each SDN switch to redirect STP traffic to the controller, which interprets STP messages to learn the path cost between any switch on the frontier and the SCT's SDNc port, but *does not reply* with any STP messages. Collectively, this behavior guarantees that each SCT is loop free. When this component notices an STP re-convergence, it notifies the path computation module, which in turn adapts the forwarding decisions at SDN switches to restore connectivity as necessary. Second, to ensure network-wide loop freedom for traffic from legacy ports, SDN switches behave as ordinary STP participants. When supported, this is achieved by configuring STP on the switches themselves. Otherwise, Panopticon can run a functionally equivalent implementation of STP.

6.3.1 Application: Consistent Updates

To showcase the "logical SDN" programming interface exposed by Panopticon, we have implemented per-packet consistency [138] for transitional networks. Our application allows an operator to specify updates to the link state of the network, while ensuring that the safety property of per-packet consistency applies over the entire network, even to legacy switches.

To implement this application, we modify the path computation to assign a unique configuration version number to every shortest path between SDNc ports. This version number is used to classify packets according to either the current or the new configuration.

When the transition from current to new configuration begins, the controller starts updating all the SDN switches along the shortest path for both the forward and backward traffic. This update includes installing a new forwarding rule and using the IP *TOS* header field (i.e., in a monotonically increasing fashion) to encode or match the version number. The rules for the old configuration with the previous version number, if there are any, are left in place and intact. This procedure guarantees that any individual packet traversing the network sees only the "old" or "new" policy, but never both.

Once all the rules for the new configuration are in place at every switch, gratuitous ARP messages are sent over to the legacy switches along the new path so that the traffic is re-routed. After a operator-defined grace-period, when the last in-flight packet labeled with the "old" tag leaves the network, the controller deletes the old configuration rules from all the SDN switches, and the process completes.

6.3.2 Evaluation

Our prototype is deployed on a network of hardware switches comprising two NEC IP8800 OpenFlow switches and one Cisco C3550XL, three Cisco C2960G, and two HP 5406zl MAC-learning Ethernet switches, interconnected as in Figure 6.2a. To emulate 6 hosts (A through F), we use an 8-core server with an 8-port 1Gbps Ethernet interface which connects to each SDNc port on the legacy switches depicted in the figure. Two remaining server ports connect to the OpenFlow switches for an out-of-band control channel.

We conduct a first experiment to demonstrate how Panopticon recovers from an STP re-convergence in an SCT, and adapts the network forwarding state accordingly. We systematically emulate 4 link failure scenarios between links (5,1) and (1,2) by disabling the respective source ports of each directed link. Host A initiates an iperf session over switch 2 to host D. After 10 seconds into the experiment, a link failure is induced, triggering an STP re-convergence. The resulting BDPU updates are observed by the controller and connectivity to host D is restored over switch 4. Figure 6.4(a) shows the elapsed time between the last received segment and first retransmitted packet over 10 repetitions and demonstrates how Panopticon quickly restores reachability after the failure event. Interestingly, we observe that Panopticon reacts faster to link changes detected via STP reconvergence (*e.g.*, `sw5_to_sw1`) than to link changes at the OpenFlow switches themselves (`sw1_to_sw2`), since our particular switches appear to briefly, internally delay sending those event notifications.

We next conduct a second experiment to explore how the SCT impacts the performance of a BitTorrent file transfer conducted among the hosts attached to SDNc ports. In this experiment, we begin by seeding a 100MB file at one host (A through F), in an iterative fashion. All other hosts are then initialized to begin simultaneously downloading the file from the seeder and amongst one another. We repeat each transfer 10 times, and measure the time for each host to complete the transfer. We then compare each time with an identical transfer in a L2 spanning tree topology. Figure 6.4(b), illustrates that some of the hosts (*i.e.*, A and D) are able to leverage the multi-path forwarding of their SCTs to finish sooner. Others, *e.g.*, B and C experience longer transfer times, as their traffic shares the same link to their frontier switch.

6.4 Incremental SDN Deployment

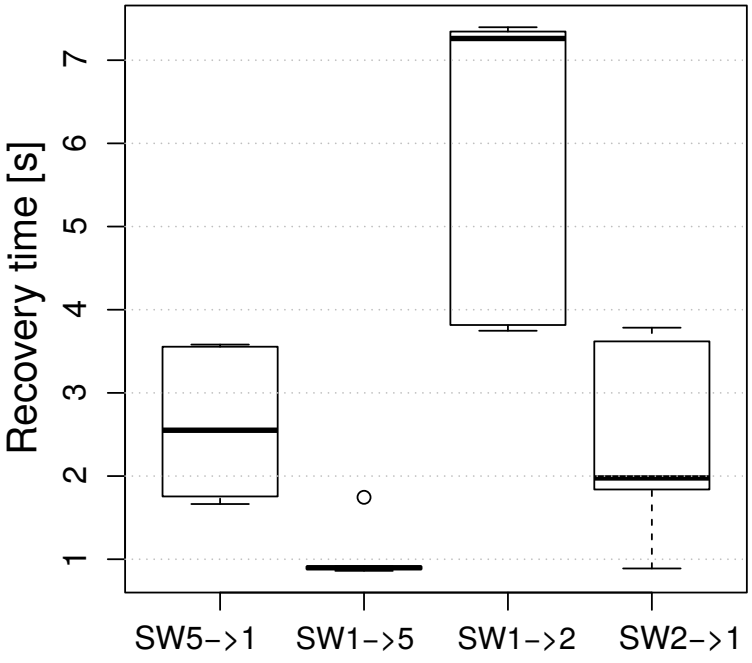
Panopticon makes no assumptions about the number of SDN switches or their locations in a partial SDN deployment. However, under practical resource constraints, an arbitrary deployment may make the *feasibility* of the logical SDN abstraction untenable, as the flow table capacities at the SDN switches and the availability of VLAN IDs on legacy switches are limited.

Beyond feasibility, the SDN deployment also influences network *performance*. By ensuring Waypoint Enforcement, SDN switches may become choke points that increase path lengths and link loads, in some cases beyond admissible values. Deployment planning therefore becomes a necessity.

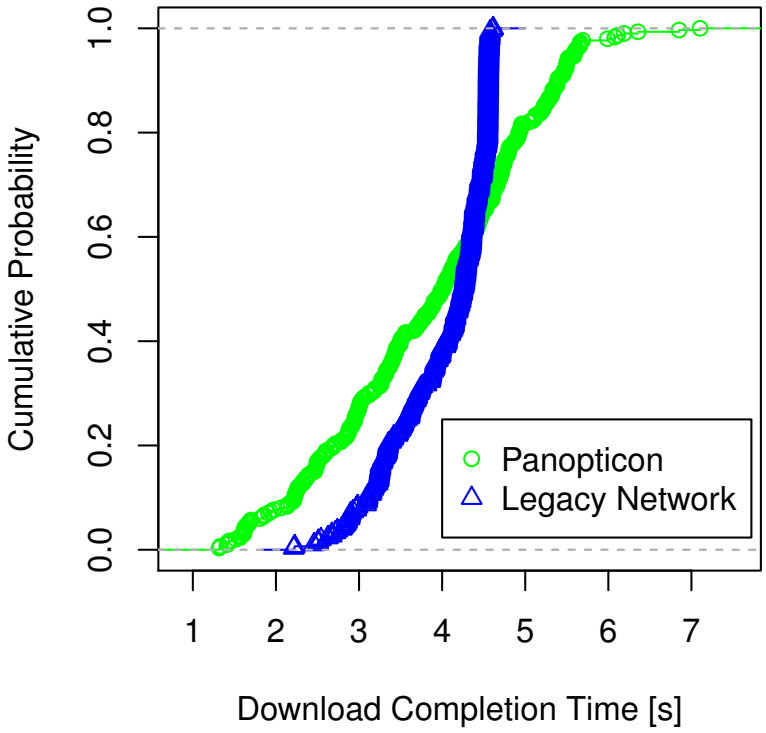
6.4.1 Deployment Planning

Deciding the number and location of SDN switches to deploy can be viewed as an optimization problem wherein the objective is to yield a good trade-off between performance and costs subject to feasibility. We envision that a tool with configurable parameters and optimization algorithms may assist operators in planning the deployment by answering questions such as "What is the minimal number of SDN switches needed to support all ports as SDNc ports?" or "Which switches should be first upgraded to SDN to reduce bottleneck link loads?"

In a companion technical report of this paper [107], we present a general integer programming algorithm to compute a partial SDN deployment optimized for different objective functions and resource constraints. This algorithm can assist operators in upgrading the network, starting from a legacy network or one that is already partially upgraded.



(a) SCT link failure recovery



(b) Bittorrent performance

Figure 6.4: Testbed experiments: (*a*) Panopticon recovers from link failure within seconds. (*b*) Panopticon enables path diversity but also increases load on some links.

Site	Access/Dist/Core	max/avg/min degree
LARGE	1296 / 412 / 3	53 / 2.58 / 1
EMULATED	489 / 77 / 1	30 / 6.3 / 1
MEDIUM	– / 54 / 3	19 / 1.05 / 1
SMALL	– / 14 / 2	15 / 3 / 2

Table 6.1: Evaluated network topology characteristics.

We observe however that specific objectives and constraints for planning an SDN deployment are likely to depend on practical contextual factors such as hardware life-cycle management, support contracts and SLAs, long-term demand evolution and more. Unfortunately, these factors are rather qualitative, vary across environments, and are hard to generalize.

Instead, we reason more generally about how deployment choices influence feasibility and performance of our approach. To navigate the deployment problem space without the need to account for all contextual factors, we focus on a few general properties of desirable solutions:

(*i*) *Waypoint Enforcement:* Every path to or from an SDNc port must traverse at least one SDN switch.

(*ii*) *Feasible:* SDN switches must have sufficient forwarding state to support all traffic policies they must enforce. VLAN requirements to realize SCTs must be within limits.

(*iii*) *Efficient:* The resulting traffic flow allocations should be efficient. We reason about efficiency using two metrics: The first metric is the path *stretch*, which we define for a given path (s, t) as the ratio between the length of the path under Waypoint Enforcement and the length of the shortest path in the underlying network. The second metric is the expected maximum load on any link.

6.4.2 Simulation-assisted Study

To explore feasibility and efficiency of Panopticon, we simulate different partial SDN deployment scenarios using real network topologies under different resource constraints and traffic conditions. These simulations let us (*i*) evaluate the feasibility space of our architecture, (*ii*) explore the extent to which SDN control extends to the entire network, and (*iii*) understand the impact of partial SDN deployment on link utilization and path stretch.

Methodology

To simulate Panopticon deployment, we first choose network topologies with associated traffic estimates and resource constraints.

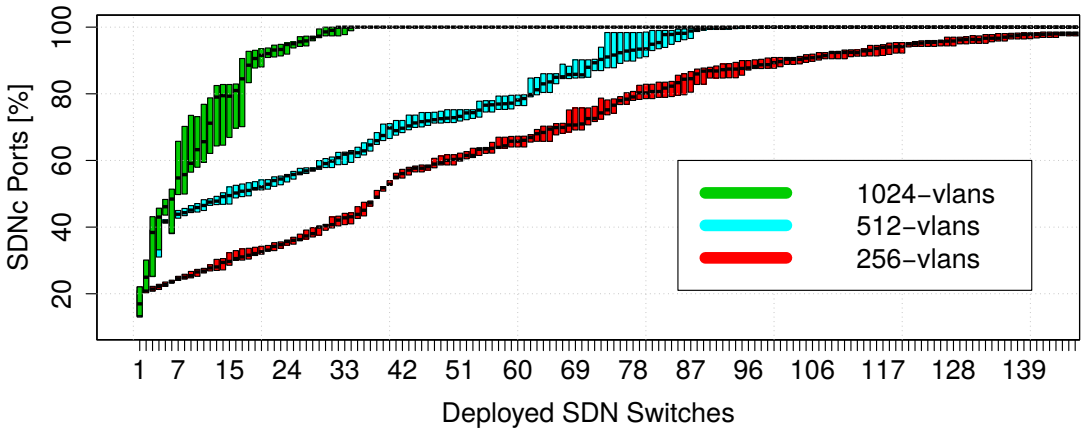
Topologies. Detailed topological information, including device-level configurations, link capacities, and end-host placements is difficult to obtain for sizeable networks: operators are reluctant to share these details due to privacy concerns. Hence, we leverage several publicly available enterprise network topologies [154, 169] and the topology of a private, local large-scale campus network. The topologies range from SMALL, comprising just the enterprise network backbone, to a MEDIUM network with 54 distribution switches, to a comprehensive large-scale campus topology derived from anonymized device-level configurations of 1711 L2 and L3 switches. Summary information on the topologies is given in Table 6.1. Every link in each topology is annotated with its respective capacity. We treat port-channels (bundled links), as a single link of its aggregate capacity.

Simulation results on the SMALL and MEDIUM network gave us early confidence in our approach, however their limited size does not clearly demonstrate the most interesting design trade-offs. Thus, we only present simulation results for LARGE.

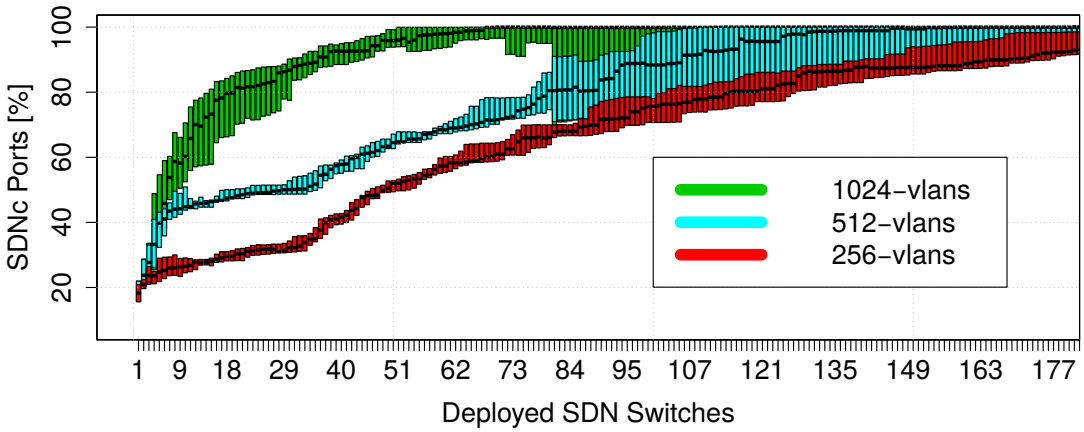
Focus on distribution switches. In our approach, we distinguish between *access switches*, *distribution switches*, and *core switches*. Access switches are identified both topologically, as well as from device-level configuration metadata. Core switches are identified as multi-chassis devices, running a L3 routing protocol. Due to their topological location, SCT construction to core switches becomes challenging, thus, we focus on distribution switches (in the following referred to as the *candidate set* for the upgrade). In case of the LARGE network, this candidate set has cardinality 412 of which, 95 devices are identified as L3 switches (running OSPF or EIGRP). Within this distribution network, we reason about legacy distribution-layer switchports as candidates to realize SDNc ports, subject to Waypoint Enforcement. Each distribution-layer switchport leads to an individual access-layer switch to which end-hosts are attached. Thus, we identify 1296 candidate SDNc ports. Unless otherwise noted, we construct SCTs connecting each SDNc port to its full frontier.

Traffic estimates. We use a methodology similar to that applied in SEATTLE [100] to generate a traffic matrix based on packet-level traces from an enterprise campus network, the *Lawrence Berkeley National Laboratory* (LBNL) [128]. The LBNL dataset contains more than 100 hours of anonymized packet level traces of activity of several thousands of internal hosts. The traces were collected by sampling all internal switchports periodically. We aggregate the recorded traffic according to source-destination pairs and for each sample, we estimate the load imposed on the network. We note that the data contains sources from 22 subnets.

To project the load onto our topologies, we use the subnet information from the traces to partition each of our topologies into subnets as well. Each of these subnets contains at least one distribution switch. In addition, we pick one node as the Internet gateway. We associate traffic from each subnet of the LBNL network in random round-robin fashion to candidate SDNc ports. All traffic within the LBNL network is aggregated to produce the intra-network traffic matrix. All destinations



(a) VOL Switch Selection Strategy



(b) Random Switch Selection Strategy

Figure 6.5: Percentage of SDNc ports as a function of deployed SDN switches, under different VLAN availability. When more VLANs are supported by legacy devices, more SDNc ports can be realized with fewer SDN switches.

outside of the LBNL network are assumed to be reachable via the Internet gateway and thus mapped to the chosen gateway node. By running 10 different random port assignments for every set of parameters, we generate different traffic matrices, which we use in our simulations. Still, before using a traffic matrix we ensure that the topology is able to support it. For this purpose we project the load on the topology using shortest path routes, and scale it conservatively, such that the most utilized gigabit link is at 50% of its nominal link capacity.

Resource constraints. Although the maximum number of VLAN IDs expressible in 802.1Q is 4096, most mid- to high-end enterprise network switches support 512-1024 VLAN IDs for simultaneous use. Accordingly, we focus on simulating scenarios where legacy switches support at most 256, 512, and 1024 simultaneous VLAN IDs.

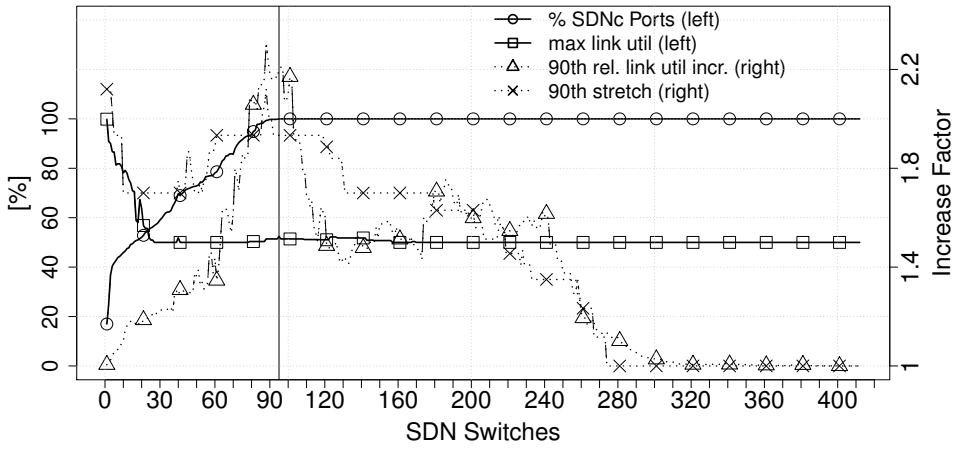
While first generation OpenFlow capable switches were limited to around 1K flow table entries many current switches readily support from 10K to 100K entries for exact and wild-card matching. Bleeding edge devices support up to 1M flow table entries [124]. To narrow our parameter space, we fix the flow table capacity of our SDN switches to 100k entries, and vary the average number of rules required to realize policy for a single SDNc port from 10 to 20. We furthermore ensure that every SDN switch maintains at all times both policy and basic forwarding state (one entry per SDNc port reached through that switch) to ensure all-to-all reachability in the absence of any policy. We note, this is a conservative setting; by comparison, if flow table entries were kept only in the temporal presence of their respective, active source-destination traffic in the LBNL dataset, the maximum number of entries would never exceed 1,200 flows/s [42].

Switch Deployment Strategies

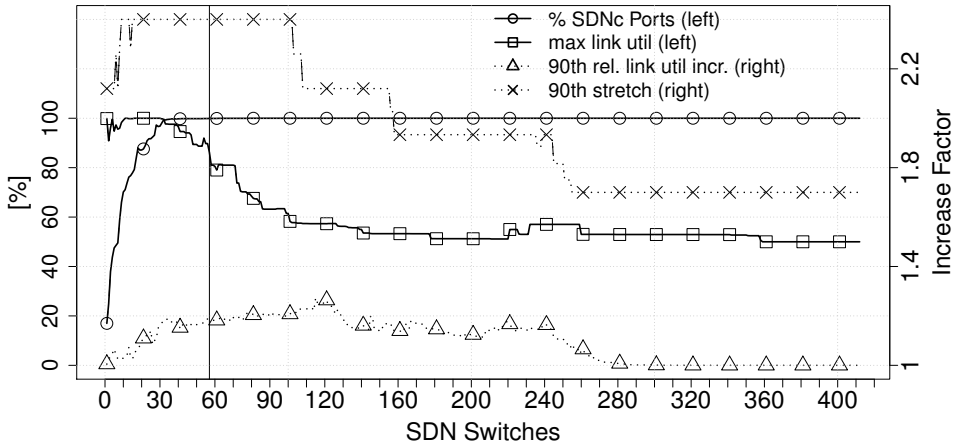
Given our topology and traffic estimates, we next explore how SDN switch deployment influences feasibility and performance. We study this through a simple yet effective deployment heuristic inspired by classical techniques such as Facility Location, called Vol.

Vol iteratively selects one legacy switch to be replaced at a time, in decreasing order of switch egress traffic volume. SDNc candidate ports are then accommodated in the following greedy fashion: SDNc ports from the previous iteration are accommodated first (we initially iterate over a random permutation of SDNc candidates). An SCT is constructed to the active frontier, whose size, chosen by the designer, defines a feasibility-efficiency trade-off we investigate later. If an SCT can be created, designated SDN switches from the active frontier are selected for each destination port, and flow table entries are allocated. If flow table policy is accommodated, the traffic matrix is consulted and traffic is projected from the candidate port to every destination along each waypoint-enforced path. When no link exceeds its maximum utilization (or safety threshold value), the port is considered SDNc. The remaining SDNc candidates are then tried and thereafter, the next SDN switch candidate is deployed and the process repeats. As Vol is a greedy algorithm and does not backtrack, it may terminate prior to satisfying all SDNc candidates, despite the existence of a feasible solution.

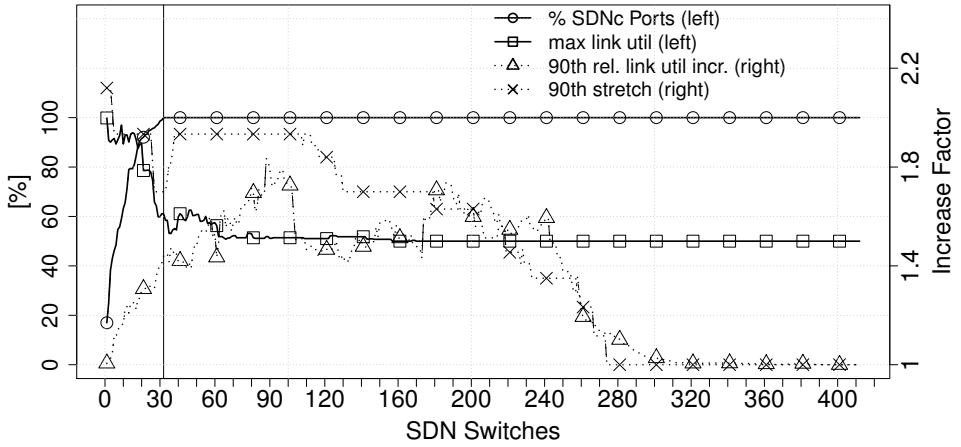
For comparison, we make use of **Rand**, which iteratively picks a legacy switch uniformly at random, subject to VLAN, flow table, and link utilization constraint satisfaction. Rand allows us to evaluate the sensitivity of the solution to the parameters we consider and the potential for sophisticated optimizations to outperform naïve approaches. We repeat every Rand experiment with 10 different random seeds.



(a) 512 VLANs, Full Frontier



(b) 512 VLANs, 2 Active Frontier



(c) 1024 VLANs, Full Frontier

Figure 6.6: SDN deployment vs. max link utilization, 90th percentile path stretch and relative link util increase. Feasible 100% SDNc port coverage can be realized with 33 SDN switches, with acceptable link utilization and path stretch.

SDNc Ports vs. Deployment Strategy

As Panopticon is designed to enable a broad spectrum of partial SDN deployments, we begin our evaluation by asking, "As a deployment grows, what fraction of candidate SDNc ports can be accommodated, under varying resource constraints?"

Scenario 1: To answer this question, we choose three values for the number of maximum simultaneous VLANs supported on any legacy switch (256, 512, 1024). We choose a policy requirement of 10 flow table entries on average for every (SDNc, destination port) pair as defined in the traffic matrix, so as to avoid a policy state bottleneck. We reason that policy state resource bottlenecks can be avoided by the operator by defining worst-case policy state needs in advance and then deploying SDN switches with suitable flow table capacity. We then compare our two deployment strategies VOL and RAND for different numbers of deployed SDN switches, as depicted by Figure 6.5 in which repeated experimental runs are aggregated into boxplots.

Observations 1: Figure 6.5 illustrates that the ability to accommodate more SDNc ports with a small number of SDN switches depends largely on the number of VLAN IDs supported for use by the legacy hardware. Under favorable conditions with 1024 VLANs, 100% SDNc port coverage can be had for as few as 33 SDN switches. VLAN ID availability is necessary to construct SCTs and in Figure 6.5(a) we see that when legacy switches support at most 256 VLANs, over 140 SDN switches must be deployed before achieving full SDNc port coverage. Figure 6.5(b) shows the importance of choosing where to deploy SDN switches, as the earliest 100% SDNc feasible solution requires 20 additional SDN switch over VOL.

How Will Panopticon Affect Traffic?

We next ask: "As more SDNc ports are accommodated, what will Waypoint Enforcement do to the traffic?"

Scenario 2: To answer this question, we evaluate the metrics path stretch and link utilization as we increase the SDN deployment, subject to two different VLAN resource constraints. As in Scenario 1, we assume average policy requirement of 10 flow table entries for every (SDNc, destination port) pair. Recall that our methodology scales up the baseline traffic matrix to ensure that the most utilized link in the original network is 50% utilized.

Figure 6.6 plots the relationship between the percentage of accommodated SDNc ports, the maximum link utilization, and the 90th percentile link utilization path stretch. Median values are shown for all metrics, across the repeated experiments. The feasible regions of each full "logical SDN" deployment with respect to all resource constraints are indicated by the vertical bar.

Observations 2: Figure 6.6(a) indicates that with 512 VLANs usable in the legacy network, a full logical SDN becomes feasible with 95 switches where the most utilized link reaches 55% of its capacity. The 90th percentile path stretch hovers around 2.1. As further switches are upgraded, the stretch and relative link utilization continue to improve. A more optimistic case is depicted in Figure 6.6(c) where full logical SDN is achieved with 33 switches. However, given fewer SDN waypoints, the maximum link utilization is higher at 60%. The key takeaway from this plot is that given conservative base link utilization, the additional burden imposed by SDN Waypoint Enforcement is small in many deployments.

Efficient 100% SDNc Port Feasibility

As we point out in our architecture section, Panopticon allows the designer to make efficiency trade-offs, where a full logical SDN can be realized with fewer SDN switches, at the expense of higher link utilization and path stretch. The parameter that governs this trade-off is the active frontier size. We next look to Figures 6.6(a) and 6.6(b), which illustrate how this trade-off plays out.

Recall from Figure 6.6(a) that for a legacy network supporting 512 VLANs, a full logical SDN becomes feasible with about 95 SDN switches when using all available frontier switches. However, each path to the frontier switches consumes a VLAN, which blocks other SDNc candidate ports later on. By limiting the active frontier to at most 2 switches, Figure 6.6(b) illustrates that a feasible solution can be achieved with 56 switches. The path stretch notably increases to a factor of 2.4, compared to less than 2 when a larger frontier is used. This trade-off underlines the flexibility of Panopticon: Operators can make design choices tailored to their individual network performance requirements.

6.4.3 Traffic Emulation Study

To compliment our simulation-based approach and further investigate the consequences of Panopticon on traffic, we conduct a series of emulation-based experiments on portions of a real enterprise network topology. These experiments (*i*) provide insights into the consequences of Waypoint Enforcement on TCP flow performance, and (*ii*) let us explore the extent to which the deployment size impacts TCP flow performance when every access port is operated as an SDNc port.

Setup. We use Mininet [69] to emulate a Panopticon deployment. Due to the challenges of emulating a large network [69], we scale down key aspects of the network characteristics of the emulation environment. We (*i*) use a smaller topology, EMULATED (see Table 6.1), which is a 567-node sub-graph of the LARGE topology obtained by pruning the graph along subnet boundaries, (*ii*) scale down the link capacities by 2 orders of magnitude, and (*iii*) correspondingly reduce the TCP

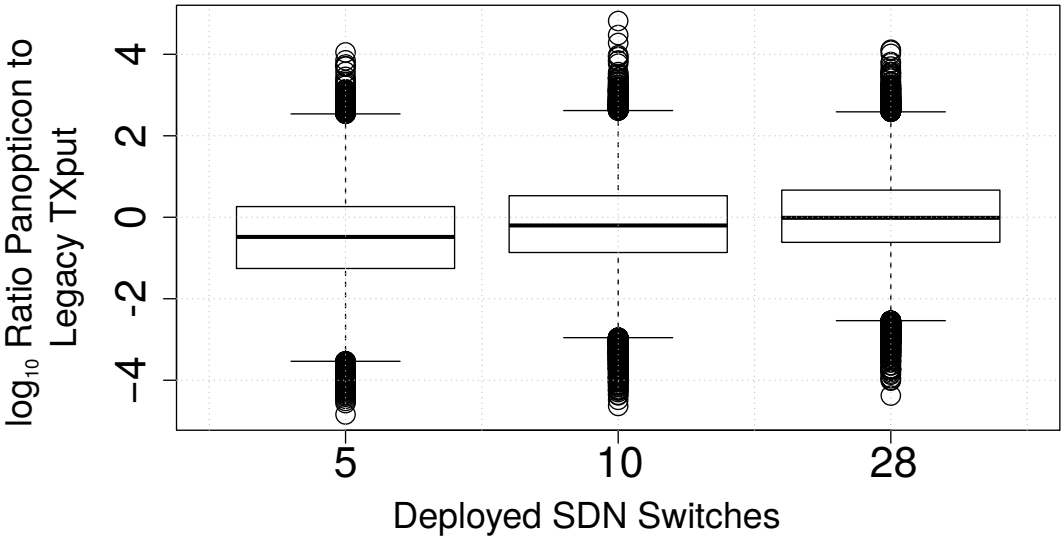


Figure 6.7: In both scenarios A and B (28 and 10 SDN switches), the median throughput over all experiments remains close to the performance of the legacy network.

MSS to 536 bytes to reduce packet sizes in concert with the reduced link capacities. This allows us to avoid resource bottlenecks that otherwise interfere with traffic generation and packet forwarding, thus influencing measured TCP throughput.

We run our experiments on a 64-core at 2.6GHz AMD Opteron 6276 system with 512GB of RAM running the 3.5.0-45-generic #68 Ubuntu Linux kernel using Open-VSwitch version 2.1.90. Baseline throughput tests indicate that our system is capable of both generating and forwarding traffic of 489 simultaneous TCP connections in excess of 34Gbps, sufficiently saturating the aggregate emulated link capacity of every traffic sender in our experiments. We note that traffic in the subsequent experiments is generated on the system-under-test itself.

Thus, our emulation experiments involve 489 SDNc ports located at "access switches" at which traffic is sent into and received from the network. The distribution network consists of 77 devices of which 28 devices are identified as IP router gateways that partition the network in Ethernet broadcast domains. Within each broadcast domain, we introduce a single spanning tree to break forwarding loops.

Traffic. We apply a traffic workload to our emulated network based on (i) a traffic matrix, defined over the 489 SDNc ports, and (ii) a synthetically generated flow size distribution where individual TCP flow sizes are obtained from a Weibull distribution with shape and scaling factor of 1, given in Table 6.2.

We re-use the traffic matrix used in the simulations to define the set of communicating source-destination pairs of SDNc ports in the network. For system scalability reasons, we limit the number of source-destination pairs to 1024, selected

	min	median	avg	max
Flow Sizes (in MB)	0.00005	6.91	9.94	101.70
Path Stretch A	1.0	1.0	1.002	1.67
Path Stretch B	1.0	1.0	1.16	3.0
Path Stretch C	1.0	1.33	1.25	3.0

Table 6.2: Traffic parameter and path stretch statistics.

randomly from the traffic matrix. For each pair of SDNc ports, we define a sequence of TCP connections to be established in iterative fashion, whose transfer sizes are determined by the aforementioned Weibull distribution. The total traffic volume exchanged between each pair is limited to 100MB. When the experiment begins, every source-destination pair, in parallel begins to iterate through its respective connection sequence. Once every traffic source has reached its 100MB limit, the experiment stops.

Scenarios. We consider three deployment scenarios in which we evaluate the effects of Panopticon on TCP traffic: Scenario A in which 28 switches out of the 77 distribution switches are operated as SDN switches, and scenarios B and C, which narrow down the number of SDN switches in A to 10 and 5 SDN switches, respectively. SDN switch locations are selected at random based on location of IP routers, identified from the topology dataset.

Results. In each scenario, we compare TCP flow throughput in the Panopticon deployment versus the original network topology (which uses shortest-path IP routes with minimum cost spanning trees). Table 6.2 lists path stretch statistics for each scenario, namely, the ratio of SDN (waypoint-enforced) to legacy path length for every source-destination pair in the network.

Figure 6.7 illustrates the impact of Waypoint Enforcement on TCP performance in the three scenarios. The first observation we make is that in scenario A, when the 28 IP routers are replaced with SDN switches, the impact on median TCP throughput is negligible. This is perhaps expected, as all traffic across subnets must traverse some IP router in the legacy network, regardless. Some flows experience congestion due to Waypoint Enforcement. Other flows actually experience a performance increase due to the availability of multiple alternate paths in Panopticon. As the SDN deployment shrinks to more conservative sizes in scenarios B and C, the effects of Waypoint Enforcement becomes more prominent, supporting our observed simulation results.

6.4.4 Discussion

Scalability. As the number of SDNc candidates increases, the resource demands grow as well. We believe that one or two SDNc ports for every access switch however is a reasonable starting point for most partial SDN deployments. Even at one SDNc

per access switch, a reasonable level of policy granularity, as end-hosts connected to the same physical access switch are often considered to be part of the same administrative unit as far as policy-specification is concerned. Should finer-grained SDNc port allocation be necessary, features such as Cisco's protected switchports (or similar ones from other vendors) may be leveraged to extend Waypoint Enforcement to individual access-switch ports without the need for additional SCTs.

Why fully deploy SDN in enterprise networks? Perhaps many enterprise networks do not need to fully deploy SDN. As our results indicate, it is a question of the trade-offs between performance requirements and resource constraint satisfaction. Our Panopticon evaluation suggests that partial deployment may in-fact be the right mid-term approach for some enterprise networks.

6.5 Related Work

Our approach toward a scalable, incrementally deployable network architecture that integrates legacy and SDN switches to expose the abstraction of a logical SDN both builds upon and complements previous research.

SDN. In the enterprise, *SANE* [43] and *Ethane* [42] propose architectures to enforce centrally-defined, fine-grained network policy. Ethane overcomes SANE [43]'s deployment challenges by enabling legacy device compatibility. Ethane's integration with the existing deployment is however, ad-hoc and the behavior of legacy devices falls out of Ethane's control. Panopticon by contrast, can guarantee SDN policy enforcement through principled interaction with legacy devices to forward traffic along safe paths. Google's transition to a software-defined WAN involved an overhaul of their entire switching hardware to improve network performance [94]. In contrast to their goals, we take an explicit stance at transitioning to an SDN control plane without the need for a complete hardware upgrade. Considering a partial SDN deployment, Agarwal *et al.* [25] demonstrate effective traffic engineering of traffic that crosses at least one SDN switch. Panopticon is an architecture that enforces this condition for all SDNc ports. The work on software-controlled routing protocols [158] presents mechanisms to enable an SDN controller to indirectly program L3 routers by carefully crafting routing messages. We view this work as complementary to ours in that it could be useful to extend Waypoint Enforcement to IP routers.

Enterprise network design and architecture. Scalability issues in large enterprise networks are typically addressed by building a network out of several (V)LANs interconnected via L3 routers [18,47]. *TRILL* [130] is an IETF Standard for so-called *RBridges* that combine bridges and routers. Although TRILL can be deployed incrementally, we are not aware of any work regarding its use for policy enforcement in enterprise networks.

Sun *et al.* [153] and Sung *et al.* [154] propose a systematic redesign of enterprise networks using parsimonious VLAN allocation to ensure reachability and provide isolation. These works focus on legacy networks only. The *SEATTLE* [100] network architecture uses a one-hop DHT host location lookup service to scale large enterprise Ethernet networks. However, such clean-slate approach is not applicable for the transitional networks we consider.

Scalable data-center network architectures. There is a wealth of recent work towards improving data-center network scalability. To name a few, *FatTree* [26], *VL2* [63], *PortLand* [123], *NetLord* [120], *PAST* [152] and *Jellyfish* [149], offer scalable alternatives to classic data-center architectures at lower costs. As clean-slate architectures, these approaches are less applicable to transitional enterprise networks, which exhibit less homogeneous structure and grow "organically" over time.

Evolvable inter-networking. The question of how to *evolve* or run a *transitional* network, predates SDN and has been discussed in many contexts, including Active Networks [161]. Generally, changes in the network layer typically pose a strain to network evolution, which lead to overlay approaches being pursued (*e.g.*, [96, 155]). In this sense, the concept of Waypoint Enforcement is grounded on previous experience.

6.6 Summary

SDN promises to ease network management through principled network orchestration. However, it is nearly impossible to fully upgrade an existing legacy network to an SDN in a single operation.

Accordingly, we have developed Panopticon, an enterprise network architecture realizing the benefits of a logical SDN control plane from a transitional network which combines legacy devices and SDN switches. Our evaluation highlights that our approach can deeply extend SDN capabilities into existing legacy networks. By upgrading between 30 to 40 of the hundreds of distribution switches in a large enterprise network, it is possible to realize the network as an SDN, without violating reasonable resource constraints. Our results motivate the argument, that partial SDN deployment may indeed be an appropriate mid-term operational strategy for enterprise networks. Our simulation source code is available at `http://panoptisim.badpacket.in`.

7

Conclusion and Outlook

The operational challenges inherent to network management and troubleshooting have provided a clear and powerful motivation for rethinking the traditional organization of network control. With the recent emergence of Software Defined Networking, new opportunities exist to refactor the network control plane, and introduce principled approaches to network management and troubleshooting. In the following sections, we summarize the contributions this thesis has made toward leveraging these new abstractions toward improving network management and troubleshooting and toward supporting their wider adoption and deployment. We then identify future directions for research based on the outcome of this thesis.

7.1 Summary of Contributions

In this thesis, we present the following contributions, addressing network management challenges through principled approaches to network design, planning, management, and troubleshooting.

In **Chapter 3**, we investigate and characterize key state distribution design trade-offs arising from the "logically centralized" network control plane. We characterize the state exchange points in a distributed SDN control plane and identify two key state distribution trade-offs. We simulate these exchange points in the context of an existing SDN load balancer application. We evaluate the impact of inconsistent global network view on load balancer performance and compare different state management approaches. Our results suggest that SDN control state inconsistency

significantly degrades performance of logically centralized control applications agnostic to the underlying state distribution.

In **Chapter 4**, we identify the problem of consistent, distributed policy composition in the logically centralized control plane. We propose an approach, **Software Transactional Networking**, to enable distributed, concurrent, multi-authorship of network-policy subject to key consistency and liveness properties. We propose an elegant policy composition abstraction based on a *transactional interface* with *all-or-nothing* semantics: a policy update is either *committed*, in which case the policy is guaranteed to compose consistently over the entire network and the update is installed in its entirety, or *aborted*, in which case, no packet is affected by it. Consequently, the control application logic is relieved from the cumbersome and potentially error-prone synchronization and locking tasks, and control applications are kept light-weight. We sketch a simple implementation of the transactional synchronization: our approach is based on fine-grained locking on network components and avoids complex state machine replication.

In **Chapter 5**, we introduce OFRewind, a novel troubleshooting tool, which leverages the logically centralized control plane to realize the abstraction of network-wide record and replay troubleshooting. We present the design of **OFRewind**, which enables *scalable*, *multi-granularity*, *temporally consistent recording* and *coordinated replay* in a network, with fine-grained, dynamic, centrally orchestrated control over both record and replay. Through *in-situ* case studies, we illustrate how OFRewind can be used to reproduce software errors, identify data-path limitations, and locate configuration errors.

In **Chapter 6**, we propose an architecture and methodology, **Panopticon**, to enable the incremental deployment of the logically centralized network control plane into existing enterprise networks, exposing the abstraction of a logical Software Defined Network. We present the design and implementation of Panopticon, and evaluate its feasibility through testbed, emulation, and simulation approaches. Our results suggest that our approach can deeply extend SDN capabilities into existing legacy enterprise networks, providing a principled approach for incremental SDN deployment and an avenue for wider adoption of network management approaches leveraging SDN.

7.2 Future Research Directions and Open Problems

Through the research work presented in this thesis, we have opened new questions that can potentially define future research directions. We revisit these open problems individually according to contribution:

The Logically Centralized Control Plane: Our study of the network load balancing application presents two control plane state distribution design trade-offs for one, specific traffic-control application. This application makes very specific assumptions about the timescales and rates of change in path-selection metrics relative to the timescales on which those changes are communicated across the network control plane; namely that the timescales are closely aligned. As a continuation of this study, it would be appropriate to systematically and more generally characterize these timescales in different network environments. Alternatively, this work could be extended by defining different safety properties over the traffic forwarding behavior, and examining how those properties may be guaranteed or violated given different assumptions about the control plane state distribution model.

Software Transactional Networking: The approach which we propose for consistent policy composition under concurrency present opportunities to extend our model to evaluate new trade-offs as well as investigate systems and implementation-level problems.

There are many ways to implement the *read-modify-write* primitive giving access to the network switches. At a high level, the implementation design depends on how the rights of accessing a switch are distributed across the controllers. If there is a single controller that is allowed to modify the configuration of the switch, then we can implement the *read-modify-write* primitive at the controller, so that it is responsible for the physical installation of the composed policy. An alternative solution is to provide a lock abstraction on the switch itself. For example, a weak form of locking can be achieved by maintaining a single *test-and-set* bit at a switch. We believe that even more fine-grained locking mechanisms are possible, where, *e.g.*, instead of acquiring locks on entire switches, transactions synchronize on *ports* only. Understanding the effects of design parameters such as lock granularity on transaction completion may inform the design of such transactional middleware.

The model upon which we define the *Consistent Policy Composition* problem also makes certain assumptions about the nature and types of failures that occur in both the data plane and the control plane. As more complex failure modes are incorporated into the model, there is the need to re-evaluate the minimal requirements (e.g., *TAG Complexity*) that enable consistent policy composition.

OFRewind: The primary goal of **OFRewind** is to assist network operators in the process of localizing the cause of faults and problematic network behaviors. To this end, **OFRewind** is designed to provide visibility into network data plane and control plane events.

Given that approaches to SDN deployment and operation continue to evolve, it is appropriate to revisit the question of how changes to SDN deployment may affect the visibility into such network events, assuming the proxy-based approach taken

by **OFRewind**. To illustrate one such example, many SDN deployments are realized using an edge-only deployment approach (As described in Section 6.1.1), one may ask what are the consequences of such a deployment scenario for the visibility and reproducibility of the symptoms associated with representative troubleshooting scenarios.

Another important open problem to be addressed when performing replay debugging in any production network is the need to guarantee that resource constraints are respected, *e.g.*that service level agreements are not in any way violated during troubleshooting. These constraints must be taken into consideration to realize a system which can ensure that its own behavior does not increase disruption or damage to the running system during the process of troubleshooting.

OFRewind presents various operational trade-offs, emerging from operational parameters governing traffic record and replay. These present opportunities for future investigations into the broader effectiveness and practical limitations of such replay-based troubleshooting approaches. One such example may include the trade-off between recording and replay timing accuracy and symptom reproducibility.

Incremental SDN Deployment: This work is concerned in part with understanding and overcoming the challenges to SDN adoption into existing networks. As we focus largely on leveraging the mechanisms of legacy Ethernet networks, a promising avenue for extending this research would involve more fundamentally integrating our architecture with distributed routing protocols running in parallel in the network. Along these lines, there are ample opportunities to more systematically explore and characterize the fault-tolerance properties of networks that run both distributed IP routing protocols along side SDN control.

As we focus on enterprise networks, we leverage their endemic topological and spatial properties to help ensure scalability and feasibility of our approach. Further investigation of our methods are required to understand their applicability to be generalized to other network types.

Finally, we observe that specific objectives and constraints for planning an SDN deployment are likely to depend on practical contextual factors such as hardware lifecycle management, support contracts and SLAs, long-term demand evolution and more. Unfortunately, these factors are rather qualitative, vary across environments, and are hard to generalize. Nevertheless such factors must be understood to fully address the challenges of SDN deployment in the wild.

Acknowledgements

Having reached this point, I can't help but recall (for whatever reason) the final scene from the Alejandro Jodorowsky Film, *The Holy Mountain*. In this scene, the pilgrims are seated around a large table, having arrived at their destination. Without warning, the Alchemist played by Jodorowsky casually breaks the fourth wall ("Zoom out, Cameraman!") to reveal that they are all but actors on a movie set. One reason why I like this scene is that it is truly, quite absurd, as is the entirety of the movie for that matter. However it effectively puts the characters' journey into perspective and also provides a nice transition out of the film world, back into the real world.

In its own way, this thesis and the research that went into making it has been a wonderful and, at times, crazy, absurd journey with diverse characters and a plot of unexpected twists and turns – not to mention a pretty crazy-looking hat at the end (Thanks INETers!). Thankfully, this thesis never took on quite the same level of surrealism as that particular Jodorowsky film.

My story here would never have begun without the support of my family, who encouraged my decision to move to Germany and work and live abroad. Over the last few years, I've also had the great fortune of making wonderful friends and working with exceptional colleagues in Berlin and in the INET and TU Berlin communities. I'd like to thank each and every one of them here and now... and so I will, with one run-on sentence per person.

I'd like to thank Anja for inviting me to join her research group, for investing her time and energy to act as my doctoral advisor over the last few years, and for (among other things) fostering a "culture of precision" which I admire, deeply appreciate, and will continue to develop going forward. I'd like to thank Marco for his role in advising my doctoral research, for his creative, pragmatic, and constructive support at all hours, as well as for introducing humor (virtual government, anyone?) and an antidote to sleep deprivation (@5am: we have plots!) into the mix. Thanks to Stefan for widening my research horizon and bringing formal methods and optimization into my toolbox. I'd like to thank Andi and Harald for getting me bootstrapped into and hooked on the messy world of networking and systems research, for sharing with me the zen of building systems and recovering from disasters, and for being all-around ninjas from whom I learned a bucket-load. Thanks to Ruben and Cigdem

for providing perspective on research-life and for helping me find my footing during my master thesis.

The reader will notice that we now begin a new paragraph, just because. I want to thank Seba and Florian for working together with me in the Routerlab, helping me to keep things running technically, but most importantly, for pushing me to mature as a team member, listener, manager, and sysadmin. Thanks to Nadi for the fun times in CA, HI (I'm glad we listened to that quarter), and the objective feedback on research. Thanks Ingmar for the ultimate car rental return experience and for sharing your perspective and experiences on taking research into business. Thanks to Lalith for good research discussions, internet meme distractions, and for adding life to the 3rd floor office. Thanks Srivatsan for the discussions and input on our STN work but also just in general. Thanks to Oliver, Benjamin, Juhoon, Steve, Julius, Enric, Philipp, Philipp, Thomas, Thomas, Carlo, Arne, Johannes and anyone else I might have missed for keeping INET excellent.

Thanks Fabian, Gregor, Grsch, Amir, Mustafa, George, Doris, Joerg, Vlad, Wolfgang, Jan, Ben, Bernhard, Vinay, Oliver, and any other member of the group I have forgotten who all set the pace and defined the culture of INET around the time I showed up. Thanks Arno, Seba, Fabian, Bernd, Paul, Julius, Ho for working on fun and interesting student projects and theses. Thanks Rainer, Jan, and Thorsten for keeping systems running in INET and thanks to Birgit and Britta for organizing "the chaos."

Thank you to Rob Sherwood and J.P. Seifert both for taking the time to serve on my committee and for taking part in my theses defense.

Thanks again to my friends and family for tolerating my absurd working hours, disappearances, deadlines, absences, antisocial or inexplicable behavior, and any other research-rooted problems that I probably should've appologized for sooner, but never did. Thanks Oscar, for being an excellent infant and for letting me get all this stuff done without significant disruption. Thanks Constanze for taking care of Oscar, and for keeping him excellent and for letting me get all this stuff done :-). You guys are the best!

I owe a huge thank you to my colleagues Michael Martinides, Fabian Schaffert, Amr Osman, and of course Marco and Stefan for helping me to get our research off of paper, into a venture, out the hangar door and onto the runway. I'll leave it at that for now, let's get this thing into the air.

List of Figures

List of Tables

Bibliography

[1] Cacti: Rrd-based monitoring. http://cacti.net/what_is_cacti.php.

[2] Endace Network Monitoring. http://www.endace.com/.

[3] EU Project Ofelia. http://www.fp7-ofelia.eu/.

[4] Gigamon Network Monitoring. http://www.gigamon.com/.

[5] IETF Working Group Forces. http://datatracker.ietf.org/wg/forces/charter/.

[6] Iperf performance benchmarking tool. http://iperf.sourceforge.net.

[7] Manage Devices Through a 'Single Pane of Glass'. http://www.microsoft.com/en-us/news/features/2012/apr12/wedmfea.aspx.

[8] NEC Programmable Networking Solutions. http://www.necam.com/PFlow/.

[9] Network virtualization platform (white paper). http://nicira.com/en/network-virtualization-platform.

[10] nmap-nping project. http://nping.sourceforge.net.

[11] OFRewind Code. http://www1.icsi.berkeley.edu/~andi/ofrewind/.

[12] POX Controller. http://noxrepo.org.

[13] Rrdtool: time series data graphing. http://oss.oetiker.ch/rrdtool/.

[14] tcpdump. http://www.tcpdump.org/.

[15] tcpreplay. http://tcpreplay.synfin.net/.

[16] United states government iaas policy. http://www.gsa.gov/portal/content/112063.

[17] VMWare NSX. http://bit.ly/1iQZzDj.

[18] Campus Networks Reference Architecture, 2010. http://juni.pr/1iR0vaZ.

[19] National survey on data center outages, September 2010. http://bit.ly/12rEs3u.

[20] Openflow and sdn: Networking's future?, Oct 2011. http://bit.ly/rkgyGw.

[21] Student email to switch to gmail beginning in august, May 2012. http://dailyprincetonian.com/news/2012/05/student-email-to-switch-to-gmail-beginning-in-august/.

[22] How kempinski hotels used cloud to improve staff efficiency, Jan. 2013. http://www.cloudpro.co.uk/node/5207.

[23] The promise of software defined networking, June 2013. http://bit.ly/1czc8LZ.

[24] M. Aguilera et al. Performance debugging for distributed systems of black boxes. In *Proc. ACM SOSP* (New York, NY, USA, 2003), ACM, pp. 74–89.

[25] Agarwal, S., Kodialam, M., and Lakshman, T. Traffic engineering in software defined networks. In *Proc. IEEE INFOCOM* (2013).

[26] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM* (2008).

[27] Alimi, R., Wang, Y., and Yang, Y. R. Shadow configuration as a network management primitive. In *Proc. ACM SIGCOMM* (2008).

[28] Altekar, G., and Stoica, I. Odr: Output-deterministic replay for multicore debugging. In *Proc. ACM SOSP* (2009).

[29] Altekar, G., and Stoica, I. Dcr: Replay debugging for the datacenter. Tech. Rep. UCB/EECS-2010-74, UC Berkeley, 2010.

[30] Altekar, G., and Stoica, I. Focus replay debugging effort on the control plane. Tech. Rep. UCB/EECS-2010-88, UC Berkeley, 2010.

[31] Anand, A., and Akella, A. Netreplay: a new network primitive. In *Proc. HOTMETRICS* (New York, NY, USA, 2009), vol. 37, ACM, pp. 14–19.

[32] Anderson, E., and Arlitt, M. Full Packet Capture and Offline Analysis on 1 and 10 Gb/s Networks. Tech. Rep. HPL-2006-156, HP Labs, 2006.

[33] Bailis, P., Venkataraman, S., Hellerstein, J. M., Franklin, M., and Stoica, I. Probabilistically bounded staleness for practical partial quorums. Tech. Rep. UCB/EECS-2012-4, EECS Department, University of California, Berkeley, Jan 2012.

[34] Ballard, J. R., Rae, I., and Akella, A. Extensible and scalable network monitoring using opensafe. In *Proc. INM/WREN* (Berkeley, CA, USA, 2010), USENIX Association, pp. 8–8.

[35] Barroso, L. A., and Hölzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[36] Benson, T., Akella, A., Shaikh, A., and Sahu, S. Cloudnaas: A cloud networking platform for enterprise applications. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 8:1–8:13.

[37] BigSwitch Networks. `http://www.bigswitch.com`.

[38] Bitincka, L., Ganapathi, A., and Zhang, S. Experiences with workload management in splunk. In *Proceedings of the 2012 Workshop on Management of Big Data Systems* (New York, NY, USA, 2012), MBDS '12, ACM, pp. 25–30.

[39] Bolte, M., Sievers, M., Birkenheuer, G., Niehörster, O., and Brinkmann, A. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe* (3001 Leuven, Belgium, Belgium, 2010), DATE '10, European Design and Automation Association, pp. 574–579.

[40] Canini, M., Kuznetsov, P., Levin, D., and Schmid, S. The Case for Reliable Software Transactional Networking. *CoRR abs/1305.7429* (2013). `http://arxiv.org/abs/1305.7429`.

[41] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE way to test openflow applications. In *Proc. 9th USENIX conference on Networked Systems Design and Implementation (NSDI)* (2012).

[42] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: taking control of the enterprise. In *Proc. ACM SIGCOMM* (2007).

[43] Casado, M., Garfinkel, T., Akella, A., Freedman, M. J., Boneh, D., McKeown, N., and Shenker, S. SANE: a protection architecture for enterprise networks. In *USENIX Security Symposium* (2006).

[44] Casado, M., Koponen, T., Shenker, S., and Tootoonchian, A. Fabric: a retrospective on evolving SDN. In *HotSDN* (2012).

[45] CBench - Controller Benchmarker. `www.openflowswitch.org/wk/index.php/Oflops`.

[46] Childers, B. Virtualization shootout: Vmware server vs. virutalbox vs. kvm. *Linux J. 2009*, 187 (Nov. 2009).

[47] Cisco. Campus Network for High Availability Design Guide, 2008. `http://bit.ly/1ffWkzT`.

[48] Clemm, A. *Network Management Fundamentals.* Cisco Press, 2006.

[49] Cooke, E., Myrick, A., Rusek, D., and Jahanian, F. Resource-aware Multi-format Network Security Data Storage. In *Proc. SIGCOMM LSAD workshop* (2006).

[50] Crimson, H. Harvard college to switch email provider to gmail. `http://www.thecrimson.com/article/2011/7/19/email-gmail-harvard-students/`.

[51] D. Geels et al. Replay debugging for distributed applications. In *Proc. Usenix Tech Conf* (2006).

[52] David L. Tennenhouse et al. *Towards an Active Network Architecture.* Computer Communication Review, 1996.

[53] Desnoyers, P., and Shenoy, P. J. Hyperion: High Volume Stream Archival for Retrospective Querying. In *Proc. 2007 USENIX Technical Conf* (2007).

[54] F. Reiss et al. Enabling Real-Time Querying of Live and Historical Stream Data. In *Proc. Statistical & Scientific Database Management* (2007).

[55] Feamster, N., Balakrishnan, H., Rexford, J., Shaikh, A., and van der Merwe, J. The case for separating routing from routers. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture* (New York, NY, USA, 2004), FDNA '04, ACM, pp. 5–12.

[56] Feamster, N., Rexford, J., and Zegura, E. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev. 44*, 2 (Apr. 2014), 87–98.

[57] Ferguson, A. D., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. Hierarchical Policies for Software Defined Networks. In *HotSDN* (2012).

[58] Flach, T., Katz-Bassett, E., and Govindan, R. Quantifying violations of destination-based forwarding on the internet. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference* (New York, NY, USA, 2012), IMC '12, ACM, pp. 265–272.

[59] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: a network programming language. In *Proc. ACM SIGPLAN ICFP* (New York, NY, USA, 2011), ICFP '11, ACM, pp. 279–291.

[60] Francois, P., Shand, M., and Bonaventure, O. Disruption free topology reconfiguration in ospf networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE* (May 2007), pp. 89–97.

[61] GENI: Global Environment for Network Innovations. `http://www.geni.net`.

[62] Gonzalez, J. M., Paxson, V., and Weaver, N. Shunting: A Hardware/Software Architecture for Flexible, High-performance Network Intrusion Prevention. In *Proc. 14th ACM CCS* (2007).

[63] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A scalable and flexible data center network. In *Proc. ACM SIGCOMM* (2009).

[64] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A clean slate 4D approach to network control and management. *SIGCOMM CCR 35* (October 2005), 41–54.

[65] Guérin, R. A., and Orda, A. QoS routing in networks with inaccurate information: theory and algorithms. *IEEE/ACM Trans. Netw. 7*, 3 (June 1999), 350–364.

[66] Gupta, D., Vishwanath, K., and Vahdat, A. Diecast: Testing distributed systems with an accurate scale model. In *Proc. USENIX NSDI* (2008).

[67] Gupta, D. et al. To infinity and beyond: Time warped network emulation. In *Proc. ACM SOSP* (2005).

[68] Hagberg, A. A., Schult, D. A., and Swart, P. J. Exploring network structure, dynamics, and function using NetworkX. In *SciPy2008* (Pasadena, CA USA, Aug. 2008), pp. 11–15.

[69] Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 253–264.

[70] Handigol, N., Seetharaman, S., Flajslik, M., McKeown, N., and Johari, R. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. SigComm Demonstration, 2009.

[71] Hassas Yeganeh, S., and Ganjali, Y. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *HotSDN* (2012).

[72] Heise. Security bericht: Innenministerium plant sichere bundes cloud, 2012.

[73] Heller, B., Sherwood, R., and McKeown, N. The Controller Placement Problem. In *HotSDN* (2012).

[74] Herlihy, M., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[75] Hower, D. R., Montesinos, P., Ceze, L., Hill, M. D., and Torrellas, J. Two hardware-based approaches for deterministic multiprocessor replay. *Comm. of the ACM 52*, 6 (2009), 93–100.

[76] IDC. European Cloud Professional Services, Cloud Management Services, and Hosted Private Cloud 2012–2016 Forecast, 2012. `http://www.idc.com/getdoc.jsp?containerId=QL05U`.

[77] IEEE. Ieee 802.1q: Virtual lans. `www.ieee802.org/1/pages/802.1Q.html`.

[78] IEEE. Ieee 802.1s: Multiple spanning trees. `www.ieee802.org/1/pages/802.1s.html`.

[79] IEEE. Ieee 802.1x: Port based network access control. `www.ieee802.org/1/pages/802.1x-2001.html`.

[80] IEEE. Ieee 802.3: Ethernet working group. `www.ieee802.org/3/`.

[81] IEEE. Ieee 802.3ad: Link aggregation. `www.ieee802.org/3/ad/index.html`.

[82] IEEE. Ieee 802.3bd: Mac flow control. `http://www.ieee802.org/3/bd/index.html`.

[83] IETF. Rfc 1052: Iab recommendations for the development of internet network management standards. `https://tools.ietf.org/html/rfc1052`.

[84] IETF. Rfc 1067: Simple network management protocol. `https://tools.ietf.org/html/rfc1067`.

[85] IETF. Rfc 1142: Osi is-is intra-domain routing protocol. `https://tools.ietf.org/html/rfc1142`.

[86] IETF. Rfc 1195: Use of osi is-is for routing in tcp/ip and dual environments. `https://tools.ietf.org/html/rfc1195`.

[87] IETF. Rfc 2328: Ospf version 2. `https://tools.ietf.org/html/rfc2328`.

[88] IETF. Rfc 4251: The secure shell protocol. `https://tools.ietf.org/html/rfc4251`.

[89] IETF. Rfc 4271: A border gateway protocol 4 (bgp-4). `https://tools.ietf.org/html/rfc4271`.

[90] IETF. Rfc 4381: Definitions of managed objects for bridges with rapid spanning tree protocol. `https://tools.ietf.org/html/rfc4318`.

[91] IETF. Rfc 7011: Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. `https://tools.ietf.org/html/rfc7011`.

[92] IETF. Rfc 854: Telnet protocol specification. https://tools.ietf.org/html/rfc854.

[93] IETF. Rfc 869: Host monitoring protocol. https://tools.ietf.org/html/rfc869.

[94] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hoelzle, U., Stuart, S., and Vahdat, A. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. ACM SIGCOMM* (2013).

[95] John, J. P., Katz-Bassett, E., Krishnamurthy, A., Anderson, T., and Venkataramani, A. Consensus routing: the internet as a distributed system. In *Proc. USENIX NSDI* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 351–364.

[96] Joseph, D., Kannan, J., Kubota, A., Lakshminarayanan, K., Stoica, I., and Wehrle, K. OCALA: An architecture for supporting legacy applications over overlays. In *Proc. USENIX NSDI* (2006).

[97] Kang, N., Liu, Z., Rexford, J., and Walker, D. Optimizing the "one big switch" abstraction in software-defined networks. In *Proc. ACM CONEXT* (2013).

[98] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: static checking for networks. In *Proc. USENIX NSDI* (2012), USENIX Association, pp. 9–9.

[99] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 15–28.

[100] Kim, C., Caesar, M., and Rexford, J. Floodless in Seattle: A scalable ethernet architecture for large enterprises. In *Proc. ACM SIGCOMM* (2008).

[101] Kim, S., Park, S., Yun, J., and Lee, Y. Automated continuous integration of component-based software: An industrial experience. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2008), ASE '08, IEEE Computer Society, pp. 423–426.

[102] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating system with time-traveling virtual machines. In *Proc. Usenix Tech Conf* (2005).

[103] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. Onix: A distributed control platform for large-scale production

networks. In *Proc. USENIX OSDI* (Berkeley, CA, USA, 2010), USENIX Association, pp. 1–6.

[104] Kotronis, V., Dimitropoulos, X., and Ager, B. Outsourcing the Routing Control Logic: Better Internet Routing Based on SDN Principles. In *HotNets* (2012).

[105] Lackey, S. Cfengine for configuration management. *Linux J. 2008*, 168 (Apr. 2008).

[106] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM 21*, 7 (1978).

[107] Levin, D., Canini, M., Schmid, S., and Feldmann, A. Panopticon: Reaping the benefits of partial sdn deployment in enterprise networks. Tech. rep., Technical Report TU Berlin `http://bit.ly/1n1U3LD`, 2013.

[108] Levin, D., Wundsam, A., Heller, B., Handigol, N., and Feldmann, A. Hotsdn 2012 sdn control simulator. `http://github.com/cryptobanana/sdnctrlsim`.

[109] Maier, G., Sommer, R., Dreger, H., Feldmann, A., Paxson, V., and Schneider, F. Enriching network security analysis with time travel. In *Proc. ACM SIGCOMM* (New York, NY, USA, 2008), ACM, pp. 183–194.

[110] Markopoulou, A., Iannaccone, G., Bhattacharyya, S., Chuah, C.-N., Ganjali, Y., and Diot, C. Characterization of failures in an operational ip backbone network. *IEEE/ACM Trans. Netw. 16*, 4 (aug 2008), 749–762.

[111] Markopoulou, A., Iannaccone, G., Bhattacharyya, S., nee Chuah, C., and Diot, C. Characterization of failures in an ip backbone. In *In IEEE Infocom2004, Hong Kong* (2004).

[112] McGrath, K. P., and Nelson, J. Monitoring & Forensic Analysis for Wireless Networks. In *Proc. Conf. on Internet Surveillance and Protection* (2006).

[113] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR 38*, 2 (2008), 69–74.

[114] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. Openflow: enabling innovation in campus networks. *SIGCOMM CCR 38* (March 2008), 69–74.

[115] Micheel, J., Donnelly, S., and Graham, I. Precision timestamping of network packets. In *Proc. ACM IMW* (2001).

[116] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing Software Defined Networks. In *NSDI* (2013).

[117] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing Software Defined Networks. In *NSDI* (2013).

[118] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proc. ACM ASPLOS* (New York, NY, USA, 2009), ACM, pp. 73–84.

[119] The Multi Router Traffic Grapher. `http://oss.oetiker.ch/mrtg/`.

[120] Mudigonda, J., Yalagandula, P., Mogul, J., Stiekes, B., and Pouffary, Y. NetLord: a scalable multi-tenant network architecture for virtualized datacenters. In *Proc. ACM SIGCOMM* (2011).

[121] Nagios: IT Infrastructure Monitoring. `http://www.nagios.org`.

[122] Cisco IOS NetFlow. `www.cisco.com/go/netflow`.

[123] Niranjan Mysore, R., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., and Vahdat, A. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *Proc. ACM SIGCOMM* (2009).

[124] NoviFlow. 1248 Datasheet. `http://bit.ly/1baQd0A`.

[125] NOX - An OpenFlow Controller. `www.noxrepo.org`.

[126] Nunes, B., Mendonca, M., Nguyen, X.-N., Obraczka, K., and Turletti, T. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. In *Communications Surveys and Tutorials, IEEE* (2014).

[127] P. Bahl et al. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. ACM SIGCOMM* (New York, NY, USA, 2007), ACM, pp. 13–24.

[128] Pang, R., Allman, M., Bennett, M., Lee, J., Paxson, V., and Tierney, B. A first look at modern enterprise traffic. In *Proc. ACM IMC* (2005).

[129] Perlman, R. An algorithm for distributed computation of a spanningtree in an extended lan. *SIGCOMM Comput. Commun. Rev. 15*, 4 (Sept. 1985), 44–53.

[130] Perlman, R., Eastlake, D., Dutt, D. G., Gai, S., and Ghanwani, A. Rbridges: Base protocol specification. In *Technical report, IETF* (2009).

[131] Phaal, P., Panchen, S., and McKee, N. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks, 2001.

[132] Qazi, Z., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM* (2013).

[133] Quagga Routing Suite. `http://www.quagga.net`.

[134] R. Fonseca et al. X-trace: A pervasive network tracing framework. In *Proc. USENIX NSDI* (2007).

[135] Radio, N. P. Colleges turn from in-house email to free gmail, April 2010.

[136] Raza, S., Huang, G., Chuah, C.-N., Seetharaman, S., and Singh, J. P. Measurouting: a framework for routing assisted traffic monitoring. *IEEE/ACM Trans. Netw. 20*, 1 (Feb. 2012), 45–56.

[137] Reitblatt, M., Canini, M., Guha, A., and Foster, N. FatTire: Declarative Fault Tolerance for Software-Defined Networks. In *HotSDN* (2013).

[138] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for Network Update. In *SIGCOMM* (2012).

[139] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. Consistent updates for software-defined networks: change you can believe in! In *Proc. ACM HotNets Workshop* (New York, NY, USA, 2011), ACM, pp. 7:1–7:6.

[140] Rexford, J., Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Xie, G., Zhan, J., and Zhang, H. Network-Wide Decision Making: Toward A Wafer-Thin Control Plane. In *Proc. ACM HotNets Workshop* (2004).

[141] Reynolds, P., Wiener, J. L., Mogul, J. C., Aguilera, M. K., and Vahdat, A. Wap5: black-box performance debugging for wide-area systems. In *Proc. ACM WWW* (2006).

[142] Saito, Y., and Shapiro, M. Optimistic replication. *ACM Comput. Surv. 37*, 1 (Mar. 2005), 42–81.

[143] Shaikh, A., Rexford, J., and Shin, K. G. Evaluating the impact of stale link state on quality-of-service routing. *IEEE/ACM Trans. Netw. 9*, 2 (Apr. 2001), 162–176.

[144] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing* (1997).

[145] Shenker, S. The Future of Networking, and the Past of Protocols., 2011. Open Networking Summit, `https://www.youtube.com/watch?v=YHeyuD89n1Y`.

[146] Shenker, S. Software defined networking at the crossroads, 2013. Stanford University Seminar, `https://www.youtube.com/watch?v=WabdXYzCAOU`.

[147] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making middleboxes someone else's problem: Network processing as a cloud service. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 13–24.

[148] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. Can the production network be the testbed? In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), OSDI'10, USENIX Association, pp. 1–6.

[149] Singla, A., Hong, C.-Y., Popa, L., and Godfrey, P. B. Jellyfish: networking data centers randomly. In *Proc. USENIX NSDI* (2012).

[150] Sommers, J., Barford, P., and Eriksson, B. On the prevalence and characteristics of mpls deployments in the open internet. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 445–462.

[151] Sommers, J., Barford, P., and Eriksson, B. On the prevalence and characteristics of mpls deployments in the open internet. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 445–462.

[152] Stephens, B., Cox, A., Felter, W., Dixon, C., and Carter, J. PAST: Scalable Ethernet for data centers. In *Proc. ACM CONEXT* (2012).

[153] Sun, X., Sung, Y.-W. E., Krothapalli, S. D., and Rao, S. G. A systematic approach for evolving vlan designs. In *Proc. IEEE INFOCOM* (2010), pp. 1451–1459.

[154] Sung, Y.-W. E., Rao, S. G., Xie, G. G., and Maltz, D. A. Towards systematic design of enterprise networks. In *Proc. ACM CONEXT* (2008).

[155] Takahashi, N., and Smith, J. M. Hybrid hierarchical overlay routing (hyho): Towards minimal overlay dilation. *IEICE Transactions 87-D*, 12 (2004), 2586–2593.

[156] Tootoonchian, A., and Ganjali, Y. Hyperflow: a distributed control plane for openflow. In *Proc. INM/WREN* (Berkeley, CA, USA, 2010), USENIX Association, pp. 3–3.

[157] Tootoonchian, A., Ghobadi, M., and Ganjali, Y. OpenTM: traffic matrix estimator for OpenFlow networks. In *Proc. PAM* (Berlin, Heidelberg, 2010), PAM'10, Springer-Verlag, pp. 201–210.

[158] Vanbever, L., and Vissicchio, S. Enabling SDN in Old School Networks with Software-Controlled Routing Protocols. In *Open Networking Summit (ONS)* (2014).

[159] Vanbever, L., Vissicchio, S., Pelsser, C., Francois, P., and Bonaventure, O. Seamless network-wide igp migrations. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 314–325.

[160] Wang, R., Butnariu, D., and Rexford, J. Openflow-based server load balancing gone wild. In *Proc. USENIX HotICE* (2011).

[161] Wetherall, D. J. Service introduction in an active network. Tech. rep., M.I.T. PhD Thesis, 1999.

[162] Wetherall, D. J., Guttag, J. V., and Tennenhouse, D. L. Ants: A toolkit for building and dynamically deploying network protocols. In *in IEEE OPENARCH* (1998).

[163] Williams, D., Jamjoom, H., and Weatherspoon, H. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 113–126.

[164] Wool, A. A Quantitative Study of Firewall Configuration Errors. *Computer 37*, 6 (2004), 62–67.

[165] Wool, A. Trends in Firewall Configuration Errors: Measuring the Holes in Swiss Cheese. *IEEE Internet Computing 14*, 4 (2010), 58–65.

[166] Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proc. USENIX ATC* (June 2011).

[167] Wundsam, A., Mehmood, A., Feldmann, A., and Maennel, O. Network Troubleshooting with Mirror VNets. In *Proc. IEEE Globecom 2010 FutureNet-III workshop* (December 2010).

[168] Yan, H., Maltz, D. A., Ng, T. S. E., Gogineni, H., Zhang, H., and Cai, Z. Tesseract: A 4D Network Control Plane. In *Proc. USENIX NSDI* (2007).

[169] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. Automatic test packet generation. In *Proc. ACM CONEXT* (2012).