USING PACKET HISTORIES TO TROUBLESHOOT NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Nikhil Ashok Handigol
June 2013

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nick McKeown, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Ramesh Johari

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Mazieres

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

Abstract

Operating networks is hard. When a network goes down, network administrators have only a rudimentary set of tools at their disposal to track down the root cause of the outage. As networks have become more complicated, with more network protocols modifying the forwarding behavior below, and more application types running above, the debugging toolkit has remained essentially unchanged, with little or no innovation in years. Today, skilled network administrators frequently use manual, heuristic-driven procedures to configure and maintain networks. Humans are involved almost every time something goes wrong, and we are still far from an era of automated troubleshooting.

In this dissertation, I show how *packet histories*—the full story of every packet's journey through the network—can simplify network diagnosis. A packet history is the route a packet takes through a network, combined with the switch state and header modifications it encounters at each switch on the route. Using packet history as the core construct, I propose an abstraction for systematic network troubleshooting, a framework with which to express the observed error symptoms and pose questions to the network.

To demonstrate the usefulness of packet histories and the practical feasibility of constructing them, I built NetSight, an extensible platform that captures packet histories and enables applications to concisely and flexibly retrieve packet histories of interest. Atop NetSight I built four applications that illustrate its flexibility: an interactive network debugger, a live invariant monitor, a path-aware history logger, and a hierarchical network profiler.

On a single modern multi-core server, NetSight can process packet histories for the traffic of multiple 10 Gb/s links. For larger networks, NetSight scales linearly with additional servers. To scale it even further to bandwidth-heavy enterprises and datacenter networks, I present two optimized NetSight variants using straightforward additions to switch ASICs and hypervisor-based switches.

Acknowledgements

My stay at Stanford has been among the most memorable and satisfying periods of my life, largely due to the wonderful people whom I have gotten to know here. I would like to take this opportunity to thank them all.

First and foremost I would like to thank my adviser, Prof. Nick McKeown. Nick is a great embodiment of the Gandhian ideal "be the change you want to see in the world." He truly believes in having an impact and changing the world through ideas, and he passionately inculcates the same values in his students. He is a visionary, a great researcher, and a caring adviser. Few teachers have influenced my thinking as much as Nick has. It has been an honor to be his student.

Special thanks to Prof. Guru Parulkar for being a great guide and mentor throughout my Ph.D. Guru has been instrumental in bringing out the very best in me by setting high standards and instilling in me the confidence to meet them.

I have also had the good fortune of collaborating with Prof. Ramesh Johari and Prof. David Mazièes. I am in awe of Ramesh's ability to articulate the most abstract of ideas. I have deep respect for David's relentlessness in asking hard questions—he does not stop till the idea is perfectly clear to him as well as the student. I would also like to thank the other members of my orals committee—Prof. Nick Bambos and Prof. George Varghese—for providing valuable feedback that helped me refine this dissertation. David Erickson also carefully reviewed an early draft of this dissertation and provided useful comments and feedback.

Brandon Heller and Vimal Jeyakumar have been great collaborators on a number of projects, including the NetSight project. I have thoroughly enjoyed many stimulating discussions and "hack sessions" with them and have learned a lot from them over the years. Many thanks also to our fourth collaborator on the Mininet project, Bob Lantz.

I would also like to thank Mario Flajslik and Srini Seetharaman, with whom I worked a lot during the initial years of my Ph.D., especially on the Aster*x load-balancing project, where we explored the idea of building an "ideal" load-balancer which could jointly select both the server and the path for incoming requests in order to optimize service performance for the clients.

It has been a pleasure to be a part of the McKeown Group; my stay at Stanford has been a special experience primarily due to the wonderful interactions with past and present McKeown Group members: Guido Appenzeller, Adam Covington, Saurav Das, David Erickson, Glen Gibb, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, Rob Sherwood, David Underhill, Tatsuya Yabe, Kok-Kiong Yap, Yiannis Yiakoumis, and James Hongyi Zeng. I have learned so much from each and every one of them.

I would like to thank all my wonderful friends, both at Stanford and outside, for all their love and support throughout.

Last but not least, I would like to thank my family—my parents and my brother Nihal—for their support and encouragement through the highs and lows of my Ph.D. I dedicate this dissertation to their unconditional love and sacrifice. This Ph.D. is as much theirs as it is mine.

To my parents, Sandhya and Ashok,

&

To my brother, Nihal.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Networks today have reached "utility" status everywhere. Just as the utility of electricity is required to power our lights and equipment, and oil or gas insures our heat in the winter, the network infrastructure of an enterprise *must* always be in place to maintain its Internet access, phone system, and a host of other mission-critical applications.

However, operating networks is hard. A network is a distributed system whose overall behavior is governed by an internal state, called the forwarding state, that is distributed across switches and routers. The logic that manages this state, called the control plane, comprises multiple network applications, all changing the forwarding state simultaneously in complex, distributed, and unpredictable ways.

The network as designed today is essentially a black box. The exact forwarding state and how it is changing are all hidden under the hood. As a consequence, despite the increase in protocols modifying the forwarding state in a distributed fashion, the network debugging toolkit has remained essentially unchanged. When a network goes down, network administrators have only a rudimentary set of tools at their disposal (`traceroute`, `ping`, `SNMP`, `NetFlow`, `sFlow`) to track down the root cause of the outage. Network administrators have become "masters of complexity," [55] who use their skill and experience to divine the root cause of each bug. Humans are involved almost every time something goes wrong, and we are still far from an era of automated troubleshooting. This, in turn, leads to longer downtimes, higher operation costs, and slower adoption of new network applications.

As an example, consider the case of the “ghost machine” problem in the Gates Computer Science Building at Stanford. After every power outage in the building, a machine at an unknown location would come online with a conflicting, statically configured IP address, rendering another machine on the network unavailable. Our admin lacked the tools to time-efficiently track it down, and therefore, chose to log into the ghost to shut it off every time. This ghost machine was never found. Another problem encountered recently at Stanford was a broken WiFi handover, where a host lost its connectivity when it moved from one access point (AP) to another. The network admins debugged this issue by running pings, periodically logging switch flow table entries, and parsing logs of control communication. After hours of debugging, they diagnosed the (surprisingly simple) root cause: upon handover to the AP, the flow table entries in the upstream wired switch were not properly updated, sending incoming packets to the original AP.

1.2 Goal: Systematic Network Troubleshooting

Debugging networks is hard for a reason: the tools available today try to reconstruct the complex and distributed state of the network in an ad-hoc fashion, even as a variety of distributed protocols, such as L2 learning and L3 routing, are constantly changing that state. We could easily diagnose many network problems if we could ask the network about suspect traffic and receive an immediate answer. For example:

- “Host A cannot talk to Host B. Show me where packets from A intended for B are going, along with any packet header modifications.”

- “I don’t want loops in my network, even transient ones. Show me every packet that passes the same switch twice.”

- “Some hosts are failing to grab IP addresses. Show me what is happening to the DHCP traffic in the network.”

- “One port is experiencing congestion losses. Show me the traffic source contributing the most to the congestion.”

Unfortunately, we cannot “just ask” these questions today, as our network diagnosis tools (1) provide no way to pose such a question, and (2) lack access to the depth of information needed to return a useful answer. My goal is to *develop a systematic method of network troubleshooting* by addressing these two shortcomings.

1.3 Solution Requirements

A solution to achieve our goal of systematic network troubleshooting should have the following characteristics:

- It should be *versatile*. Network administrators and network application developers should be able to troubleshoot a wide range of frequently encountered problems.
- It should impose *minimal constraints* on the applications, ideally none.
- It should be *scalable* to a large enterprise or a datacenter network with hundreds of switches and multiple Gb/s of traffic.
- It should be *practical*. We should be able to implement it with only simple changes to the network hardware.

1.4 Solution Overview

My solution is based on the observation that most network bugs manifest themselves as errant packet behavior in the dataplane, i.e., when there is a problem, there is an observable symptom in the form of misbehaving packets. For example, when there is a forwarding loop in the network, packets take a cyclic path, and when there is a connectivity problem, packets do not reach their intended destination.

Building on this observation, my solution monitors packet-forwarding events and state changes in a consistent fashion, so that starting from error symptoms we can reconstruct the sequence of events that led to the errant behavior. The solution is divided into three main parts:

- An abstraction to express the observed error symptoms and pose questions to the network.
- A platform to monitor network events and collect the necessary information to support the abstraction.
- Applications built on top of the platform to troubleshoot real-world problems.

1.4.1 Packet History: The Troubleshooting Abstraction

A number of network diagnosis questions, including those mentioned in Section 1.1, could be answered with an omniscient view of every packet's journey through the network. I call this notion a *packet history.* More specifically,

Definition A packet history is the route a packet takes through a network, combined with the switch state and header modifications it encounters at each switch on the route.

A single packet history can be the "smoking gun" that tells us *why*, *when*, and *where* a network failed, evidence that would otherwise be hidden in gigabytes of control messages, flow records, and data-plane packet logs. To concisely and flexibly specify paths, switch state, and packet header fields in packet histories of interest, I developed a regular-expression-like language called *Packet History Filter* (PHF).

1.4.2 NetSight: The Troubleshooting Platform

To support the packet history abstraction, I built *NetSight*, an extensible platform to capture and filter packet histories of interest. With a view of every packet history in the network, NetSight supports both real-time and postmortem analysis via an API based on the Packet History Filter (PHF) language. NetSight assembles packet histories using *postcards*—event records created whenever a packet traverses a switch. Each postcard contains a copy of the complete packet header, the switch ID, the input/output ports, and a version of the switch forwarding state that is updated on every change. The challenge for any system offering packet histories is to efficiently process a stream of postcards into archived, queryable packet histories. Surprisingly, a *single* NetSight server suffices to assemble and store every packet history in many campus and enterprise networks. To support larger networks, NetSight scales out on general-purpose servers—increasing its assembly, query, and storage capabilities almost linearly with the number of processing cores, servers, and disks. The NetSight prototype that I built works by transparently interposing on the control channel between switches and controllers.

1.4.3 Troubleshooting Applications

Atop NetSight, I built four applications that illustrate its flexibility: (1) *ndb*, an interactive network debugger, (2) *netwatch*, a live network invariant monitor, (3) *netshark*, a network-wide packet history logger, and (4) *nprof*, a hierarchical network profiler. They all use the PHF-based NetSight API to query for packet histories of interest. The problems described previously are a small sample from the set of problems these applications can help solve.

1.5 Dissertation Scope

One “smoking gun” packet history can single-handedly confirm or disprove a hypothesis about the cause of a network problem by showing events that actually transpired in the network, along with all relevant states. This method of network analysis nicely complements techniques that model network behavior [35, 37]. Rather than *predicting* the behavior of the network on *hypothetical* packets, NetSight shows the *actual* behavior of the network on *real* packets. Thus, the scope of this dissertation spans:

1. a wide range of people associated with the development and operation of networks—network operators, control program developers, and switch implementers;

2. a variety of network problems—errors in firmware, hardware, or control protocols; and
3. a variety of network environments—enterprise, wide-area, and datacenter.

1.6 Dissertation Outline

In this first chapter, I have outlined my motivation for seeking a systematic network troubleshooting solution and my approach of creating the Packet History construct.

In Chapter 2, I will describe the abstraction for systematic network troubleshooting, a framework to express the observed error symptoms and pose questions to the network. The framework is centered around the packet history construct, the complete story of a packet's journey through the network. I will then describe Packet History Filter (PHF), a regular-expression-like language to flexibly express and retrieve packet histories of interest.

In Chapter 3, I will describe the design and implementation of NetSight, an extensible network troubleshooting platform to capture and filter packet histories of interest.

In Chapter 4, I will describe the four applications that I built on top of NetSight—an interactive network debugger, a live invariant monitor, a path-aware history logger, and a hierarchical network profiler—to help diagnose real-world problems. I will show how the applications enable diagnostics that would otherwise be impractical, time-consuming, or impossible for a network administrator using conventional tools.

In Chapter 5, I discuss the scalability and performance of NetSight, and present two optimized NetSight variants that will help it to scale to bandwidth-heavy enterprises and datacenter networks. NetSight-SwitchAssist moves postcard processing into switch ASICs, while NetSight-HostAssist spreads postcard and history processing among all virtualized servers.

In Chapter 6, I discuss the limitations of packet histories and those of NetSight, along with the opportunities to make troubleshooting a fundamental primitive of the network.

Finally, I describe related work in Chapter 7 before concluding in Chapter 8.

Chapter 2

A Troubleshooting Abstraction

2.1 Motivating Packet Histories

In order to build a systematic troubleshooting framework, we first need a construct to pose questions and describe the errors. A large class of network problems show up as errant behavior in the dataplane, packets showing unintended behavior such as not reaching their intended destination or going around in a loop. This errant behavior, in turn, is caused by the network state. These two observations lead us to the notion of a *packet history*.

In this chapter, I define packet histories, give examples of their utility in constructing network diagnosis tools, and show how Software-Defined Networking (SDN) can help us collect packet histories in a network.

Packet History Definition. A *packet history* tells the full story of a packet's journey through the network. More precisely, a packet history describes:

- *what* the packet looked like as it entered the network (headers)
- *where* the packet was forwarded (switches + ports)
- *how* it was changed (header modifications)
- *why* it was forwarded that way (matched flow/actions + flow table).

Figure 2.1 shows an example of a packet history.

```
packet [dl_src: 0x123, ...]:
     switch 1: { inport: p0, outports: [p1]
                 mods: [dl_dst -> 0x345]
                 matched flow: 23 [...]
                 matched table version: 3 }
     switch 2: { inport: p0, outports: [p2]
                 mods: []
                 matched flow: 11 [...]
                 matched table version: 7 }
     ...
     switch N: { inport: p0
                 table miss
                 matched table version: 8 }
```

Figure 2.1: A packet history shows the path taken by a packet along with the modifications and switch state encountered by it at each hop.

2.1.1 Why Packet Histories?

Put simply, packet histories provide direct evidence to diagnose network problems, covering what, where, how, and why. One “smoking gun” packet history can single-handedly confirm or disprove a hypothesis about the cause of a network problem by showing events that *actually* transpired in the network, along with all relevant states. The stream of packet histories enables diagnostics that would otherwise be impractical, time-consuming, or impossible for a network administrator using conventional tools—typically, a combination of control message logs, flow records [16, 48], and passively captured packets [4, 5, 10].

2.1.2 Bug Stories

To show exactly how packet histories can be used to solve real problems, I next walk through some of the problems encountered in the production network of the Gates Building at Stanford, which contains a mix of traditional Ethernet/VLAN-based and OpenFlow-based [45] switches.

Ghost machine

After every power outage in the building, a machine at an unknown location would come online with a statically assigned IP address that conflicted with another machine, rendering it unreachable. Every time, our local network admin, Charlie, would log into the "ghost machine" and shut it off, restoring access to the right machine. Charlie could have manually iteratively searched the MAC learning tables in switches to locate the port the machine is connected to, but the time required led him to use the SSH-shutdown shortcut instead.

With packet histories: Charlie could have used the packet histories with the conflicting IP address to locate the ghost machine and permanently shut it off. This machine, however, remained a ghost, and was never found.

Incomplete handover

In the OpenFlow portion of the network, when a WiFi host B moved from one access point to another while running a ping to another host A, the ICMP reply stopped reaching B. To debug this issue, the admin, Masa, inspected the flow tables of all the involved switches, as well as the logs of the control communication between the involved switches and the controller, and manually correlated the two. After multiple hours of debugging, he diagnosed the (surprisingly simple) root cause: after handover to the new WiFi access point, the flow table entries in the upstream wired switch were not updated for the ICMP reply flow, causing the ICMP reply packets to be sent to the old location of the WiFi host B.

With packet histories: Masa could have uncovered the wrong turn of the misdirected ICMP replies by requesting packet histories of all ICMP reply traffic sent from the IP address of host A.

Race condition bug in a switch

At one point, some users of the OpenFlow deployment observed unexpectedly poor Web browsing experience. Another admin, Srini, debugged the problem by dumping the TCP handshake packets—SYN and SYN-ACK—of HTTP sessions at both the clients and the servers using tcpdump, then correlated it with the control plane events using `wireshark`. The TCP handshake packets showed a surprisingly high drop rate. The correlation further revealed that the control plane events were as expected. The packets must not have been properly forwarded by the switches, even though the flow table of each switch in the network appeared to be accurate.

This loss of the TCP handshake packets caused a poor Web browsing experience, because the TCP session had to wait for a prolonged TCP timeout before proceeding further. Since this packet loss behavior was not consistent, Srini hypothesized that there was a race condition. With this race condition, a packet (the TCP SYN packet) arriving at the second-hop switch while the switch was in the middle of adding a flow table entry to forward it, was sometimes dropped. A delay box placed between the first and the second switch confirmed the diagnosis: packets that were delayed long enough to give the second switch time to finish adding the flow table entry showed no SYN drops.

With packet histories: Srini could have used packet histories of all HTTP traffic from a host to immediately know where the drops were occurring, as well as to immediately disprove the hypothesis about incorrect flow table configuration. He could have picked other hosts behind other switches to easily compare their drop rates, confirm that only one vendor switch suffered the race condition bug, and confirm a fix from the vendor. A model-based approach would not have been useful in diagnosing this problem.

FlowVisor Isolation Bugs

FlowVisor [56] is a control-plane proxy that “slices” a network into isolated regions, by port or by packet headers. Each slice can then be managed by a separate controller or network application. A load balancer application, Aster*x [25], ran in one such slice handling all HTTP packets sent to or received from the shared IP address of a set of replicated web servers. Aster*x required access to ARP packets of its slice, since it would gratuitously send ARP replies with a dummy MAC address, then dynamically rewrite the destination MAC address of the subsequent IP packets with that of the chosen server at the first hop. At one stage in its development, FlowVisor did not have the feature to assign ARPs to the right slice because IP addresses are hidden in the ARP request payload. Consequently, ARP packets were getting handled by another controller, causing connectivity errors. However, this problem was not immediately apparent to the administrator, causing him to spend many fruitless hours looking for bugs in the Aster*x code.

With packet histories: The admin could have used packet histories of the ARP packets to quickly discover that the requests were being forwarded all the way to the end hosts, instead of being gratuitously replied to by the Aster*x controller. By looking at the flow table entries that the ARP packets encountered, the admin could have figured out that the ARP packets were being handled by the wrong controller.

These examples only touch on the bugs seen in the last few years, and tell the best stories. The admins encountered many more "standard" errors like improperly separated VLANs, link failures, and even load balancer bugs [24], all of which could be diagnosed using packet histories.

2.1.3 Challenges

While the packet history appears to be a useful construct, generating packet histories in operational networks is not trivial. First, we must be able to view and record the paths taken by *every* packet in the network. The bigger challenge is determining the *exact* switch state encountered by a packet at each hop. Observing the switch states from an external vantage point, say by either logging the control messages or querying the switches for their state, will not guarantee precise state-packet correlation. I argue that the only place where packets can be correlated with the exact switch state is the data plane itself. With the lack of concurrency and consistency in distributed protocols, this problem only gets harder. Finally, the system built to record the packet histories of all the packets in the network should be scalable. It should be able to collect, process, and store multiple gigabytes of packet history data per second.

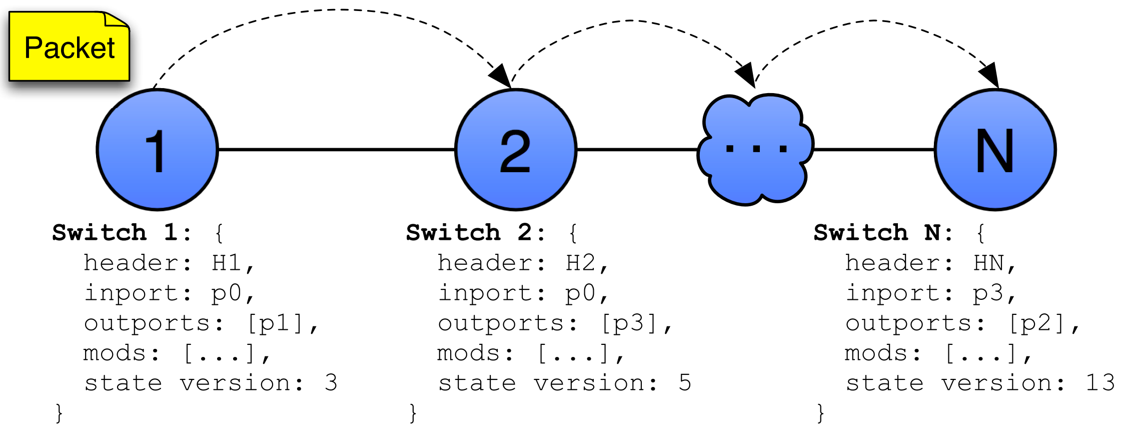


Figure 2.2: A packet history looks like a string of packet-hop information.

2.1.4 Opportunities with SDN

Software-Defined Networking (SDN), which refactors the relationship between network control and data planes, has a number of useful properties that can aid in overcoming these challenges: (1) standardized representation of switch state (flow tables), (2) a standardized interface to manage switch state (e.g., the OpenFlow protocol), and (3) a logically centralized control plane to manage the state. In Chapter 3, I show how we can leverage all three properties to precisely correlate packets with the state used to forward them. Next, I propose a language for specifying and filtering packet histories of interest.

2.2 Packet History Filter

The next challenge is to provide a framework for troubleshooting applications to query the network for packet histories of interest. As shown in Figure 2.2, a packet history essentially looks like a string of packet-hop information. Regular expression provides a concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters [34]. Inspired by the flexibility of regular expressions at parsing strings, I developed a regular-expression-like language—Packet History Filter (PHF)—to express interest in packet histories with specific paths, encountered switch state, and header fields.

2.2.1 Postcard Filters

The building block of Packet History Filter is the *postcard filter* (PF). A PF is a filter to match a packet at a switch. Syntactically, a PF is a conjunction of filters on various qualifiers: packet headers, switch ID (known as datapath ID, or dpid), input port, output port, and the switch state encountered by the packet (referenced by a "version" as described in Chapter 3). A PF is written as follows:

```
--bpf [not] <BPF> --dpid [not] <switch ID> --inport [not] <input port>
--outport [not] <output port> --version [not] <version>
```

where, `<BPF>` is a Berkeley Packet Filter expression. The `not`'s are optional and negate matches. A PF must have at least one of the above qualifiers. For example, a PF for an IP packet with source IP address `A`, entering switch `S` at any input port other than port `P` is written as:

`--bpf "ip src A" --dpid S --inport not P`.

2.2.2 Packet History Filter Grammar

A Packet History Filter is a regular expression built with PFs, where each PF is enclosed within double braces. Appendix A shows the complete PHF grammar. In addition to the simple regular expressions, PHFs also include extended regex features such as backreferences [14]. Backreferences (written as `\<number>`), can be used to match the same postcards that matched the `<number>`th group in the PHF. An example use case of backreferences is in loop detection.

2.2.3 Packet History Filter Examples

The following example PHFs, built using PFs `X` and `Y`, match packets that:

- start at `X`: `^{{X}}`
- end at `X`: `{{X}}$`

- go through `X`: `{{X}}`
- traverse link `XY`: `{{X}}{{Y}}`
- go through `X`, and later `Y`: `{{X}}.*{{Y}}`
- start at `X`, never reach `Y`: `^{{X}}[^{{Y}}]*$`
- experience a loop: `(.).*(\1)`

Using PHFs, we can describe a whole range of error conditions commonly encountered by network operators. For example:

- **Error class:** Reachability
  Symptom: Host `A` is not able to talk to host `B`.
  PHF: `^{{--bpf "ip src A and dst B" --dpid X --inport p1}}`
  `[^{{--dpid Y --outport p2}}]*$`
  where, (`X`, `p1`) is the location of `A` and (`Y`, `p2`) is the location of `B`.

- **Error class:** Isolation
  Symptom: Host `A` should not be able to talk to host `B`, but it is doing so.
  PHF: `^{{--bpf "ip src A and dst B" --dpid X --inport p1}}`
  `.*{{--dpid Y --outport p2}}$`
  where, (`X`, `p1`) is the location of `A` and (`Y`, `p2`) is the location of `B`.

- **Error class:** Forwarding Loop
  Symptom: Packet going through same switch twice.
  PHF: `(.).*(\1)`

2.2.4 Benefits and Limitations of Packet History Filter

The regular-expression-based Packet History Filter language has the following benefits:

- It is *comprehensible*. PHF captures semantics understandable to humans (e.g., packets not reaching their destination, or packets going through a forwarding loop).

- It is *expressive.* More specifically, it covers the set of languages known as regular languages [28].
- It is *intuitive.* Regular expressions are extensively used in scripting and programming. Thus, network administrators are already equipped with the necessary skills to use PHFs.

The PHF language also has its limitations. Because it is based on regular expressions, it can not describe richer languages such as context-free languages, also known as Type-2 languages [28]. However, in practice, Packet History Filter is sufficiently expressive to describe most error symptoms encountered by network administrators.

In the next chapter, I will describe the design and implementation of a system to capture and filter packet histories of interest.

Chapter 3

NetSight: A Troubleshooting Platform

In this chapter, I present the second major component of the systematic troubleshooting solution: NetSight, a platform to monitor network events and to collect the necessary information to support the Packet History abstraction. In short, NetSight collects, stores, and filters *all* packet histories, and presents a Packet History Filter-based API to troubleshooting applications, upon which one can build a range of applications to troubleshoot networks. In the remainder of the chapter, I describe the design and implementation of NetSight, and how NetSight handles some of the corner cases of operation. Finally, I describe the NetSight API and show how it interacts with the troubleshooting applications built on top of it.

3.1 How NetSight Works

The astute reader is likely to doubt the scalability of *any* system that attempts to store the header of every packet traversing a network, along with its corresponding path, state, and modifications, as well as apply complex filters to it. This is a lot of data to forward, let alone process and archive.

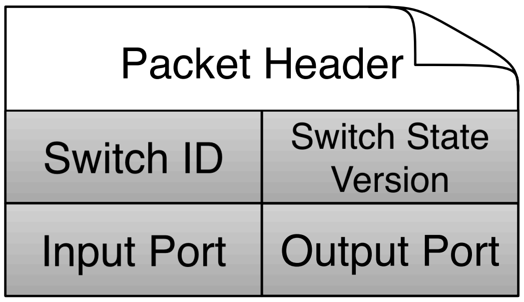


Figure 3.1: A postcard contains the packet hearder, switch ID, input/output ports, and the version of the switch state encountered by the packet.

Hence, NetSight is designed from the beginning to scale out and see linear improvements with increasing compute and storage resources. All the processing elements of NetSight, such as table lookups, compression operations, and querying, are simple enough to enable hardware implementations. As an existence proof that such a system is indeed feasible, the implementation described in Section 3.2 and evaluated in Section 5.3 can perform all packet history processing and storage steps for a moderately-sized network on a single server. For faster networks, this number increases linearly with the number of servers, as shown in Chapter 5.

3.1.1 NetSight Philosophy

NetSight assembles packet histories using *postcards*; event records sent out whenever a packet traverses a switch. Each postcard contains the packet header, switch ID, input/output ports, and current version of the switch state, as shown in Figure 3.1. Combining topology information with the postcards generated by a packet, we can reconstruct the complete packet history: the exact path taken by the packet along with the state and header modifications encountered by it at each hop along the path.

An alternative way to assemble packet histories is to use a *passport*-based approach, where history information is directly appended to the original packet at each hop. Unlike the passport-based design, NetSight's postcard-based design decouples the fate of packets from the fate of their history information. In addition, placing minimal trust in the data-plane simplifies the design and does not risk affecting the

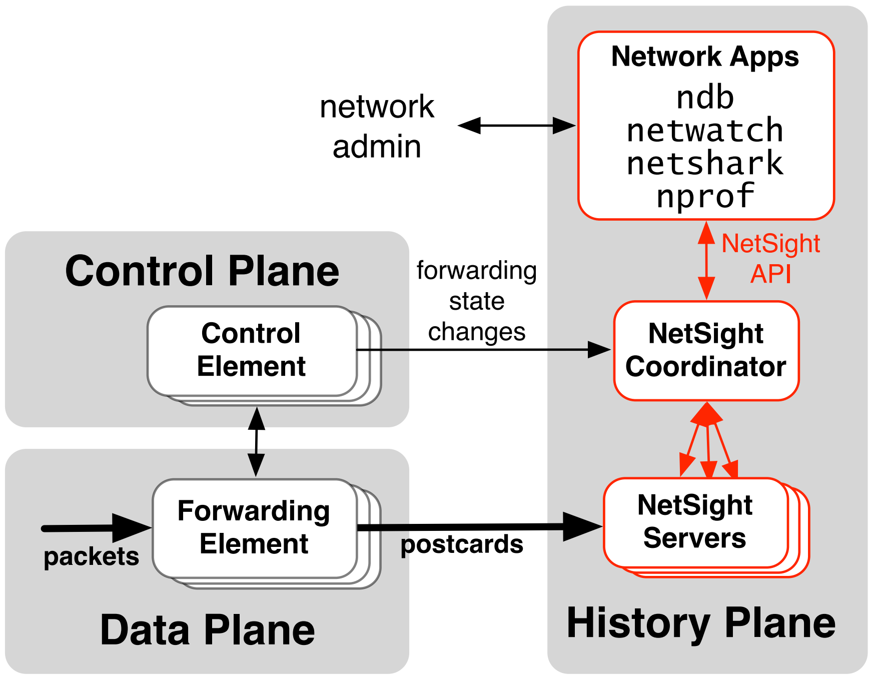


Figure 3.2: NetSight architecture.

actual packet's forwarding behavior.

Compared to passports, postcards require more network bandwidth and additional processing to assemble packet history from out-of-order postcards. The challenges are to avoid (or at least minimize) changes to the forwarding hardware, to keep network traffic and storage rates reasonable, and to scale these steps using multiple servers. I first explain how NetSight works in the *common case*, where:

1. the network does not drop postcards;
2. the network does not modify packets; and
3. all packets are unicast.

Later, Section 3.3 describes how NetSight handles these edge cases.

3.1.2 System Architecture

Figure 3.2 sketches the architectural components of NetSight. Similar to MapReduce, NetSight employs a central coordinator to manage multiple workers (called NetSight servers). Troubleshooting applications issue PHF-based triggers and queries to the coordinator, which then returns a stream or batch of matching packet histories. The coordinator connects to each element in the network (e.g., using the switch CLI or by interposing on the control channel) to (1) set up the transmission of postcards from the forwarding elements to the NetSight servers and (2) monitor the state changes in the switches. State change monitoring can be implemented by modifying switch firmware or interposing on the control channel of externally-controlled switches as in OpenFlow/SDN [24]. Finally, the coordinator performs periodic liveness checks, broadcasts queries and triggers, and communicates topology information for the workers to use when assembling packet histories. The network topology can be obtained from reading a configuration file, asking the control plane, or dynamic learning (e.g., using the Link Layer Discovery Protocol (LLDP) [42]).

3.1.3 NetSight Design

In brief, NetSight turns postcards into packet histories. To explain this process, I now follow the steps performed inside the NetSight servers, as shown in Figure 3.3.

Postcard Generation

Goal: *to record all information relevant to a forwarding event and send it for analysis.*

As a packet enters a switch, the switch creates a postcard by duplicating the packet, truncating it to the minimum packet size, marking it with relevant state, and forwarding it to a NetSight server. The marked state includes the switch ID, the output port to which this packet is about to be forwarded, and a version ID representing the exact state of this switch when the packet was forwarded. The "original packet" remains untouched, and continues on its way.

This design requires that switches implement the above sequence of actions to generate postcards. Switches already perform similar packet duplication actions to

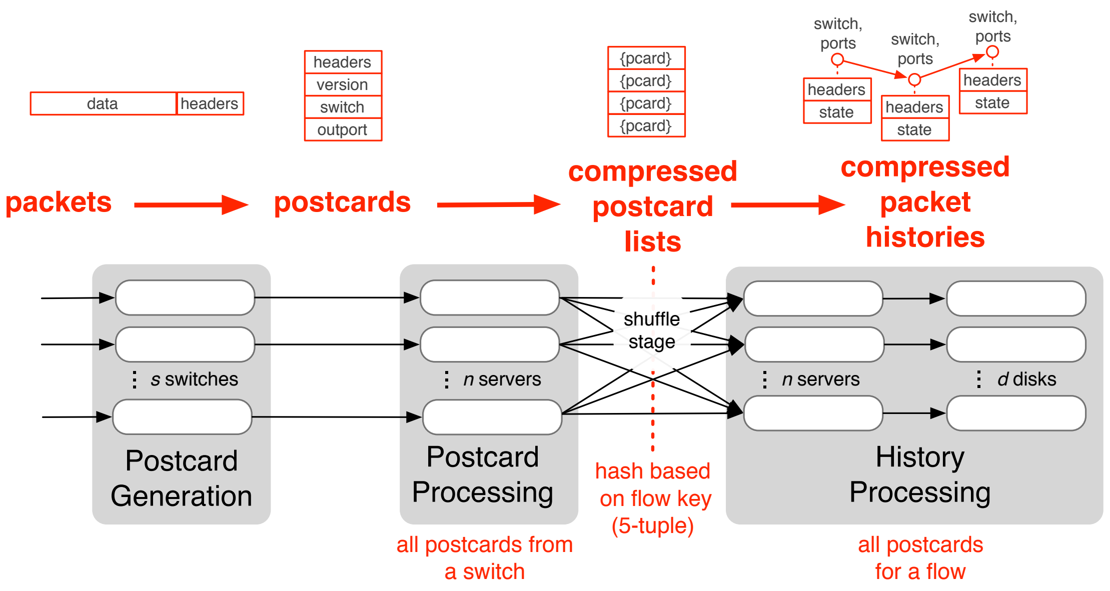


Figure 3.3: Processing flow used in NetSight to turn packets into packet histories across multiple servers.

divert flows for security purposes (e.g. DMCA violations), lawful intercept, or intrusion monitoring. RSPAN is an example of a mechanism that could be repurposed because it collects all the traffic on one or more ports, encapsulates it, and forwards it to a remote destination [31].

Postcard Collection

Goal: *to send all postcards of a packet to one server, so that its packet history can be assembled.*

In order to reconstruct packet histories, NetSight needs to collect all postcards corresponding to a single packet on a single server. To scale processing, NetSight needs to ensure that these groups of postcards are evenly spread across the set of available servers. NetSight achieves this by shuffling the postcards between the NetSight servers, à la MapReduce, using a hash function that ensures "postcard locality."

To do this, when a postcard arrives at a NetSight server, it is temporarily placed in a data structure called the *Postcard Stage*. The Postcard Stage consists of a collection of lists, one per NetSight server, each containing postcards that will eventually go to that server. NetSight picks the destination server for a postcard based on a hash of the postcard's flow ID (the `(srcip, dstip, srcport, dstport, protocol)` 5-tuple) and enqueues it in the corresponding list of Postcard Stage. Thus, all the postcards of all the packets belonging to a flow are assembled at one NetSight server.

Postcard collection is organized into "rounds," during which a Postcard Stage fills up. When a round is over, servers empty their Postcard Stages and send postcard lists to their final destination, where the corresponding packet histories can be assembled and archived. The Postcard Stage provides an opportunity to compress postcard data before shuffling it and sending it across the network, by exploiting the redundancy of header values, both within a flow and between flows. By sending the postcards of all the packets belonging to a flow to a single destination server, NetSight improves the effectiveness of this compression. Section 3.2 details the fast network-specific compression technique employed by NetSight to reduce network bandwidth usage.

History Assembly

Goal: *to assemble packet histories from out-of-order postcards.*

Packet histories must be properly ordered for both humans and PHF matching engines to make sense of them. Postcards arrive without timestamps, obviating the need for switches to use fine-grained time synchronization methods such as Precision Time Protocol [30]. They can also arrive out-of-order due to varying propagation and queuing delays from switches to NetSight servers. Hence, NetSight must use topology information to make sense of postcard ordering.

When a NetSight server receives the complete round of postcard lists from all other servers, it decompresses and merges each one into the *Path Table*, a data structure that helps combine all postcards for a single packet into a group. To identify all postcards corresponding to a packet, NetSight combines immutable header fields such as IP ID, fragment offset, and TCP sequence number fields into a "packet ID," which uniquely identify a packet within a flow. The Path Table is simply a hash table indexed by the packet ID, where values are lists of corresponding postcards.

The NetSight server extracts these postcard groups, one at a time, to assemble

```
# Input: pcard_lst (Postcards with a given packet ID)
# Input: topo (Topology)
pcard_table = HashTable(dpid -> pcard)
populate(pcard_table, pcard_list)
for i in (0, len(pcard_lst)):
  pcard = pcard_lst[i]
  dpid, outport = get_metadata(pcard)
  nbr = get_neighbor(topo, dpid, outport)
  if nbr in pcard_table:
    pcard->next = nbr
    nbr->prev = pcard
for i in (0, len(pcard_lst)):
  pcard = pcard_lst[i]
  if (pcard->prev == NULL):
    return pcard
```

Figure 3.4: Topological sort algorithm to assemble postcards into packet history.

them into packet histories. For each group, NetSight then performs a topological sort, using switch IDs and output ports, along with topology data. The resulting sorted list of postcards is the packet history. This sort, shown in Figure 3.4, runs in $O(p)$, where p is the number of postcards in the path; typically, p will be small.

Filter triggers

Goal: *to immediately notify applications of fresh packet histories matching a pre-installed PHF.*

Once the packet history is assembled at a NetSight server, the server matches it against any "live" PHFs pre-installed by applications, and on a successful match, immediately triggers notifications back to the application via the coordinator.

History archival

Goal: *to efficiently store the full set of packet histories.*

Next, the stream of packet histories generated in each round is written to a file. NetSight compresses these records using the same compression algorithm before the shuffle phase to exploit redundancy between postcards of a packet and between packets of a flow.

Historical query

Goal: *to enable applications to issue PHF queries against archived packet histories.*

When an application issues a historical PHF query to a specified time region, that query runs in parallel on all NetSight servers. Compression helps to improve effective disk throughput here, and hence query completion times, albeit at the cost of extra CPU resources to perform the compression/decompression. Ideally the filesystem is log-structured, so that individual rounds can be restored at the full disk throughput with minimal seeking [52].

3.2 NetSight Implementation

My first NetSight prototype implementation, written entirely in Python, could collect and assemble about a million postcards per second and filter a few thousand postcards per second—enough to be useful for debugging emulated networks or small test networks. The implementation had two processes: one interposed between an OpenFlow controller and its switches to report configuration changes, while another did all postcard and history processing. To verify that it operated correctly on physical switches, I tested it on a chain topology of 6 NEC IP8800 switches [44]. To verify that it ran with unmodified controllers, I tested it on the Mininet emulation environment [40] with multiple controllers (OpenFlow reference, NOX [23], POX [49], RipL-POX [51]) on multiple topologies (chains, trees, and fat trees).

This naive prototype demonstrated the potential of NetSight as a platform. Clearly, for a modest trial deployment in an operational network, the system must run orders of magnitude faster, and scale out to multiple servers. In addition, I learned from our network operators that most debugging is not done live, but post-mortem, and therefore, we needed a way to efficiently store histories.

This section describes the individual pieces of my second prototype, which implements all fast-path processing in C++ and implements the slow-path coordinator

```
# Assume postcards are routed to the server via outport C.
# Interpose on the control channel:
while True:
  M = next message
  S = switch targeted by M
  if M is a flow modification message:
    F = flow entry specified by M
    S.version += 1
    tag_actions = []
    for action in F.actions:
      if action == Output:
        tag = pack_to_48bits(one_byte(S.id),
                             two_bytes(action.port),
                             three_bytes(S.version))

        tag_actions.append(SetDstMac(tag), Output(port=C))
    F.actions.append(tag_actions)
  S.send_message(M)
```

Figure 3.5: Pseudocode of the Flow Table State Recorder that rewrites control messages to generate postcards.

and applications in Python.

3.2.1 Postcard Generation

The NetSight prototype, based on OpenFlow/SDN, leverages the fact that network state changes are coordinated by a controller, which provides an ideal place to monitor and intercept switch state changes. The prototype uses a transparent proxy called the *flow table state recorder* (recorder for short) that sits on the control path between the controller and OpenFlow switches.

When a controller modifies flow tables on a switch, the recorder intercepts the message and stores it in a database. For each OpenFlow rule sent by the controller to the switch, the recorder *appends* new actions to generate a postcard for each packet matching the rule in addition to the controller-specified forwarding.

Specifically, the actions create a copy of the packet and tag it with the switch

```
Postcard forwarding algorithm:

The recorder installs two sets of rules at each switch:
(1) Highest priority rules: Forward postcards out of
    the spanning tree output port if received on
    valid spanning tree input ports.
(2) Lower priority rule: Drops all postcards.

Postcard generation algorithm:

If in_port not wildcarded:
  if input_port != spanning_tree_out_port:
    output postcard out of spanning_tree_out_port
  else:
    output postcard out of OFPP_IN_PORT
else:
  output postcard out of both
    spanning_tree_out_port and OFPP_IN_PORT
```

Figure 3.6: Algorithm to generate postcards in-band.

ID,[1] the output port, and a version number for the matching flow entry. The version number is simply a counter that is incremented for every flow modification message. The pseudocode in Figure 3.5 shows how the tags are stored in the postcard. The tag values overwrite the destination MAC address (the header of the original packet remains completely unchanged). Once created, postcards are sent to a NetSight server over a separate VLAN. Postcard forwarding can be handled out-of-band via a separate network, or in-band over the regular network. In the in-band mode, switches recognize postcards using a special VLAN tag to avoid generating postcards for postcards.

The out-of-band case is straightforward; postcards simply exit the “collection port” of each switch and a separate network forwards them to the NetSight servers.

In in-band collection, the postcards can be forwarded either along spanning trees rooted at the NetSight servers, or using a more general multipath routing mechanism. However, in-band collection has two caveats:

[1]To fit into the limited tag space, NetSight uses a locally created “pseudo switch ID” (PSID) and maintains an internal mapping from the 8 byte datapath ID to the PSID.

1. **Correctness**: The OpenFlow protocol specification makes the seemingly simple task of in-band postcard collection over spanning tree paths to a single NetSight server tricky. An OpenFlow rule does not forward a packet back out of the input port unless explicitly instructed to do so via `OFPP_IN_PORT`. Therefore, if a packet's input port at a switch is the same as the postcard's spanning tree output port, the postcard will never be generated by that switch. The problem is further complicated if the input port field in the rule is wildcarded, as the flow table rule will now have to be split into two—one specifying `OFPP_IN_PORT` if the input port is the same as the spanning tree output port, and another for the remaining input ports. Such naive flow entry splitting can affect the correctness of the controller logic. NetSight overcomes this problem by using the algorithm shown in Figure 3.6. In the case where the input port is wildcarded, the algorithm generates two postcards; one sent out of the spanning tree output port, and the other out of `OFPP_IN_PORT`. This guarantees that at least one postcard is generated for the packet. An additional low-priority rule in the neighboring switch drops the duplicate postcard, if generated.

2. **Performance**: In in-band collection, postcards must share the network with "regular" packets, and, therefore, may affect the performance of the network. The impact of postcards on the original packets is minimized by placing the postcards in low-priority queues. However, these separate queues must be sufficiently well provisioned to ensure that the postcards are not dropped.

3.2.2 Compression

NetSight compresses postcards (devoid of payload) in two places: (1) before shuffling them to servers, and (2) before archiving assembled packet histories to disk. Compression algorithms work by building a dictionary of repetitive bit-patterns. While we can use off-the-shelf compression algorithms such as gzip or LZMA to compress the stream of postcards, we can do better by leveraging the structure in packet headers and the fact that all the packets in a flow—identified by the 5-tuple flow ID `(srcip, dstip, srcport, dstport, protocol)`—look similar.

The goal of the NetSight compression algorithm is to compress the packets in such a way that it can later reconstruct the entire packet stream, including the order of the packets, the complete packet headers, and their timestamps. NetSight compresses packets by computing diffs between successive packets in the same stream. A diff is a `(Header,Value)` pair, where `Header` uniquely identifies the field that changed and `Value` is its new value. Certain fields (e.g. IPID and TCP Sequence numbers) can be compressed better if we just store the successive *deltas*. NetSight compresses packets as follows. The first packet of each flow is stored verbatim. Subsequent packets are encoded as a `DiffRecord` that only stores packed `(Header,Value)` tuples that change, along with a back-reference to the previous packet in the same stream, with respect to which the DiffRecord is calculated. The compressor only stores the non-zero octets of each `Value` and treats path information, such as the switch ID and flow version, as if they were header fields in the packet. NetSight (optionally) pipes the stream of encoded diffs through a standard fast compression algorithm (e.g., gzip at level 1).

Format

NetSight's compression outputs three files: a list of "first packets (`FP`)," a list of timestamps (`TS`) (if available, e.g., from a pcap file or a hardware capture device [4]) and a list of packet diffs (`DF`). `FP` is the dictionary of packets against which subsequent packets are diffed. NetSight outputs a `DiffRecord` for every packet. Figure 3.7 shows a `DiffRecord` encoding, it consists of:

1. an index to the previous packet in the same stream or the FP table;
2. the number of diffs in this packet; and
3. a list of `FieldRecord`s for every diff.

A `FieldRecord` encodes the field name (e.g. IPID, TCPSEQ, etc.), and its new value with respect to the packet referenced by the `DiffRecord`.

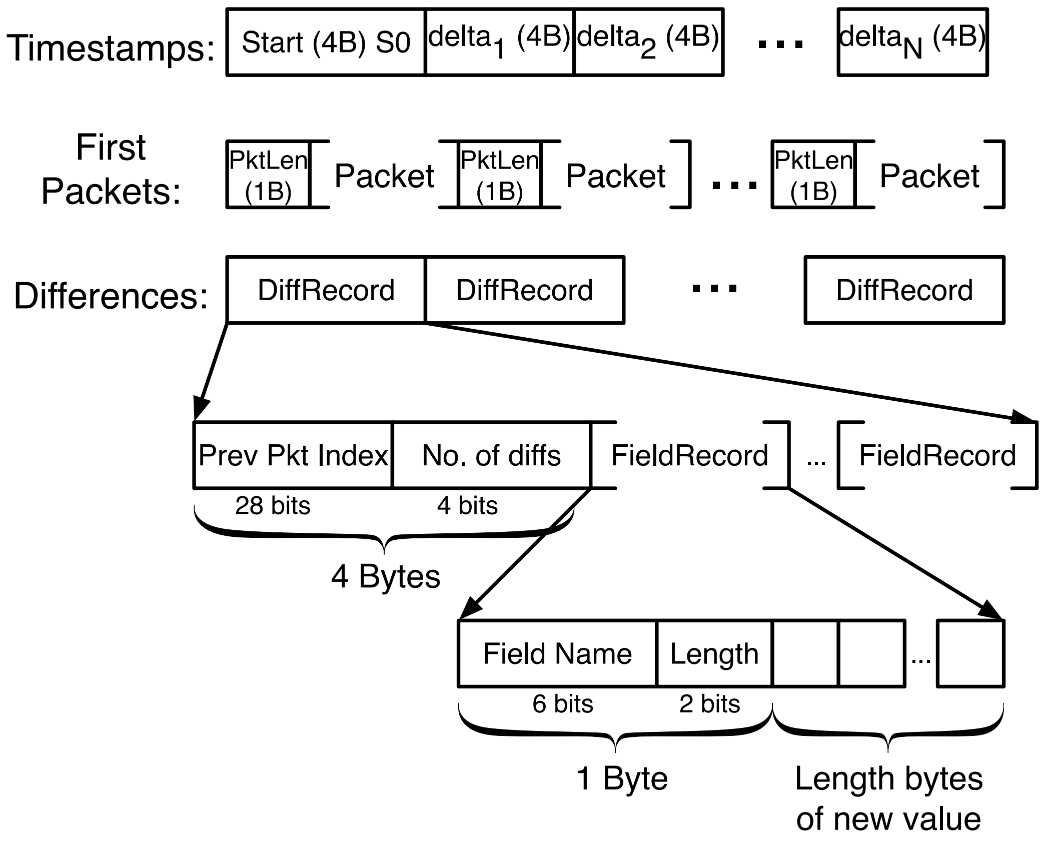


Figure 3.7: The NetSight postcard compression format.

Algorithm and Encoding

NetSight makes a single pass through the packet stream. For each packet, NetSight constructs a flow key[2] and inserts it into a hash table. If the flow key is absent in the hash table, NetSight appends it to `FP` and notes its position in the list. The `DiffRecord` for the first packet of a flow is therefore empty and references the packet just appended to `FP`. If the key is present in the hash table, the `DiffRecord` encodes a 28-bit index into the same stream, and a 4-bit number denoting the number of changes. Each `FieldRecord` contains a 6-bit encoding of the field name that changed; a 2-bit number indicating the length of the value to follow (1, 2, 3 or 4 bytes). Though the field length is implicit from the field type, NetSight uses a variable length encoding

[2]A flow key is the 5-tuple (srcip, dstip, srcport, dstport, protocol).

that only stores a value's non-zero octets. Each packet's `DiffRecord` is emitted to the differences file, `DF`. Empirical results show that at most 12 fields change within a packet in many traces, and therefore the 4-bit number suffices. NetSight uses the all-1 magic number (`0xf`) to denote the first packet in the stream. Also, there are only about 40 commonly encountered fields in a packet header, and therefore, 6-bits suffice.

Compression

The `TS` file contains the timestamp for the first packet and only encodes 4-byte deltas between successive records. `FP` is just a flat record of packets that triggered a new flow key. These files are by no means efficiently compressed, but NetSight offers a tunable parameter to select the compression level after generating the above files. NetSight uses `gzip` which offers good compression at reasonable CPU cost. The `TS`, `FP`, and `DF` files are then passed through `gzip`.

Compressing Packet Histories

Packet histories contain path and switch state information in addition to packet headers. However, the paths used by networks come from a small set, and switch state changes are relatively infrequent. NetSight exploits this fact and compresses packet histories using the same algorithm above by treating paths and switch state version as additional packet fields.

The NetSight compression algorithm is a generalization of Van Jacobson's compression of TCP packets over slow links [32]. The compressor (and decompressor) is implemented in the NetSight prototype in about 2200 lines of C++ code. The prototype implementation has sufficient scope for further optimization using techniques like Huffman coding [29].

3.2.3 PHF Matching

The PHF matching engine in NetSight is based on the RE1 regex engine [18] and uses the Linux x86 BPF JIT compiler [9] to match packet headers against BPF filters. RE1 compiles a subset[3] of regular expressions into byte codes. This byte code implements a Nondeterministic Finite Automaton (NFA) which RE1 executes on an input string. In RE1, character matches trigger state machine transitions; in NetSight, the character-equality-check function is "overloaded" to match postcards against postcard filters. The PHF matching engine is implemented in approximately 2000 lines of C/C++ code.

3.3 Relaxing the Assumptions

We now relax three assumptions we made earlier in our description of NetSight.

Assumption 1: The network does not drop postcards. One way to handle drops is to simply keep incomplete packet histories for further analysis. For instance, if only a single postcard is missing from a packet history, the topological sorting can be extended to consider both possible orderings, which can both be matched against filters.

The other way to handle drops is to avoid them in the first place. Many networks have dedicated out-of-band links for controlling switches; this additional capacity can reduce the chances of dropping a postcard. Alternatively, even if postcards are routed in-band, one can statically reserve bandwidth for postcards on each link by marking postcards with a special VLAN tag.

Assumption 2: The network does not modify packets. NetSight uses the flow key to shuffle postcards to their final NetSight server. Network Address Translation (NAT) boxes modify the header fields that are used in creating the flow key, causing postcards for the same packet to arrive at multiple NetSight servers and making it much harder to assemble packet histories. To correctly handle this case, we must never use a field that gets modified during the course of packet forwarding in the flow key. For example, using immutable headers, or even a hash of the packet contents, as the flow key in the shuffle would ensure that all postcards of a packet

[3]RE1 supports concatenation, alternation, and the Kleene star operations. NetSight handles loops separately as a special case.

arrive at the same server.[4] These flow key choices, however, can make the packet compression for storage less effective, as each of n NetSight servers will then receive $1/n$-th of the packets of each flow.

The addition of a second shuffle stage can provide both correctness and storage efficiency. In the first stage, packet histories are shuffled for assembly using the packet ID, while in the second stage, they are shuffled for storage using a hash of the 5-tuple flow key of their *first packet*. This way, flow locality is maximized at the cost of additional network traffic and processing. In practice, NAT boxes tend to be at the edge of a network, making in-network modifications less of a concern.

Assumption 3: Packets are all unicast. Broadcast, multicast, and loop traffic are all examples of non-unicast paths. In such cases NetSight returns packet histories as directed graphs, rather than lists. In the case of a broadcast or multicast packet, it returns a tree. In the case of a loop, it returns the packet history with an arbitrary starting point and indicates it as a loop.

3.4 Resolving Ambiguity

The NetSight prototype is not perfect. In some cases, flow table and packet ambiguity may prevent NetSight from unambiguously identifying a packet history.

3.4.1 Flow Table Ambiguity

Postcards generated by the tagging algorithm (Figure 3.5) uniquely identify a switch, the matching flow entry, and output port(s). In most cases, a packet history constructed from these postcards provides sufficient information to reason about a bug. However, a developer cannot reason about the *full* flow table state when postcards specify only the matching flow entry. This gap in knowledge can lead to the ambiguity shown in Figure 3.8. Suppose a controller inserts two flow entries, A and B, in succession. If NetSight sees a postcard generated by entry A, it may either be the case that entry B was not installed (yet), or that the packet did not match entry B.

[4]That is, if middleboxes do not modify packet *payloads*.

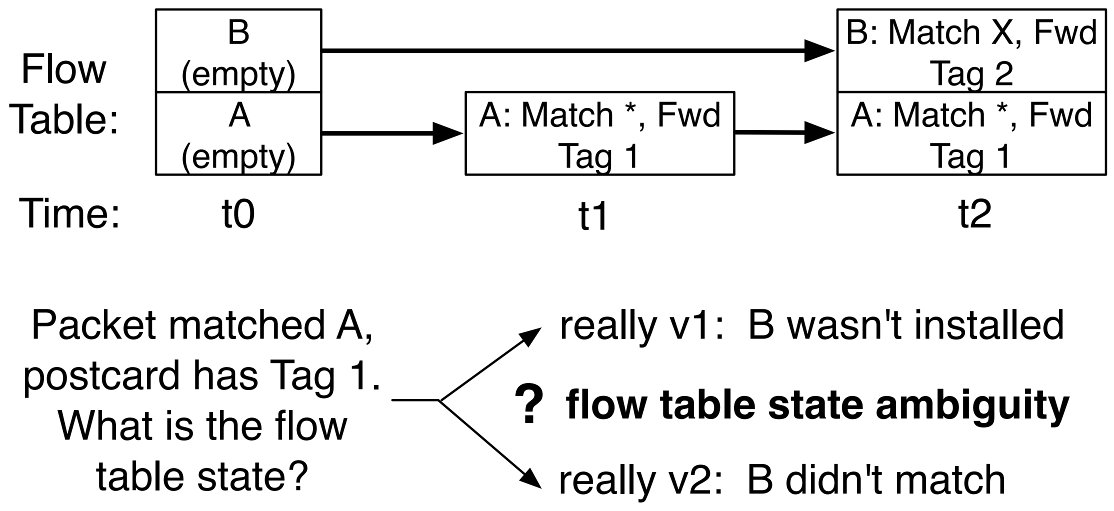

Flow Table:
B (empty)
A (empty)
A: Match *, Fwd Tag 1
B: Match X, Fwd Tag 2
A: Match *, Fwd Tag 1
Time: t0 t1 t2
Packet matched A, postcard has Tag 1. What is the flow table state?
really v1: B wasn't installed
? flow table state ambiguity
really v2: B didn't match

Figure 3.8: Simple two-entry flow table highlights a possible ambiguity that cannot be resolved just by knowing the flow entry a packet matched.

Moreover, a switch can timeout flow entries, breaking NetSight's view of flow table state.

To resolve such ambiguities and produce the state of the *entire* flow table at the time of forwarding, the NetSight recorder should observe and control every change to flow table state, in three steps.

First, to prevent the switch state from advancing on its own, the recorder must emulate timeouts. Hard timeouts can be emulated with one extra flow-delete message, while soft timeouts require additional messages to periodically query statistics and remove inactive entries.

Second, the recorder must ensure that flow table updates are ordered. An OpenFlow switch is not guaranteed to insert successive entries in order, so the recorder must insert a `barrier` message after every flow modification message.

Third, the recorder must version the entire flow table rather than individual entries. One option is to update the version number in *every* flow entry after *any* flow entry change. This bounds flow table version inconsistency to one flow entry, but increases the `flow-mod` rate by a factor equal to the number of flow entries in the switch. A cleaner option is atomic flow updates. Upon each state change, the recorder could issue a transaction to atomically update all flow entries to tag packets with the

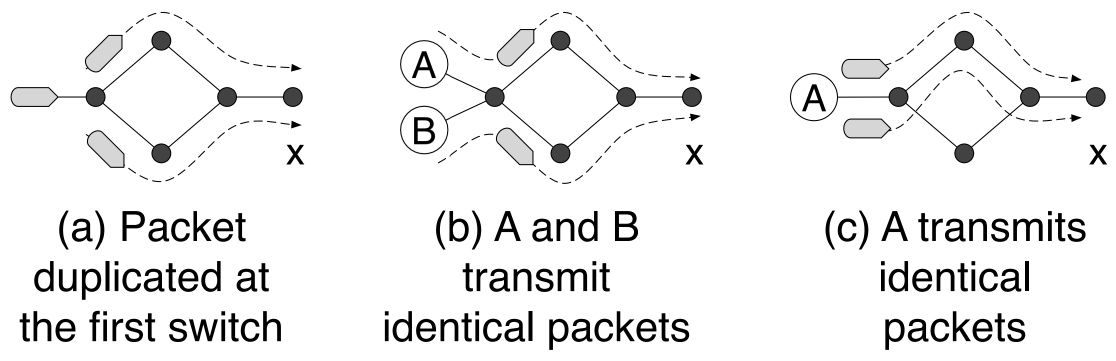

A
B
x
(a) Packet duplicated at the first switch
(b) A and B transmit identical packets
(c) A transmits identical packets

Figure 3.9: Ambiguous scenarios for packet identification.

Window	**% identical pkts**	**Split**
10ms	11.34%	IP: 11.29%, ARP: 0%, Other: 0.04%
1s	11.46%	IP: 11.46%, ARP: $\sim$ 0%, Other: 0.06%

Table 3.1: Effectiveness of unique packet identification by using a hash of immutable header fields. Over a time window of 1 second, I observed 7 identical ARP packets, out of a total of 400K packets.

new version number, either by swapping between two tables or by pausing forwarding while updating. The preferred option is to have a separate register that is atomically incremented on each flow table change. This decouples versioning from updates, but requires support for an action that stamps packets with the register value.

3.4.2 Packet Ambiguity

Identifying packets uniquely within the timescale of a packet lifetime is a requirement for unambiguous packet histories. As shown in Figure 3.9, ambiguity arises when (a) a switch duplicates packets within the network, (b) different hosts generate identical packets, or (c) a host repeats packets (e.g., ARP requests). For example, in Figure 3.9(a), a packet duplicated at the first switch, but packet histories along the upper and lower paths are equally feasible. In these situations, where NetSight cannot resolve this ambiguity, it returns all possibilities.

To evaluate the strategy of using immutable header fields to uniquely identify packets and their corresponding postcards, I analyzed a trace of enterprise packet headers, consisting of about 400,000 packets [41]. Table 3.1 summarizes the results. Nearly 11.3% of packets were indistinguishable from at least one other packet within a one-second time window. Closer inspection revealed that these were mostly UDP packets with IPID 0 generated by an NFS server. Ignoring these removed all IP packet ambiguity, leaving only seven ambiguous ARP packets. This analysis suggests that most of the packets have enough entropy in their immutable header fields to be uniquely identified.

3.5 NetSight API

NetSight allows applications to specify and query packet histories of interest via the NetSight API. More specifically, it allows applications to *add*, *delete*, and *list* installed PHFs.

3.5.1 NetSight API Goals

In addition to providing the above functionality, the NetSight API should also meet the following goals:

- It should allow multiple troubleshooting applications to run simultaneously.
- It should allow applications to dynamically start and stop without affecting the NetSight coordinator or the NetSight servers.
- It should not impose any programming language restrictions on the troubleshooting applications.
- It should be flexible enough to allow each application to scale across multiple CPU cores or a cluster of machines.

Message queue systems, which provide an asynchronous communication protocol above low-level transport protocols like TCP, PGM (reliable multicast), inter-process communication (IPC), and inter-thread communication (ITC), possess the flexibility and capability to meet all the above goals.

3.5.2 Using ØMQ

The NetSight API is implemented based on the ØMQ (ZeroMQ) message queue library [27]. In addition to meeting the API goals listed in Section 3.5.1, ØMQ has the following advantages:

- It carries messages across a variety of transport protocols: *inproc*, IPC, TCP, and multicast.
- It supports a wide variety of N-to-N communication abstractions: fanout, publish-subscribe, pipeline, request-reply, etc.
- It is supported on most operating systems including Linux, Windows, and OS X.
- It has bindings for more than 30 languages including C, C++, Java, .NET, and Python.
- It has a large and active open-source community built around it.

Unlike message-oriented middleware, ØMQ is a messaging library, and can run without a dedicated message broker. The library is designed to have a familiar socket-style API.

3.5.3 NetSight API Design

At a high level, NetSight needs to perform two tasks:

1. Exchange control messages with the applications to add, delete, and list installed PHFs.
2. Communicate matched packet histories to the applications.

These two tasks naturally lead to the NetSight API design. As shown in Figure 3.10, the NetSight coordinator exposes two message queue sockets[5] to the troubleshooting applications:

[5]The message queue sockets are not the same as TCP or Unix sockets.

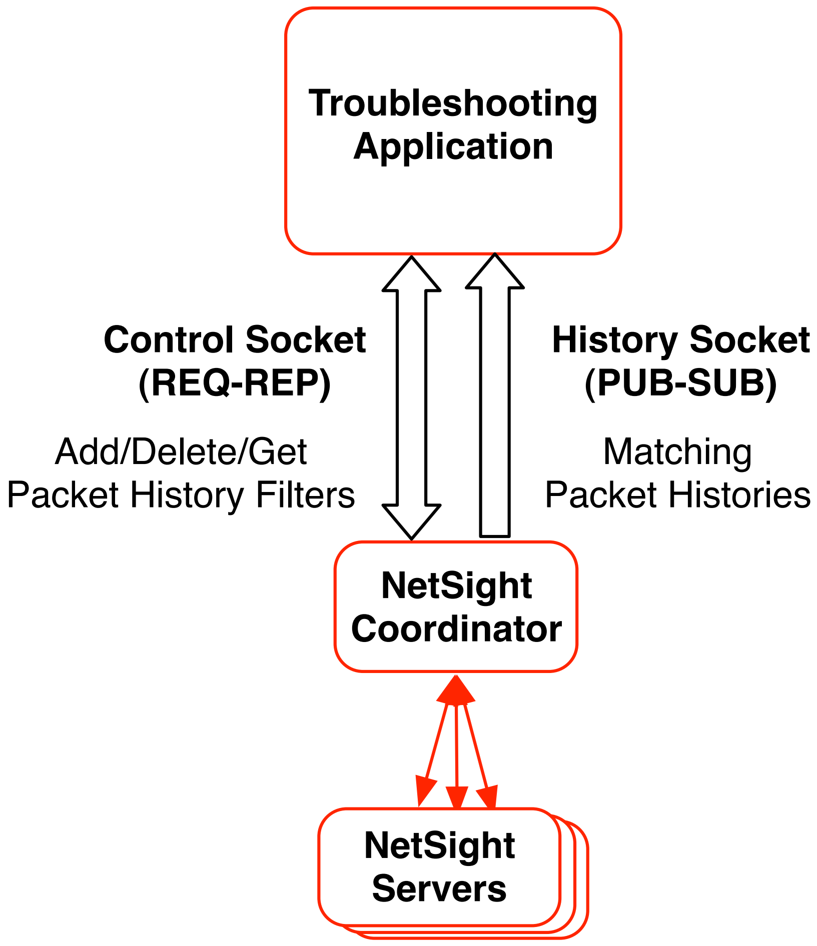

Troubleshooting Application
Control Socket (REQ-REP)
Add/Delete/Get Packet History Filters
History Socket (PUB-SUB)
Matching Packet Histories
NetSight Coordinator
NetSight Servers

Figure 3.10: NetSight supports the NetSight API via two message queue sockets: control socket and history socket.

1. *Control Socket*: A request-reply (REQ-REP) message queue socket, and

2. *History Socket*: A publish-subscribe (PUB-SUB) message queue socket

The applications use the control socket to send control messages to add, delete, and list installed PHFs. NetSight internally maintains a hash table mapping the application ID (a randomly generated unique ID for each application) to the list of PHFs added by the application. For each packet history matching any of the added PHFs, NetSight publishes the packet history (serialized using JSON encoding [8]) along with the matched PHF over the history socket. The application can get the matched packet history by subscribing for the corresponding PHF over the history socket. In ØMQ, clients of a PUB-SUB socket can subscribe to or unsubscribe from specific types of messages by calling the `zmq_setsockopt` function on the socket.

The NetSight API is available to the troubleshooting applications in the form of a shared library.

The remainder of this section describes each function of the NetSight API in detail.

3.5.4 Add Packet History Filter

```
add_filter(packet_history_filter, control_socket, subscribe_socket,
             start_time, end_time)
```

Upon calling this function, the API library performs two tasks:

1. It sends an `ADD_FILTER_REQUEST` message to the NetSight coordinator over the control socket. NetSight responds to the `ADD_FILTER_REQUEST` message with an `ADD_FILTER_REPLY` message that contains a unique ID (`PHF_ID`) that corresponds to the installed PHF.

2. It subscribes for packet histories corresponding to the `PHF_ID` on the history socket.

The optional `start_time` and `end_time` arguments can be used for historical querying. When specified, NetSight matches packet histories archived between `start_time` and `end_time` against `packet_history_filter` and publishes them over the history socket.

3.5.5 Delete Packet History Filter

```
delete_filter(PHF_ID, control_socket, subscribe_socket)
```

Upon calling this function, the API library performs the following tasks:

1. It sends a `DELETE_FILTER_REQUEST` message to the NetSight coordinator over the control socket, and
2. It unsubscribes from packet histories corresponding to `packet_history_filter` on the history socket.

NetSight responds to the `DELETE_FILTER_REQUEST` message with a `DELETE_FILTER_REPLY` message indicating whether the PHF was successfully deleted from its internal hash table.

3.5.6 Get Installed Packet History Filters

`get_filters(control_socket)`

This function gets the list of all filters installed by an application that are currently active. Upon calling this function, the API library sends a `GET_FILTERS_REQUEST` message to the NetSight coordinator over the control socket.

NetSight responds to the `GET_FILTERS_REQUEST` message with a `GET_FILTERS_REPLY` message which contains a list of all the currently active PHFs installed by the application.

3.5.7 Send Periodic Heartbeats

`send_echo_request(control_socket)`

The applications periodically send heartbeat messages to the NetSight coordinator in the form of `ECHO_REQUEST` messages. The NetSight coordinator responds to them with `ECHO_REPLY` messages. If NetSight does not receive heartbeats for a predefined timeout period, it assumes that the application is dead and erases all data associated with it—application ID and the installed PHFs. My NetSight prototype uses a heartbeat interval of 1 second, and a heartbeat timeout period of 5 seconds.

Appendix B enumerates the NetSight API message types and the structure of each message.

In the next chapter, I describe the four troubleshooting applications I built on top of NetSight using the NetSight API. For each application, I describe the motivation behind it, its design and implementation, and provide example use cases.

Chapter 4

Troubleshooting Applications

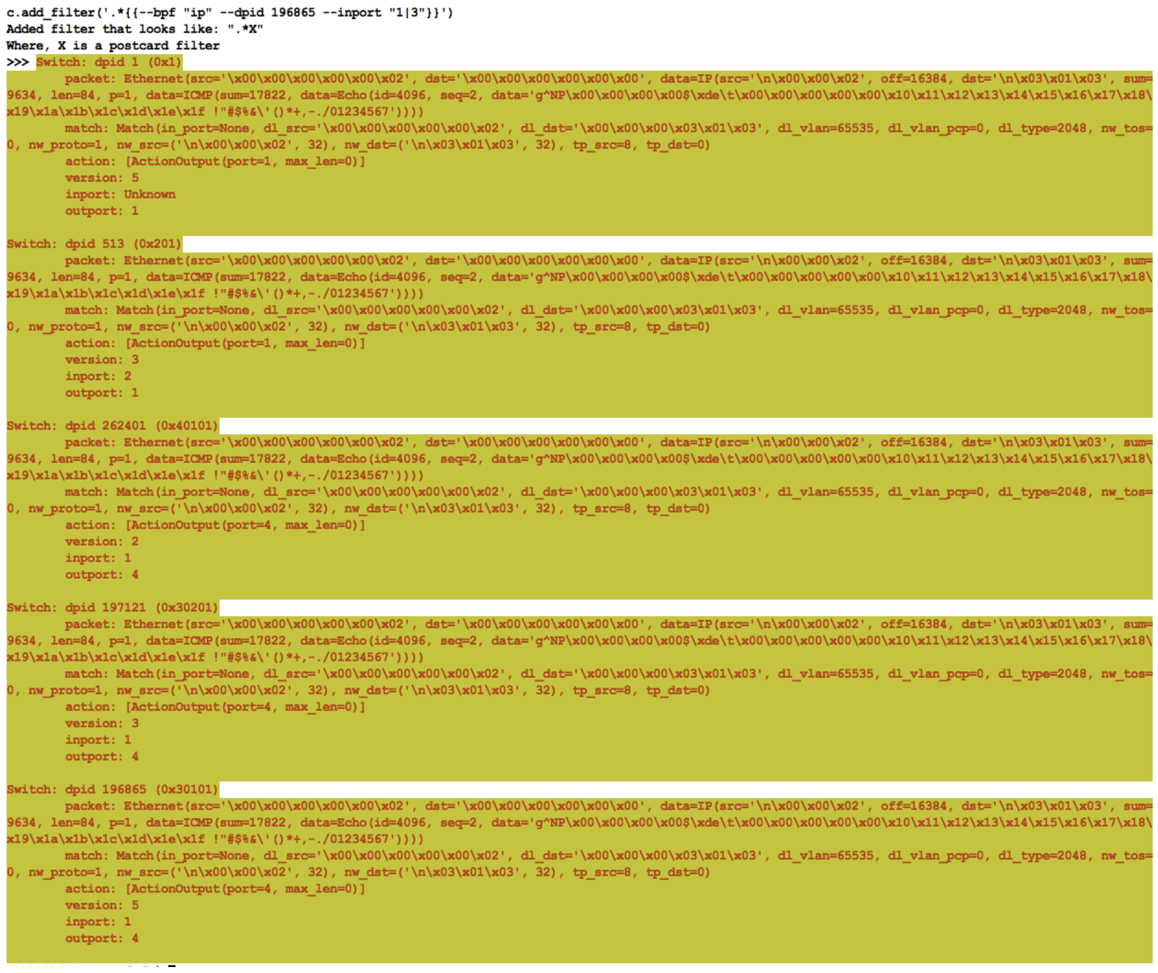
To demonstrate the utility of the NetSight API, I built four troubleshooting applications using it. The two key enablers of these applications are:

1. The ability to access packet histories in real-time or postmortem, and
2. A flexible way to specify packet histories of interest.

Without these two features of NetSight, it would have been either impossible, or at least very hard, to implement the applications. In this chapter, I describe each application—the motivation behind it and its design and implementation, and provide example use cases.

4.1 `ndb`: Interactive Network Debugger

The first application, as well as the original motivating application for NetSight, is `ndb`, an interactive network debugger [24]. The goal of `ndb` is to provide interactive debugging features for networks analogous to those provided by `gdb` for software programs. Using `ndb`, network application developers can set PHFs on errant network behavior. The returned packet histories contain the sequence of switch forwarding events leading to the errant behavior. The following examples cover common bugs for which PHFs installed by `ndb` are a natural fit.



```
c.add_filter('.*{{--bpf "ip" --dpid 196865 --inport "1|3"}}')
Added filter that looks like: ".*X"
Where, X is a postcard filter
>>> Switch: dpid 1 (0x1)
        packet: Ethernet(src='\x00\x00\x00\x00\x00\x02', dst='\x00\x00\x00\x00\x00\x00', data=IP(src='\n\x00\x00\x02', off=16384, dst='\n\x03\x01\x03', sum=
9634, len=84, p=1, data=ICMP(sum=17822, data=Echo(id=4096, seq=2, data='g^NP\x00\x00\x00\x00$\xde\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\
x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'))))
        match: Match(in_port=None, dl_src='\x00\x00\x00\x00\x00\x02', dl_dst='\x00\x00\x00\x03\x01\x03', dl_vlan=65535, dl_vlan_pcp=0, dl_type=2048, nw_tos=
0, nw_proto=1, nw_src=('\n\x00\x00\x02', 32), nw_dst=('\n\x03\x01\x03', 32), tp_src=8, tp_dst=0)
        action: [ActionOutput(port=1, max_len=0)]
        version: 5
        inport: Unknown
        outport: 1

Switch: dpid 513 (0x201)
        packet: Ethernet(src='\x00\x00\x00\x00\x00\x02', dst='\x00\x00\x00\x00\x00\x00', data=IP(src='\n\x00\x00\x02', off=16384, dst='\n\x03\x01\x03', sum=
9634, len=84, p=1, data=ICMP(sum=17822, data=Echo(id=4096, seq=2, data='g^NP\x00\x00\x00\x00$\xde\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\
x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'))))
        match: Match(in_port=None, dl_src='\x00\x00\x00\x00\x00\x02', dl_dst='\x00\x00\x00\x03\x01\x03', dl_vlan=65535, dl_vlan_pcp=0, dl_type=2048, nw_tos=
0, nw_proto=1, nw_src=('\n\x00\x00\x02', 32), nw_dst=('\n\x03\x01\x03', 32), tp_src=8, tp_dst=0)
        action: [ActionOutput(port=1, max_len=0)]
        version: 3
        inport: 2
        outport: 1

Switch: dpid 262401 (0x40101)
        packet: Ethernet(src='\x00\x00\x00\x00\x00\x02', dst='\x00\x00\x00\x00\x00\x00', data=IP(src='\n\x00\x00\x02', off=16384, dst='\n\x03\x01\x03', sum=
9634, len=84, p=1, data=ICMP(sum=17822, data=Echo(id=4096, seq=2, data='g^NP\x00\x00\x00\x00$\xde\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\
x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'))))
        match: Match(in_port=None, dl_src='\x00\x00\x00\x00\x00\x02', dl_dst='\x00\x00\x00\x03\x01\x03', dl_vlan=65535, dl_vlan_pcp=0, dl_type=2048, nw_tos=
0, nw_proto=1, nw_src=('\n\x00\x00\x02', 32), nw_dst=('\n\x03\x01\x03', 32), tp_src=8, tp_dst=0)
        action: [ActionOutput(port=4, max_len=0)]
        version: 2
        inport: 1
        outport: 4

Switch: dpid 197121 (0x30201)
        packet: Ethernet(src='\x00\x00\x00\x00\x00\x02', dst='\x00\x00\x00\x00\x00\x00', data=IP(src='\n\x00\x00\x02', off=16384, dst='\n\x03\x01\x03', sum=
9634, len=84, p=1, data=ICMP(sum=17822, data=Echo(id=4096, seq=2, data='g^NP\x00\x00\x00\x00$\xde\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\
x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'))))
        match: Match(in_port=None, dl_src='\x00\x00\x00\x00\x00\x02', dl_dst='\x00\x00\x00\x03\x01\x03', dl_vlan=65535, dl_vlan_pcp=0, dl_type=2048, nw_tos=
0, nw_proto=1, nw_src=('\n\x00\x00\x02', 32), nw_dst=('\n\x03\x01\x03', 32), tp_src=8, tp_dst=0)
        action: [ActionOutput(port=4, max_len=0)]
        version: 3
        inport: 1
        outport: 4

Switch: dpid 196865 (0x30101)
        packet: Ethernet(src='\x00\x00\x00\x00\x00\x02', dst='\x00\x00\x00\x00\x00\x00', data=IP(src='\n\x00\x00\x02', off=16384, dst='\n\x03\x01\x03', sum=
9634, len=84, p=1, data=ICMP(sum=17822, data=Echo(id=4096, seq=2, data='g^NP\x00\x00\x00\x00$\xde\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\
x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'))))
        match: Match(in_port=None, dl_src='\x00\x00\x00\x00\x00\x02', dl_dst='\x00\x00\x00\x03\x01\x03', dl_vlan=65535, dl_vlan_pcp=0, dl_type=2048, nw_tos=
0, nw_proto=1, nw_src=('\n\x00\x00\x02', 32), nw_dst=('\n\x03\x01\x03', 32), tp_src=8, tp_dst=0)
        action: [ActionOutput(port=4, max_len=0)]
        version: 5
        inport: 1
        outport: 4
```

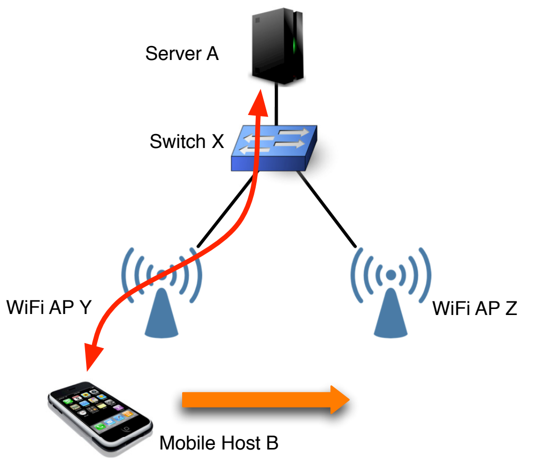
Figure 4.1: A screenshot of the packet history output of `ndb`.

4.1.1 Reachability Error

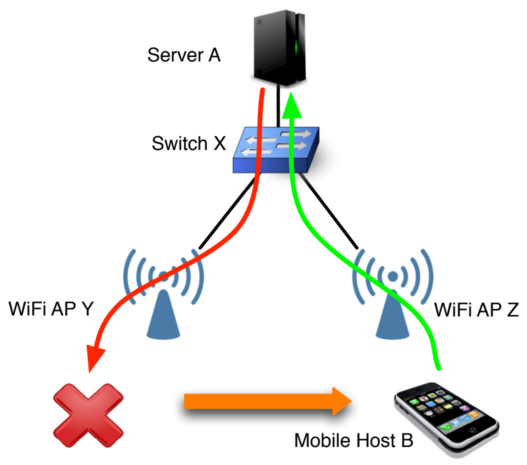
Consider the common reachability problem of two hosts, A and B, which are unable to talk to each other. Using `ndb`, the developer would use a PHF to specify packets from A destined for B that never reach the intended final hop, the switch to which B is attached:

```
^{{--bpf "ip src A and dst B" --dpid X --inport p1}}
    [^{{--dpid Y --outport p2}}]*$
```

where, (X, p1) and (Y, p2) are the (switch, port) tuples at which hosts A and B are attached, respectively. The returned sequences of events that led to the observed



(a) WiFi host B attached to AP Y sends periodic ICMP `ping` requests to server A.



(b) When host B moves to AP Z, the ICMP reply packets continue going to AP Y and get dropped there.

Figure 4.2: Network topology used to emulate the incomplete handover bug.

errant behavior would aid the developer in homing in on the root cause of the problem.

Using the Mininet network emulator [26], I reproduce the incomplete handover bug encountered at our production OpenFlow network at Stanford described in Section 2.1.2. Figure 4.2 shows the topology of the emulated network, with a mobile host B attached to access point (AP) Y, continuously sending ICMP `ping` requests to a server A. The network has a (buggy) OpenFlow controller running as the mobility manager that manages flows as WiFi hosts move from one AP to another. When the mobile client moves to AP Z, the controller correctly installs new flow entries for the ICMP `ping` request from B to A, but does not do so for ICMP replies coming back from A to B. Consequently, the ping replies continue going to AP Y and get dropped there. Thus, when the mobile host moves from AP Y to AP Z, it stops receiving ICMP `ping` replies.

Using `ndb`, I install the following PHF:

```
^{{--bpf "icmp and ip src A and dst B" --dpid X}} {{--dpid not Z}}*$
```

The packet history output from `ndb` shows us the exact path taken by the ICMP reply packet and the flow table state it encountered at each hop along the path, telling us why it took that path. This packet history provides direct evidence to show that the flow entry for ICMP reply packets was not updated by the controller, helping us identify the bug in minutes instead of hours.

Figure 4.1 shows an example output from `ndb`.

4.1.2 Race condition

A controller may insert new flow entries on multiple switches in response to network events such as link failures or new flow arrivals. If a controller's flow entry insertions are delayed, packets can get dropped, or the controller can get spurious "packet-in" notifications. The following Packet History Filter captures such events, by matching packet histories that *terminate* at switch X, port p:

```
{{--dpid X --inport p}}$
```

4.1.3 Incorrect packet modification

A packet can be modified at any hop in the network. The presence of a large number of rules at each hop makes it difficult to diagnose packet modification errors. Packets reaching the destination with unexpected headers due to incorrect modifications can be captured by the following Packet History Filter:

```
^{{--bpf "BPF1"}}.*{{--bpf "BPF2"}}$
```

Where `BPF1` is a Berkeley Packet Filter matching the packet as it entered the network and `BPF2` is the observed (incorrect) packet as it reaches the destination.

4.2 `netwatch`: Live Invariant Monitor

The second application is `netwatch`, a live network invariant monitor. `netwatch` allows the operator to specify desired network behavior in the form of invariants, and trigger alarms whenever a packet violates any invariant (e.g., freedom from traffic loops). `netwatch` is a library of invariants written using Packet History Filters to match packets that violate those invariants. Once Packet History Filters are pushed to NetSight, the callback returns the packet history that violated the invariant(s). The callback not only notifies the operator of an invariant violation, but the Packet

History Filter provides useful context around *why* it happened. `netwatch` currently supports the following network invariants:

4.2.1 Isolation

Hosts in group A should not be able to communicate with hosts in group B. An alarm should be raised whenever this condition is violated. The function `isolation(a_host_set, b_host_set, topo)` pushes down two Packet History Filters:

```
^{{ GroupA }}.*{{ GroupB }}$
^{{ GroupB }}.*{{ GroupA }}$
```

GroupA and GroupB can be described by a set of host IP addresses, or by network locations (switch, port) where the hosts are attached. This Packet History Filter matches packets that are routed from group A to group B. Extending `isolation` to multiple groups is easy.

4.2.2 Loop Freedom

The network should have no traffic loops. The function `loop_freedom()` pushes down one Packet History Filter:

```
(.).*(\1)
```

This Packet History Filter matches any packet history where the packet goes through a forwarding loop.

4.2.3 Black Hole Freedom

Packets should start at an edge port and finish at an edge port; they should never get dropped in the middle. The function `blackhole_freedom(topo)` pushes down Packet History Filters of type:

```
{{--outport "core_port"}}$
```

where core ports are the network internal ports derived from the supplied topology. Such an invariant could be used to correlate edge-to-non-edge packet histories with drops. If the drop rate at a port is abnormally high, it indicates a silent black hole or link failure.

4.2.4 Waypoint routing

Certain types of traffic should go through specific waypoints. For example, all HTTP traffic should go through the proxy, or guest traffic should go through the IDS and the firewall. The function `waypoint_routing(traffic_class, waypoint_id)` installs a Packet History Filter of the form:

```
{{--bpf "traffic_class" --dpid not "waypoint_id"}}{{--dpid not "waypoint_
id"}}*$
```

This Packet History Filter catches packet histories of packets that belong to `traffic_class` and never go through the specified waypoint.

4.2.5 Max-path-length

No path should ever exceed a specified maximum length, such as the diameter of the network. The function `max_path_length(n)` installs a Packet History Filter of the form:

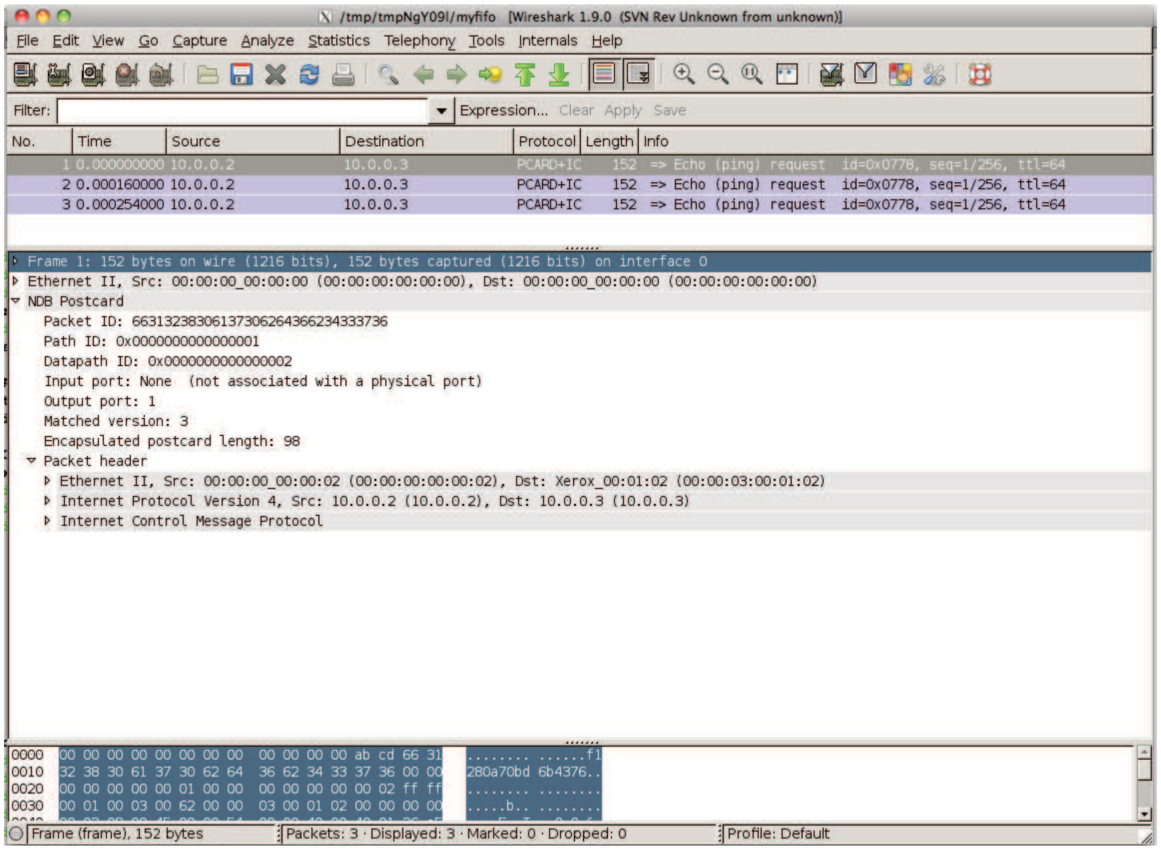
```
.{n+1}
```

This Packet History Filter catches all paths whose lengths exceed `n`.

4.3 `netshark`: Network-wide Path-Aware Packet Logger

The third application is `netshark`, a `wireshark`-like application that allows users to set filters on the entire history of packets, including their paths and header values at each hop.

For example, a user could look for all HTTP packets with src IP A and dst IP B arriving at (switch X, port p) that have also traversed through switch Y. `netshark` accepts Packet History Filters from the user, returns the collected packet histories


/tmp/tmpNgY09l/myfifo [Wireshark 1.9.0 (SVN Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No.  Time         Source    Destination  Protocol  Length  Info
  1 0.000000000 10.0.0.2  10.0.0.3     PCARD+IC     152  => Echo (ping) request  id=0x0778, seq=1/256, ttl=64
  2 0.000160000 10.0.0.2  10.0.0.3     PCARD+IC     152  => Echo (ping) request  id=0x0778, seq=1/256, ttl=64
  3 0.000254000 10.0.0.2  10.0.0.3     PCARD+IC     152  => Echo (ping) request  id=0x0778, seq=1/256, ttl=64

▷ Frame 1: 152 bytes on wire (1216 bits), 152 bytes captured (1216 bits) on interface 0
▷ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▽ NDB Postcard
    Packet ID: 66313238306137306264366234333736
    Path ID: 0x0000000000000001
    Datapath ID: 0x0000000000000002
    Input port: None  (not associated with a physical port)
    Output port: 1
    Matched version: 3
    Encapsulated postcard length: 98
  ▽ Packet header
    ▷ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: Xerox_00:01:02 (00:00:03:00:01:02)
    ▷ Internet Protocol Version 4, Src: 10.0.0.2 (10.0.0.2), Dst: 10.0.0.3 (10.0.0.3)
    ▷ Internet Control Message Protocol

0000  00 00 00 00 00 00 00 00  00 00 00 00 ab cd 66 31   ........ ......f1
0010  32 38 30 61 37 30 62 64  36 62 34 33 37 36 00 00   280a70bd 6b4376..
0020  00 00 00 00 00 01 00 00  00 00 00 00 00 02 ff ff   ........ ........
0030  00 01 00 03 00 62 00 00  03 00 01 02 00 00 00 00   .....b.. ........

Frame (frame), 152 bytes   Packets: 3 · Displayed: 3 · Marked: 0 · Dropped: 0   Profile: Default
```

Figure 4.3: A screenshot of the netshark wireshark output.

matching the query, and includes a `wireshark` dissector to analyze the results. Figure 4.3 shows a screenshot of the `netshark wireshark` output. The user can then view properties of a packet at a hop (packet header values, switch ID, input port, output port, and matched flow table version) as well as properties of the packet history to which it belongs (packet ID, path traversed, and path length).

4.4 `nprof`: Hierarchical Network Profiler

Software profiling tools like `gprof` show users where a program spent its time, split at the granularity of a function, along with which functions called which other functions while it was executing. This information reveals those pieces of a program that might yield the most benefit when optimized.

App	`ndb`	`netwatch`	`netshark`	`nprof`
Lines of Code	32	103	179	111

Table 4.1: Application sizes (lines of code).

The fourth troubleshooting application is `nprof`, a hierarchical network profiler. The goal of `nprof` is to "profile" any collection of network links to understand the traffic characteristics and routing decisions that contribute to link utilization. For example, to profile a particular link, `nprof` first pushes a Packet History Filter specifying the link of interest:

```
{{--dpid X --outport p}}
```

`nprof` combines the resulting packet histories with the topology information to provide a live hierarchical profile, showing which switches are sourcing traffic to the link, and how much. The profile tree can be further expanded to show which particular flow entries in those switches are responsible.

`nprof` can be used to not only identify end hosts (or applications) that are congesting links of interest, but also to identify how a subset of traffic is being routed across the network. This information can suggest better ways to distribute traffic in the network, or show packet headers that cause uneven load distributions on routing mechanisms, such as equal/weighted cost multi-path routing.

The Python versions of the 4 applications described above—`ndb`, `netwatch`, `netshark`, and `nprof` —are each less than 200 lines of code, as shown in Table 4.1.

Chapter 5

Scalability and Performance

While NetSight supports a versatile and flexible abstraction, to be really useful, it needs to be able to scale to handle the traffic generated by a large operational network. It is natural to doubt the scalability of any system that attempts to store the header of every packet traversing a network along with its corresponding path, state, and modifications and also to apply complex filters to it. This is a lot of data to forward, let alone process and archive. In this chapter, I study the cost associated with running NetSight by evaluating the performance of each component of NetSight, and I suggest modifications to the NetSight design that will enable it to scale to large networks handling multiple gigabytes of data per second.

5.1 NetSight Scalability

The analysis of NetSight's scalability can be divided into two components:

1. Scalability in the control path
2. Scalability in the datapath

5.1.1 Scalability in the Control Path

In the current implementations of software-defined networks (SDN), the control channel tends to be a scarce resource. It is therefore important for NetSight to impose minimal cost on the control channel. The two costs that we need to be concerned about are:

- **Latency cost**: The extra latency incurred by control messages traversing through NetSight
- **Bandwidth cost**: The extra traffic on the control channel caused due to NetSight

Latency Cost

Any control path cost of NetSight is due to the NetSight recorder. Past works—Flowvisor [56], NetPlumber [36], VeriFlow [38]—have shown that it is feasible to build an SDN control channel proxy that adds very little extra latency to the control messages. The task performed by the NetSight recorder involves less computation than that of Flowvisor, NetPlumber, or VeriFlow; NetSight only records the control message, adds pre-computed extra actions, and asynchronously stores the message in a database, as described in Section 3.2.1. Therefore, the latency cost is small.

Bandwidth Cost

In today's SDN switches, the bandwidth of the control channel (the connection between the switch datapath and the controller) tends to be a scarce resource. The current commercial SDN switches support only a few hundred flow insertion messages (`flow-mod`) and `packet-in` messages per second. This bottleneck resource should not be made scarcer by any new element added on the control channel. The NetSight recorder adds no additional (expensive) flow insertions on the control channel; it only adds extra actions to flow insertion messages already created by the SDN controller. Recent work has demonstrated OpenFlow controllers that scale to millions of messages per second, beyond the capabilities of entire networks of hardware switches [60]. In addition, the postcards generated by NetSight are strictly confined to the datapath; they do not enter the control path. Therefore, the bandwidth cost is small, too.

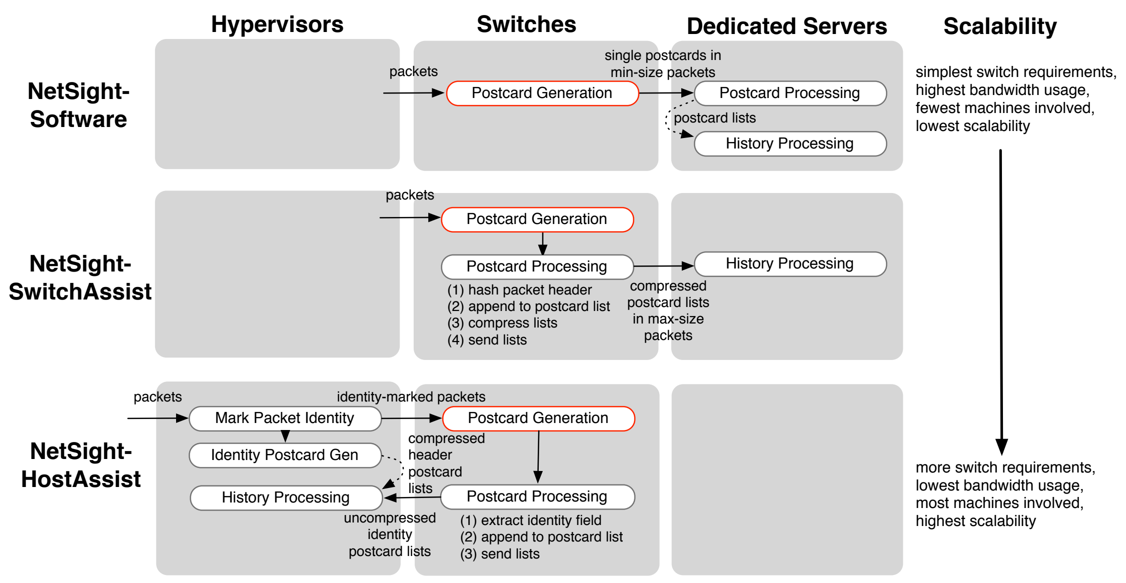


Figure 5.1: NetSight places the bulk of its functionality in dedicated servers, but switch hardware resources and virtual machine hosts can also be employed to reduce its bandwidth costs and to increase its scalability. The common element here is the generation of postcards at switches.

5.1.2 Scalability in the Datapath

Recall that every packet creates a postcard at every hop across which it traverses in the network. This clearly creates a bandwidth cost in the datapath. If we do not compress postcards before sending them over the network, we need to send them each as a min-sized packet. We can calculate the bandwidth cost in the datapath as a fraction of the data traffic as,

$$\text{bandwidth cost} = \frac{\text{postcard packet size}}{\text{avg. packet size}} * \text{avg. number of hops} \qquad (5.1)$$

For example, the Stanford campus backbone network has the following characteristics: 14 internal routers connected by 10 Gb/s links, two Internet-facing routers, a network diameter of 5 hops, and an average packet size of 1031 bytes. If we assume that postcards are minimum-sized Ethernet packets, they increase traffic by

$\frac{64B}{1031B} \times 5(hops) = 31\%$. [1]

The average aggregate utilization of the Stanford backbone is about 5.9 Gb/s, for which the postcard traffic adds 1.8 Gb/s. If we conservatively add together the peak traffic of every campus router, it leads to a total of 25 Gb/s of packet data, which will, in turn, generate 7.8 Gb/s of postcard traffic. This postcard traffic can be handled by one NetSight server, as I will show later in Section 5.4. If the postcards are sent in-band, the extra traffic will likely affect the network performance.

For a low-utilization network, especially one in the bring-up phase or a test network, the bandwidth costs may be acceptable for the debugging functionality that NetSight provides. However, for a live production network with longer paths, or with smaller packets, or with higher traffic rate, the bandwidth cost of NetSight, as currently designed, will likely be too high.

To scale NetSight to a large data center or enterprise, I next present two design modifications to reduce the datapath bandwidth cost while scaling out postcard collection, packet history assembly, and packet history archival. The modifications move postcard generation functionality into switch ASICs and end hosts, as shown in Figure 5.1. These modifications trade off deployment convenience for improved scalability.

5.2 NetSight Design Modifications for Scalability

5.2.1 NetSight-SwitchAssist

NetSight-SwitchAssist reduces the bandwidth cost of sending postcards in the datapath. In this design modification, postcard processing is directly handled by the switches; switch ASICs directly generate, compress, and shuffle postcards. History processing continues to be handled by the NetSight servers.[2] Because switches send

[1] If we overcome the min-size requirement by aggregating the 40 byte postcards into larger packets before sending them, the bandwidth cost reduces to 19%.

[2] This design is functionally equivalent to directly attaching a NetSight postcard processor to each switch.

compressed aggregates of postcards to NetSight servers (rather than individual uncompressed postcards), the bandwidth requirement diminishes. For example, as I will show in Section 5.3, with NetSight's compression algorithm, the average size of a postcard comes down to 15 bytes per compressed postcard. Using this number in Equation 5.1 leads to a bandwidth cost of:

$$\frac{15 \text{ Bytes}}{1031 \text{ Bytes}} * 5 \text{ hops} = 0.07$$

The additional bandwidth requirement in the datapath reduces from 31% to 7%.

Postcard processing at the switch is simple enough to be implemented in hardware with relatively few pieces of logic. The first is the logic to extract and hash flow IDs to index into a small SRAM table and identify the destination NetSight server. This hardware is identical to Equal-Cost Multi-Path (ECMP), which selects the next hop for a packet using a hash of its 5-tuple flow ID. The second addition is logic to compress postcards. The compressor described in Section 3.2 could be implemented in hardware using a small amount of SRAM to cache recently seen packets and group them by flow IDs.[3] The memory only stores the "diffs" between successive packets and places them in separate buffers, one each per NetSight server. I believe it would be practical for future switch chips to incorporate this logic running at line-rate.

5.2.2 NetSight-HostAssist

The next design modification, NetSight-HostAssist, is suited for environments such as data centers, where the end hosts can be modified. This design modification reduces postcard traffic by having a thin shim layer at each end host (e.g., in the hypervisor or in a software switch such as Open vSwitch [46]) tag packets to help switches to succinctly describe postcards. The shim layer at the end host tags each outgoing packet with a sequentially incrementing packet ID and locally stores a copy of the packet header and a mapping between the packet ID and the packet header. One way to incorporate packet IDs into the existing packet header structure is by adding it as an IP option. The shim uses the NetSight compression algorithm to reduce the

[3]100,000 postcards would require 4 MB of memory.

cost of locally storing a copy of the outgoing packet headers. When a switch receives a packet, it extracts the tag and generates a mini-postcard that contains only the packet ID, the flow table state, and the input/output ports. This state is appended to a hash table entry keyed by the source address of this packet. At the end of each round, the switch dispatches the hash entry (a list of packet IDs and state) to the source. The hosts, then, locally assemble and archive the packet history (as shown at the bottom of Figure 5.1).

Because a packet ID is unique to a particular end host, the shim can use fewer bytes (e.g., 4 bytes) to uniquely identify a packet. For example, for traffic with an average packet size of 1031 bytes, the amount of data that an end-host can send with a packet ID of 4 bytes before wrapping around is:

$$1031 \text{ Bytes} * 2^{32} \text{ Packet IDs} = 4428 \text{ GBytes}$$

In other words, for a host with a 10 Gb/s link, when sending traffic at full line-rate, it will take more than 3500 seconds for a 4-byte packet ID field to wrap around.

If, on average, it takes 15 bytes per packet to store compressed headers at the end hosts (as shown in Section 5.3) and 6 bytes per mini-postcard, the bandwidth cost to collect postcards in the network reduces to 3%.

$$\frac{6 \text{ Bytes}}{1031 \text{ Bytes}} * 5 \text{ hops} = 0.03$$

This is in contrast with a cost of 31% if postcards are naively collected as min-sized packets, each without any compression. Because each end host stores packet histories of its own traffic, the mechanism scales with the number of hosts in the network.

The task of the switch is simpler in NetSight-HostAssist than it is in NetSight-SwitchAssist. For each arriving packet, the switch needs to read a fixed header value (the packet ID) and create mini-postcards with the stored packet IDs, the corresponding flow table state, and the input/output ports. This task is simple enough to be implemented in the switch ASIC. At the end of each round, it needs to send the mini-postcards to the appropriate end host, a task manageable by the switch CPU or an attached processing unit.

The NetSight-HostAssist design has a few caveats besides ASIC implementation costs:

- The packet ID should be correctly preserved as a packet traverses through the network.
- The extra packet ID adds a fixed-size cost to every packet.
- The traffic arriving to the datacenter from the Internet needs special handling. Such traffic should either be forwarded through middleboxes, or through border routers that can generate full postcards and insert packet IDs into packets.

5.2.3 Distributed Filtering with Two-Stage Filtering

Many troubleshooting applications written on top of NetSight (e.g., `netwatch`, `netshark`, and `nprof`) only install "live" PHF triggers and are interested only in a subset of all of the packet histories. One way to reduce the bandwidth cost of the system in such cases is by efficiently distributing the filtering (PHF matching) itself and by pruning out *useless* postcards even before they are assembled into packet histories at a NetSight server. In this context, useless postcards are those that belong to a packet history that does not match any of the live PHF triggers. This problem is analogous to distributed string regex matching, when the characters of the string themselves are dispersed across multiple machines. The goal of distributed PHF matching is to minimize the number of postcards collected at any centralized location for packet history assembly. NetSight's solution, two-stage filtering (TSF), leverages additional processing elements attached to the network, called *secondary servers* (secondaries for short). These can be server-level CPUs on a switch [1, 7], directly attached CPU linecards [3, 6], virtual machines at the edge, or any commodity server attached to the network.

As the name suggests, TSF works in two stages.

Stage 1

In the first stage, secondary servers prune out useless postcards using the concept of *imperative predicates.* Imperative predicates are the postcard filters (PF) in a PHF

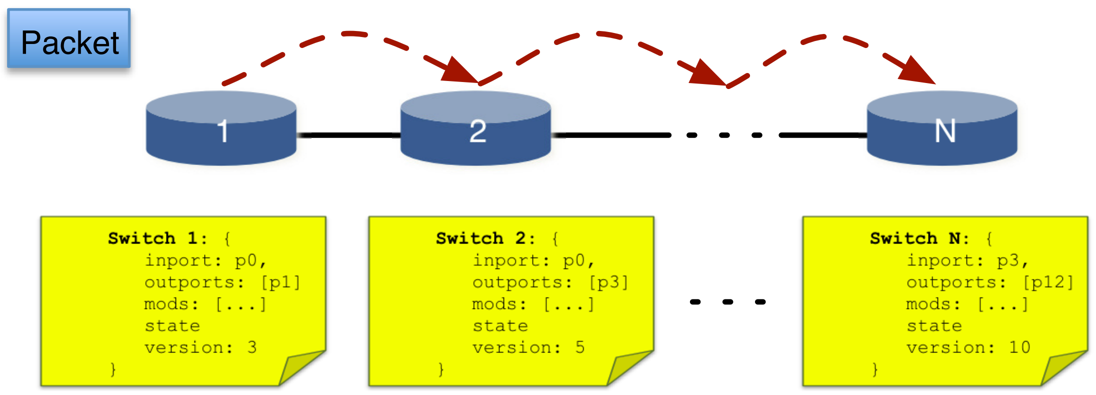


Figure 5.2: A packet traversing a path from switch 1 to switch N can be represented by the corresponding sequence of postcards.

that can be verified with local knowledge (i.e., from one switch) and must be matched by at least one postcard in a matching packet history.

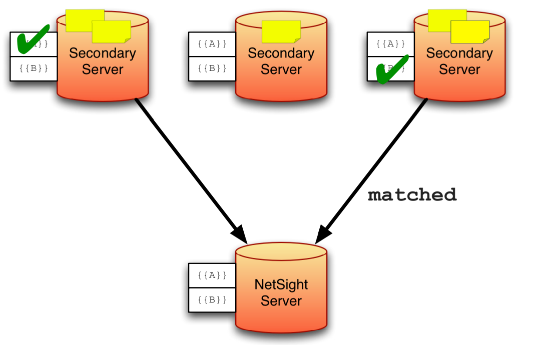
For example, suppose we have a live PHF trigger "`^{{--dpid 1}} .* {{--dpid N}}$`" that matches all packet histories starting at switch 1 and ending at switch `N`. Suppose we have a packet traversing a path from switch 1 through switch `N`. Its packet history can be represented by a set of topologically sorted postcards generated by the packet along its path, as shown in Figure 5.2. Now, the packet history will match the PHF only if it contains at least one postcard matching the PF "`--dpid 1`" and at least one other postcard matching the PF "`--dpid N`." Therefore, the PFs "`--dpid 1`" and "`--dpid N` are the imperative predicates. As another example, in a PHF that looks like "`^{{--dpid 1}} {{--dpid K}}* {{--dpid N}}$`," only "`--dpid 1`" and "`--dpid N`" are imperative predicates, as a matching packet history can have zero or more postcards matching the PF "`--dpid K`."

As a secondary collects postcards, it temporarily stores them in a local buffer.[4] When a postcard matches any imperative predicate, it immediately sends a small `matched` notification to the NetSight server:

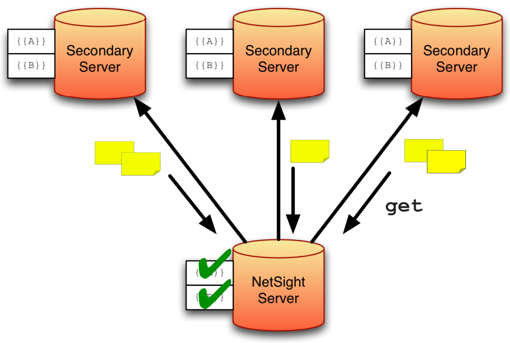
```
matched(pkt_id, list(matched_imperative_predicates))
```

as shown in Figure 5.3(a). If a PHF contains no imperative predicates, the secondaries

[4]This buffer should be sized based on the network RTT, switch data rate, and the round duration.



(a) Stage-1: The secondary servers send notify messages to the NetSight server for any postcard matching an imperative preciate.



(b) Stage2: If the NetSight server receives notify messages for all imperative predicates with a particular packet ID, it gets all the postcards with that packet ID from the secondaries.

Figure 5.3: Two-Stage Filtering.

send `matched` notifications for all postcards.

Stage 2

The NetSight server waits for `matched` notifications from the secondaries, up to a short configurable collection timeout. If it receives `matched` notifications for all imperative predicates in a PHF for a particular packet ID, it proceeds to query all of the secondaries for postcards with that packet ID (Figure 5.3(b)). Once the server receives all of the postcards matching that packet ID, it performs the regular history assembly and PHF matching, as described in Section 3.1.3 and Section 3.1.3, respectively.

I next discuss the pros and cons of TSF.

Pros of Two-Stage Filtering

- Simplicity of the architecture: TSF does not depend on the number of secondary servers or how the switches are assigned to them. In the extreme case, the operator can choose to have one secondary per switch in the network.

- Simplicity of the communication: There are no complex queries at secondaries, and secondaries only interact with the NetSight servers, not one another.

Cons of Two-Stage Filtering

- TSF works only in cases where the network operator is interested only in matching packet histories against live PHF triggers and not in archiving them. In other words, if TSF is used, the operator cannot run PHF queries on archived packet histories.
- TSF captures only the presence or absence of postcards matching the imperative predicate, not their order. This results in false positives, where some “useless” postcards do not get pruned out at the secondaries.
- TSF becomes ineffective when the PHF contains no imperative predicates.

5.3 NetSight Performance Evaluation

This section quantifies the performance of the mechanisms that comprise NetSight in an effort to investigate the feasibility of collecting and storing every packet history. Figure 5.4 show the various computational pieces that comprise NetSight: compression, decompression, packet history assembly, and PHF matching. From the speed of each piece, we can estimate the data rate that a single CPU core can process and determine the number of NetSight servers needed to support deployments in the enterprise, data center, and wide-area networks (WAN).

In the remainder of this section, I will evaluate the performance of each computational piece of NetSight and use the combined performance to study the feasibility of running NetSight in a production network. All of the evaluations are on a single core of a 3.20 GHz Intel(R) Core(TM) i7 CPU with 12 GB RAM, unless stated otherwise.

5.3.1 Compression

NetSight compresses postcards before the shuffle phase to reduce network bandwidth, and then it compresses packet histories again during the archival phase to reduce

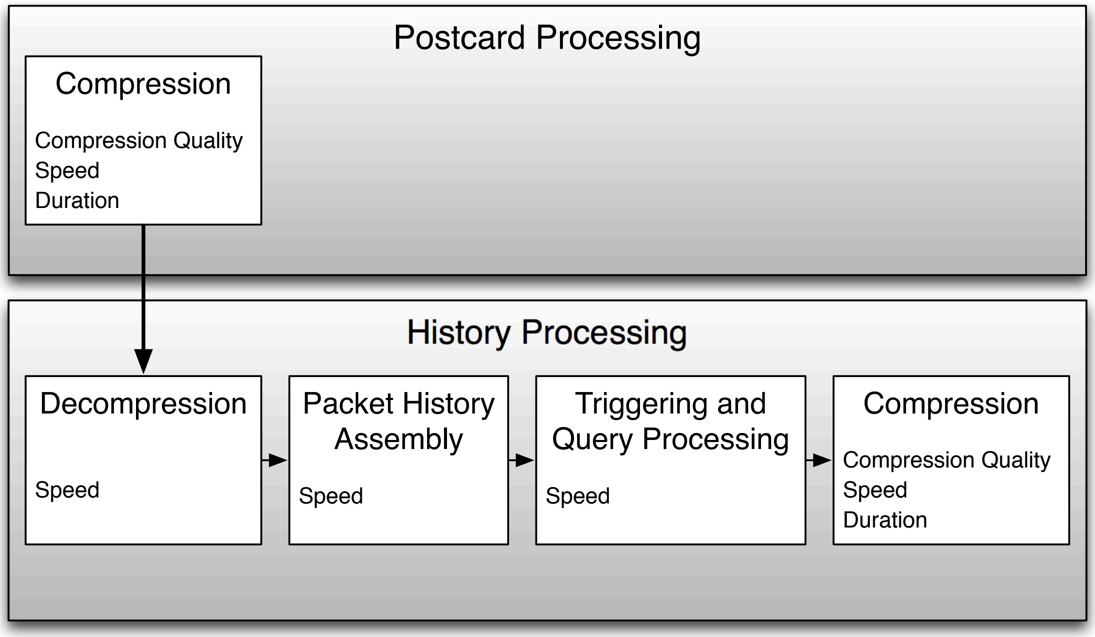


Figure 5.4: The mechanisms that comprise NetSight: compression, decompression, packet history assembly, and PHF matching.

storage costs. This section investigates three questions:

- **Compression:** How tightly can NetSight compress packet headers, and how does this compare with off-the-shelf options?
- **Speed:** How expensive are the compression and decompression routines, and what are their time vs. size tradeoffs?
- **Duration:** How does the round length (time between data snapshots) affect compression properties, and is there a sweet spot?

Traces

To answer performance questions, I use thirteen packet capture (pcap) data sets: one from a university enterprise network (UNIV), two from university data centers (DCs),

Compression Type	Description
Wire	Raw packets on the wire
PCAP	All IP packets, truncated up to layer 4 headers
gzip	PCAP compressed by gzip level 6
NetSight (NS)	Van Jacobson-style compression for all IP 5-tuples
NetSight + gzip (NS+GZ)	Compress `DiffRecords` with gzip level 1

Table 5.1: Compression techniques.

and nine from a wide-area network (WAN). I preprocessed all traces and removed all non-IPv4, non-TCP, and non-UDP packets, and I stripped packet headers beyond the layer 4 TCP header, which accounted for less than 1% of all of traffic.[5] UNIV is the largest trace (31 GB pcap trace) that contains headers with an average packet size of 687 bytes per packet, collected over two hours on a weekday afternoon. The average flow size over the duration of this trace is 76 packets. The data center traces DC1 and DC2 have a larger average packet size (about 765 bytes and 715 bytes per packet, respectively) and a larger average flow size (about 333 packets per flow). However, in the WAN traces, I observed that flows have less than 3 packets over the duration the trace.

The UNIV trace contains packets seen at one core router connecting Clemson University to the Internet. The data center traces—DC1 and DC2—are from [13], whose IP addresses were anonymized using SHA1 hash. Finally, each WAN trace (from [2]) accounts for a minute of packet data collected by a hardware packet capture device. IP addresses in this trace are anonymized using a CryptoPan prefix-preserving anonymization.

[5]In a production deployment, such packets can either be ignored, or if necessary for troubleshooting, used uncompressed without incurring much performance overhead.

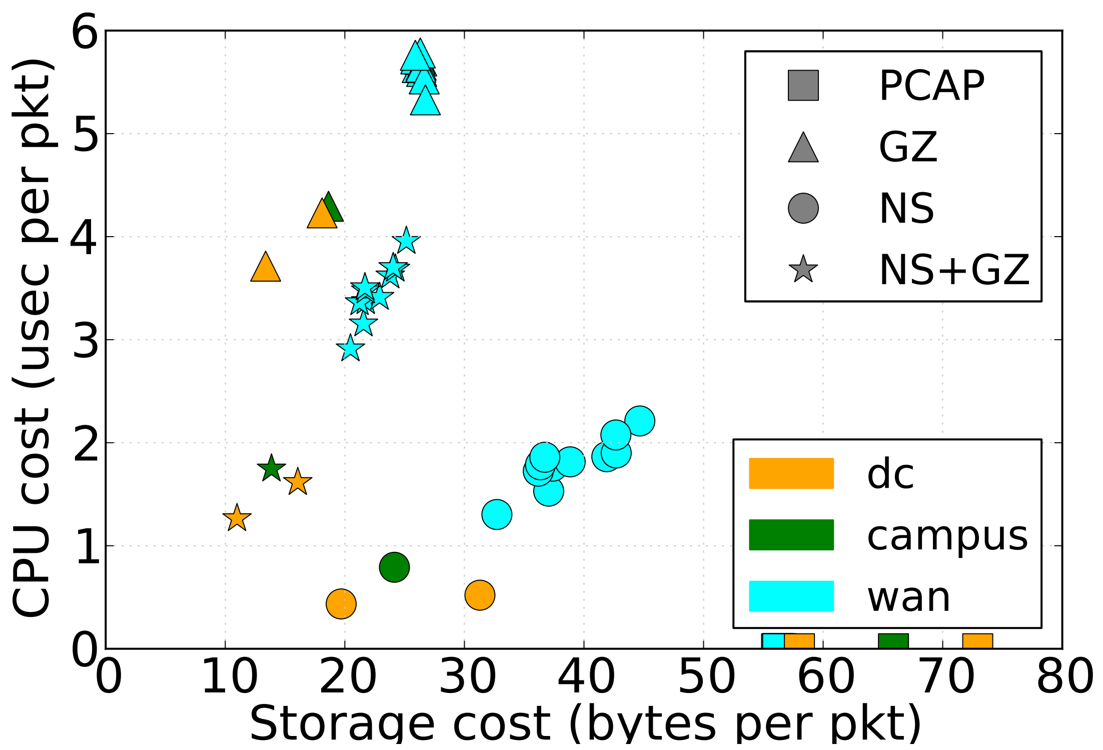


Figure 5.5: NetSight reduces storage relative to PCAP files, at a low CPU cost. Combining NS with gzip (NS+GZ) reduces the size better than gzip does, at a fraction of gzip's CPU costs. The WAN traces compress less as they have fewer packets in a flow compared with other traces.

Storage vs CPU Costs

Figure 5.5 answers many of the performance questions, showing the tradeoff between compression storage costs and CPU costs for different traces and compression methods. This graph compares four candidate methods, listed in Table 5.1: (a) PCAP: the uncompressed list of packet headers, (b) gzip compression run directly on the pcap file, (c) NS: the adaptation of Van Jacobson's compression algorithm, and (d) NS+GZ: output of (c) followed by gzip compression (level 1, fastest). Each one is lossless with respect to headers; they recover all header fields and timestamps and maintain the packet ordering.

We find that all candidates reduce storage relative to PCAP files by up to 4x and, as expected, their CPU costs vary. GZ, an off-the-shelf option, compresses well but has a higher CPU cost than do both NS and NS+GZ, which leverage knowledge of packet formats in their compression. *NetSight uses NS+GZ because for every trace, it compresses better than pure GZ does, at a reasonably low CPU cost.*

We also find that the compressed sizes depend heavily on the average flow size of the trace. Most of the benefits come from storing differences between successive packets of a flow, and a smaller average flow size reduces opportunities to compress. We see this in the WAN traces, which have shorter flows and compress less. Most of the flow entropy is in a few fields such as IP identification (IP ID), IP checksums, and TCP checksums, and the cost of storing diffs for these fields is much lower than the cost of storing a whole packet header.

To put these speeds into perspective, consider our most challenging scenario, NS+GZ in the WAN, shown by the blue stars. The average process time per packet is 3.5μs, meaning that one of the many cores in a modern CPU can process 285,000 postcards/sec. Assuming an average packet size of 600 bytes, this translates to about 1.37 Gb/s of network traffic, and this number scales linearly with the number of cores. Moreover, the storage cost (for postcards) is about 6.84 MB/s; a 1 TB disk array can store all postcards for an entire day. Most of this storage cost goes into storing the first packet of a flow. As the number of packets per flow increases (e.g., in datacenter traces), the storage costs reduce further.

Duration

A key parameter for NetSight is the round length. Longer rounds present opportunities for better postcard compression, but they increase the delay until the applications see matching packet histories. On the other hand, smaller rounds reduce the size of the hash table used to store flow keys in NS compression, which speeds up software implementations and makes hardware implementations feasible. Figure 5.6 shows the compression quality of NS+GZ for varying number of packets in a round. This graph suggests that a round length of 1 million postcards is sufficient to attain most compression benefits. On most lightly loaded 10 Gb/s links, this translates to about

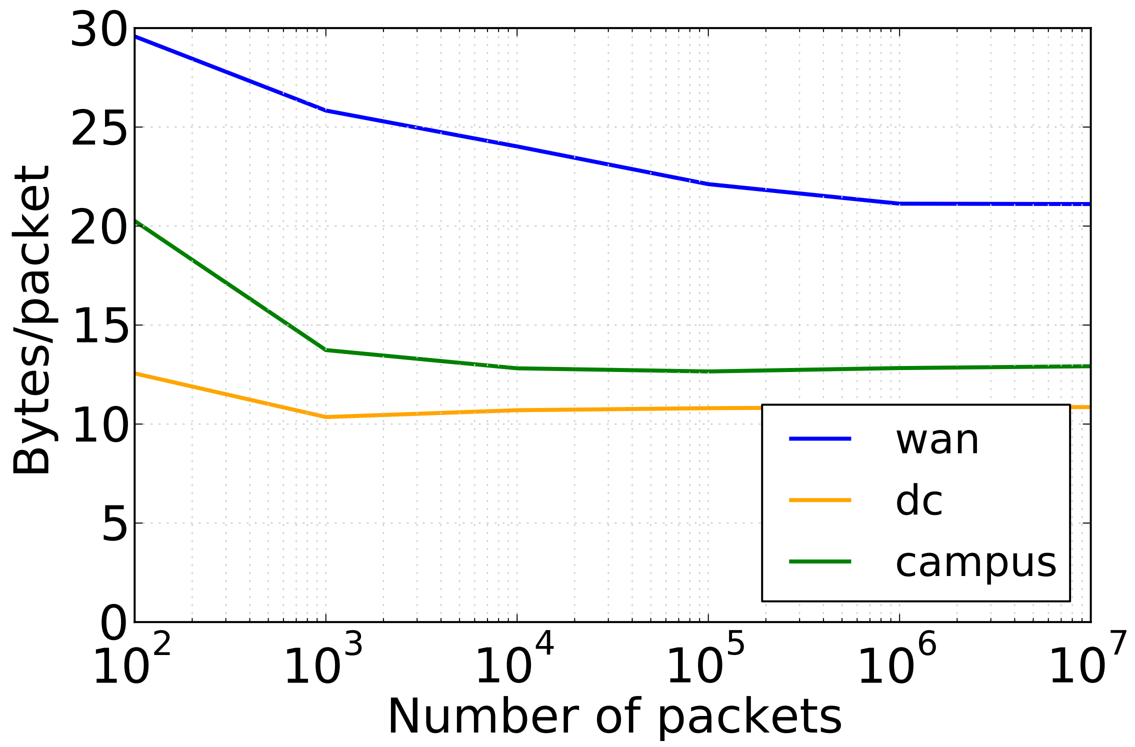


Figure 5.6: Packet compression quality for NS+GZ as a function of seen packets in the trace. In our traces from three operating environments, we find that NetSight quickly benefits from compression after processing a few 100s of thousands of packets.

Scenario	Enterprise	WAN	Data Center
CPU cost per packet	0.725μs	0.434μs	0.585μs

Table 5.2: Decompression latency.

a second. In other words, a round size as small as one second is sufficient to derive most of the compression benefits in NetSight.

Decompression Speed

Table 5.2 shows NS+GZ decompression latency (decompression time per postcard) for one trace from each of the environments. In every case, NS+GZ decompression is

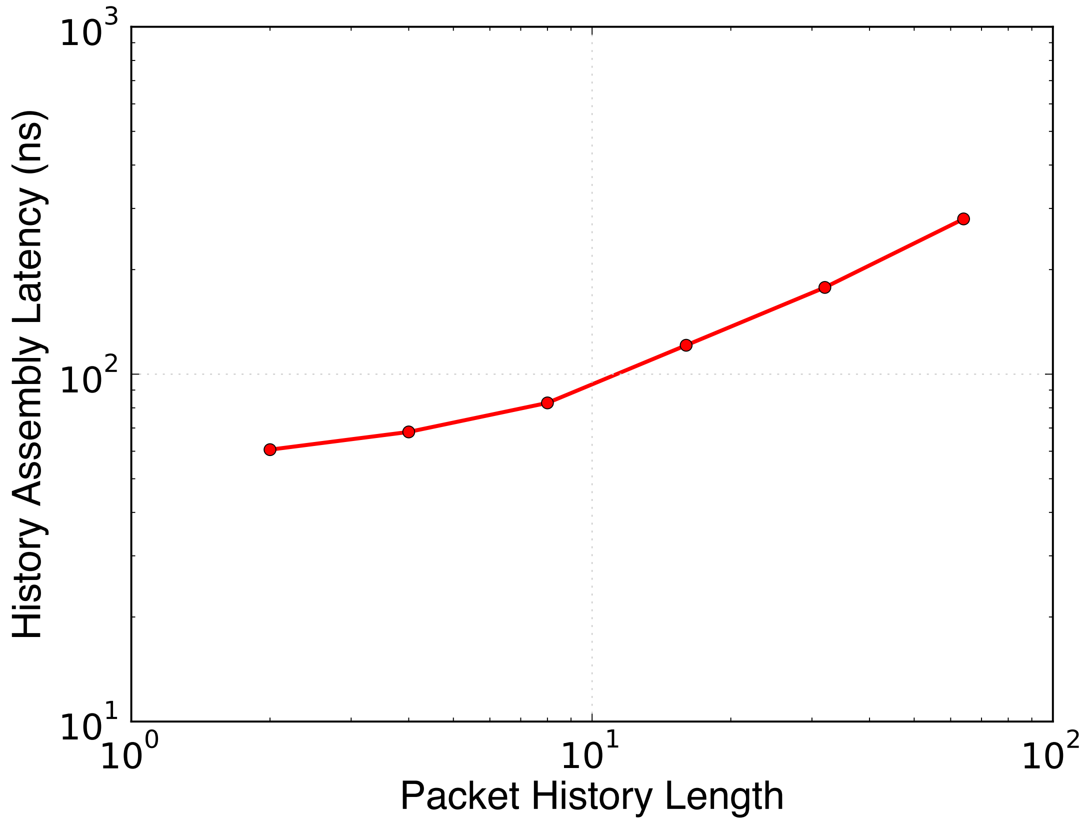


Figure 5.7: History assembly latency microbenchmark for packet histories of increasing length.

significantly faster than compression. These numbers underrepresent the achievable per-postcard latencies because the implementation loads the entire set of first packets and timestamps into memory before iterating through the list of `DiffRecord`s. As with compression, a shorter round duration would improve cache locality and use less memory.

5.3.2 Packet History Assembly

At the end of the shuffle phase, each NetSight server assembles packet histories by topologically sorting the postcards it received that may have arrived out-of-order. I measure the speed of NetSight's history assembly module written in C++. Topological sorting is fast—it runs in $O(p)$ time, where p is the number of postcards in the packet history, and typically, p will be small. As shown in Figure 5.7 for typical packet history lengths of 2 to 8, history assembly takes less than 100 nanoseconds. In other words, a single NetSight server can assemble more than 10 million packet histories per second per core.

5.3.3 Triggering and Query Processing

NetSight needs to match PHFs against assembled packet histories, either in a live stream of packet histories or in an archive. In this section, I measure the speed of packet history matching using both microbenchmarks and a macrobenchmark suite, looking for scenarios where matching might be slow. The PHF match latency depends on a number of factors:

- The length of the packet history
- The size and type of the PHF
- Whether the packet history matches the PHF

Microbenchmarks

Figure 5.8 shows the performance of NetSight's PHF matching engine (described in Section 3.2.3) for sample PHFs of varying size matching against packet histories of varying length. The sample PHFs are of the type "`.*X`," "`.*X.*`," "`X.*X`," and "`X.*X.*X`," where each `X` is a postcard filter and contains filters on packet headers (BPF), switch ID, and input ports. I match a large number of packet histories awith each PHF and calculate the average latency per match. In order to avoid any data caching effects, the evaluation script reads the packet histories from a 6 GB file, and it ignores the I/O latency incurred while reading packet histories from the disk.

The dashed lines show the latency when the packet history matches the PHF ("match"), and the solid lines show the latency when the packet history does not

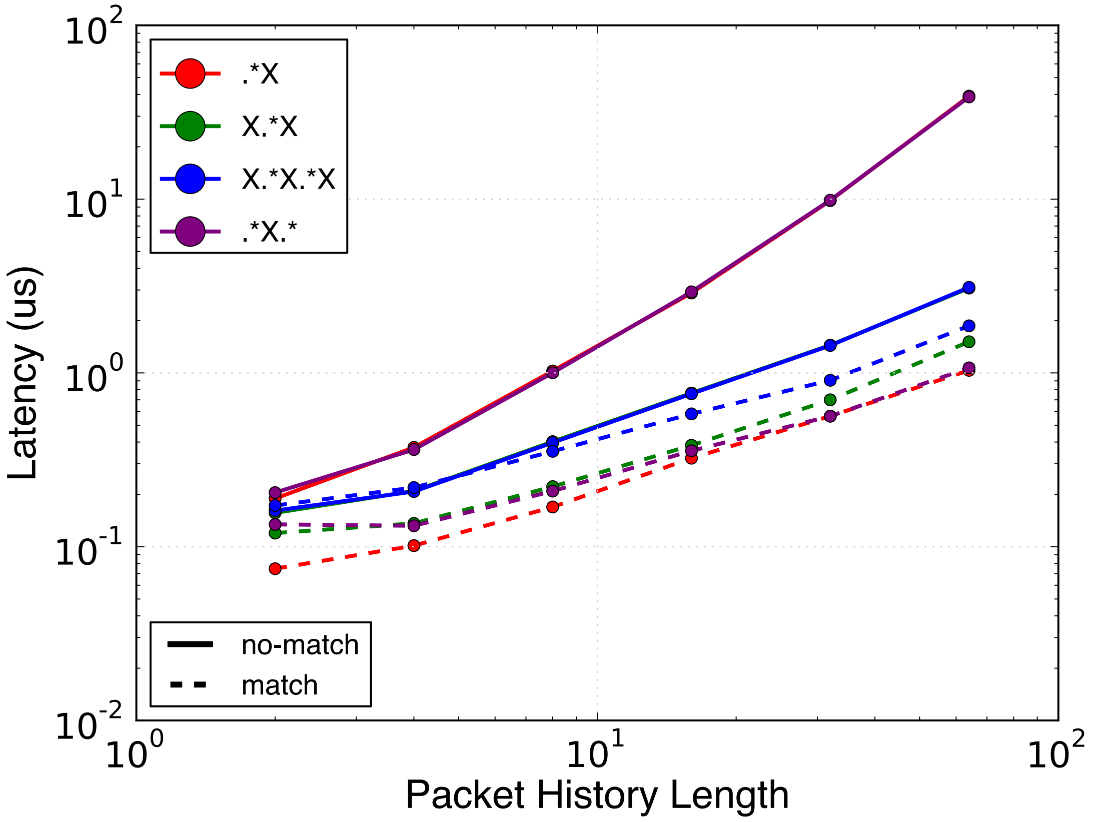


Figure 5.8: PHF matching latency microbenchmark for various sample PHFs and packet histories of increasing length.

match the PHF ("no-match"). We see that the "match" latencies are typically smaller than are the corresponding "non-match" latencies because the program can return as soon as a match is detected. We also see that the match latency increases with the number of PFs in the PHF as well as the length of the packet history. Importantly, the region of interest is the bottom left corner—packet histories of lengths 2 to 8. Here, the match latency is low—a few hundred nanoseconds.

Macrobenchmarks

The UNIV trace was captured at the core router of the Clemson backbone network connecting two large datacenters and 150 buildings to the Internet. I reconstruct

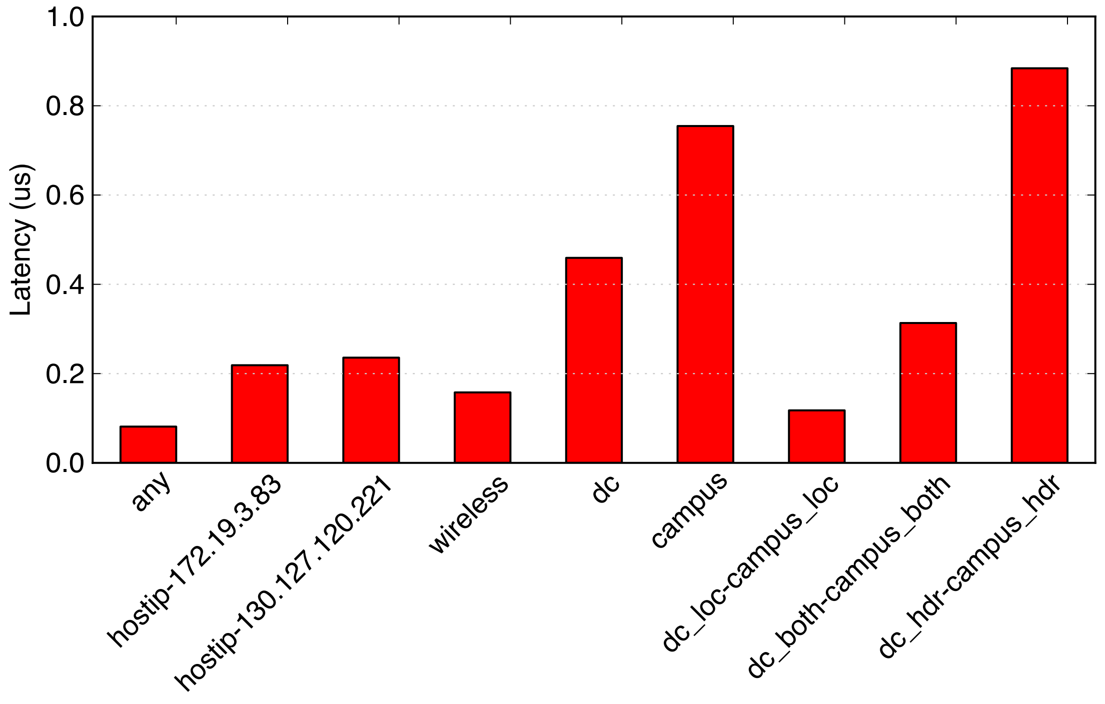


Figure 5.9: Representative results from the macrobenchmark suite of queries run on the Clemson trace. The most expensive queries were those with complex BPF expressions.

packet histories for packets in this trace using topology and subnet information. Then, I run a suite of 28 benchmark PHF queries that include filters on specific hosts, locations (datacenter, campus, and Internet), paths between the locations, and packet headers. Figure 5.9 shows the average PHF match time (on a single Intel(R) Core(TM) i7 CPU core) for a representative set of queries on hosts, subnets (campus) and paths (dc_hdr–campus_hdr). Most matches execute quickly (less than 300 ns/match); the most expensive ones (900 ns/match) are complex BPF queries that contain a predictate on 24 subnets. PHF matching is data parallel and scales linearly with the number of cores.

The above results show that even an unoptimized single-threaded implementation of PHF matching can achieve high throughput. In addition, PHF matching is embarrassingly parallel—each packet history can be matched against a PHF independent of all other packet histories. A future optimized implementation can also perform the matching directly on compressed archives of packet histories rather than on each individual packet history.

5.3.4 Effeciency of Two-Stage Filtering

In this section, I evaluate the efficiency of TSF using both microbenchmark and macrobenchmark tests.

Efficiency Intuition

Consider a hypothetical application interested in packet histories of all packets that traverse a switch A. It installs a PHF in the form of "`.*{{--dpid A}}.*`," which has a single imperative predicate "`--dpid A`." In this case, the two-stage filtering method is optimally efficient; it ensures that postcards are requested by the NetSight server *if and only if* they are a part of a matching packet history.

However, for filters such as "`{{--dpid A}}.*{{--dpid B}}`," TSF is not optimal in that it results in false positives: All paths starting at B and ending at A will also result in a `get` request by the NetSight server, but the postcards will never match the PHF. This example shows the necessity of the final PHF match; all imperative predicates may match, but the order of the postcards may not, as with this reversed path.[6]

Microbenchmarks

The efficiency of TSF depends heavily on the scenario (topology and traffic matrix) as well as on the installed PHFs. To better understand these effects, I start with a simple 3-switch, 4-host tree topology as shown in Figure 5.10 with an all-to-all traffic pattern. Figure 5.11 shows the results, which compare the ratio of postcards collected at the NetSight server with TSF to those collected without TSF (none) and the theoretical lower bound (optimal). The figure also shows the number of `matched` notification messages generated by TSF. The query suite covers invariant checks (*loop,*

[6]These "useless" postcards can be further reduced by tagging the input/output ports as edge or core in the PHF to match on direction, not just the switch datapath ID.

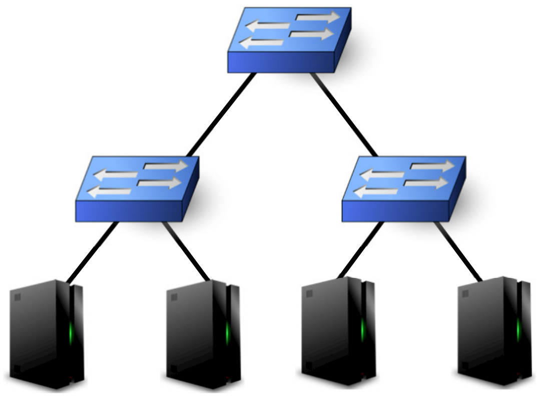


Figure 5.10: A 3-switch, 4-host tree topology used for the TSF microbenchmarks.

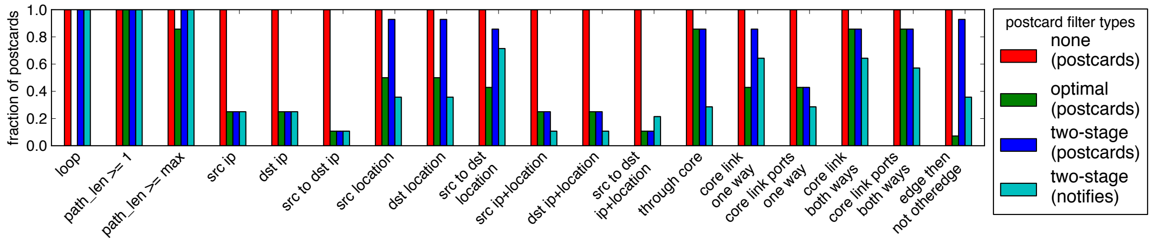


Figure 5.11: Postcard traffic reduction from TSF on a 3-switch, 4-host tree topology with all-to-all traffic.

path_len), tcpdump-style header filters (*src ip*), path queries (*src to dst location*), and their combinations; "src" and "dst" are hosts on opposite edge switches.

When TSF provides no benefit. Starting at the left side of Figure 5.11, TSF provides no benefit for PHFs that need to see all traffic, or when the PHF has no imperative predicates (e.g., *path_len* queries). Some of these queries can be improved with context-specific optimizations, e.g., loops can be checked locally at a secondary, and path lengths can be checked just by looking at the number of `matched` notification messages.

When TSF provides benefit. For all of the other queries, where the PHFs match a strict subset of the traffic and contain imperative predicates, TSF provides a significant benefit; it is optimal in many cases. In larger networks, PHF triggers for

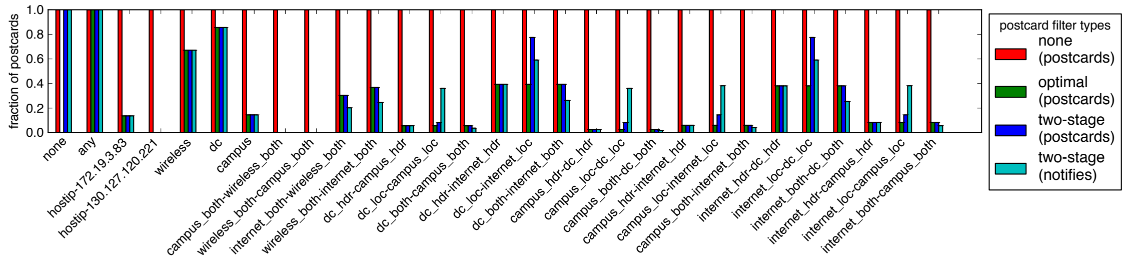


Figure 5.12: Postcard traffic reduction from TSF for sample queries on the Clemson campus backbone.

individual hosts, links, or switches will yield a much lower matched fraction, pruning an even larger fraction of "useless" postcards.

Evaluation with Real Traffic Traces

To get a more realistic sense of postcard reductions in practice, I next move to the UNIV trace. Figure 5.12 shows the results from running the suite of 28 benchmark PHF queries (same as in Section 5.3.3 Macrobenchmark) on the UNIV trace. The PHFs include specific hosts, areas (datacenter, campus, and Internet), and paths between areas, filtering on headers, paths, and both. The results show improvements for all query types. In this more realistic scenario, the benefits range from a 15% reduction to more than 95% reduction in the number of postcards assembled, showing the value of TSF at reducing postcard traffic. If we used more specific filters for subnets or individual switches rather than areas, these reductions would be even more significant.

5.4 Provisioning Scenario

In Section 1.1 of this dissertation, I suggested a set of questions, each of which maps to a PHF in NetSight. With performance numbers for each piece of NetSight, we can estimate the traffic rate it can handle as it answers those questions.

Adding up the end-to-end processing cost in NetSight—compressing, decompressing, assembling, and filtering with "live" PHFs—yields a per-core throughput of

240,000 postcards/second. With five hops on the typical path and 1000-byte average packet size (as in the Stanford University backbone network trace), a single 16-core server, available for under $2000, can handle 6.1 Gb/s of network traffic. This is approximately the average rate of the transit traffic in our university network. To handle the peak, a few additional servers would suffice, and as the network grows, the administrator can add servers knowing that NetSight will scale to handle the added demand.

If this network were to get upgraded to NetSight-SwitchAssist, one of the expensive compression steps would go away and yield a higher throughput of 7.3 Gb/s of network traffic per server. Adding NetSight-HostAssist would yield a throughput of 55 Gb/s per server because mini-postcards require no compression or decompression. The processing costs are heavily dominated by compression and decompression, and reducing these costs seem like a worthwhile future direction to improve software-based NetSight systems.

In summary, NetSight is able to handle the load for an entire campus backbone with $\sim 20,000$ *users, with a small number of servers.*

Chapter 6

Limitations and Opportunities

NetSight is not a panacea for network troubleshooting. There are a number of issues that it can not diagnose. In this chapter, I discuss the limitations of both the troubleshooting abstraction (packet histories and Packet History Filter) and the troubleshooting platform (NetSight) described in this dissertation. I then present a wishlist of features in future versions of SDN standards (e.g., OpenFlow) to enable better troubleshooting. Finally, I discuss the exciting opportunities presented by this line of research and directions for future work.

6.1 Limitations of the Abstraction

The troubleshooting abstraction presented in this dissertation is comprised of packet histories and the Packet History Filter language. The limitations of the troubleshooting abstraction are:

- It can not preemptively catch bugs. The packet history abstraction requires bugs to manifest themselves as errant behavior in the dataplane.
- Packet histories, by themselves, do not reveal the root cause of the problems. Just like a software debugger (e.g., `gdb`), they provide the context around the bugs and the sequence of events that led to the errant behavior. Packet histories can not look into payloads, measure rates, or (on their own) correlate to control

plane or application-level data. However, it is possible to build troubleshooting applications on top to automate some of these tasks.

- The Packet History Filter language, because it is based on regular expressions, can only describe regular languages. It can not describe more expressive languages, such as context-free languages (a.k.a. Type-2 languages).

  In automata theory, an example of a context-free language that is not regular is that of strings with well-formed parentheses. A string has well-formed parentheses if it has an equal number of opening and closing parentheses and if they open/close in the right order. An analogous problem in the context of network troubleshooting could be finding the set of all packet histories that have "well-formed zone traversal." Here, the network is divided into zones, and packets have well-formed zone traversal if and only if:

  1. Packets that enter a zone also leave it, and
  2. Packets enter and exit zones in the right order.

  Because the PHF language is based on regular expressions, the above question can not be expressed as a PHF. This was a conscious choice made at the design stage to strike a balance between flexibility (expressibility) and ease of use (Section 2.2.4).

6.2 Limitations of the Platform

In deploying NetSight, I found that the hardware switch in our lab was processing NetSight's particular sequences of forwarding actions in software. Talking with firmware and ASIC implementers [58, 59, 61, 62], I learned practical limitations that prevent NetSight from reaching its potential today but that appear to be addressable.

The first problem is a **lack of forwarding flexibility** that directly leads to issues of scalability. Our vendor contacts were unaware of any network firmware that uses the "postcard action sequence" [send, modify, send-to-other-port]. Without customer demand, the switch vendor understandably turned to software. Fortunately, Broadcom and Intel (Fulcrum) switch silicon can support this sequence; the switch can even send a modified and unmodified packet to the same port to enable in-band postcard generation [59, 62].

Another problem is a lack of **flexible modification and encapsulation**. NetSight must write bits into a header field to store postcard information, and the current options are constraining. Some commercial switches implement port mirroring to a remote host (e.g., [17]) which does [send, encap, send] but can only send to 32 distinct destinations. Lacking a good encap option in OpenFlow 1.0 (VLAN is only 15 bits), the NetSight prototype overwrites the destination MAC address field with postcard data; but, this choice obscures modifications to this field, and possibly bugs [47]. Support for layer-2 encapsulation, such as MAC-in-MAC, would remove this limitation.

A third problem is an **inability to truncate packets**. OpenFlow does not expose this feature, though some switch hardware is capable.

6.3 SDN Feature Wishlist

The NetSight design process has shown that SDN protocols in general, and future versions of OpenFlow in particular, could greatly ease the design of network troubleshooting platforms if they support a few specific features. As a concrete first step, I make three suggestions to make OpenFlow networks more "debuggable."

Atomic flow table updates. As discussed in Section 3.4.1, tagging postcards with just the matching flow table entry leads to flow table state ambiguity. To produce a fully deterministic packet history, we need the ability to perform concurrent updates—either across multiple entries within a single flow table or over a flow table entry and a register. Modifying SDN protocols to support atomic update primitives makes returning a fully deterministic packet history possible.

Flexible encapsulation. The current version of NetSight uses the source MAC address field to tag postcards (Section 3.2). This choice interferes with the troubleshooting process when a bug modifies source MAC addresses [47]. Modifying SDN protocols to support flexible layer-2 encapsulation, e.g., MAC-in-MAC, would remove this limitation.

Flexible actions. To avoid a need for input-port-specific flow entries, the current prototype of NetSight avoids tagging postcards with the input port and instead infers a packet's input port from the output port of the previous hop and knowledge of the topology. Protocol-level support to more flexibly tag packets with the input port or register contents, as well as to support packet truncation, would simplify the construction of NetSight.

Together, these examples reiterate a key message of this dissertation: SDN protocols (and implementations) should be designed with troubleshooting in mind. Some features in this direction are already being considered for future OpenFlow versions (such as layer-2 encapsulation), but not specifically for their debugging benefits. My hope is that, having demonstrated their utility for debugging, other suggestions will be prototyped, improved, and maybe one day even considered for standardization.

While we could just use a network processor-based switch, these abilities are desirable in commodity silicon and the switches based on them. I hope the case for packet histories in this dissertation will spur the switch vendors to expose the features that are already present and will motivate ASIC designers to make chips with more flexible actions so as to make platforms such as NetSight better-performing and easier to build.

6.4 Extensions and Opportunities

NetSight already supports four network troubleshooting applications, but its extensibility is more exciting. Besides those already implemented, the NetSight API enables a whole range of other applications and extensions. For example, `ndb` could be integrated with network control programs or software debuggers, such as `gdb` and `pdb`, to automatically point to a specific line of code that led to the error. Traffic profiles from `nprof` could close the loop to the control plane, enabling real-time traffic engineering decisions and fault monitoring. Applications could use packet histories from `netshark` to replay specific events in the same network, a shadow network, or a simulator.

Ideally, we would catch *every* bug preemptively, before it manifests in the network, with tools incuding model checkers, invariant checkers, languages, and update abstractions [15, 22, 35, 37, 50]. NetSight nicely complements these tools by addressing real problems (such as hardware, firmware, and configuration errors) when they inevitably occur.

Early SDN practitioners seem more interested in using NetSight for development than for administration. Few tools available today provide a consistent view of a packet's life in spite of a dynamically changing forwarding state. As the SDN ecosystem grows, the number of potential users for such tools will expand—from network admins debugging their latest scripts to software and hardware vendors wanting immediate, automatic, and detailed bug reports from the field and even to operators tracking down the sources of misbehavior in a multi-vendor deployment. NetSight can support each of these users.

Chapter 7

Related Work

In this chapter, I discuss some of the work—both commercial and academic—that is most closely related to NetSight. This includes work in the areas of network troubleshooting and also systems and techniques that are not specifically meant for network troubleshooting but have architectural elements similar to those of NetSight.

7.1 Network Troubleshooting

7.1.1 Commercial Work

The most closely related commercial works are tools to provide packet-level visibility, including packet sampling [16, 48], port mirroring (Switched Port Analyzer (SPAN) and Roving Analysis Port (RAP)), and dedicated network-visibility boxes [4, 5, 10].

7.1.2 Academic Work

In the academic sphere, plenty of systems have been proposed to enable better network troubleshooting, many of which exploit the logically centralized visibility and the programmatic interface to switch forwarding tables provided by SDNs.

OFRewind: The most closely related academic work, OFRewind [64], records and plays back SDN control plane traffic; like NetSight, it also records flow table states via a proxy and logs packet traces. However, OFRewind and NetSight differ in their overall approach to debugging. OFRewind aids debugging via scenario re-creation, whereas NetSight helps debugging via live observation. Also, OFRewind lacks the complete visibility and automatic path inference found in NetSight.

The next class of related work checks or prevents network bugs, often by leveraging the central control plane visibility of SDNs.

Anteater: Anteater [43] statically analyzes the dataplane configuration to monitor invariant violations such as connectivity and isolation errors. Anteater uses SAT solvers for invariant checking.

Header Space Analysis: Header Space Analysis (HSA) is a framework designed to model and check networks for failure conditions in a protocol-independent way. It is a generalization of the geometric approach to packet classification pioneered by Lakshman and Stiliadis [39], in which classification rules for K packet fields are viewed as subspaces in a K dimensional space. A number of tools have been built using HSA as the base framework. The Hassel library enables static analysis of the dataplane configuration, whereas NetPlumber [36] enables real-time policy verification by incrementally checking compliance of state changes.

VeriFlow: Like NetPlumber, VeriFlow [38] is also designed to check network-wide invariants in real time by intercepting communication between the OpenFlow controller and the network. It uses a trie structure to search rules based on equivalence classes (ECs), and, upon an update, determines the affected ECs and updates the forwarding graph for that class. This, in turn, triggers a rechecking of affected policies.

Consistent Updates: Consistent Updates [50] provides primitives for consistent state changes across the network. It prevents inconsistent configuration errors by ensuring that packets either see the initial configuration state or the final, nothing in between.

Frenetic and Nettle: Frenetic [22] and Nettle [63] are high-level programming languages for OpenFlow networks inspired by previous work on functional reactive programming. They have been designed to reducing programming errors by abstracting aspects of SDNs to improve code correctness and composability.

NICE: NICE [15] combines model checking and symbolic execution to verify controller codes. It applies model checking to explore the state space of the entire system to identify representative packets that exercise code paths on the controller.

The above techniques *model* network behavior, but bugs can creep in and break this idealized model. NetSight, on the other hand, takes a more direct approach—it finds bugs that manifest themselves as errantly forwarded packets and provides direct evidence to help identify their root cause.

7.2 Architectural Elements

Finally, I discuss some other academic works that employ architectural elements similar to those of NetSight but use them for a different purpose.

IP Traceback: IP Traceback builds paths to locate denial-of-service attacks on the Internet [19, 53, 57].

Trajectory Sampling: Trajectory sampling monitors traffic and its distribution [21] or improves sampling efficiency and fairness [11, 20, 54]. NetSight has a different goal (network diagnosis) and uses different methods. Rather than inferring sample path properties to monitor traffic engineering performance, NetSight's techniques build a consistent picture of the full forwarding state to analyze errant packets. Still, NetSight can benefit from some of the optimizations used in trajectory sampling for reducing postcard traffic overhead.

Packet Obituaries: Packet Obituaries [12] proposes an accountability framework to provide information about the fate of packets. Its notion of a "report" is similar to a packet history but provides only the inter-AS path information. It also lacks a systematic framework to pose questions about these reports in a scalable way.

Van Jacobson TCP/IP Header Compression: Van Jacobson TCP/IP Header Compression (VJ compression) [33] is a data compression protocol designed by Van Jacobson to improve TCP/IP performance over slow serial links. VJ compression reduces the normal 40-byte TCP/IP packet headers down to 3-4 bytes for the average case. It does this by saving the state of TCP connections at both ends of a link and only sending the differences in the header fields that change. NetSight's postcard compression techniques, although used for a completely different purpose, are based on those of VJ compression.

In summary, the packet history-based troubleshooting framework and NetSight borrow techniques from a number of previous systems. They complement the growing number of troubleshooting systems that exploit the centralized visibility and programmatic control provided by SDNs to build more predictable and robust networks.

Chapter 8

Conclusion

It is without doubt that debuggers have been an important part of the software development toolchain. This dissertation introduces concepts of systematic troubleshooting to the field of networking analogous to its powerful counterpart in the world of software programming.

Networks offer notoriously poor visibility into their behavior and performance. In particular, networks are inherently distributed. They move data at aggregate rates greater than any single machine can process. Because of concurrency, network experiments are seldom repeatable. There is no way to pause or "single-step" a network the way we like with other kinds of systems. Finally, control and data traffic may compete for the same resources and affect each other in subtle yet pernicious and hard-to-reproduce ways.

Despite these challenges, NetSight significantly improves network visibility. In particular, it demonstrates the surprising practicality of collecting and storing complete packet histories for all traffic on moderate-sized networks. Using an ad-hoc, lightweight compression technique, NetSight archives packet headers and the switch configurations with which they were processed at an average size as low as 10 bytes per packet. This makes it possible to aggregate information in dedicated servers that reassemble full packet histories.

Furthermore, NetSight demonstrates the perhaps less surprising result that, given access to a network's complete packet histories, one can implement a number of compelling new applications. NetSight provides a Packet History Filter (PHF) API for matching packet histories. Given this API, I implemented four applications—network debugger, invariant monitor, packet logger, and hierarchical network profiler—none of which required more than 200 lines of code. These tools manifested their utility almost immediately when a single, incompletely assembled packet history revealed a switch configuration error within minutes of NetSight's first test deployment.

This is just the beginning of an exciting opportunity to build robust, bug-free networks. While packet history captures the history of a packet in the dataplane, the programmer could benefit from a more complete trace that is tightly integrated with controller execution. It remains to be seen how application controllers written at higher levels of abstraction (e.g., Onix, Frenetic) can benefit from low-level debugging information from the network.

As shown in Section 3.4, currently used SDN control protocols have limitations for debugging; the semantics for modifying forwarding state and packets could be more flexible. Other communities have found support for easier debugging so valuable that they have standardized hardware support for it, such as JTAG in-circuit debugging in embedded systems and programmable debug registers in x86 processors. As a community, we should explore whether to augment hardware or to use software workarounds with caveats such as those in Section 6.2. I believe that NetSight is only one of many SDN-specific tools that, provided sufficient hardware support, will make networks easier to debug, benefiting networking researchers, vendors, and operators.

Appendix A

Packet History Filter Grammar

```
           <RE>  ::=  <union> | <simple-RE>

        <union>  ::=  <RE> "|" <simple-RE>

    <simple-RE>  ::=  <concatenation> | <basic-RE>

<concatenation>  ::=  <simple-RE> <basic-RE>

     <basic-RE>  ::=  <star> | <plus> | <num> |
                      <numrange> | <elementary-RE>

         <star>  ::=  <elementary-RE> "*"

         <plus>  ::=  <elementary-RE> "+"

          <num>  ::=  <elementary-RE> "{" <number> "}"
```

Continued on next page...

```
      <numrange>  ::=  <elementary-RE> "{" <number> "," <number> "}"

 <elementary-RE>  ::=  <group> | <any> | <bos> |
                       <eos> | <char> | <set>

         <group>  ::=  "(" <RE> ")" | <backref>

       <backref>  ::=  "(" "\" <num> ")"

           <any>  ::=  "."

           <bos>  ::=  "^"

           <eos>  ::=  "$"

          <char>  ::=  "{{" <postcard-filter> "}}"

           <set>  ::=  <positive-set> | <negative-set>

  <positive-set>  ::=  "[" <set-items> "]"

  <negative-set>  ::=  "[^" <set-items> "]"

     <set-items>  ::=  <char> | <char> <set-items>
```

Table A.1: Packet History Filter grammar.

Appendix B

NetSight API Messages

B.1 Control Socket Messages

NetSight API supports control socket messages of the following types:

```
enum MessageType {
    ECHO_REQUEST = 1,
    ECHO_REPLY,
    ADD_FILTER_REQUEST,
    ADD_FILTER_REPLY,
    DELETE_FILTER_REQUEST,
    DELETE_FILTER_REPLY,
    GET_FILTERS_REQUEST,
    GET_FILTERS_REPLY
};
```

A control socket message contains two parts—message type and data—as shown below:

```
struct Message {
MessageType type;
void *data;
};
```

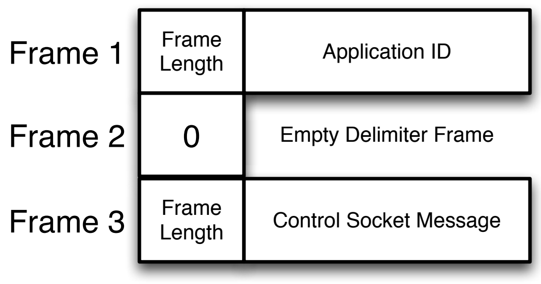


Figure B.1: The control socket message framing includes the application ID and the control socket message.

The wire format of the control socket messages sent over the ØMQ REQ-REP socket consists of three frames, as shown in Figure B.1. The first frame contains the ID of the application sending the message. The second frame is an empty delimiter frame, as required by ØMQ. The third frame contains the actual message.

The following is the structure of the individual control socket messages:

```
struct EchoRequestMessage: Message {
            type = ECHO_REQUEST;
            data = timestamp;
};

struct EchoReplyMessage: Message {
            type = ECHO_REPLY;
            data = timestamp;
};

struct AddFilterRequestMessage: Message {
            type = ADD_FILTER_REQUEST;
            data = packet_history_filter;
};

struct AddFilterReplyMessage: Message {
```

```
            type = ADD_FILTER_REPLY;
            data = {"filter": packet_history_filter, "id": PHF_ID,
                    "result": result};
};

struct DeleteFilterRequestMessage: Message {
            type = DELETE_FILTER_REQUEST;
            data = PHF_ID;
};

struct DeleteFilterReplyMessage: Message {
            type = DELETE_FILTER_REPLY;
            data = {"filter": packet_history_filter, "id": PHF_ID,
                    "result": result};
};

struct GetFiltersRequestMessage: Message {
            type = GET_FILTERS_REQUEST;
            data = NULL;
};

struct GetFiltersReplyMessage: Message {
            type = GET_FILTERS_REPLY;
            data = vector<{"filter": packet_history_filter,
                           "id": PHF_ID}>;
};
```

B.2 History Socket Messages

NetSight publishes packet histories that match installed PHFs over the history socket. The structure of the packet history message published over the history socket looks

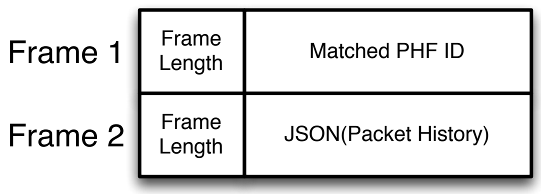


Figure B.2: The history socket message framing includes the matched PHF ID and the corresponding packet history.

as follows:

```
struct PostcardList {
        int length;
        vector<Postcard> postcard_list;
};

struct Postard {
        Packet *pkt;
        int dpid;
        int inport;
        int outport;
        int version;
};
```

As shown in Figure B.2, the wire format of the message contains two frames:

1. The ID of the PHF that the packet history matched. ØMQ uses this frame to route the message to the corresponding application that installed the PHF.
2. The actual packet history (serialized using JSON-encoding).

Bibliography

[1] Arista networks. www.aristanetworks.com.

[2] The caida ucsd anonymized internet traces 2012 – nov 15 2012. http://www.caida.org/data/passive/passive_2012_dataset.xml.

[3] Cisco unified computing system. http://www.cisco.com/en/US/products/ps10265/index.html.

[4] Endace inc. http://www.endace.com/.

[5] Gigamon. http://www.gigamon.com/.

[6] Hp allianceone services zl module series. http://h17007.www1.hp.com/us/en/whatsnew/docs/advancemoduleds.pdf.

[7] Intel seacliff trail 640 gbps switch reference platform for intel alta fm6000. http://www.adiengineering.com/products/seacliff-trail.

[8] Introducing json (javascript object notation). http://www.json.org/.

[9] A jit for packet filters. http://lwn.net/Articles/437981/.

[10] Net optics: Architecting visibility into your network. http://www.netoptics.com/.

[11] A. Arefin, A. Khurshid, M. Caesar, and K. Nahrstedt. Scaling data-plane logging in large scale networks. In *MILCOM*. IEEE, 2011.

[12] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *ACM HotNets-III*, 2004.

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

[14] Cezar Campeanu, Huiwen Li, Kai Salomaa, and Sheng Yu. Regex and extended regex.

[15] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A nice way to test openflow applications. In *NSDI*. USENIX, 2012.

[16] B. Claise. Rfc 5101: Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. `http://tools.ietf.org/html/rfc5101`.

[17] Hewlett Packard Company, June 2012. HP Networking Management and Configuration Guide K.15.09.

[18] Russ Cox. Regular expression matching: the virtual machine approach. `http://swtch.com/~rsc/regexp/regexp2.html`.

[19] D. Dean, M. Franklin, and A. Stubblefield. An algebraic approach to ip traceback. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):119–137, 2002.

[20] N. Duffield. Fair sampling across network flow measurements. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 367–378. ACM, 2012.

[21] N. G. Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. In *IEEE/ACM Transactions on Networking*, 2001.

[22] N. Foster, R. Harrison, M.J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[23] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, and N. McKeown. Nox: Towards an operating system for networks. In *ACM SIGCOMM CCR: Editorial note*, July 2008.

[24] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *HotSDN*. ACM, 2012.

[25] N. Handigol and others. Aster* x: Load-balancing web traffic over wide-area networks. *GENI Engineering Conf. 9*, 2010.

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

[27] Pieter Hintjens. ZeroMQ: The Guide, 2010.

[28] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.

[29] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[30] Precision Time Protocol. `http://ieee1588.nist.gov/`.

[31] Cisco Inc. Configuring local span, rspan, and erspan. Official Cisco ERSPAN documentation.

[32] V. Jacobson. Rfc 1144: Compressing tcp. `http://tools.ietf.org/html/rfc1144`.

[33] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144, February 1990.

[34] A. Josey, DW Cragun, N. Stoughton, M. Brown, C. Hughes, et al. The open group base specifications issue 6 ieee std 1003.1. *The IEEE and The Open Group*, 20, 2004.

[35] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*. USENIX, 2012.

[36] Peyman Kazemian, M Chang, H Zheng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[37] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *HotSDN*, August 2012.

[38] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[39] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '98, pages 203–214, New York, NY, USA, 1998. ACM.

[40] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *HotNets*. ACM, 2010.

[41] LBNL/ICSI Enterprise Tracing Project. `http://www.icir.org/enterprise-tracing/download.html`.

[42] IEEE Std 802.1AB-2009, IEEE Standard for Local and Metropolitan Area Networks—Station and Media Access Control Connectivity Discovery.

[43] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *SIGCOMM*. ACM, 2011.

[44] Nec ip8800 openflow-enabled switch. `http://support.necam.com/pflow/legacy/ip8800/`.

[45] The openflow switch. `http://www.openflow.org`.

[46] Open vSwitch: An Open Virtual Switch. `http://openvswitch.org/`.

[47] [ovs-discuss] setting mod-dl-dst in action field of openflow corrupts src mac address. `http://openvswitch.org/pipermail/discuss/2012-March/006625.html`.

[48] Peter Phaal. sflow version 5. `http://sflow.org/sflow_version_5.txt`.

[49] The pox controller. `http://github.com/noxrepo/pox`.

[50] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, pages 323–334. ACM, 2012.

[51] Ripl-pox (ripcord-lite for pox): A simple network controller for openflow-based data centers. `https://github.com/brandonheller/riplpox`.

[52] M. Rosenblum and J.K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[53] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical network support for ip traceback. In *SIGCOMM*. ACM, 2000.

[54] V. Sekar, M.K. Reiter, W. Willinger, H. Zhang, R.R. Kompella, and D.G. Andersen. Csamp: a system for network-wide flow monitoring. In *NSDI*. USENIX, 2008.

[55] Scott Shenker. The Future of Networking, and the Past of Protocols. In *Open Networking Summit*, October 2011.

[56] R. Sherwood et al. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[57] Alex C. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer. Hash-based ip traceback. In *SIGCOMM*. ACM, 2001.

[58] Dan Talayco. personal communication, 2012. Technical Led, Indigo open-source firmware, BigSwitch Networks.

[59] Joe Tardo. personal communication, 2012. Associate Technical Director at Broadcom.

[60] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Workshop on Research on enterprise networking (INM/WREN)*. USENIX Association, 2010.

[61] Jean Tourrilhes. personal communication, 2012. Implementer of HP ProCurve OpenFlow Implementation.

[62] Sean Varley. personal communication, 2012. Strategic Marketing Manager at Intel Corporation.

[63] A. Voellmy and P. Hudak. Nettle: Functional reactive programming of openflow networks. In *Practical Aspects of Declarative Languages*, 2011.

[64] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. Ofrewind: enabling record and replay troubleshooting for networks. In *ATC*. USENIX, 2011.