Scalable Management of Enterprise and Data-Center Networks

Minlan Yu

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science
Adviser: Jennifer Rexford

September 2011

Abstract

The networks in campuses, companies, and data centers are growing larger and becoming more complicated to manage. Today, network operators devote tremendous time and effort to three key management tasks — routing, access control, and troubleshooting. Rather than trying to make today's brittle networks easier to manage, we focus on new network designs that are inherently easier to manage and scale to many hosts, switches, and applications.

We design and develop a new management system that scales the routing, access control, and performance diagnosis in enterprise and data center networks. The key challenges are the large number of hosts, switches, and applications in these networks and the need for flexible policies, while faced with strict memory and power constraints in the switches. To address these challenges, we propose three key ideas: (1) designing new data structures and algorithms that make effective use of limited memory in switches; (2) redirecting traffic when simple switches do not have enough memory to handle packets; (3) rethinking the division of labor among switches, hosts, and a centralized management system to make the network both flexible and scalable.

Based on the key ideas, we propose a new management system that addresses the scalability challenges of routing, supporting flexible policies, and performance diagnosis with three key components:

(i) BUFFALO: A scalable packet forwarding architecture that reduces the switch memory usage for storing the forwarding table using Bloom filters — a compact way of representing a set of elements. To gracefully handle the false positives caused by Bloom filters, BUFFALO sends packets through a slightly longer path.

(ii) DIFANE: A scalable way to enforce flexible management policies from the centralized management system to the switches. DIFANE rethinks the division of labor between the centralized management system and the switches, by pulling some rule processing functions back to the switches, to achieve better scalability.

(iii) SNAP: A scalable network performance diagnosis architecture that exposes the interactions between the network and applications in data centers. SNAP passively logs traffic statistics in the end-host network stack and pinpoints problems that occur at the network device, network stack and the application software.

Our systems can be easily implemented with small modifications in today's switches and end hosts, as demonstrated by our prototypes built using the OpenFlow switches and Microsoft Windows servers, and our evaluation using configuration data from AT&T networks and a deployment in a production data center.

Acknowledgments

I would like to thank my advisor Jennifer Rexford for bringing me into the field of data networking, for her guidance and constant encouragement. She has taught me how to find research problems from practice, conduct solid research with real world impact, and present research results to different audiences. Being a student of hers is one of the luckiest experiences in my life.

I've been very lucky to get a chance to work with and learn from Albert Greenberg. He keeps reminding me to find real problems from engineers and to provide practical tools that can solve these real problems. I also learnt that good research projects should not only be intellectually exciting, but also be practical and useful. This will be a goal for my future research.

Mike J. Freedman has been another awesome colleague and mentor. His good taste of research problems, his diligence on systems research, and passion on building real systems served as a good model for me.

I also benefited a lot from collaborations with many people that led to this thesis. I especially thank Alex Fabrikant for contributing to the theoretical proofs in Chapter 2; Michael J. Freedman and Jia Wang for insightful discussions on the distributed system solutions and rule partitioning and caching algorithms in Chapter 3; Albert Greenberg, Dave Maltz, Lihua Yuan, Srikanth Kandula, and Changhoon Kim for the great help on the motivations of designing the SNAP tool and evaluating SNAP in the production data centers in Chapter 4. I also thank Ed Felten and Vivek Pai for being part of my committee and giving great feedback.

I would also like to thank my collaborators on various finished and ongoing projects including Matthew Caesar, Mung Chiang, Nick Feamster, Lavanya Jose, Eric Keller, Steven Ko, Li Erran Li, Lucian Popa, Sanjay Rao, Sylvia Ratnasamy, Michael Schapira, Ion Stoica, Peng Sun, Xin Sun, Marina Thottan, Yung Yi for sharing your expertise and ideas with me. It is an exciting experience to collaborate with you!

I thank previous and current cabernet group members Nate Foster, Sharon Goldberg, Rob Harrison, Wenjie Jiang, Elliot Karpilovsky, Changhoon Kim, Haakon Ringberg, Martin Suchara, Yi Wang, Rui Zhang-Shen, Yaping Zhu for their inspiring discussions and many thoughtful comments on papers and practice talks. I also thank Kai Li for giving me great advice on how to be a good researcher over the years.

I benefited a lot from my three summer internships. Many thanks to Marina Thottan at Bell Labs; Angela Chiu, Robert Doverspike, Guangzhi Li, and Dahai Xu at AT&T Labs - Research; Chuanxiong Guo, Randy Kern, Varugis Kurien, Saby Mahajan, Jitendra Padhye, Murari Sridharan, and Ming Zhang at Microsoft for giving me great industrial internship experiences and letting me appreciate the importance of solving practically challenging problems.

I would like to thank my parents Gang and Yuli for their love and support; I would like to thank Zhiwei for sharing his life with me; I would also like to thank my friends, both inside and outside Princeton University, for bringing happiness and wisdom to my life. Finally, I would like to thank Princeton University for giving me wonderful memory as a graduate student.

This dissertation work was supported by DARPA Coronet PO S000004001, NSF grants CNS 05-09447, 08-31646, and 09-04729.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many networks at the edge of the Internet—in campuses, companies, data centers, and homes—are growing larger in recent years and are becoming the important components of the Internet. To manage these networks, campuses, companies, and data centers usually need a huge army of people to perform daily management tasks such as registering new devices, assigning network addresses, configuring routing policies, setting up firewalls, troubleshooting, and maintaining efficient and reliable network connectivity. According to a Yankee group report [2], in most enterprises, network management is responsible for 80% of the IT budget and 62% of the outages.

The key question that this thesis answers is how to build a management system that both flexibly supports various management tasks and scales with the growth of these edge networks.

1.1 Edge Networks and Their Scalability Challenges

Edge networks are those networks that are located at the edge of the Internet, such as enterprise networks (*i.e.*, networks in campuses and corporations) and data center networks. These networks have grown dramatically in the number of hosts, switches, and applications. The management of these networks is getting more complex because operators have to assign network addresses for each device, configure routing policies for each host and application, set up firewalls, and troubleshoot. It is a challenging problem to scale the management of these networks because the networks are growing large with complex management tasks, but only have cheap, simple switches.

1.1.1 Edge Networks Are Large

Large enterprises can easily have tens to hundreds of thousands of heterogeneous hosts such as laptops, mobile phones, desktops and computing clusters [3]. Some hosts can move around to different locations in the enterprise. There are thousands of switches and routers connecting these hosts together. There are also hundreds of applications running on these hosts with different traffic patterns and performance requirements.

Data center networks can easily have hundreds of thousands of servers with tens of virtual machines on each server. These virtual machines can migrate to different servers. There are multiple layers of switches connecting these servers. Data center applications include many map-reduce jobs, search applications, and cloud storage applications and have stricter performance requirements than enterprise applications.

1.1.2 Edge Networks Need Fine-grained Policies

Operators of the enterprise and data center networks have many considerations for network management such as security, mobility, application performance, energy consumption, cost saving, debugging and maintenance, etc. To express these management considerations, operators need to specify fine-grained policies for different types of traffic. Here are some example types of fine-grained policies:

Routing: Different applications have different requirements for routing. For example, a banking application may require secure paths that malicious users do not have access to; a video streaming application may need high-bandwidth paths; the financial applications that exchange real-time stock information may need low-latency paths.

Access control: Operators use access control rules to block the traffic from malicious users, in order to improve the security of their networks. For example, in a medical company, operators should make sure that attackers should not have access to the server which stores patients' private information. These access control rules should be fine-grained enough to capture various types of malicious traffic without affecting the normal traffic.

Measurement: Operators need to *measure* the behaviors in the network for accounting, anomaly detection, troubleshooting, and traffic engineering. For example, cloud providers need to accurately count the traffic from different users and applications in order to charge the users for the bandwidth they use. Operators in data centers may need to identify the large flows in the network to better configure their routing for traffic engineering (A flow is a sequence of packets that share the same packet header properties such as source address/port, destination address/port, and/or protocol).

1.1.3 Simple, Cheap Switches Have Limited Memory

Unfortunately, to support ever growing networks with more complicated policies, there are only *simple and cheap* switches in today's enterprise and data center networks. This is because these switches must process each packet within hundreds of nanoseconds to support high link speeds (*e.g.*, 10 Gbps, 40 Gbps and more). To store the state needed to process these packets, the switches must use expensive, power-hungry on-chip memory. Due to cost and power constraints, the switch memory is limited in size.

There are two types of memory in switches — SRAM (Static Random-Access Memory) and TCAM (Ternary Content-Addressable Memory): The SRAM is used for direct lookup of key-value pairs. For example, the SRAM in switches is used to store forwarding tables which match the destination MAC addresses to the outgoing interfaces. In contrast, the TCAM is used for wildcard matching where the keys can contain not only 0s and 1s, but also "don't care" bits. For example, the TCAM is used to store ACLs (Access Control Lists) at switches, which permit, block, or rate-limit the flows whose packet headers match the patterns defined in the ACLs. The SRAM is much cheaper and consume less power than the TCAM.

While the switch memory is very small due to its cost and energy constraints[1], operators hope to store a lot of state in the switch. For example, just to perform the simple forwarding action, each switch may need to store 10K entries for an enterprise network with 10K hosts, because one entry is needed for each host destination. Furthermore, switches may need to provide different access control policies and quality of service (QoS) to different users and applications — requiring even more memory. Switches may also need to maintain counters for specific flows for accounting and diagnosis.

[1]The study [4] shows that a 48-port switch takes about 100-200 Watts power.

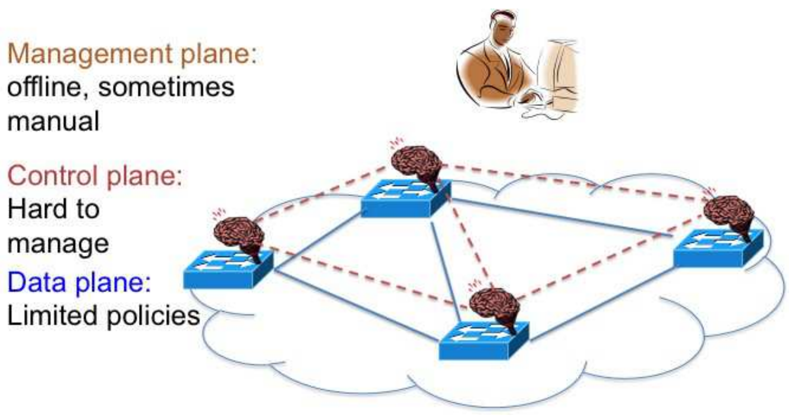


Figure 1.1: Three layers in today's management systems.

1.2 Management of Today's Edge Networks

To address the scalability challenges of enterprise and data center networks, most research focuses on making them faster and more efficient. However, the management of these edge networks is an important yet under-explored area in networking research.

There are three key scalability challenges of managing enterprise and data center networks: How to make the basic function of routing scale with many hosts and switches? How to scalably enforce fine-grained polices at switches? How to diagnose performance problems with many hosts and complicated applications?

Today, all these management tasks are performed with a three-layer architecture as shown in Figure 1.1. The first layer is the data plane—the *fast path* of these networks that actually processes packets. The data plane usually has fast on-chip memory such as SRAM and TCAM to store the network state and packet processing rules. The second layer is the control plane—the brains of the networks that run protocols to decide how to handle traffic. The third layer is the management plane—the place where operators configure policies. Switches and routers from different vendors typically have different data plane implementations, all of which only support limited policies. In addition, the control plane is tightly bundled with the data plane. Therefore, different switches and routers may have different control plane APIs and different levels of protocol support, making management even harder. As a result, operators have to configure individual network devices separately in an offline fashion, sometimes even manually. For example, to prevent attackers from accessing the web server, operators have to identify the IP or MAC addresses of all the attackers' machines, locate the routers that direct traffic to the web server, and then install ACLs at these routers.

Now, we will describe how the key management tasks — routing, access control, and diagnosis — are done in today's networks with the three layer architecture. We will also discuss the scalability limitations in this architecture.

1.2.1 Hybrids of Ethernet Islands Connected by IP Routers

In an Ethernet network, hosts are allocated with MAC addresses and the switches forward Ethernet frames based on their destination MAC addresses. The MAC address is tied to the device and is independent of the hosts' locations. Thus, MAC addresses are a good choice for the enterprise networks where laptops and mobile devices are moving frequently and data center networks where virtual machine migration is common.

To deliver traffic, the switches cooperate to construct a *spanning tree* in the control plane to provide loop-free routing paths. In the data plane, each switch maintains a forwarding table which records how to reach different destinations. When a switch receives a frame and it contains no forwarding-table entry for the frame's destination MAC address, the switch *floods* each frame over the entire spanning tree. The switch *learns* how to reach a MAC address by remembering the incoming link for frames with a particular source MAC addresses and saving the mapping.

However, these Ethernet protocols do not scale to large networks: MAC addresses cannot be aggregated, so we need one entry for each destination at each switch. Flooding requires large amounts of state and control messages that grow with the size of the network [5]. Since spanning tree networks can leave some links completely unused, they fail to take advantage of multiple paths for higher bisectional bandwidth. Two nodes near each other may have to reach each other through much longer paths if the shorter paths are not in the spanning tree. Sometimes, operators must configure the switches carefully so that the switches can construct a spanning tree with low latency for most pairs of nodes [6]. When a link or node fails, operators have to reconfigure the entire network to construct a new spanning tree with low latency for most nodes.

To address the scalability and efficiency problems of the Ethernet, most enterprise and data center networks today use small Ethernet islands interconnected by routers. The network is divided into many Ethernet islands, each with only a limited number of hosts. Within each Ethernet island, MAC addresses are used to support host mobility and run spanning tree protocols to route packets. Each Ethernet island is allocated with an IP subnet (*i.e.*, the list of IP addresses that belongs to the same IP prefix). A few IP routers deliver traffic based on the IP subnet a destination host belong to. IP routers make Ethernet more scalable at the expense of configuration complexity. Since IP addresses represent the *locations* of hosts, it is very hard to support host mobility. When hosts move between Ethernet islands, these hosts must get new IP addresses or the routing configuration at routers should change.

1.2.2 Policy Support with Access Control Lists Between Virtual LANs

In order to support flexible policies (*e.g.*, customized routing, access control, and measurement), operators group hosts sharing the same networking requirements using Virtual Local Area Networks (VLANs), instead of grouping hosts based on their locations. Each VLAN is usually allocated an IP subnet, and hosts within the VLAN communicate using Ethernet protocols. Between VLANs, there are some routers where operators can specify policies such as configuring different queues with different quality of service (QoS), installing access control lists that block malicious flows, or counting the amount of traffic from different applications.

VLANs can only support limited granularity of policy because operators can only express their management policies at a few routers located in these networks and have limited control and measurement of the traffic. VLANs are not scalable because a network can have only a limited number of VLANs and each VLAN can have only a limited number of hosts [6]. VLAN configuration is complex and error-prone because operators must ensure that each host's IP address belongs to the IP subnet of its VLAN, and that each VLAN has an efficient spanning tree that remains connected under common failure scenarios [6].

1.2.3 Ad-hoc Troubleshooting with Application and Network Logs

To troubleshoot for reachability and performance problems, today's enterprise and data center networks often collect application logs on the servers about the liveness, correctness, and performance of applications. At switches, these networks collect SNMP (Simple Network Management Protocol) logs about the number of bytes/packets transferred through each switch. In addition, operators maintain trouble ticket systems where users can report their problems. For example, operators usually pick out the trouble ticket with the most significant problems, manually investigate the problem description in the ticket, and then look into the application logs to identify the problem. If they find that the problem is related to the network, they will study the network logs, or run a few tools to identify the problems (*e.g.*, ping to test reachability, traceroute to verify routing).

Although these logs can reveal the significant problems in these networks, it is still a challenging job to infer the causal relations between these problems and identify the root causes. Worse still, for large enterprises and data centers, there are application and network teams each in charge of different components in the system. It is challenging to identify which team is responsible for performance problems or failures. It is even more challenging to automatically report problems when they happen, quickly point out the root causes, and fix or get around these problems in real time before the problems start to affect application performance.

1.2.4 Summary

In summary, today's enterprise and data center networks lack the scalability and flexibility to support various management tasks. Often, in order to keep the networks manageable, operators impose a lot of restrictions on the growth of the networks and the kind of applications that can run. Therefore, to realize the full power of the enterprise and data center networks, we should make the network easier and less expensive to manage. Future networks should be totally transparent to applications, so that they can use the network freely without worrying about the network configurations and resource constraints. However, today's networks are far from that goal.

1.3 Emerging Trends of Redesigning for Manageability

Most of these problems of managing today's brittle networks arise from the fact that these networks were not designed with management in mind. Yet it is challenging to simplify the management of today's networks. Instead, we focus on how to design future networks to make them inherently easier to manage.

There have been several recent directions arising from the research community with the goal of making management easier: flat addresses, shortest path routing, logically centralized control,

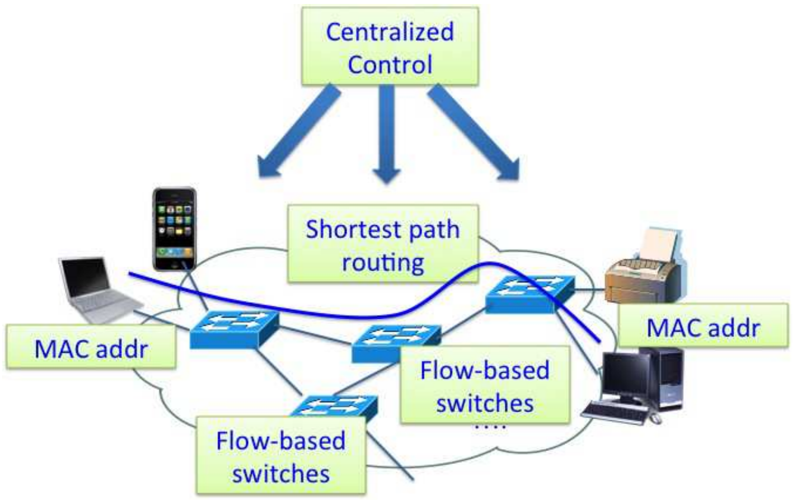


Figure 1.2: Emerging techniques for better network management.

and flow-based switches (as shown in Figure 1.2). While all these trends enable more scalable and flexible management of these edge networks, they introduce new scalability challenges in the data plane. We will discuss the benefits of these new trends and their scalability challenges below.

1.3.1 Flat Addresses to Support Zero Configuration and Host Mobility

To avoid complex configuration and better support host mobility, new protocols with flat addresses such as SEATTLE [5], RBridges/TRILL [7, 8] and SPBM (Shortest-Path Bridging MAC) [9] have been proposed. Each host has a fixed identifier that does not change when the host moves around, so the host can be plugged into the network and start using the network immediately without any extra configuration. Even when the network topology changes, administrators do not need to reassign addresses. Furthermore, with flat addresses for routing, we can assign IP addresses to express policies in a more aggregatable way. For example, for a campus network, we can assign one IP prefix to the students' machines although these machines may be located at different places on campus. As a result, we only need a single IP prefix to express the policies for the student group.

However, with all the hosts using MAC addresses that cannot be easily combined, we have too many forwarding-table entries at each switch to reach a large amount of hosts.

1.3.2 Shortest Path Routing to Improve Performance

Recent advances in Ethernet such as Rbridges/TRILL [7, 8] and SEATTLE [5] compute shortest paths based on link weights between all pairs of hosts. We can also tune the link weights to control the flow of traffic to minimize the delay and network congestion. Since shortest path routing uses all the links in the network, it provides more efficient connectivity than spanning tree protocols. We can also split traffic across multiple shortest paths, which is common in data center networks. In addition, these shortest path routing protocols can quickly propagate link and node failure events in the network, and automatically recalculate the new shortest paths.

1.3.3 Centralized Control to Make Policy Configuration Easier

Instead of configuring each switch and router separately, recent works propose to replace the control-plane software at individual switches with a logically-centralized controller in the management plane [10, 11, 12, 13]. With the centralized controller, operators only need to configure policies at a single place without worrying about reconfiguring the network when network topology or traffic changes. The centralized controller is responsible for keeping information about the address assignment, network topology, and routing. With this information, the centralized controller can automatically configure the data plane of network devices to express the management policies.

The logically centralized controller imposes new scalability and reliability challenges. Instead of making decisions on their own, the switches send the packets to the controller if they do not know how to handle those packets, which significantly increases the load on the centralized controller [13]. The centralized controller can also be a single point of failure.

1.3.4 Flow-based Switches to Enable Fine-grained Policies

Traditional switches in enterprise and data center networks do not have much support for policies such as customized routing, access control, and measurement. Flow-based switches [14, 15] are proposed to support the policies. These flow-based switches perform simple actions based on rules. The rules match on bits in the packet header (*e.g.*, source address/port, protocol, etc.). Possible actions include dropping packets from malicious users, forwarding packets to a specific outgoing interface, and counting packets that belong to the web traffic. These flow-based switches are already available from several switch vendors (*e.g.*, HP, NEC, etc.), and have been deployed in several enterprises and backbones (*e.g.*, National LambdaRail and Internet2).

With flow-based switches enabling fine-grained policies, operators can now specify new policies that can have fine-grained control of their networks. As a result, operators may be tempted to deploy many policies, which in turn introduce further scalability challenge of fitting these policies into the small switch memory.

1.4 Key Design Principles

This thesis describes the design, implementation, and evaluation of new architectures that address these key challenges: improving scalability and handling flexible policies with simple switches for enterprise and data center networks. There are three key design principles that guide the design of these new systems:

Traffic indirection that minimizes the state required at each switch. Switches need to store various network state such as forwarding tables and ACLs. To minimize the state at each switch, we propose to compress the state information or only store part of the state. As a result, one switch may not be able to make decisions on its own. Instead, we redirect the traffic to other switches that have stored the required state to make the decisions. In general, we reduce switch memory usage by sending a small fraction of packets on a slightly longer path. Since most enterprise and data center networks are local area networks with low end-to-end delay, the slightly longer path would not introduce significant performance problems.

New data structures that work with today's commodity switches. We design and build new data structures for the data plane to reduce the memory usage at the switches and to support traffic indirection. These new designs can be implemented with today's commodity switches without much hardware changes, because our data structures match the hardware features such as hashing support at switches and wildcard rule support in TCAM. By leveraging the key features in the enterprise and data center network settings (*e.g.*, all the nodes are in a single domain, managed by a centralized controller, and are well connected via shortest path routing), we can guarantee the reachability, reduce the extra delay of traffic indirection, and quickly adapt to the dynamics of these edge networks such as host mobility, traffic changes, and topology changes.

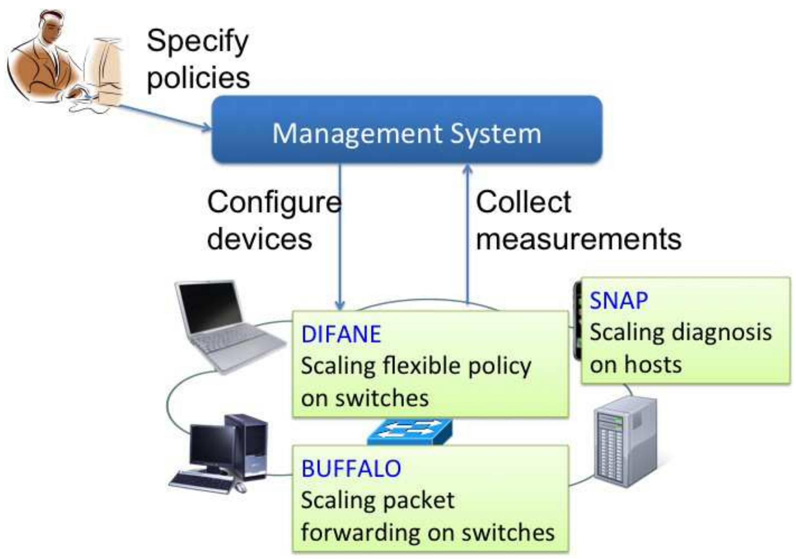

Specify policies
Management System
Configure devices
Collect measurements
DIFANE
Scaling flexible policy on switches
SNAP
Scaling diagnosis on hosts
BUFFALO
Scaling packet forwarding on switches

Figure 1.3: A Scalable Management System for Edge Networks.

Rethinking the division of labor among hosts, switches and the centralized management system. Hosts, switches, and the centralized controller are the three key components in enterprises and data centers. It is important to place the right function at the right component to achieve the scalability and flexibility at the same time. For example, we should place the function of configuring high-level policies in the centralized controller to make management easier, and process all the packets in the data plane at the switches to achieve the best performance. We also rethink the division of labor between the network and the end hosts. For example, we place the performance monitoring function at end hosts to leverage the computing and storage resources at millions of end hosts.

1.5 A Scalable Management System for Edge Networks

Following the key design principles, we propose a scalable management system which consists of three parts to scale packet forwarding, policy enforcement, and performance diagnosis as shown in Figure 1.3. In this management system, operators can configure high-level policies at a centralized controller. The controller enforces the high-level policies at the switches in the network, and collects measurement data from both the switches and the end hosts to understand network behavior. To improve the scalability of the system, we build three components: BUFFALO — a scalable packet forwarding data plane that provides basic network connectivity, DIFANE — a system that scalably enforces fine-grained policies at the switches, and SNAP — a scalable performance diagnosis system for data centers. We have built prototypes for all three parts and evaluated them with industrial configuration data and real-world deployment.

BUFFALO [16]: A scalable packet forwarding architecture using Bloom filters and random walks. As the link speed and the number of hosts and switches grow, simply building switches with larger amounts of faster memory is not appealing, since high-speed memory is both expensive and power-hungry. Given the fixed size of fast on-chip memory SRAM in switches, BUFFALO uses a compact data structure, called a *Bloom filter*, to store the set of addresses associated with each outgoing link. Bloom filters reduce SRAM usage at the expense of false positives.If a Bloom filter reports that an address is in the set of addresses, it may not be the case. In contrast, a Bloom filter does not have any false negatives. That is, if a Bloom filter indicates that an address is not in the set, it must be the case. We propose a new way to gracefully handle false positives without reducing the packet-forwarding rate, by *redirecting* those packets which experience false positives through a slightly longer path. To reduce the extra delay of packet redirection, we leverage the shortest-path routing features in enterprises and data centers. Using *random walk* theory, we proved a bound of the average delay that packets could experience. Our extensive analysis, simulation, and prototype implementation in Click (a kernel-level software router) show that BUFFALO significantly reduces memory cost, increases the scalability of the data plane, and improves packet-forwarding performance.

DIFANE [17]: A distributed flow-based architecture built on OpenFlow switches. Emerging flow-based networking and commercially available OpenFlow switches [18] use a centralized controller software. With the centralized controller, enterprise administrators can easily specify fine-grain policies to control how the underlying switches forward, drop, and measure traffic. However, existing techniques rely too heavily on the controller to install rules based on fine-grained policies, leading to poor scalability and high latency. DIFANE relegates the controller to the simpler task of translating high-level policies to low-level rules, and decentralizes rule installation and *all* the packet processing in the underlying switches. To reduce the memory usage of storing rules at switches, DIFANE builds a *distributed rule directory service* among the switches by partitioning the rules among several switches, and selectively redirecting packets through these switches. DIFANE can be easily implemented with small software modifications to commercial OpenFlow switches, as demonstrated by our prototypes built using the Click modular router. We evaluated DIFANE on a 40-node testbed and conducted simulations of realistic topologies consisting of thousands of nodes and millions of access-control rules extracted from real corporate and campus networks.

SNAP [19]: A scalable network-application diagnosis system for data center applications. Network performance problems are notoriously difficult to diagnose, and this difficulty is magnified when applications are often split into multiple tiers of application components spread across thousands of servers in a data center. We observe that performance problems often arise in the communication between the tiers, where either the application or the network (or both!) could be to blame. Therefore, we designed and built SNAP, a scalable network-application profiler that guides developers in identifying and fixing performance problems. SNAP passively collects network stack information (*e.g.*, congestion window size, send buffer size, number of packet loss, etc.) with low computation and storage overhead. Then SNAP leverages *statistical techniques* to correlate connections across shared resources (*e.g.*, host, link, switch) to pinpoint the locations of the performance problem. Our one-week deployment of SNAP in a production data center (with over 8,000 servers and over 700 application components) has already helped developers uncover 15 major performance problems in application software, the network stack on the server, and the underlying network.

The remainder of this thesis is organized as follows: Chapter 2 presents BUFFALO, the scalable forwarding layer. Chapter 3 describes DIFANE, the scalable way to enforce flexible polices at switches. Chapter 4 shows SNAP, the scalable diagnosis system for data center applications. Chapter 5 discusses how the three systems work together and concludes the thesis.

Chapter 2

BUFFALO: Scaling Packet Forwarding on Switches

This chapter focuses on the scalability challenges for providing basic packet forwarding under flat addressing. Simply building switches with larger amounts of faster memory is not appealing, since high-speed memory is both expensive and power hungry. Implementing hash tables in SRAM is not appealing either because it requires significant overprovisioning to ensure that all forwarding table entries fit. Instead, we propose the BUFFALO architecture that provides a scalable packet forwarding layer on switches.

BUFFALO uses a small SRAM to store one Bloom filter of the addresses associated with each outgoing link. We provide a practical switch design leveraging flat addresses and shortest-path routing. BUFFALO gracefully handles false positives without reducing the packet-forwarding rate, while guaranteeing that packets reach their destinations with bounded stretch with high probability. We tune the sizes of Bloom filters to minimize false positives for a given memory size. We also handle routing changes and dynamically adjust Bloom filter sizes using counting Bloom filters in slow memory. Our extensive analysis, simulation, and prototype implementation in kernel-level Click show that BUFFALO significantly reduces memory cost, increases the scalability of the data plane, and improves packet-forwarding performance.

2.1 Introduction

The Ethernet switches used in today's enterprise and data-center networks do not scale well with increasing forwarding-table size and link speed. Rather than continuing to build switches with ever larger and faster memory in the data plane, we believe future switches should leverage Bloom filters for a more scalable and cost-effective solution.

2.1.1 Memory Problems in the Data Plane

Enterprises and data centers would be much easier to design and manage if the network offered the simple abstraction of a virtual layer-two switch. End hosts could be identified directly by their hard-coded MAC addresses, and retain these addresses as they change locations (*e.g.*, due to physical mobility or virtual-machine migration). The hosts could be assigned IP addresses out of a large pool, without the artificial constraints imposed by dividing a network into many small IP subnets. However, traditional Ethernet can only support this abstraction in small network topologies, due to a heavy reliance on network-wide flooding and spanning tree. Recent advances [7, 8, 5] have made it possible to build much larger layer-2 networks, while still identifying hosts by their MAC addresses. These new architectures focus primarily on improving the *control plane*, enabling the use of shortest-path routing protocols (instead of spanning tree) and efficient protocols for disseminating end-host information (instead of flooding data packets).

As these new technologies enable the construction of ever larger "flat" networks, the scalability of the *data plane* becomes an increasingly serious problem. In today's Ethernet and in proposed solutions like TRILL [7, 8], each switch maintains a forwarding-table entry for each active MAC address. Other solutions [5] cache a smaller number of end-host MAC addresses, but still require a relatively large amount of data-plane state to reach every switch in the network. Large networks can easily have tens or hundreds of thousands of end-host MAC addresses, due to the proliferation of PDAs (in enterprises) and virtual machines (in data centers). In addition, link speeds are increasing rapidly, forcing the use of ever-faster—and, hence, more expensive and power-hungry—memory for the forwarding tables. This motivates us to explore new ways to represent the forwarding table that require less memory and (perhaps more importantly) do not require memory upgrades when the number of end hosts inevitably grows.

To store the forwarding table, one simple solution is to use a hash table in SRAM to map MAC addresses to outgoing interfaces. However, this approach requires significant overprovisioning the fast memory for three reasons: First, when switches are out of memory, the network will either drop packets in some architectures [7, 8] or crash in others [5]. Second, it is difficult and expensive to upgrade the memory for all the switches in the networks. Third, collisions in hash tables (*i.e.*, different destination addresses mapped to the same place in the hash table) require extra memory overhead to handle them, and affect the throughput of the switch.

Given these memory problems in the data plane, our goal is to make efficient use of a small, fast memory to perform packet forwarding. Such small fast memory can be the L1 or L2 cache on commodity PCs serving as software switches, or dedicated SRAM on the line cards. When the memory becomes limited with the growth of forwarding table, we ensure that all packet-forwarding decisions are still handled within the SRAM, and thus allow the switches last longer with the increase of forwarding table size.

2.1.2 The BUFFALO Forwarding Architecture

Most enterprise and data center networks are "SPAF networks", which uses Shortest Path routing on Addresses that are Flat (including conventional link-state, distance-vector, and spanning tree protocols). Leveraging the unique properties in SPAF networks, we propose BUFFALO, a Bloom Filter Forwarding Architecture for Large Organizations. BUFFALO performs the *entire* address lookup for all the packets in a small, fast memory while occasionally sending the packets through a slightly longer path.

To make all packet-forwarding decisions with a small fast memory, we use a *Bloom filter* [20], a hash-based compact data structure for storing a set of elements, to perform the flat address lookup. Similar to previous work on resource routing [20, 21][1], we construct one Bloom filter for each next hop (*i.e.*, outgoing link), and store all the addresses that are forwarded to that next hop. By checking which Bloom filter the addresses match, we perform the *entire* address lookup within the fast memory for all the packets. In contrast, previous work [22] uses Bloom filters to *assist* packet lookup and every address lookup still has to access the slow memory at least once.

[1]These studies design the algorithms of locating resources by using one Bloom filter to store a list of resources that can be accessed through each neighboring node.

To apply our Bloom filter based solution in practice, we provide techniques to resolve three issues:

Handling false positives: False positives are one key problem for Bloom filters. We propose a simple mechanism to forward packets experiencing false positives without any memory overhead. This scheme works by randomly selecting the next hop from all the matching next hops, excluding the interface where the packet arrived. We prove that in SPAF networks the packet experiencing one false positive (which is the most common case for the packet) is guaranteed to reach the destination with constant bounded stretch. We also prove that in general the packets are guaranteed to reach the destination with probability 1. BUFFALO gracefully degrades under higher memory loads by gradually increasing stretch rather than crashing or resorting to excessive flooding. In fact, in enterprise and data center networks with limited propagation delay and high-speed links, a small increase in stretch would not run the risk of introducing network congestion. Our evaluation with real enterprise and data center network topologies and traffic shows that the expected stretch of BUFFALO is only 0.05% of the length of the shortest path when each Bloom filter has a false-positive rate of 0.1%.

Optimizing memory and CPU usage: To make efficient use of limited fast memory, we *optimize* the sizes and number of hash functions of the Bloom filters to minimize the overall false-positive rate. To reduce the packet lookup time, we let the Bloom filters share the same group of hash functions and reduce the memory access times for these Bloom filters. Through extensive analysis and simulation, we show that BUFFALO reduces the memory usage by 65% compared to hash tables.

Handling routing dynamics: Since routing changes happen on a much longer time scale than packet forwarding, we separate the handling of routing changes from the packet forwarding and use counting Bloom filters in the large, slow memory to assist the update of the Bloom filters. To reduce the false-positive rate under routing changes, we *dynamically* adjust the sizes and number of hash functions of Bloom filters in fast memory based on the large fixed-size counting Bloom filters in slow memory.

We implement a prototype in the Click modular router [23] running in the Linux kernel. By evaluating the prototype under real enterprise and data center network topologies and traffic, we show that in addition to reducing memory size, BUFFALO forwards packets 10% faster than traditional hash table based implementation. BUFFALO also reacts quickly to routing changes with the support of counting Bloom filters.

The rest of the chapter is organized as follows: Section 2.2 describes the underlying SPAF networks we focus on in this chapter. Section 3.2 presents an overview of the BUFFALO architecture. Section 2.4 describes how to handle false positives and proves the packet reachability. In Section 2.5, we adjust the sizes of Bloom filters to make the most efficient use of limited fast memory. In Section 2.6, we show how to dynamically adjust the sizes of Bloom filters using counting Bloom filters. Section 3.6 presents our prototype implementation and the evaluation. Section 2.8 discusses several extensions of BUFFALO. Section 2.9 and 4.9 discuss related work and conclude the chapter.

2.2 Shortest Paths & Flat Addresses

This chapter focuses on *SPAF networks*, the class of networks that perform Shortest-Path routing on Addresses that are Flat. In fact, most enterprise and data center networks are SPAF networks.

Flat addresses: Flat addresses are used widely in enterprise and data center networks. For example, MAC addresses in Ethernet are flat addresses. New protocols with flat address spaces (*e.g.*, SEATTLE [5], ROFL [24], AIP [25]) have been proposed to facilitate network configuration and management, because they simplify the handling of topology changes and host mobility without requiring administrators to reassign addresses. Even IP routing can be done based on flat addresses, by converting variable-length IP prefixes into multiple non-overlapping fixed-length (*i.e.*, /24) sub-prefixes.

Shortest path routing: We also assume shortest-path routing on the network topology, based on link-state protocols, distance-vector protocols, or spanning-tree protocols.[2] Recent advances in Ethernet such as Rbridges [7, 8] and SEATTLE [5] all run link-state protocols that compute shortest paths.

Based on the above two properties, we define the SPAF network as a graph $G = (V, E)$, where V denotes the set of switches in the network, and E denotes all the links viewed in the data

[2] In today's Ethernet the control plane constructs a spanning tree and the data plane forwards packets along shortest paths within this tree.

plane. In the SPAF network we assume all the links in E are actively used, *i.e.*, the weight on link $e(A, B)$ is smaller than that on any other paths connecting A and B. This is because if a link is not actively used, it should not be seen in the data plane. Let $P(A, B)$ denote the set of all paths from A to B. Let $l(A, B)$ denote the length of the shortest path from A to B, *i.e.*, the length of $e(A, B)$, and the length of a path p is $l(p) = \sum_{e \in p} l(e)$. We have:

$$\forall A, B \in V, p \in P(A, B), l(A, B) \leq l(p).$$

In this chapter, we propose an efficient data plane that supports any-to-any reachability between flat addresses over (near) shortest paths. We do not consider data-plane support for Virtual LAN (VLANs) and access-control lists (ACLs), for three main reasons. First, the new generation of layer-two networks [7, 8, 5] do not perform any flooding of data packets, obviating the need to use VLANs simply to limit the scope of flooding. Second, in these new architectures, IP addresses are opaque identifiers that can be assigned freely, allowing them to be assigned in ways that make ACLs more concise. For example, a data center could use a single block of IP addresses for all servers providing a particular service; similarly, an enterprise could devote a small block of IP addresses to each distinct set of users (*e.g.*, faculty vs. students). This makes ACLs much more concise, making it easier to enforce them with minimal hardware support at the switches. Third, ACLs are increasingly being moved out of the network and on to end hosts for easier management and better scalability. In corporate enterprises, distributed firewalls [26, 27], managed by Active Directory [28] or LDAP (Lightweight Directory Access Protocol), are often used to enforce access-control policies. In data-center networks, access control is even easier since the operators have complete control of end hosts. Therefore, the design of BUFFALO focuses simply on providing any-to-any reachability, though we briefly discuss possible ways to support VLAN in Section 2.8.

2.3 Packet Forwarding in BUFFALO

In this section, we describe the BUFFALO switch architecture in three aspects: First, we use one Bloom filter for each next hop to perform the entire packet lookup in the fast SRAM. Second, we leverage shortest-path routing to forward packets experiencing false positives through slightly longer paths without additional memory overhead. Finally, we leverage counting Bloom filters

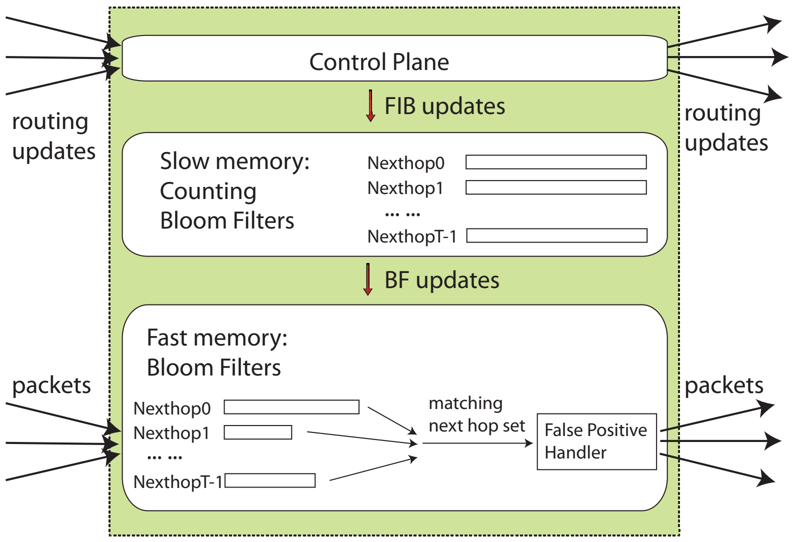


Figure 2.1: BUFFALO Switch Architecture

in slow memory to enable fast updates to the Bloom filters after routing changes. Figure 3.7 summarizes the BUFFALO design.

2.3.1 One Bloom Filter Per Next Hop

One way to use a small, fast memory is route caching. The basic idea is to store the most frequently used entries of the forwarding table (FIB) in the fast memory, but store the full table in the slow memory. However, during cache misses, the switch experiences low throughput and high packet loss. Malicious traffic with a wide range of destination addresses may significantly increase the cache miss rate. In addition, when routing changes or link failures happen, many of the cached routes are simultaneously invalidated. Due to its bad performance under *worst-case workloads*, route caching is hardly used today.

To provide *predictable* behavior under various workloads, we perform the *entire* packet lookup for *all* the packets in the fast memory by leveraging Bloom filters, a hash-based compact data structure to store a set of elements. We set one Bloom filter $BF(h)$ for each next hop h (or outgoing link), and use it to store all the addresses that are forwarded to that next hop. For a switch with T next hops, we need T Bloom filters. A Bloom filter consists of an array of bits. To insert an address into Bloom filter $BF(h)$, we compute k hash values of the address, each denoting a position in the array. All the k positions are set to 1 in the array. By repeating the same procedure for all the addresses with next hop h, Bloom filter $BF(h)$ is constructed.

It is easy to check if an address belongs to the set with Bloom filter $BF(h)$. Given an address, we calculate the same k hash functions and check the bits in the corresponding k positions of the array. If all the bits are 1, we say that the element is in the set; otherwise it is not. To perform address lookup for an address $addr$, we check which $BF(h)$ contains $addr$, and forward the packet to the corresponding next hop h.

Note that there are different number of addresses associated with each next hop. Therefore we should use different size for each Bloom filter according to the number of addresses stored in it, in order to minimize the overall false-positive rate with a fixed size of fast memory. We formulate and solve the false-positive rate optimization problem in Section 2.5.

2.3.2 Handling False Positives in Fast Memory

One key problem with Bloom filters is the false positive — an element can be absent from the set even if all k positions are marked as 1, since each position could be marked by the other elements in the set. Because all the addresses belong to one of the Bloom filters we construct, we can easily detect packets that experience false positives if they match in multiple Bloom filters.[3]

One way to handle packets experiencing false positives is to perform a full packet lookup in the forwarding table stored in the slow memory. However, the lookup time for packets experiencing false positives will be much longer than others, leading to the throughput decrease. Attackers may detect those packets with longer latency and send a burst of them. Therefore, we must handle false positives in fast memory by picking one of the matching next hops.

For the packets that experience false positives, if we do not send them through the next hop on the shortest path, they may experience stretch and even loops. One way to prevent loops is to use a deterministic solution by storing the false positive information in the packets (similar to FCP [29]). When a packet is detected to have false positives in a switch, we store the switch and the next hop it chooses in the packet. So next time the packet travels to the same switch, the

[3]The handling of addresses that should have *multiple* matches (*e.g.*, Equal-Cost Multi-Path) are discussed in Section 2.8. Addresses that have *no match* in the FIB should be dropped. Yet these packets may hit one Bloom filter due to false positives. We cannot detect these addresses, but they will eventually get dropped when they hit a downstream switch that has no false positives. In addition, an adversary cannot easily launch an attack because it is hard to guess which MAC addresses would trigger this behavior.

switch will choose a different next hop for the packet. However, this method requires modifying the packets and has extra payload overhead.

Instead, we use a probabilistic solution without any modification of the packets. We observe that if a switch sends the packet back to the interface where it comes from, it will form a loop. Therefore, we avoid sending the packet to the incoming interface. To finally get out the possible loops, we randomly pick one from all the remaining matching next hops. In Section 2.4, we prove that in SPAF networks, the packet experiencing one false positive (which is the most common case for the packet) is guaranteed to reach the destination with constant bounded stretch. In general, packets are guaranteed to reach the destination with probability 1. This approach does not require any help from the other switches in the network and thus is incrementally deployable.

2.3.3 CBFs for Handling Routing Changes

When routing changes occur, the switch must update the Bloom filters with a new set of addresses. However, with a standard Bloom filter (BF) we cannot delete elements from the set. A *counting Bloom filter* (CBF) is an extension of the Bloom filter that allows adding and deleting elements [30]. A counting Bloom filter stores a counter rather than a bit in each slot of the array. To add an element to the counting Bloom filter, we increment the counters at the positions calculated by the hash functions; to delete an element, we decrement the counters.

A simple way to handle routing changes is to use CBFs instead of BFs for packet forwarding. However, CBFs require much more space than BFs. In addition, under routing changes the number of addresses associated with each next hop may change significantly. It is difficult to dynamically increase/decrease the sizes of CBFs to make the most efficient use of fast memory according to routing changes.

Fortunately, since routing changes do not happen very often, we can store CBFs in slow memory, and update the BFs in small fast memory based on the CBFs. We store CBF in slow memory rather than a normal FIB because it is easier and faster to update BFs from CBFs under routing changes. With a CBF layer between BFs and control plane, we can even change the sizes of BFs to the optimal values with low overhead. By using both CBFs and BFs, we make an efficient use of small fast memory without losing the flexibility to support changes in the FIB. The details of using CBFs are discussed in Section 2.6.

As shown in Figure 3.7, there are three layers in our switch architecture. The control plane can be either today's Ethernet or new Ethernet techniques such as Rbridges [7, 8] and SEATTLE [5]. CBFs are stored in slow memory and learn about routing changes from the control plane. BFs are stored in the fast memory for packet lookup. During routing changes, the related BFs will be updated according to the corresponding CBFs. If the number of addresses associated with a BF changes significantly, BFs are reconstructed with new optimal sizes from CBFs.

2.4 Handling False Positives

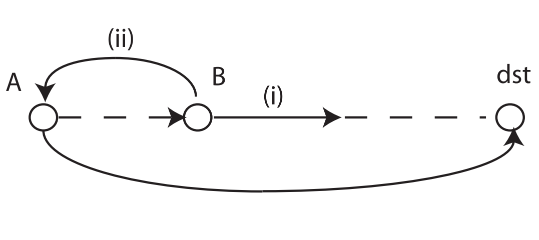
When BUFFALO detects packets that experience false positives, it randomly selects a next hop from the candidates that are different from the incoming interface, as discussed in Section 2.3.2. In the case of a single false positive, which is the most common, avoiding sending the packet to the incoming interface guarantees that packets reach destination with tightly bounded stretch. In the improbable case of multiple false positives, this randomization guarantees packet reachability with probability 1, with a good bounds on expected stretch.

Notation	**Definition**
$NH_{sp}(A)$	The next hop for shortest path at switch A
$NH_{fp}(A)$	The matching next hop due to a sole false positive at switch A
$l(A,B)$	The length of the shortest path from switch A to B
$P(A,B)$	All the paths from switch A to B

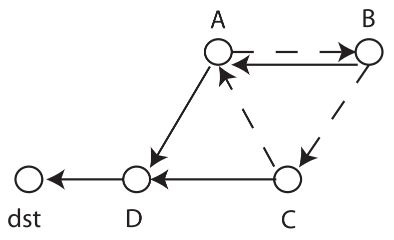
Table 2.1: Notations for the false-positive handler

2.4.1 Handling One False Positive by Avoiding Incoming Interface

We first investigate the case that a packet only experiences one false positive at one switch, *i.e.*, at switch A, the packet has two matching next hops. Let $NH_{sp}(A)$ denote the next hop A should forward the packet to on the shortest path, and $NH_{fp}(A)$ denote the additional next hop matched by a Bloom filter false positive. (The notations are summarized in Table 2.4.) Note that this single false positive case is the most common case, because with reasonable Bloom filter parameters the probability of multiple false positives is much lower than one false positive. Since switch A connects



(a) One false positive



(b) Multiple false positives

Figure 2.2: Loops caused by false positives ($\dashrightarrow$ false positive link, $\longrightarrow$ shortest path link)

to each end host through one switch port, there is no false positives for the ports that connect to end hosts. Therefore, $NH_{fp}(A)$ must be a switch rather than an end host.

The one false positive case is shown in Figure 2.2(a). (We hereafter use $\dashrightarrow$ to denote false positive link, and $\rightarrow$ for shortest path link.) Switch A has a false positive, and randomly picks a next hop B ($NH_{fp}(A) = B$). Switch B receives the packet, and may (i) send it to a next hop different from A, which leads to the destination or (ii) send it back to A. For (i), we will prove that there are not any loops. For (ii), when A receives the packet, it will not pick B since the packet comes from B.

In general, we have the following theorem:

Theorem 1. *In SPAF networks, if a packet only experiences one false positive in one switch, it is guaranteed to reach the destination with no loops except a possible single transient 2-hop one.*

Proof. Suppose the packet matches two next hops at switch A: $NH_{sp}(A)$ and some $B = NH_{fp}(A)$. If A picks $NH_{sp}(A)$, there is no loop. If B is selected, it will have a path $B \rightarrow \ldots \rightarrow dst$ to forward the packet to the destination, since the network is connected. With a single false positive, a packet will follow the correct shortest-path hops at all nodes other than A. Thus, the only case that can cause a loop is the packet going through A again ($A \dashrightarrow B \rightarrow \ldots \rightarrow A \rightarrow \ldots \rightarrow dst$). However, in SPAF networks, we assume the link $e(B, A)$ is actively used (This assumption is introduced in Sec. 2.2), *i.e.*,

$$l(NH_{fp}(A), A) \leq l(p), \forall p \in P(NH_{fp}(A), A).$$

Thus, on a shortest path from B to dst, if the packet visits A at all, it would be immediately after B. If the packet is sent back to A, A will avoid sending it to the incoming interface $NH_{fp}(A)$, and thus send the packet to $NH_{sp}(A)$, and the shortest path from there to dst can't go through

A. Thus, the packet can only loop by following $A \dashrightarrow NH_{fp}(A) \rightarrow A \rightarrow \ldots \rightarrow dst$. So the path contains at most one 2-hop loop. $\square$

Now we analyze the stretch (*i.e.*, latency penalty) the packets will experience. For two switches A and B, let $l(A, B)$ denote the length of the shortest path from A to B. Let $l'(A, B)$ denote the latency the packet experiences on the path in BUFFALO. We define the stretch as:

$$S = l'(A, B) - l(A, B)$$

.

Theorem 2. *In SPAF networks, if a packet experiences just one false positive at switch A, the packet will experience a stretch of at most* $l(A, NH_{fp}(A)) + l(NH_{fp}(A), A)$.

Proof. Since there is one false positive in A, A will choose either $NH_{sp}(A)$ or $NH_{fp}(A)$. If A picks $NH_{sp}(A)$, there is no stretch. If A picks $NH_{fp}(A)$, there are two cases: (i) If $NH_{fp}(A)$ sends the packet to the destination without going through A, the shortest path from $NH_{fp}(A)$ to the destination is followed. Based on the triangle inequality, we have:

$$l(NH_{fp}(A), dst) \leq l(NH_{fp}(A), A) + l(A, dst)$$
$$l'(A, dst) - l(A, dst) \leq l(A, NH_{fp}(A)) + l(NH_{fp}(A), A)$$

(ii) If $NH_{fp}(A)$ sends the packet back to A, the stretch is $l(A, NH_{fp}(A)) + l(NH_{fp}(A), A)$. $\square$

Therefore, we prove that in the one false positive case, packets are guaranteed to reach the destination with bounded stretch in SPAF networks.

2.4.2 Handling Multiple False Positives with Random Selection

Now we consider the case where all the switches in the network apply our Bloom filter mechanism. We choose different hash functions for different switches so that the false positives are independent among the switches. (We can choose different keys for key-based hash functions to generate a group of hash functions.) Thus it is rare for a packet to experience false positives at multiple switches.

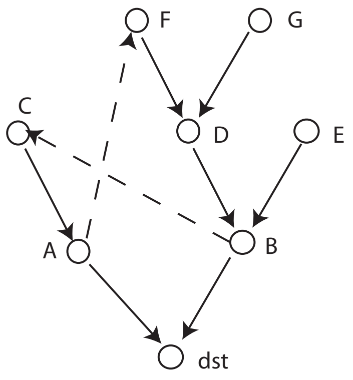


Figure 2.3: BUFFALO forwarding graph ($\dashrightarrow$ false positive link, $\longrightarrow$ shortest path tree)

Let us fix a destination *dst*, and condition the rest of this section on the fixed forwarding table and memory size (which, per Section 2.5, means the Bloom filter parameters are fixed, too). Let $f(h)$ be the probability of Bloom filter h erroneously matching *dst* (a priori independent of *dst* if *dst* shouldn't be matched). Then, k, the number of Bloom filters mistakenly matching d is, in expectation, $F = \sum_h f(h)$. If all values of f are comparable and small, F is roughly binomial, with $\Pr[k > x]$ decays exponentially for $x \gg 2f|E|$.

There may exist loops even when we avoid picking the incoming interface. For example, in Figure 2.2(b), A may choose the false-positive hop to B at first, and B may choose the false-positive next hop C. However, the packet can eventually get out of the loop if each switch chooses the next hop randomly: any loop must contain a false-positive edge, and the source node of that edge will with some probability choose its correct hop instead. For example, A will eventually choose the next hop D to get out of the loop. Such random selection may also cause out-of-order packets. This may not be a problem for some applications. For the applications that require in-order packet forwarding, we can still reorder packets in the end hosts.

In general, in SPAF networks there is a shortest path tree for each destination *dst*, with each packet to *dst* following this tree. In BUFFALO, packets destined to *dst* are forwarded in the *directed* graph consisting of a shortest path tree for *dst* and a few false positive links. We call this graph *BUFFALO forwarding graph*. An example is shown in Figure 2.3, where two false positives occur at A and B. Note that, if the shortest-path links are of the same length (e.g., in terms of latency), the link from A to F cannot be less than about twice that length: otherwise F's shortest

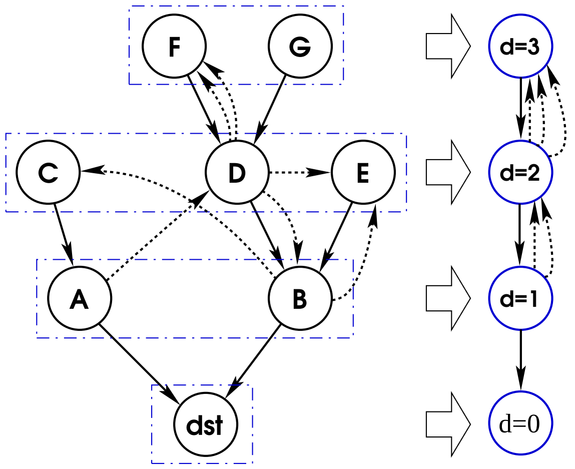


Figure 2.4: Coupling for the expected stretch proof: all nodes at the same hop distance from dst are collapsed into 1 node. Forward false positive links, like D to B, are dropped. The number of backward edges on the line graph is the maximum over all nodes at that distance of the number of backward and same-depth false-positive links (B's edges at $d = 1$, D's edges at $d = 2$). A random walk on the line graph converges no faster than a random walk on the original. The line graph itself is a valid network, thus allowing for tight bounds.

path would go through A. If all links are the same length , no false positive edge can take more than one "step" away from the destination.

If a packet arrives at the switch that has multiple outgoing links in BUFFALO forwarding graph due to false positives, we will randomly select a next hop from the candidates. Thus, the packet actually takes a random walk on the BUFFALO forwarding graph, usually following shortest-path links with occasional "setbacks".

Theorem 3. *With an adversarially designed BUFFALO network with uniform edge latencies, and even worst-case placement of false positives, the expected stretch of a packet going to a destination associated with k false positives is at most* $S(k) = \rho \cdot (\sqrt[3]{3})^k$, *where* $\rho = \frac{529}{54 \cdot \sqrt[3]{3}} < 6.8$.

To prove this theorem, we first present the model used in our proof:

Step 1: Model.

We make the following two assumptions:

1. We assume the underlying SPAF computation is based on "hop count" only, or, equivalently, equal edge weights on all edges. We expect that the bounds we derive under this assumption

will not differ qualitatively from the typical case of edge weights that are diverse but roughly on the same order of magnitude.

2. BUFFALO implements a "no-bounce" rule: if a node detects a false positive, it randomly selects a next hop out of the several candidates that the Bloom filters then offer, but *excludes* from consideration the hop that the packet arrived from. We allow the selection of the latter, as well, ignoring the "no-bounce" optimization in BUFFALO. This simplifies the analysis, and we expect that the upper bounds should not get any worse in the presence of this feature.

We represent the underlying network with an undirected graph $G = (V, E)$, with n nodes, each representing a switch or an end host. Consider forwarding to a particular destination $\mathsf{dst} \in V$ with some, possibly all, switches in the network implementing BUFFALO. The underlying protocol builds a shortest-path tree $(V, T \subset E)$, directed toward dst.

After any given FIB update, some BUFFALO-using routers may have false positives for destination dst. A false positive at node v for next hop v' will cause BUFFALO to possibly forward from v to v' — but this can only happen if the edge $\{v, v'\}$ exists in G. Let (V, B) be the directed "BUFFALO graph," containing all the edges along which BUFFALO switches might forward, given the current false positives. That is, B consists of T, and, for each false positive at node v that matches next hop v' for destination dst, an additional (directed) edge (v, v'). We use P to denote the false-positive edges alone ($P = B \setminus T$).

Observation 2.4.1. *Without the "no-bounce" rule, if some or all routers use BUFFALO, a packet to* dst *performs an unbiased random walk on* (V, B) *until reaching* dst.

For background on random walks, we refer the reader to any of a number of standard books, such as [31, 32, 33].

We denote by $d_T(v, w)$ the hop distance from v to w in T, and use $d(v)$ as shorthand for $d_T(v, \mathsf{dst})$, the distance from v to the destination in T, which we call the *depth* of v.

Observation 2.4.2. $d(v) = d_E(v, \mathsf{dst}) = d_B(v, \mathsf{dst})$*, since T is a shortest-path tree, and is contained in B and E. The shortest-path property also ensures that, for any edge $\{v, w\} \in E$,* $|d(v) - d(w)| \leq 1$.

We say a packet traveling from v to dst experiences stretch S if it traverses $d(v)+S$ hops before reaching dst. The expected stretch of a packet starting at v is directly connected to the random walk's expected hitting time from v to dst: $T_{v,\mathsf{dst}} = d(v) + \mathrm{E}[S]$.

Now, Let's prove Theorem 3.

Step 2: Calculate expected stretch.

First, "project" the whole graph (the shortest path tree and the "noise" false positive edges) onto a line (multi)graph L where node l_i corresponds to all nodes in the real network at depth i away from dst, as shown in Figure 2.4. Let's say l_0, corresponding to the destination, is on the "right" while l_x, where x is the equivalent of "diameter" (the length of the longest shortest path to d), is on the left.

l_i has 1 edge pointing to the right, and k_i edges pointing to the left, where k_i is the max, over all nodes v at depth i in the original graph, of the number of false positives at v that point to nodes of depth i or $i+1$.

Lemma 2.4.3. *The expected stretch of an unbiased random walk on B starting at v is at most the expected stretch of an unbiased random walk on L starting at $l_{d(v)}$.*

Proof. The lemma follows from a conventional coupling argument, between the random walk's depth in B and the random walk's depth in L.

Consider the next hop from some node v in B, at depth d, which has $i \geq 1$ edges to nodes of depth $d-1$, j edges to nodes of depth d, and k edges to nodes of depth $d+1$. By construction, the corresponding node l_d has at least $k+j$ edges to l_{d+1} and one edge to l_{d-1}. The probability of the walk moving to depth $d-1$ in L at most $1/(k+j+1)$, while the walk in B moves to depth $d-1$ with probability $i/(i+k+j) \geq 1/(k+j+1)$. This allows a natural coupling for depth of the next hop in B and in L:

If the L walk takes a deeper next hop than the B walk, we can "suspend" the B walk, and let the L walk continue until the next time it reaches a node at the same depth as the B walk — this is guaranteed to happen, since depth never changes by more than 1 hop. After that point, proceed with the next hop on both random walks. Since L and B random walks start at the same depth, this coupling guarantees that L will never move to a lower depth than B, and thus that L will take at least as many hops as B. ☐

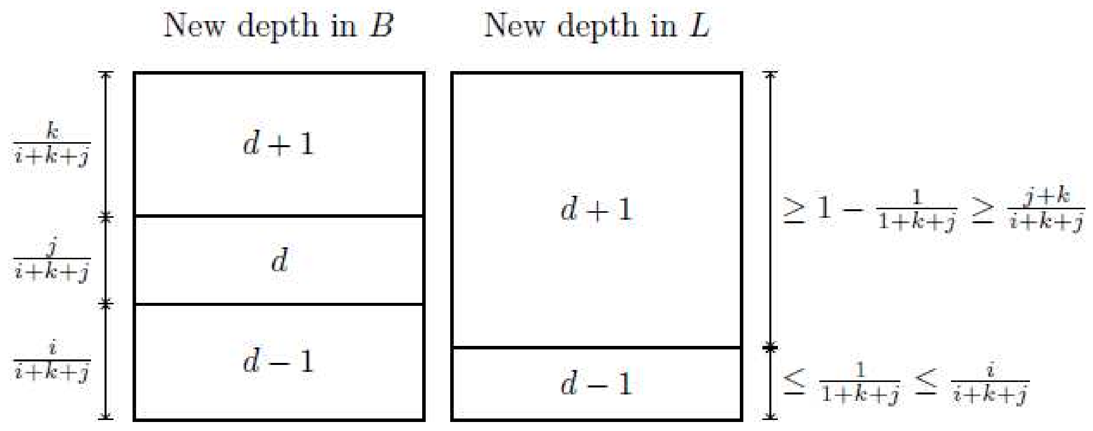


Figure 2.5: Coupling the depth process in B and L

We now analyze the worst-case expected hitting time for an unweighted random walk on L. For any $i > j$, the hitting time from l_i to l_0 must include at least one visit to l_j, guaranteeing that the expected hitting time to l_0 is maximized at l_x. The transition probability matrix (where $P_{i,j}$ is the probability, when at l_i, of going to l_j on the next step) is:

$$P = \begin{pmatrix} 0 & 1 & 0 & \cdots & & & 0 \\ \frac{k_{x-1}}{k_{x-1}+1} & 0 & \frac{1}{k_{x-1}+1} & 0 & \cdots & & 0 \\ 0 & \frac{k_{x-2}}{k_{x-2}+1} & 0 & \frac{1}{k_{x-2}+1} & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & \cdots & 0 & \frac{k_2}{k_2+1} & 0 & \frac{1}{k_2+1} & 0 \\ 0 & & \cdots & 0 & \frac{k_1}{k_1+1} & 0 & \frac{1}{k_1+1} \\ 0 & & & \cdots & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Following the expected hitting time analysis detailed in, *e.g.*, [34], which cites [35], we let Q be the matrix P with the last row and last column removed. The expected hitting time from v_x to v_0 is then:

$$T = \sum_j (I-Q)^{-1}_{0,j} \tag{2.2}$$

Inversion of $(I-Q)$ via Gaussian elimination on the augmented matrix yields:

$$
\begin{array}{r}
R_x : \\ R_{x-1} : \\ R_{x-2} : \\ \vdots \\ R_2 : \\ R_1 :
\end{array}
\left(\begin{array}{cccccc|cccccc}
1 & -1 & 0 & \cdots & & 0 & 1 & & & & & \\
\frac{-k_{x-1}}{k_{x-1}+1} & 1 & \frac{-1}{k_{x-1}+1} & 0 & \cdots & 0 & & 1 & & & & \\
0 & \frac{-k_{x-2}}{k_{x-2}+1} & 1 & \frac{-1}{k_{x-2}+1} & \cdots & 0 & & & 1 & & & \\
\vdots & & \ddots & \ddots & \ddots & \vdots & & & & \ddots & & \\
0 & \cdots & 0 & \frac{-k_2}{k_2+1} & 1 & \frac{-1}{k_2+1} & & & & & 1 & \\
0 & & \cdots & 0 & \frac{-k_1}{k_1+1} & 1 & & & & & & 1
\end{array}\right)
\begin{array}{l}
\\ \leftarrow R'_{x-1} = (k_{x-1}+1)R_{x-1} + k_{x-1}R'_x \\ \leftarrow R'_{x-2} = (k_{x-2}+1)R_{x-2} + k_{x-2}R'_{x-1} \\ \vdots \\ \leftarrow R'_2 = (k_2+1)R_2 + k_2R'_3 \\ \leftarrow R'_1 = (k_1+1)R_1 + k_1R'_2
\end{array}
$$

$$
\rightsquigarrow
\begin{array}{r}
R'_x : \\ R'_{x-1} : \\ R'_{x-2} : \\ \vdots \\ R'_2 : \\ R'_1 :
\end{array}
\left(\begin{array}{cccccc|cccccc}
1 & -1 & 0 & \cdots & & 0 & 1 & 0 & \cdots & & & \\
0 & 1 & -1 & 0 & \cdots & 0 & k_{x-1} & (k_{x-1}+1) & 0 & \cdots & & \\
0 & 0 & 1 & -1 & \cdots & 0 & k_{x-1}k_{x-2} & (k_{x-1}+1)k_{x-2} & (k_{x-2}+1) & 0 & \cdots & \\
\vdots & & \ddots & \ddots & \ddots & \vdots & & & & \ddots & & \\
0 & \cdots & 0 & 0 & 1 & -1 & k_{x-1}k_{x-2}\cdots k_2 & (k_{x-1}+1)k_{x-2}\cdots k_2 & \cdots & & (k_2+1) & 0 \\
0 & & \cdots & 0 & 0 & 1 & k_{x-1}k_{x-2}\cdots k_2k_1 & (k_{x-1}+1)k_{x-2}\cdots k_2k_1 & \cdots & & (k_2+1)k_1 & (k_1+1)
\end{array}\right)
\begin{array}{l}
\leftarrow R''_x = \sum_i R'_i \\ \\ \\ \\ \\ \\
\end{array}
\tag{2.3}
$$

The expecting hitting time, the sum of the top row of the inverse matrix, is thus the sum of all the entries on the right side. We split the sum of each column c except the first into $\sum_{i=1}^{x-c+1}\prod_{j=i}^{x-c+1} k_j + \sum_{i=1}^{x-c}\prod_{j=i}^{x-c} k_j + 1$, and then group the first term with the previous column, $c-1$. This yields:

$$
\begin{aligned}
T &= x + 2\sum_{c=1}^{x}\sum_{i=1}^{x-c}\prod_{j=i}^{x-c} k_j && (2.4) \\
&= x + 2\sum_{1\le a\le b\le x}\prod_{i=a}^{b} k_i && (2.5)
\end{aligned}
$$

Since, with no false positives, a packet may take up to x hops to get to d, the stretch S is thus upper bounded by $S \le T - x = 2\sum_{1\le a\le b\le x}\prod_{i=a}^{b} k_i$.

We'll use the symbol S_p^q for $\sum_{p\le a\le b\le q}\prod_{i=a}^{b} k_i$, i.e. the sum of products over each possible substring of integer sequence $\{k_i\}$.

The S_p^q notation, as well as the L^q and R_p notations introduced below, leave implicit their dependence on the sequence $\{k_i\}$. When discussing proposed changes to the sequence, these symbols are to always be interpreted in terms of the sequence *before* the change in question.

Step 3: Exact patterns for maximizing S_1^x.

The remainder of the proof of Theorem 3 is a detailed combinatorial treatment of maximizing S_1^x over all sequences whose sum, i.e. the total number of false positives $\sum_i k_i$, is fixed at some k. We proceed by listing a series of transformations of any candidate maximizing sequence that never change the sum, never decrease the sum of substring products, and converge to a very specific family of maximum sequences that allows S to be explicitly computed.

We forewarn the reader that this subsection is relatively technical and is unlikely to be of broader interest outside this proof.

In all the statements below, we implicitly require that $\sum_i k_i = k$.

Observation 2.4.4. *To upper-bound stretch, we can assume, WLOG, that $k_i > 0$ for all i. Otherwise, removing that k_i and thus shrinking x by 1 cannot decrease S: the walk before the edge is removed can never return to the part of L above this edge, so contracting the edge only adds possible extra upward traversals, which would still have to return to this node before continuing to* dst*.*

Lemma 2.4.5. *There is a sequence $\{k_i\}$ maximizing S_1^x that has $k_i \le 3$ for all i.*

Proof. Suppose there is a $k_i \ge 4$ in some sequence $\{k_i\}$ maximizing S_1^x. Then, split k_i into two adjacent sequence elements, 2 and $k_i - 2$. Since $2(k_i - 2) \ge k_i$, each substring of the original sequence that contained k_i will have a corresponding substring including both 2 and $k_i - 2$ in the new sequence, with a product at least as big as the original substring. Any substring that didn't include k_i originally will still exist as-is. Also, the new sum of substring products will also include the substring containing just the new 2 element, which does not correspond to any of the original substrings, thus guaranteeing a strictly greater substring product sum. $\square$

Lemma 2.4.6. *There is a sequence $\{k_i\}$ that maximizes S that has $1 \le k_i \le 3$ and is* dip-free*: if $a < b < c$, and $k_a > k_b$, then $k_b \ge k_c$; and if $a < b < c$ and $k_b < k_c$, then $k_a \le k_b$. That is, $\{k_i\}$ matches the regular expression* $1^*2^*3^*2^*1^*$.

Proof. Consider a sequence $\{k_i\}$ maximizing S. We will see that it is dip-free by considering transpositions of two adjacent sequence elements.

Let L^p denote $\sum_{1\le a\le p}\prod_{i=a}^{p} k_i$, the partial sum of substring products covering all substrings ending exactly at p. Similarly, let R_q denote $\sum_{q\le b\le x}\prod_{i=q}^{b} k_i$, the partial sum of substring products covering all substrings starting exactly at q. We define $\mathsf{L}^0 = \mathsf{R}_{x+1} = 0$.

Note that, since $k_i \geq 1$, $\mathsf{L}^p = k_p\mathsf{L}^{p-1} + k_p$ monotonically strictly increases with p, and $\mathsf{R}_q = k_q\mathsf{R}_{q+1} + k_q$ monotonically strictly decreases with q.

For any $1 \leq i \leq x-1$, we can now split the full sum of S_1^x into several categories, based on where the substring lies in relation to k_i and k_{i+1}:

$$\begin{aligned}\mathsf{S}_1^x =& \mathsf{S}_1^{i-1} + \mathsf{S}_{i+2}^x + k_ik_{i+1} + k_i + k_{i+1} + k_ik_{i+1}\left(\mathsf{L}^{i-1} + \mathsf{R}_{i+2} + \mathsf{L}^{i-1}\mathsf{R}_{i+2}\right)\\ &+ k_i\mathsf{L}^{i-1} + k_{i+1}\mathsf{R}_{i+2}\end{aligned}$$

All *except* the last two of the terms add up to an expression that is symmetric with respect to transposing k_i and k_{i+1}. Now, suppose $k_i < k_{i+1}$. Then, since the sequence is maximizing, its S_1^x is greater than it would be if k_i and k_{i+1} were transposed, requiring:

$$\begin{aligned}k_i\mathsf{L}^{i-1} + k_{i+1}\mathsf{R}_{i+2} &\geq k_{i+1}\mathsf{L}^{i-1} + k_i\mathsf{R}_{i+2}\\ (k_{i+1} - k_i)(\mathsf{R}_{i+2} - \mathsf{L}^{i-1}) &\geq 0\\ \mathsf{R}_{i+2} - \mathsf{L}^{i-1} &\geq 0\end{aligned}$$

Then, for any $i' < i$, strict monotonicity gives us $\mathsf{R}_{i'+2} - \mathsf{L}^{i'-1} > 0$. If $k_{i'} > k_{i'+1}$, we would have $(k_{i'+1} - k_{i'})\left(\mathsf{R}_{i'+2} - \mathsf{L}^{i'}\right) < 0$, contradicting maximality of $\{k_i\}$, since that would allow us to raise S_1^x by transposing $k_{i'}$ and $k_{i'+1}$. Inductively applying this argument shows that, if $a > b > c$ and $k_b < k_c$, then $k_a \leq k_b$. The other side follows by symmetry. $\square$

Lemma 2.4.7. *There exists a maximizing sequence obeying the constraints of Lemma 2.4.6 that starts and ends with a 1,* i.e., *it matches the regular expression* 11*2*3*2*1*1.

Proof. If $k_1 > 1$ in a maximizing sequence, consider splitting it into adjacent elements 1 and $k_1 - 1$. This transforms the original sum of substring products from $k_1 + k_1\mathsf{R}_2 + \mathsf{S}_2^x$ into $1 + k_1 - 1 + 1(k_1 - 1) + (k_1 - 1)\mathsf{R}_2 + 1(k_1 - 1)\mathsf{R}_2 + \mathsf{S}_2^x = 1 + 2(k_1 - 1) + 2(k_1 - 1)\mathsf{R}_2 + \mathsf{S}_2^x$. Since $k_1 \leq 2(k_1 - 1)$, this increases the total sum of substring products by 1. The same argument holds for k_x. $\square$

Observation 2.4.8. *An adjacent* $(1,3)$ *pair can always be replaced with a* $(2,2)$ *pair, strictly increasing* S_1^x*. It changes from* $\mathsf{S}_1^{i-2}+\mathsf{S}_{i+1}^x+(1+3)\mathsf{L}^{i-2}+(3+3)\mathsf{R}_{i+1}+3\mathsf{L}^{i-2}\mathsf{R}_{i+1}+1+3+3$ *to* $\mathsf{S}_1^{i-2}+\mathsf{S}_{i+1}^x+(2+4)\mathsf{L}^{i-2}+(2+4)\mathsf{R}_{i+1}+4\mathsf{L}^{i-2}\mathsf{R}_{i+1}+2+2+4$*, an increase by* $2\mathsf{L}^{i-2}+\mathsf{L}^{i-2}\mathsf{R}_{i+1}+1$*, which is always positive.*

Lemma 2.4.9. *For* $k \geq 7$*, there is an maximizing sequence satisfying 2.4.2 that has at most two 1's on each side.*

Proof. Consider a sequence satisfying 2.4.2 with at least three 1's at the start. If the first non-1 element is some $k_i = 3$, Observation allows us to replace it and the preceding 1 with (2,2).

If the first non-1 element is some $k_i = 2$, consider replacing $(1,2)$ with a 3. S_1^x changes from $\mathsf{S}_1^{i-2}+\mathsf{S}_{i+1}^x+(1+2)\mathsf{L}^{i-2}+(2+2)\mathsf{R}_{i+1}+2\mathsf{L}^{i-2}\mathsf{R}_{i+1}+1+2+2$ to $\mathsf{S}_1^{i-2}+\mathsf{S}_{i+1}^x+3\mathsf{L}^{i-2}+3\mathsf{R}_{i+1}+3\mathsf{L}^{i-2}\mathsf{R}_{i+1}+3$. The increment is $\mathsf{L}^{i-2}\mathsf{R}_{i+1}-\mathsf{R}_{i+1}-2$, non-negative if $(\mathsf{L}^{i-2}-1)\mathsf{R}_{i+1} \geq 2$.

For $k \geq 7$, this is satisfied, since either:

1. There are four or more ones before k_i. Then, $\mathsf{L}^{i-2}-1 \geq 2$. By 2.4.2, there is at least one element after k_i, guaranteeing $\mathsf{R}_{i+1} \geq 1$;

2. Or $\sum_{j=i+1}^{x} k_j \geq 2$, guaranteeing $\mathsf{R}_{i+1} \geq 2$. Since the sequence starts with three or more ones, $\mathsf{L}^{i-2}-1 \geq 1$.

A symmetrical argument applies to the other end of the sequence. □

Lemma 2.4.10. *For* $k \geq 18$*, any maximizing sequence satisfying 2.4.9 has at least one 3.*

Proof. With $k \geq 18$ no 3's, and at most two 1's at each end, the sequence must then have at least seven 2's. Let k_{i-2} be the first 2. We consider replacing the third, fourth, and fifth 2's (k_i, k_{i+1}, and k_{i+2})) with two 3's.

Before the change, S_1^x is $\mathsf{S}_1^{i-1}+\mathsf{S}_{i+3}^x+(2+4+8)\mathsf{L}^{i-1}+(2+4+8)\mathsf{R}_{i+3}+8\mathsf{L}^{i-1}\mathsf{R}_{i+3}+2+2+2+4+4+8=\mathsf{S}_1^{i-1}+\mathsf{S}_{i+3}^x+14(\mathsf{L}^{i-1}+\mathsf{R}_{i+3})+8\mathsf{L}^{i-1}\mathsf{R}_{i+3}+22$. After the change, it becomes $\mathsf{S}_1^{i-1}+\mathsf{S}_{i+3}^x+(3+9)\mathsf{L}^{i-1}+(3+9)\mathsf{R}_{i+3}+9\mathsf{L}^{i-1}\mathsf{R}_{i+3}+3+3+9=\mathsf{S}_1^{i-1}+\mathsf{S}_{i+3}^x+12(\mathsf{L}^{i-1}+\mathsf{R}_{i+3})+9\mathsf{L}^{i-1}\mathsf{R}_{i+3}+15$. This is an increase of $\mathsf{L}^{i-1}\mathsf{R}_{i+3}-2(\mathsf{L}^{i-1}+\mathsf{R}_{i+3})-7=(\mathsf{L}^{i-1}-2)(\mathsf{R}_{i+3}-2)-11$. With $k_{i-2}=k_{i-1}=k_{i+3}=k_{i+4}=2$, we are guaranteed that both L^{i-1} and R_{i+3} are at least 6, making the increase positive. □

Lemma 2.4.11. *For $k \geq 18$, any maximizing sequence satisfying 2.4.10 has exactly one 1 on each side.*

Proof. Suppose, WLOG, there are exactly two 1's in the beginning (Lemmas 2.4.2 and 2.4.9 allow us to reach such a maximizing sequence from any other). Use b for the number of 2's between the last 1 and the first 3 ($b = i - 3$). Consider replacing $k_2 = 1$ and the first 3, $k_i = 3$ (which Lemma 2.4.10 guarantees to exist) with $k_2 = k_i = 2$. The increment in S_1^x is:

$$\begin{aligned}
&\Bigg(\mathsf{S}_{i+1}^x + \mathsf{S}_3^{i-1} + \mathsf{R}_{i+1}\left(1 \cdot 2 \cdot 2^b \cdot 2 + 2 \cdot 2^b \cdot 2 + 2\sum_{j=0}^{b} 2^j\right) \\
&\qquad + 2 \cdot \left(\sum_{j=0}^{b} 2^j\right) + 2 \cdot 2^b \cdot 2 + 2 \cdot 2^b \cdot 2 \cdot 1 + 2 \cdot \left(\sum_{j=0}^{b} 2^j\right) + 1 \cdot 2 \cdot \left(\sum_{j=0}^{b} 2^j\right) + 1\Bigg) - \\
&\qquad\qquad \Bigg(\mathsf{S}_{i+1}^x + \mathsf{S}_3^{i-1} + \mathsf{R}_{i+1}\left(1 \cdot 1 \cdot 2^b \cdot 3 + 1 \cdot 2^b \cdot 3 + 3\sum_{j=0}^{b} 2^j\right) \\
&\qquad + 3 \cdot \left(\sum_{j=0}^{b} 2^j\right) + 3 \cdot 2^b \cdot 1 + 3 \cdot 2^b \cdot 1 \cdot 1 + 1 \cdot \left(\sum_{j=0}^{b} 2^j\right) + 1 \cdot 1 \cdot \left(\sum_{j=0}^{b} 2^j\right) + 1\Bigg) \\
&\qquad = \mathsf{R}_{i+1}\big((12 \cdot 2^b - 2) - (12 \cdot 2^b - 3)\big) + \big((20 \cdot 2^b - 5) - (16 \cdot 2^b - 4)\big) \\
&\qquad\qquad = \mathsf{R}_{i+1} + 2^{b+2} - 1 \qquad (2.6)
\end{aligned}$$

Since $b \geq 1$, the increment is strictly positive, violating maximality. A symmetric argument applies for two 1's at the end. $\square$

Lemma 2.4.12. *For $k \geq 18$, any maximizing sequence satisfying 2.4.11 has at most three 2's on each side of the 3's.*

Proof. Consider a maximizing sequence with at least four 2's before the 3's. First, apply to ensure that we're considering a maximizing sequence starting and ending with a 1.

By Observation 2.4.2, in a maximizing sequence, there must be at least one 2 after the 3's — else the total is strictly increased by replacing a $(3, 1)$ pair with $(2, 2)$.

Like in Lemma 2.4.10 above, we consider replacing the last three 2's before the 3's with two new 3's, which will increase the total if $(\mathsf{L}^{i-1} - 2)(\mathsf{R}_{i+3} - 2) > 11$. But the existence of at least one 3, at least one 2, and a 1 after the triple we're replacing guarantees that $\mathsf{R}_{i+3} - 2 \geq 3 + 6 + 6 - 2 = 13$. Since there must be at least one extra 2 before the triple, and at least one 1, we have $\mathsf{L}^{i-1} \geq 4$, guaranteeing the strict inequality. $\square$

We are now guaranteed, for $k \geq 18$, the existence of a maximizing sequence of form $12^b3^c2^d1$ where $1 \leq b, d \leq 3$ and $c \geq 1$. We can now obtain a convenient closed form for S_1^x for such sequences. First note that the substring products of a homogeneous sequence z^n add up to:

$$\begin{aligned} m(z,n) &= \sum_{1 \leq a \leq b \leq n} \prod_{i=a}^{b} k_i \\ &= z \sum_{a=1}^{n} \sum_{b=a}^{n} z^{b-a} \\ &= z \sum_{a=1}^{n} \frac{z^{(n-a+1)} - 1}{z-1} \\ &= \frac{z}{z-1} \sum_{i=1}^{n} (z^i - 1) \\ &= \frac{z}{z-1} \cdot \left(\frac{z^{n+1} - 1}{z-1} - 1 - n \right) \\ &= \frac{z^{n+2} - (n+1)z^2 + nz}{(z-1)^2} \end{aligned}$$

Using this, we can split up the computation of S_1^x based on which part of the sequence the substring starts and ends in:

$$
\begin{aligned}
\mathsf{S}_1^x &= 1 + m(2,b) + m(3,c) + m(2,d) + 1 \\
&\quad + 1 \cdot \sum_{i=1}^{b} 2^i + 1 \cdot 2^b \cdot \sum_{i=1}^{c} 3^i + 1 \cdot 2^b \cdot 3^c \cdot \sum_{i=1}^{d} 2^i + 1 \cdot 2^b \cdot 3^c \cdot 2^d \cdot 1 \\
&\quad + \left(\sum_{i=1}^{b} 2^i\right)\left(\sum_{i=1}^{c} 3^i\right) + \left(\sum_{i=1}^{b} 2^i\right) \cdot 3^c \cdot \left(\sum_{i=1}^{d} 2^i\right) + \left(\sum_{i=1}^{b} 2^i\right) \cdot 3^c \cdot 2^d \cdot 1 \\
&\quad + \left(\sum_{i=1}^{c} 3^i\right)\left(\sum_{i=1}^{d} 2^i\right) + \left(\sum_{i=1}^{c} 3^i\right) \cdot 2^d \cdot 1 \\
&\quad + \left(\sum_{i=1}^{d} 2^i\right) \cdot 1 \\
&= 4 \cdot 2^b - 2b - 4 + 4 \cdot 2^d - 2d - 4 + \frac{9}{4}3^c - \frac{9}{4}c - \frac{9}{4} + \frac{3}{4}c \\
&\quad + (2 - 2 + 3) + (2 - \frac{3}{2} - 3)2^b + (2 - \frac{3}{2} - 3)2^d + (-3 + 4 - 3)3^c \\
&\quad + (\frac{3}{2} - 2 + 3 - 4)2^b3^c + (\frac{3}{2} - 2 + 3 - 4)3^c2^d + (2 + 1 + 4 + 2)2^b3^c2^d \\
&= 9 \cdot 2^b3^c2^d - 1.5 \cdot 3^c(2^b + 2^d) + 0.25 \cdot 3^c + 1.5(2^b + 2^d) - 2(b + d) - 1.5c - 6.25
\end{aligned}
\tag{2.7}
$$

Lemma 2.4.13. *For $k \geq 18$, any maximizing sequence satisfying 2.4.12 cannot contain three consecutive 2's.*

Proof. Suppose, WLOG, $b = 3$. We consider two cases for d:

Suppose $d = 1$. Consider "balancing" the sequence, by setting $b = d = 2$, and leaving c fixed. In equation 2.7, this preserves the value of $b + d$ and $2^b 2^d$. Thus, the only change to S_1^x is in the terms involving $(2^b + 2^d)$, which goes from 9 to 8. Since $c \geq 1$ by Lemma 2.4.10, this guarantees a strict increase in S_1^x, by $1.5 \cdot 3^c - 1.5 \geq 3$.

Suppose $2 \leq d \leq 3$. Consider replacing k_3 and k_4, the second and third 2 at the beginning, and k_{x-2}, the first 2 at the end, with two 3's. This changes b from 3 to 1, d to $d - 1$, and c to $c + 2$. Thus, S_1^x is incremented by:

$$\begin{aligned}
9 \cdot 3^c(2 \cdot 2^{d-1} \cdot 9 - 8 \cdot 2^d) - 1.5 \cdot 3^c(9(2 + 2^{d-1}) - (8 + 2^d)) + 0.25 \cdot 3^c(9 - 1) \\
+ 1.5(2 + 2^{d-1} - 8 - 2^d) - 2(1 + (d-1) - (3 + d)) - 1.5 \cdot 2 = \\
&= 7.5 \cdot 3^c 2^{d-1} - 13 \cdot 3^c - 1.5 \cdot 2^{d-1} - 6 \\
&\geq 15 \cdot 3^c - 13 \cdot 3^c - 1.5 \cdot 4 - 6 \\
&= 2 \cdot 3^c - 12
\end{aligned}$$

Since there are at most three 2's and exactly one 1 on each side of the sequence, $k \geq 18$ requires $c \geq 2$, which renders the increment strictly positive. □

We now know that $1 \leq b, d \leq 2$ in a maximizing sequence. But, at this point, if $k \bmod 3 = 0$, this only allows $b = d = 1$; if $k \bmod 3 = 1$, this only allows $b = d = 2$; and if $k \bmod 3 = 2$ this only allows $b = 2$ and $d = 1$ or vice versa — and since the formula for S_1^x is symmetric with respect to reversing the sequence, in all three cases, this specifies the exact sequence maximizing S_1^x, and hence the expected stretch.

For $k \geq 18$, Theorem 3 follows from substituting each of the three possibilities into equation 2.7:

$$\begin{aligned}
S(k) &\leq 2 \max \begin{cases} (9 \cdot 4 - 6 + 0.25)3^{(k-6)/3} - 1.5\frac{k-6}{3} + 6 - 4 - 6.25 & (k \bmod 3 = 0) \\ (9 \cdot 16 - 12 + 0.25)3^{(k-10)/3} - 1.5\frac{k-10}{3} + 12 - 8 - 6.25 & (k \bmod 3 = 1) \\ (9 \cdot 8 - 9 + 0.25)3^{(k-8)/3} - 1.5\frac{k-8}{3} + 9 - 6 - 6.25 & (k \bmod 3 = 2) \end{cases} \\
&< 2 \cdot 3^{k/3} \max \begin{cases} 121/36 & (k \bmod 3 = 0) \\ 529/(108\sqrt[3]{3}) & (k \bmod 3 = 1) \\ 253/(36\sqrt[3]{9}) & (k \bmod 3 = 2) \end{cases} \\
&= \frac{529}{54\sqrt[3]{3}} \cdot 3^{k/3}
\end{aligned}$$

k	$\max S(k)$	$\left\lfloor \frac{529}{54\sqrt[3]{3}} 3^{k/3} \right\rfloor$	Maximum stretch at:
1	2	9	1
2	6	14	1 1
3	12	20	1 1 1
4	20	29	1 1 1 1
5	32	42	1 1 2 1
6	52	61	1 2 2 1
7	76	88	1 1 2 2 1
8	120	127	1 2 2 2 1
9	170	183	1 2 3 2 1
10	260	264	1 2 2 2 2 1
11	370	381	1 2 2 3 2 1
12	544	550	1 2 2 2 2 2 1
13	786	793	1 2 2 3 2 2 1
14	1126	1144	1 2 2 3 3 2 1
15	1622	1650	1 2 2 2 3 2 2 1
16	2370	2380	1 2 2 3 3 2 2 1
17	3400	3433	1 2 2 3 3 3 2 1

Table 2.2: Optimal sequences for $1 \leq k \leq 17$, obtained by brute-force search.

For $1 \leq k \leq 17$, brute force search among sequences adding up to k and obeying Lemma 2.4.6 shows that the sequences in Table 2.2 maximize stretch, which obeys the bound of Theorem 3, as well, concluding the proof of the latter.

Step 4: Tight lower bound.

Observation 2.4.14. *Assuming that there can be multiple (a, b) edges from a particular node a to a particular node b, the line graph L* itself *is then a valid "tree and false positives" configuration, which means the sequence defined by Lemma 2.4.13 corresponds to a concrete, valid example that makes the above bound exactly tight for $k \mod 3 = 1$.*

Observation 2.4.15. *If multiple edges between the same pair of nodes are not allowed, we expect that the worst-case stretch will be somewhat better, but can still be exponential under worst-case false positive placements. An example of such an arrangement is in Figure 2.6.*

□

Though the expected stretch is exponential, this is counterbalanced by the exponentially low probability of k false positives. Tuning the Bloom filter parameters to optimize memory usage will allow us to bound F, yielding at least a superlinear tail bound on the probability of large stretches, assuming f values are comparable to $1/2m$ or smaller, yielding $F = O(1)$:

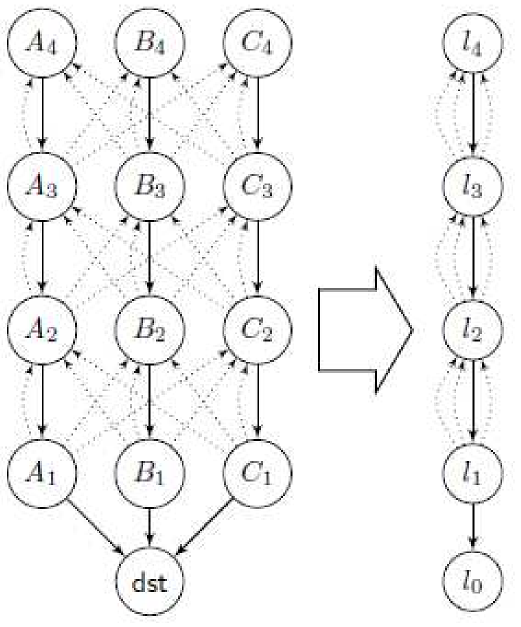


Figure 2.6: Exponential stretch with k false positives and no multiple edges: an $i+1$-row network like the one on the left has $9i$ false positives, with worst-case stretch equivalent to the line graph on the right: $S(k) = \Theta(3^i) = \Theta(3^{k/9})$

Theorem 2.4.16. *For any $z \geq 60.3 \cdot 7.3221^F$, the probability of stretch exceeding z is bounded by $2/z$. Asymptotically, the tail will decay as $\tilde{O}(1/z^{1.54})$, to within polylog factors.*

Proof. Assume the worst-case arrangement of false positive locations. Consider packets starting at the source with the worst expected stretch to the destination. With k false positives, by the Markov bound, the stretch will be at most $2S(k)$ with probability at least $1/2$. After every $2S(k)$ steps, any such packet not at the destination can be "reset" to the worst possible starting location without shortening its current expected arrival time. Thus, for any integer α, the probability of the stretch being more than $2\alpha S(k)$ is at most $1/2^\alpha$.

We can bound the overall stretch by setting k, with foresight, to the solution of $k3^{k/3} = \frac{z}{\rho(2-1/e)}$ (here and below, e is Euler's constant), which is a unique positive value, by monotonicity[4].

[4]We forgo the details of dealing with rounding and integrality.

This value of k will allow us to productively apply the following union bound:

$$\Pr[\text{stretch} > z] \leq \Pr[\text{stretch} > z | \leq k \text{ false pos}] + \tag{2.8}$$

$$+ \Pr[> k \text{ false pos}] \tag{2.9}$$

The above iterated Markov bound covers the first term, which is then bounded by $1/2^{z/2S(k)} = 1/2^{k(1-1/2e)}$.

For the second term, let $F = \sum f(h)$ be the expected number of false positives anywhere in the system for any fixed destination, i.e. the sum of the probabilities of false positives along each possible edge. If $k > 2eF$ (where e is Euler's constant), we can use the fact that the hashes for each Bloom filter are chosen independently and apply the following form of the Chernoff bound [32]: $\Pr[X > (1+\delta)\,\mathrm{E}[X]] < 2^{-\delta\,\mathrm{E}[X]}$, for any $\delta > 2e-1$. With X as the random variable counting the false positives for our destination, we set $\delta = k/F - 1$ yielding $\Pr[X > k] < 2^{-(k/F-1)F} = 2^{-k+F} < 1/2^{k(1-1/2e)}$, bounding the overall probability by $2/(2^{1-1/2e})^k \leq 2/1.76^k$.

To satisfy the $k > 2eF$ requirement of the Chernoff bound, we need $z = (2-1/e)\rho k 3^{k/3} \geq 60.3 \cdot 7.3221^F$. The same constraint, since $F > 0$, guarantees $z \geq 29$, ensuring that $2/1.76^k \leq 2/((2-1/e)\rho k 3^{k/3}) = 2/z$.

With $z = 3^{k/3+O(log(k))}$, the tail asymptotically decays as $\tilde{O}(1/z^{\log_{\sqrt[3]{3}} 1.76})$, to within polylog factors, yielding the second part of the theorem. □

This bound characterizes the worst-case configuration the network may end up in after any particular control-plane event. As a description of the *typical* behavior, on the other hand, this bound is quite crude. We expect that an *average-case analysis* over false positive locations, corresponding to the typical behavior of BUFFALO, will yield polynomial expected stretch for fixed k: the exponential worst-case behavior relies on all the false-positives carefully "conspiring" to point away from the destination, and randomly placed false positives, as with real Bloom filters, will make the random walk behave *similarly* to a random walk on an undirected graph, producing polynomial hitting times. This will allow $z = poly(k)$ and hence an *exponentially* decaying stretch distribution.

While our scheme works with any underlying network structure, it works particularly well with a tree topology. Tree topologies are common in the edges of enterprise and data center networks.

A logical tree is usually constructed with the spanning tree protocols in today's Ethernet. Roughly speaking, in a tree, a lot of distinct false positives are needed at each distance from the destination in order to keep "pushing" the packet away from the destination.

Claim 4. *If the underlying network is a tree, with no multiple links between any one pair of routers, the expected stretch with k false positives, even if they are adversarially placed, is at most* $2(k-1)^2$.

Proof. Consider any configuration of the shortest path tree and the extra false positive edges. Since there are no multiedges, each edge of the shortest path tree has 1 or 0 antiparallel false positive edges. Contracting all the tree edges without a corresponding false positive edge will not decrease the expected stretch, by an argument similar to Observation 2.4.4: Any such edge renders the subtree behind it unreachable after it's traversed. Shrinking this edge only adds the possibility of an extra detour back into the subtree, which would have to pass through this node again before proceeding — at that point, the expected remaining time until destination will be the same as when this node was first visited in the original graph.

The resulting graph is effectively undirected, with each edge having a corresponding antiparallel edge. A random walk is thus identical to a random walk on an undirected tree, shown in, *e.g.*, Sec. 5.3 of [31] to have the expected hitting time of at most $2(k-1)^2$. □

We believe that similar results should apply when we allow heterogeneous latencies, especially when the per-link latencies are within a small constant factor of each other, as is likely in many geographically-local networks.

2.4.3 Stretch in Realistic Networks

We evaluate the stretch in three representative topologies: Campus is the campus network of a large (roughly 40,000 students) university, consisting of 1600 switches [36]. AS 1239 is a large ISP network with 315 routers [37]. (The routers are viewed as switches in our simulation.) We also constructed a model topology similar to the one used in [5], which represents a typical data center network composed of four full-meshed core routers each of which is connected to a mesh of twenty one aggregation switches. This roughly characterizes a commonly-used topology in data centers [38].

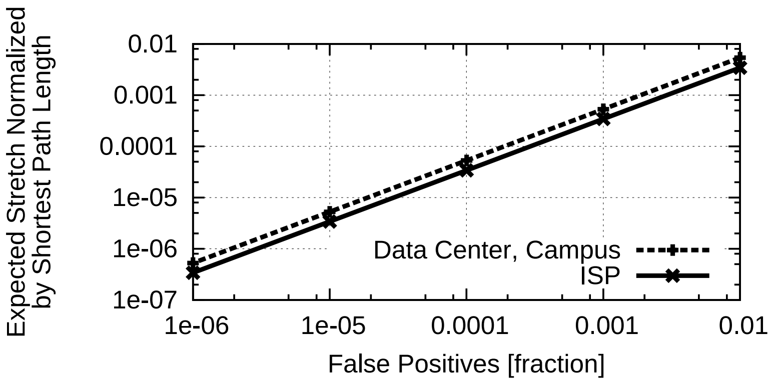


Figure 2.7: Expected stretch

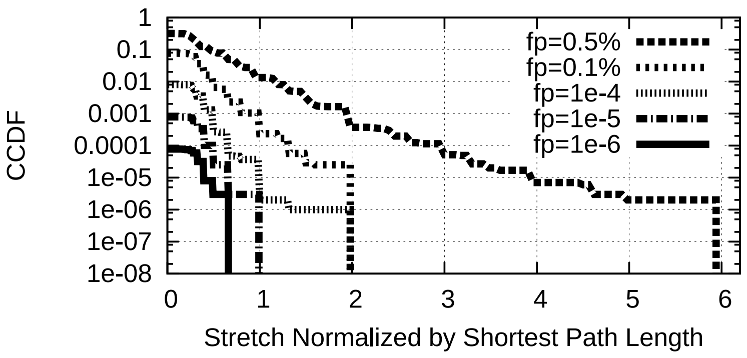


Figure 2.8: Stretch with real traces in campus network

In the three topologies, we first analyze the expected stretch given the false-positive rate. We then use simulation to study the stretch with real packet traces.

Analysis of expected stretch: We pick the false-positive rate of Bloom filters and calculate the expected stretch for each pair of source and destination in the network by analyzing all the cases with different numbers and locations of false positives and all the possible random selections. The expected stretch is normalized by the length of the shortest path. We take the average stretch among all source-destination pairs. We can see that the expected stretch increases linearly with the increase of the false-positive rate. This is because the expected stretch is dominated by the one false-positive case. Since we provide a constant stretch bound for the one false-positive case in BUFFALO, the expected stretch is very small. Even with a false-positive rate of 1%, the expected stretch is only 0.5% of the length of the shortest path (Figure 3.16).

Simulation on stretch distribution: We also study the stretch of BUFFALO with packet traces collected from the Lawrence Berkeley National Lab campus network by Pang et. al. [39]. There are four sets of traces, each collected over a period of 10 to 60 minutes, containing traffic to and from roughly 9,000 end hosts distributed over 22 different subnets. Since we cannot get the network topology where the trace is collected, we take the same approach in [5] to map the trace to the above campus network while preserving the distribution of source-destination popularity of the original trace. Figure 2.8 shows the distribution of stretch normalized by shortest path length. When the false-positive rate is 0.01%, 99% of the packets do not have any stretch and 0.01% of the packets have a stretch that is twice as long as the length of the shortest path. Even when the false-positive rate is 0.5%, only 0.0001% of the packets have a stretch of 6 times of the length of the shortest path. Note that in an enterprise or data center, the propagation delays are small, so the stretch caused by false positives is tolerable.

2.5 Optimizing Memory Usage

In this section, we consider a switch with M-bit fast memory (*i.e.*, SRAM) and a fixed routing table. We formulate the problem of minimizing the overall false-positive rate through tuning the sizes in the Bloom filters. This optimization is done by a Bloom filter manager implemented in a BUFFALO switch. We then show numerical results of false positives with various sizes of memory and forwarding tables.

2.5.1 Optimizing Bloom-Filter Sizes

Our goal is to minimize the *overall false-positive rate*. If any one of the T Bloom filters has a false positive, an address will hit in multiple Bloom filters. In this case, we send the packets through a slightly longer path as described in Section 2.4. To reduce the stretch, we must minimize the false positives in each switch. We define the overall false-positive rate for a switch as the probability that any one of the T Bloom filters has a false positive. As above, let $f(h)$ denote the false-positive rate of Bloom filter $BF(h)$. Since Bloom filters for different next hops store independent sets of addresses, and thus are independent of each other, the overall false-positive rate of T Bloom filters is

$$F = 1 - \prod_{h=1}^{T}(1 - f(h)) \approx \sum_{h=1}^{T} f(h)$$
$$(when\ f(h) \ll 1/T, \forall h = 1..T)$$

Optimizing the sum approximation for F also directly optimizes the applicability threshold for Theorem 2.4.16, expressed in terms of the sum as such.

Since there are different numbers of addresses per next hop, we should use different sizes for the Bloom filters according to the number of addresses stored in them, in order to minimize the overall false-positive rate with the M-bit fast memory.

In addition to constraining the fast memory size, we should also avoid overloading the CPU. We bound the packet lookup time, which consists of hash computation time and memory access time. To reduce the computational overhead of address lookup, we apply the same group of hash functions to all T Bloom filters. Since we use the same hash functions for all the Bloom filters, we need to check the same positions in all the Bloom filters for an address lookup. Therefore, we put the same positions of the Bloom filters in one memory unit (*e.g.*, a byte or a word), so that they can be accessed by one memory access. In this scheme, the packet lookup time is determined by the maximum number of hash functions in T Bloom filters ($k_{max} = \max_{h=1}^{T}(k(h))$, where $k(h)$ denotes the number of hash functions used in $BF(h)$). Let u_{hash} denote the number of hash functions that can be calculated in a second. We need k_{max}/u_{hash} time for hash computation. Let t_{fmem} denote the access time on small, fast memory. Assume there are b bits in a memory unit which can be read by one memory access. We need $\lceil (Tk_{max}/b)t_{fmem} \rceil$ memory access time. Since both hash computation and memory access time are linear in the maximum number of hash functions k_{max}, we only need to bound k_{max} in order to bound the packet lookup time.[5]

We minimize the lookup time for each packet by choosing $m(h)$ (the number of bits in $BF(h)$) and $k(h)$ (*i.e.*, the number of hash functions used in $BF(h)$), with the constraint that Bloom filters must not take more space than the size of the fast memory and must have a bounded number of hash functions.

[5]In switch hardware, the 4-8 hash functions can be calculated in parallel. We also assert that fabrication of 6 to 8 read ports for an on-chip Random Access Memory is attainable with today's embedded memory technology [40]. The cache line size on a Intel Xeon machine is about 32 bytes to 64 bytes, which is enough to put all the positions of T Bloom filters in one cache line.

Let $n(h)$ denote the number of addresses in Bloom filter $BF(h)$. The optimization problem is formulated as:

$$\begin{aligned}
Min \quad & F = \sum_{h=1}^{T} f(h) & (2.10)\\
s.t. \quad & f(h) = (1 - e^{-k(h)n(h)/m(h)})^{k(h)} & (2.11)\\
& \sum_{h=1}^{T} m(h) = M & (2.12)\\
& k(h) \leq k_{max}, \forall h \in [1..T] & (2.13)\\
given \quad & T, M, k_{max}, \textit{ and } n(h)(\forall h \in [1..H]) &
\end{aligned}$$

Equation (2.10) is the overall false-positive rate we need to minimize. Equation (2.11) shows the false-positive rate for a standard Bloom filter. Equation (2.12) is the size constraint of the fast memory. Equation (2.13) is the bound on the number of hash functions. We have proved that this problem is a convex optimization problem.[6] Thus there exists an optimal solution for this problem, which can be found by the IPOPT [41] (Interior Point OPTimizer) solver. Most of our experiments converge within 30 iterations, which take less than 50 ms. Note that the optimization is executed only when the forwarding table has significant changes such as a severe link failure leading to lots of routing changes. The optimization can also be executed in the background without affecting the packet forwarding.

2.5.2 Analytical Results of False Positives

We study in a switch the effect of forwarding table size, number of next hops, the amount of fast memory, and number of hash functions on the overall false-positive rate.[7] We choose to analyze the false positives with synthetic data to study various sizes of forwarding tables and different memory and CPU settings. We have also tested BUFFALO with real packet traces and forwarding tables. The results are similar to the analytical results and thus omitted in the thesis. We studied a forwarding table with 20K to 2000K entries (denoted by N), where the number of next hops (T) varies from 10 to 200. The maximum number of hash functions in the Bloom filters (k_{max}) varies

[6]The proof is omitted due to lack of space.

[7]In Section 2.4, the false-positive rate is defined for each Bloom filter. Here the overall false-positive rate is defined for the switch because different Bloom filters have different false-positive rates. The overall false-positive rate can be one or two orders of magnitude larger than individual Bloom-filter false-positive rate.

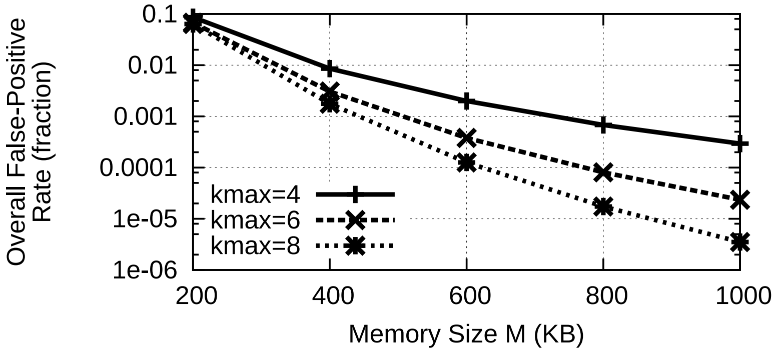


Figure 2.9: Effect of memory size ($T = 10, N = 200K$)

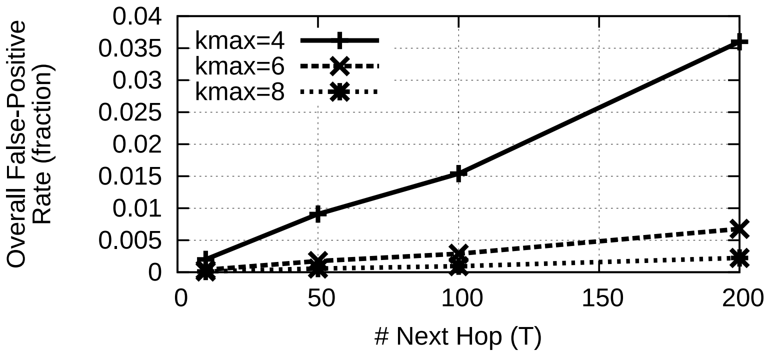


Figure 2.10: Effect of number of next hops ($M = 600KB, N = 200K$)

from 4 to 8. Since next hops have different popularity, Pareto distribution is used to generate the number of addresses for each next hop. We have the following observations:

(1) A small increase in memory size can reduce the overall false-positive rate significantly. As shown in Figure 2.9, to reach the overall false-positive rate of 0.1%, we need 600 KB fast memory and 4-8 hash functions to store a FIB with 200K entries and 10 next hops. If we have 1 MB fast memory, the false-positive rate can be reduced to the order of 10^{-6}.

***(2) The overall false-positive rate increases almost linearly with the increase of** T**.* With the increase of T and thus more Bloom filters, we will have larger overall false-positive rate. However, as shown in Figure 2.10, even for a switch that has 200 next hops and a 200K-entry forwarding table, we can still reach a false-positive rate of 1% with 600KB fast memory ($k_{max} = 6$). This is because if we fix the total number of entries N, with the increase of T, the number of addresses for each next hop drops correspondingly.

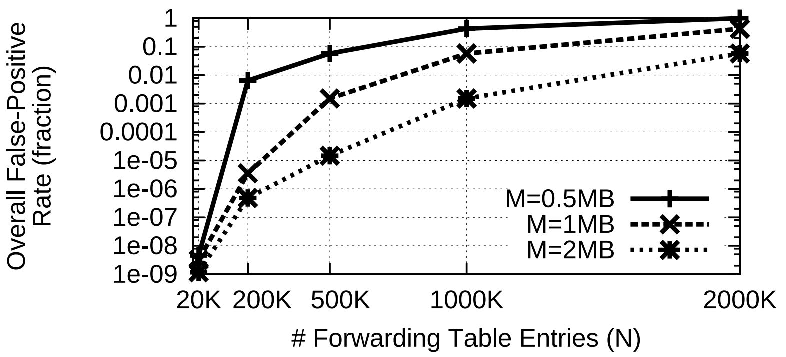


Figure 2.11: Effect of number of entries ($T = 10, k_{max} = 8$)

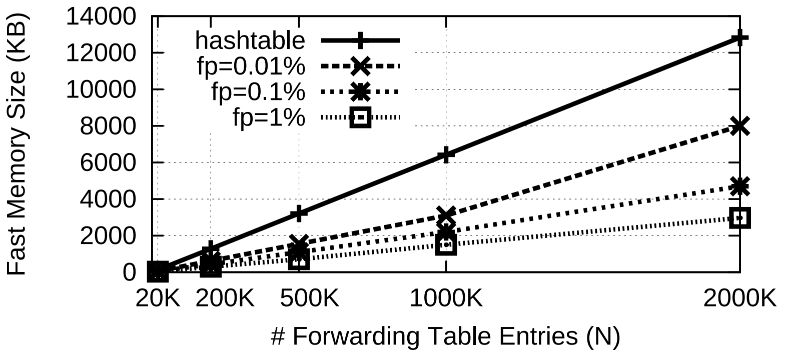


Figure 2.12: Comparison of hash table and Bloom filters ($T = 10, k_{max} = 8$)

(3) BUFFALO switch with fixed memory size scales well with the growth of forwarding table size. For example, as shown in Figure 2.11, if a switch has a 1MB fast memory, as the forwarding table grows from 20K to 1000K entries, the false-positive rate grows from 10^{-9} to 5%. Since packets experiencing false positives are handled in fast memory, BUFFALO scales well with the growth of forwarding table size.

(4) BUFFALO reduces fast memory requirement by at least 65% compared with hash tables at the expense of a false-positive rate of 0.1%. We assume a perfect hash table that has no collision. Each entry needs to store the MAC address (48 bits) and an index of the next hop ($\log(T)$ bits). Therefore the size of a hash table for an N-entry forwarding table is $(\log(T) + 48)N$ bits. Figure 2.12 shows that BUFFALO can reduce fast memory requirements by 65% compared with hash tables for the same number of FIB entries at the expense of a false-positive rate of 0.1%. With the increase of forwarding table size, BUFFALO can save more memory. However in practice, handling collisions in hash tables requires much more memory space and affects the throughput. In contrast, BUFFALO can handle false positives without any memory overhead. Moreover, the packet forwarding performance of BUFFALO is independent of the workload.

2.6 Handling Routing Changes

In this section, we first describe the use of CBFs in slow memory to keep track of changes in the forwarding table. We then discuss how to update the BF from the CBF and how to change the size of the BF without reconstructing the CBF.

2.6.1 Update BF Based on CBF

In a switch, the control plane maintains the RIB (Routing Information Base) and updates the FIB (Forwarding Information Base) in the data plane. When FIB updates are received, the Bloom filters should be updated accordingly. We use CBFs in slow memory to assist the update of Bloom filters. We implement a group of T CBFs, each containing the addresses associated with one next hop. To add a new route of address *addr* with next hop h, we will insert *addr* to $CBF(h)$. Similarly, to delete a route, we remove *addr* in $CBF(h)$. The insertion and deletion operations on the CBF are described in Section 2.2. Since CBFs are maintained in slow memory, we set the sizes of CBFs large enough, so that even when the number of addresses in one CBF increases significantly due to routing changes, the false-positive rates on CBFs are kept low.

After the $CBF(h)$ is updated, we update the corresponding $BF(h)$ based on the new $CBF(h)$. We only need to modify a few BFs that are affected by the routing changes without interrupting the packet forwarding with the rest BFs. To minimize the interruption of packet forwarding with the modified BFs, we implement a pointer for each BF in SRAM. We first generate new snapshots of BFs with CBFs and then change the BF pointers to the new snapshots. The extra fast memory for snapshots is small because we only need to modify a few BFs at a time.

If the CBF and BF have the same number of positions, we can easily update the BF by checking if each position in the CBF is 0 or not. The update from CBF to BF becomes more challenging when we have to dynamically adjust the size of the BF to reduce the overall false-positive rate.

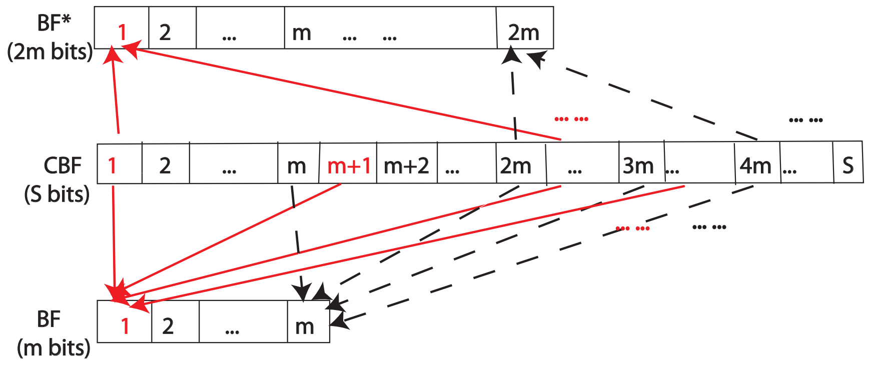


Figure 2.13: Adjust BF size from m to $2m$ based on CBF

2.6.2 Adjust BF Size Without Reconstructing CBF

When the forwarding table changes over time, the number of addresses in the BF changes, so the size of the BF and the number of hash functions to achieve the optimal false-positive rate also change. We leverage the nice property that to halve the size of a Bloom filter, we just OR the first and second halves together [20]. In general, the same trick applies to reducing the size of a Bloom filter by a constant c. This works well in reducing the BF size when the number of addresses in the BF decreases. However, when the number of addresses increases, it is hard to expand the BF.

Fortunately, we maintain a large, fixed size CBF in the slow memory. we can dynamically *increase or decrease* the size of the BF by mapping multiple positions in the CBF to one position in the BF. For example in Figure 2.13, we can easily expand the BF with size m to BF^* with size $2m$ by collapsing the same CBF.

To minimize the overall false-positive rate under routing changes, we monitor the number of addresses in each CBF, and periodically reconstruct BFs to be of the optimal sizes and number of hash functions. Since resizing a BF based on a CBF requires the BF and CBF to use the same number of hash functions. We need to adjust the number of hash functions in the CBF before resizing the BF. The procedure of reconstructing a BF with an optimal size from the corresponding CBF is described in three steps:

Step 1: Calculate the optimal BF size and the number of hash functions. Solving the optimization problem in Section 2.5, we first get the optimal size of each BF and denote it by m^*. Then we round m^* to m', which is a factor of S,

$$m' = S/c, \textit{where } c = \lceil S/m^* \rceil.$$

Finally we calculate the optimal number of hash functions to minimize false positives with size m' and the number of addresses n in the BF, which is $m' \ln 2/n$ based on standard Bloom filter analysis [20]. We also need to bound the number of hash functions by k_{max}. Thus the number of hash functions is $k' = \min(k_{max}, m' \ln 2/n)$.

Step 2: If $k \neq k'$, change the number of hash functions in the CBF from k to k'. The number of hash functions does not always change because routing changes are sometimes not significant and we have the k_{max} bound. When we must change k, there are two ways with either more computation or more space: (i) If $k' > k$, we obtain all the addresses currently in the forwarding table from the control plane, calculate the hash values with the $k' - k$ new hash functions on all the addresses currently in the BF, and update the CBF by incrementing the counters in corresponding positions. If $k' < k$, we also calculate $k - k'$ hash values, and decrementing the corresponding counters. (ii) Instead of doing the calculation on the fly, we can pre-calculate the values of these hash functions with all the elements and store them in the slow memory.

Step 3: Construct the BF of size $m' = S/c$ based on the CBF of size S. As shown in Figure 2.13, the value of the BF at position x ($x \in [1..m']$) is updated by c positions in CBF x, $2x$, ... cx. If all the counters in the c positions of CBF are 0, we set the position x in BF to 0; otherwise, we set it to 1. During routing changes, the BFs can be updated based on CBFs in the same way.

2.7 Implementation and Evaluation

To verify the performance and practicality of our mechanism through a real deployment, we built a prototype *BuffaloSwitch* in kernel-level Click [23]. The overall structure of our implementation is shown in Figure 3.7. *BuffaloSwitch* consists of four modules:

Counting Bloom filters: The counting Bloom filter module is used to receive routing changes from the control plane and increment/decrement the counters in the related CBFs correspondingly.

Bloom filters: The Bloom filter module maintains one Bloom filter for each next hop. It also performs the packet lookup by hash calculation and checking all the Bloom filters. When it finds out the multiple next hop candidates due to false positives, it will call the false positive handler.

Bloom filter manager: The Bloom filter manager monitors the number of addresses in each BF. If the number of addresses in one BF changes significantly (above threshold TH), we recalculate the optimal size of the BF and reconstruct it based on the CBF.

False positive handler: The false positive handler module is responsible for selecting a next hop for the packets that experience false positives.

To evaluate our prototype, we need a forwarding table and real packet traces. We map the Lawrence Berkeley National Lab Campus network traces [39] to the campus network topology [36] as described in Section 2.4.3. We then calculate shortest paths in the network and construct the forwarding table accordingly. The forwarding table consists of 200K entries.

We run *BuffaloSwitch* on a 3.0 GHz 64-bit Intel Xeon machine with a 8 KB L1 and 2 MB L2 data cache. The main memory is a 2 GB 400 Mhz DDR2 RAM. We take the fast memory size M as 1.5MB to make sure Bloom filters fit in the L2 cache. The Bloom filter manager optimizes sizes of Bloom filters given the forwarding table and M. To avoid the potential bottleneck at the Ethernet interfaces, we run the Click packet generator on the same machine with *BuffaloSwitch*. We send the packet with constant rate and measure the peak forwarding rate of *BuffaloSwitch*. The packet size is set as 64 bytes, which is the minimum Ethernet packet size, so that the packet payload does not pollute the cache much. For comparison, we also run *EtherSwitch* — a standard Click element which performs Ethernet packet forwarding using hash tables.

Our experiment shows that *BuffaloSwitch* achieves a peak forwarding rate of 365 Kpps, 10% faster than *EtherSwitch* which has 330 Kpps peak forwarding rate. This is because all the Bloom filters in *BuffaloSwitch* fit in the L2 cache, but the hash table in *EtherSwitch* does not and thus takes longer time to access memory. The forwarding rate with *BuffaloSwitch* can be further improved by parallelizing the hash calculations on multiple cores.

To measure the performance of *BuffaloSwitch* under routing changes, we generate a group of routing updates which randomly change FIB entries and replay these updates. It takes 10.7 μsec for *BuffaloSwitch* to update the Bloom filters for one route change. Under significant routing changes, it takes an additional 0.47 seconds to adjust the Bloom filter sizes based on counting

Bloom filters. This is because CBFs are very large and takes longer time to scan through them. However, it is much faster than recalculating hash functions for all FIB entries to reconstruct Bloom filters.

2.8 Extensions to BUFFALO

In this section we discuss the extensions of BUFFALO to support ECMP, VLAN, broadcast and multicast packets, and backup routes.

Supporting ECMP: In shortest-path routing protocols like OSPF and IS-IS, ECMP (Equal-Cost Multi-Path) is used to split traffic among shortest paths with equal cost. When a destination address has multiple shortest paths, the switch inserts the destination address into the Bloom filter of each of the next hops. Since packets with this address match multiple next hops, the BUFFALO false-positive handler will randomly choose one next hop from them, achieving even splitting among these equal-cost multiple paths.

Supporting virtual LANs: VLAN is used in Ethernet to allow administrators to group multiple hosts into a single broadcast domain. A switch port can be configured with one or more VLANs. We can no longer use just a single Bloom filter for each port because due to false positives a packet in VLAN A may be sent to a switch which does not know how to reach VLAN A and thus get dropped. To support VLANs in BUFFALO, we use one Bloom filter for each (VLAN, next hop) pair. For a packet lookup, we simply check those Bloom filters that have the same VLAN as the packet. However, this does not scale well with a large number of VLANs in the network. For future architectures that have simpler network configuration and management methods rather than VLAN, we do not have this problem.

Broadcast and multicast: In this chapter, we have focused on packet forwarding for unicast traffic. To support Ethernet broadcast, the switch can identify the broadcast MAC address and forward broadcast packets directly without checking the Bloom filters. Supporting multicast in the layer-2 network is more complex. One way is to broadcast all the multicast packets and let the NIC on the hosts decide whether to accept or drop the packets. Another way is to allow switches to check whether the destination IP address of the packet is a multicast-group packet, and leverage IP multicast solutions such as storing the multicast forwarding information in packets [42, 43].

Fast failover to backup routes: When a significant failure happens such as one of the switch's own links fail, many routes change in a short time. In order to quickly recover from significant failures, we provide an optional optimization of BUFFALO. The control plane calculates backup routes for every link/node failure case in advance, and notifies the data plane about the failure event. In addition to the original routes stored in $CBF(h)$ ($h \in [1..T]$), we pre-calculate backup counting Bloom filters $CBF(h_1, h_2)$ (for all $h_1 \in [1..T], h_2 \in [1..T]$), which denotes the set of addresses that are originally forwarded to next hop h_1, but if h_1 is not accessible, they should be forwarded to next hop h_2. When the failure happens and thus h_1 is not accessible, we simply need to update the Bloom filters based on the original Bloom filters and backup counting Bloom filters. For example, we update $BF(h_2)$ based on the old $BF(h_2)$ and $CBF(h_1, h_2)$. This is fast because merging two Bloom filters is just OR operations.

2.9 Related Work

Bloom filters have been used for IP packet forwarding, and particularly the longest-prefix match operation [22]. The authors use Bloom filters to determine the *length* of the longest matching prefix for an address and then perform a direct lookup in a large hash table in slow memory. The authors in [44] design a *d-left scheme* using d hash functions for IP lookups. To perform an IP lookup, they still need to access the slow memory at least d times. The paper [45] stores Bloom filter in the fast memory, and stores the values in a linked structure in the slow memory such that the value can be accessed via one access on the slow memory most of the times. Different from these works, we focus on flat addresses and perform the *entire* lookup in fast memory at the expense of a few false positives. We also propose a simple scheme that handles false positives within fast memory, and proves its reachability and stretch bound.

Bloom filters have also been used in resource routing [20, 21], which applies Bloom filters to probabilistic algorithms for locating resources. Our "one Bloom filter per next hop" scheme is similar to their general idea of using one Bloom filter to store the list of resources that can be accessed through each neighboring node. To keep up with link speed in packet forwarding with a strict fast memory size constraint, we *dynamically* tune the *optimal* size and the number of hash functions of Bloom filters by keeping large fixed-size counting Bloom filters in slow memory. We also handle false positives without any memory overhead. BUFFALO is also similar to *Bloomier filters* [46] in that we both use a group of Bloom filters, one for each value of a function that maps the key to the value. However, Bloomier filters only work for a *static* element set.

Bloom filters are also been used for multicast forwarding. LIPSIN [43] uses Bloom filters to encode the multicast forwarding information in packets. False positives in Bloom filters may cause loops in its design. LIPSIN caches packets that may experience loops and send the packets to a different link when a loop is detected. However, they do not show how well they can prevent loops and the cache size they need. In contrast, our loop prevention mechanism is simple and effective, and does not have any memory overhead.

To handle routing changes, the paper [45] store counting Bloom filters (CBFs) in fast memory, which uses more memory space than the Bloom filters (BFs). Both our paper and the paper [22] and leverage the fact that routing changes happen on a much longer time scale than address lookup, and thus store only the BF in fast memory, and use the CBF in slow memory to handle routing changes. The idea of maintaining both the CBF and BF is similar to the work in [30], which uses BFs for sharing caches among Web proxies. Since cache contents change frequently, the authors suggest that caches use a CBF to track their own cache contents, and broadcast the corresponding BF to the other proxies. The CBF is used to avoid the cost of reconstructing the BF from scratch when an update is sent; the BF rather than the CBF is sent to the other proxies to reduce the size of broadcast messages. Different from previous work, we dynamically adjust the size of the BF without reconstructing the corresponding CBF, which may be useful for other Bloom filter applications.

Our idea of using one Bloom filter per port is similar to SPSwitch [47] which forward packets on flat identifiers in content-centric networks. Our workshop paper [48] applies Bloom filters for enterprise edge routers by leveraging the fact that edge routers typically have a small number of next hops. However, it does not deal with loops caused by false positives. The paper uses one Bloom filter for each (next hop, prefix length) pair and discusses its effect on false positives. It also proposes the idea of using CBF to assist the BF update and resizing. We consider flat address lookup in SPAF networks in this paper, and thus eliminate the effect of various prefix lengths. We also propose a mechanism to handle false positives in the network without extra memory. We perform extensive analysis, simulation, and prototype implementation to evaluate our scheme.

2.10 Summary

With recent advances in improving control plane scalability, it is possible now to build large layer-2 networks. The scalability problem in the data plane becomes challenging with increasing forwarding table sizes and link speed. Leveraging flat addresses and shortest path routing in SPAF networks, we proposed BUFFALO, a practical switch design based on Bloom filters. BUFFALO performs the entire packet forwarding in small, fast memory including those packets experiencing false positives. BUFFALO gracefully degrades under higher memory loads by gradually increasing stretch rather than crashing or resorting to excessive flooding. Our analysis, simulation and prototype demonstrate that BUFFALO works well in reducing memory cost and improving the scalability of packet forwarding in enterprise and data center networks.

Chapter 3

DIFANE: Scaling Flexible Policy Support on Switches

This chapter focuses on the scalability challenges of supporting flexible policies. Ideally, enterprise administrators could specify fine-grain policies that drive how the underlying switches forward, drop, and measure traffic. However, existing techniques for flow-based networking rely too heavily on centralized controller software that installs rules reactively, based on the first packet of each flow. In this chapter, we propose DIFANE, a scalable and efficient solution for enforcing flexible policies at switches.

DIFANE keeps all traffic in the data plane by selectively directing packets through intermediate switches that store the necessary rules. DIFANE relegates the controller to the simpler task of partitioning these rules over the switches. DIFANE can be readily implemented with commodity switch hardware, since all data-plane functions can be expressed in terms of wildcard rules that perform simple actions on matching packets. Experiments with our prototype on Click-based OpenFlow switches show that DIFANE scales to larger networks with richer policies.

3.1 Introduction

The emergence of flow-based switches [14, 15] has enabled enterprise networks that support flexible policies. These switches perform simple actions, such as dropping or forwarding packets, based on rules that match on bits in the packet header. Installing all of the rules in advance is not attractive, because the rules change over time (due to policy changes and host mobility) and the switches have relatively limited high-speed memory (such as TCAMs). Instead, current solutions rely on directing the first packet of each "microflow" to a centralized controller that reactively installs the appropriate rules in the switches [13, 10]. In this thesis, we argue that the *switches themselves* should collectively perform this function, both to avoid a bottleneck at the controller and to keep all traffic in the data plane for better performance and scalability.

3.1.1 DIFANE: Doing It Fast ANd Easy

Our key challenge, then, is to determine the appropriate "division of labor" between the controller and the underlying switches, to support high-level policies in a scalable way. Previous work has demonstrated that a logically-centralized controller can track changes in user locations/addresses and compute rules the switches can apply to enforce a high-level policy [10, 11, 12]. For example, an access-control policy may deny the engineering group access to the human-resources database, leading to low-level rules based on the MAC or IP addresses of the current members of the engineering team, the IP addresses of the HR servers, and the TCP port number of the database service. Similar policies could direct packets on customized paths, or collect detailed traffic statistics. The controller can generate the appropriate switch rules simply by substituting high-level names with network addresses. The policies are represented with 30K - 8M rules in the four different networks we studied. This separation of concerns between rules (in the switches) and policies (in the controller) is the basis of several promising new approaches to network management [13, 49, 50, 51, 52].

While we agree the controller should *generate* the rules, we do not think the controller should (or needs to) be involved in the real-time handling of data packets. Our DIFANE (DIstributed Flow Architecture for Networked Enterprises) architecture, illustrated in Figure 3.1, has the following two main ideas:

- The controller **distributes the rules** across (a subset of) the switches, called "authority switches," to scale to large topologies with many rules. The controller runs a partitioning algorithm that divides the rules evenly and minimizes fragmentation of the rules across multiple authority switches.

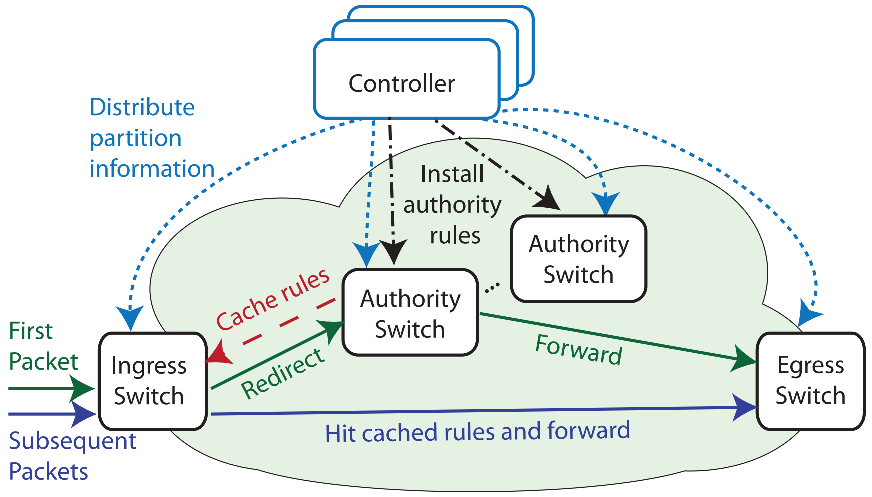


Figure 3.1: DIFANE flow management architecture. (Dashed lines are control messages. Straight lines are data traffic.)

- The switches handle **all packets in the data plane** (*i.e.*, TCAM), diverting packets through authority switches as needed to access the appropriate rules. The "rules" for diverting packets are themselves naturally expressed as TCAM entries.

All data-plane functionality in DIFANE is expressible in terms of wildcard rules with simple actions, exactly the capabilities of commodity flow switches. As such, a DIFANE implementation requires only modifications to the control-plane software of the authority switches, and no data-plane changes in any of the switches. Experiments with our prototype, built on top of the Click-based OpenFlow switch [53], illustrate that distributed rule management in the data plane provides lower delay, higher throughput, and better scalability than directing packets through a separate controller.

Section 3.2 presents our main design decisions, followed by our DIFANE architecture in Section 3.3. Next, Section 3.4 describes how we handle network dynamics, and Section 3.5 presents our algorithms for caching and partitioning wildcard rules. Section 3.6 presents our switch implementation, followed by the performance evaluation in Section 4.5. Section 3.8 describes different deployment scenarios of DIFANE. The chapter concludes in Section 4.9.

3.1.2 Comparison to Related Work

Recent work shows how to support policy-based management using flow switches [14, 15] and centralized controllers [10, 11, 12, 13]. The most closely related work is the Ethane controller that reactively installs flow-level rules based on the first packet of each TCP/UDP flow [13]. The Ethane controller can be duplicated [13] or distributed [54] to improve its performance. In contrast, DIFANE distributes wildcard rules amongst the switches, and handles all data packets in the data plane. Other recent work capitalizes on OpenFlow to rethink network management in enterprises and data centers [49, 50, 51, 52]; these systems could easily run as applications on top of DIFANE.

These research efforts, and ours, depart from traditional enterprise designs that use IP routers to interconnect smaller layer-two subnets, and rely heavily on inflexible mechanisms like VLANs. Today, network operators must configure Virtual LANs (VLANs) to scope broadcast traffic and direct traffic on longer paths through routers that perform access control on IP and TCP/UDP header fields. In addition, an individual MAC address or wall jack is typically associated with just *one* VLAN, making it difficult to support more fine-grained policies that treat different traffic from the same user or office differently.

Other research designs more scalable networks by selectively directing traffic through intermediate nodes to reduce routing-table size [5, 55, 56]. However, hash-based redirection techniques [5, 55], while useful for flat keys like IP or MAC addresses, are not appropriate for look-ups on rules with wildcards in arbitrary bit positions. ViAggre [56] subdivides the IP prefix space, and forces some traffic to always traverse an intermediate node, and does not consider on-demand cache or multi-dimensional, overlapping rules.

3.2 DIFANE Design Decisions

On the surface, the simplest approach to flow-based management is to install all of the low-level rules in the switches in advance. However, preinstalling the rules does not scale well in networks with mobile hosts, since the same rules would need to be installed in multiple locations (*e.g.*, any place a user might plug in his laptop). In addition, the controller would need to update many switches whenever rules change. Even in the absence of mobile devices, a network with many rules might not have enough table space in the switches to store all the rules, particularly as the network grows or its policies become more complex. Instead, the system should install rules on demand [13].

To build a flow-processing system that has high performance and scales to large networks, DIFANE makes four high-level design decisions that reduce the overhead of handling cache misses and allow the system to scale to a large number of hosts, rules, and switches.

3.2.1 Reducing Overhead of Cache Misses

Reactively caching rules in the switches could easily cause problems such as packet delay, larger buffers, and switch complexity when cache misses happen. More importantly, misbehaving hosts could easily trigger excessive cache misses simply by scanning a wide range of addresses or port numbers — overloading TCAM and introducing extra packet-processing overhead. DIFANE handles "miss" packets *efficiently* by keeping them in the data plane and reduces the *number* of "miss" packets by caching wildcard rules.

Process all packets in the data plane: Some flow management architectures direct the first packet (or first packet header) of each microflow to the controller and have the switch buffer the packet awaiting further instructions [13].[1] In a network with many short flows , a controller that handles "miss" packets can easily become a bottleneck. In addition, UDP flows introduce extra overhead, since multiple (potentially large) packets in the same flow may be in flight (and need to visit the controller) at the same time. The switches need a more complex and expensive buffering mechanism, because they must temporarily store the "miss" packets while continuing to serve other traffic, and then retrieve them upon receiving the rule. Instead, DIFANE makes it cheap and easy for switches to forward *all* data packets in the data plane (*i.e.*, hardware), by directing "miss" packets through an intermediate switch. Transferring packets in the data plane through a slightly longer path is much faster than handling packets in the control plane.

Efficient rule caching with wildcards: Caching a separate low-level rule for each TCP or UDP microflow [13], while conceptually simple, has several disadvantages compared to wildcard rules. For example, a wildcard rule that matches on all destinations in the 123.132.8.0/22 subnet would require up to 1024 microflow rules. In addition to consuming more data-plane memory on the

[1]Another solution to handle cache miss is for the switch to encapsulate and forward the entire packet to the controller. This is also problematic because it significantly increases controller load.

switches, fine-grained rules require special handling for more packets (*i.e.*, the first packet of each microflow), leading to longer delays and higher overhead, and more vulnerability to misbehaving hosts. Instead, DIFANE supports wildcard rules, to have fewer rules (and fewer cache "misses") and capitalize on TCAMs in the switches. Caching wildcard rules introduces several interesting technical challenges that we address in our design and implementation.

3.2.2 Scaling to Large Networks and Many Rules

To scale to large networks with richer policies, DIFANE divides the rules across the switches and handles them in a distributed fashion. We also keep consistent topology information among switches by leveraging link-state protocols.

Partition and distribute the flow rules: Replicating the controller seems like a natural way to scale the system and avoid a single point of failure. However, this requires each controller to maintain all the rules, and coordinate with the other replicas to maintain consistency when rules change. (Rules may change relatively often, not only because the policy changes, but also because host mobility triggers changes in the mapping of policies to rules.) Instead, we *partition* the space of rules to reduce the number of rules each component must handle and enable simpler techniques for maintaining consistency. As such, DIFANE has one primary controller (perhaps with backups) that manages policies, computes the corresponding rules, and divides these rules across the switches; each switch handles a portion of the rule space and receives updates only when those rules change. That is, while the switches *reactively* cache rules in response to the data traffic, the DIFANE controller *proactively* partitions the rules across different switches.

Consistent topology information distribution with the link-state protocol: Flow-based management relies on the switches having a way to communicate with the controller and adapt to topology changes. Relying on rules for this communication introduces circularity, where the controller cannot communicate with the switches until the appropriate rules have been installed. Rather than bootstrapping communication by having the switches construct a spanning tree [13], we advocate running a link-state protocol amongst the switches. Link-state routing enables the switches to compute paths and learn about topology changes and host location changes without involving the controller, reducing overhead and also removing the controller from the critical path of failure recovery. In addition, link-state routing scales to large networks, enables switches to direct packets through intermediate nodes, and reacts quickly to switch failure [5]. As such, DIFANE runs a link-state routing protocol amongst the switches, while also supporting flow rules that allow customized forwarding of traffic between end hosts. The controller also participates in link-state routing to reach the switches and learn of network topology changes.[2]

3.3 DIFANE Architecture

The DIFANE architecture consists of a *controller* that generates the rules and allocates them to the *authority switches*, as shown in Figure 3.1. Authority switches can be a subset of existing switches in the network (including ingress/egress switches), or dedicated switches that have larger memory and processing capability.

Upon receiving traffic that does not match the cached rules, the ingress switch encapsulates and redirects the packet to the appropriate authority switch based on the partition information. The authority switch handles the packet in the data plane and sends feedback to the ingress switch to cache the relevant rule(s) locally. Subsequent packets matching the cached rules can be encapsulated and forwarded directly to the egress switch.

In this section, we first discuss how the controller partitions the rules and distributes the authority and partition rules to the switches. Next, we describe how a switch directs packets through the authority switch and caches the necessary rules, using link-state routing to compute the path to the authority switch. Finally, we show that the data-plane functionality of DIFANE can be easily implemented on today's flow-based switches using wildcard rules.

3.3.1 Rule Partition and Allocation

As shown in Figure 3.2, we use the controller to *pre-compute* the low-level rules, generate *partition rules* that describe which low-level rules are stored in which authority switches, and then distribute the partition rules to all the switches. The partition rules are represented by coarse-grained wildcard rules on the switches.

[2]The links between the controller and switches are set with high link-weights so that traffic between switches do not go through the controller.

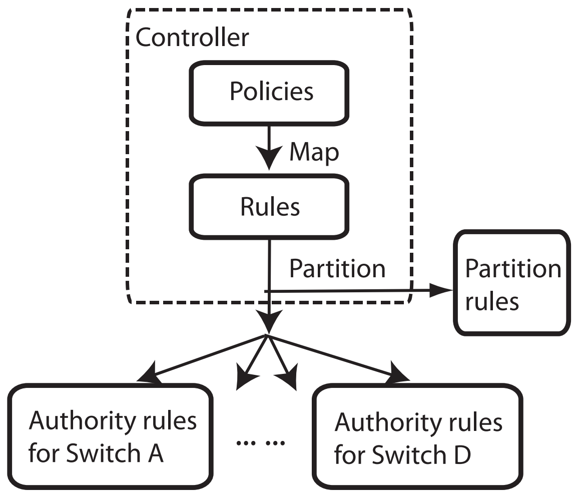
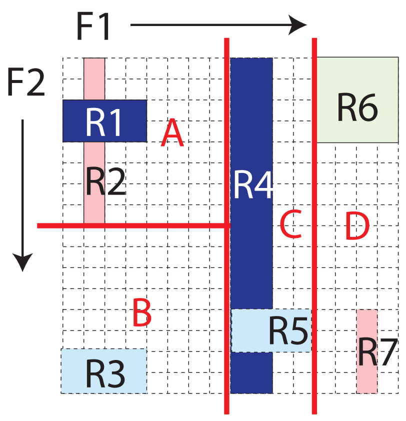


Figure 3.2: Rule operations in the controller.

Precompute low-level rules: The controller pre-computes the low-level rules based on the high-level policies by simply substituting high-level names with network addresses. Since low-level rules are pre-computed and installed in the TCAM of switches, we can always process packets in the fast path. Most kinds of policies can be translated to the low-level rules in advance because the controller knows the addresses of the hosts when the hosts first connect to the ingress switch, and thus can substitute the high-level names with addresses. However, precomputation is not an effective solution for policies (like traffic engineering) that depend on dynamically changing network state.

Use partitioning to subdivide the space of all rules: Hashing is an appealing way to subdivide the rules and direct packets to the appropriate authority switch. While useful for flat keys like an IP or MAC address [5, 55], hashing is not effective when the keys can have wildcards in arbitrary bit positions. In particular, packets matching the same wildcard rule would have different hash values, leading them to different authority switches; as a result, multiple authority switches would need to store the same wildcard rule. Instead of relying on hashing, DIFANE *partitions* the rule space, and assigns each portion of rule space to one or more authority switches. Each authority switch stores the rules falling in its part of the partition.



(a) Low-level rules and the partition

Type	Priority	F_1	F_2	Action	Timeout	Note
Cache rules	210	00**	111*	Encap, forward to B	10 sec	R_3
	209	1110	11**	Drop	10 sec	R_7
	...	...	...	... ...	...	...
Authority rules	110	00**	001*	Encap, forward to D, trigger ctrl. plane func.	∞	R_1
	109	0001	0***	Drop, trigger ctrl. plane func.	∞	R_2
Partition rules	15	0***	000*	Encap, redirect to B	∞	Primary
	14	0***	1***	Encap, redirect to C	∞	
	13	11**	****	Encap, redirect to D	∞	
	5	0***	000*	Encap, redirect to B'	∞	Backup
	...	...	...	... ...	...	...

(b) Wildcard rules in the TCAM of Switch A

Figure 3.3: Wildcard rules in DIFANE (A-D are authority switches).

Run the partitioning algorithm on the controller: Running the partitioning algorithm on the switches themselves would introduce a large overhead, because they would need to learn the rules from the controller, run the partitioning algorithm, and distribute the results. In contrast, the controller is a more natural place to run the partitioning algorithm. The controller is already responsible for translating policies into rules and can easily run the partitioning algorithm periodically, as the distribution of low-level rules changes. We expect the controller would recompute the partition relatively infrequently, as most rule changes would not require rebalancing the division of rule space. Section 3.4 discusses how DIFANE handles changes to rules with minimal interruption to the data traffic.

Represent the partition as a small collection of partition rules: Low-level rules are defined as actions on a *flow space*. The flow space usually has seven dimensions (source/destination IP addresses, MAC addresses, ports, the protocol) or more. Figure 3.3(a) shows a two-dimensional flow space (F_1, F_2) and the rules on it. The bit range of each field is from 0 to 15 (*i.e.*, $F_1 = F_2 = [0..15]$). For example, F_1, F_2 can be viewed as the source/destination fields of a packet respectively. Rule R_2 denotes that all packets which are from source 1 ($F_1 = 1$) and forwarded to a destination in [0..7] ($F_2 = [0..7]$) should be dropped.

The controller partitions the flow space into M *ranges*, and assigns each range to an authority switch. The resulting partition can be expressed concisely as a small number of coarse-grain *partition rules*, where M is proportional to the number of authority switches rather than the number of low-level rules. For example, in Figure 3.3(a), the flow space is partitioned into four parts by the straight lines, which are represented by the partition rules in Figure 3.3(b). Section 3.5 discusses how the controller computes a partition of overlapping wildcard rules that reduces TCAM usage.

Duplicate authority rules to reduce stretch and failure-recovery time: The first packet covered by an authority rule traverses a longer path through an authority switch. To reduce the extra distance the traffic must travel (*i.e.*, "stretch"), the controller can assign each of the M ranges to *multiple* authority switches. For example, if each range is handled by two authority switches, the controller can generate two partition rules for each range, and assign each switch the rule that would minimize stretch. That way, on a cache miss,[3] a switch directs packets to the *closest* authority switch responsible for that range of rules. The placement of multiple authority switches is discussed in Section 3.5.

Assigning multiple authority switches to the same range can also reduce failure-recovery time. By pushing *backup* partition rules to every switch, a switch can quickly fail over to the backup authority switch when the primary one fails (see Section 3.4). This requires each switch to store more partition rules (*e.g.*, $2M$ instead of M), in exchange for faster failure recovery. For example, in Figure 3.3(b), switch A has a primary partition rule that directs packets to B and a backup one that directs packets to B'.

[3] In DIFANE, every packet matches some rule in the switch. "Cache miss" in DIFANE means a packet does not match any cache rules, but matches a partition rule instead.

3.3.2 Packet Redirection and Rule Caching

The authority switch stores the authority rules. The ingress switch encapsulates the first packet covered by an authority switch and redirects it to the authority switch.[4] The authority switch processes the packet and also caches rules in the ingress switch so that the following packets can be processed at the ingress switch.

Packet redirection: In the ingress switch, the first packet of a wildcard flow matches a partition rule. The partition rule indicates which authority switch maintains the authority rules that are related to the packet. For example, in Figure 3.3 a packet with $(F_1 = 9, F_2 = 7)$ hits the primary partition rule for authority switch B and should be redirected to B. The ingress switch encapsulates the packet and forwards it to the authority switch. The authority switch decapsulates the packet, processes it, re-encapsulates it, and forwards it to the egress switch.

Rule Caching: To avoid redirecting all the data traffic to the authority switch, the authority switch caches the rules in the ingress switch.[5] Packets that match the cache rules are encapsulated and forwarded directly to the egress switch (*e.g.*, packets matching R_3 in Figure 3.3(b) are encapsulated and forwarded to D). In DIFANE "miss" packets do not wait for rule caching, because they are sent through the authority switch rather than buffered at the ingress switch. Therefore, we can run a simple caching function in the control plane of the authority switch to generate and install cache rules in the ingress switch. The caching function is triggered whenever a packet matches the authority rules in the authority switch. The cache rule has an idle timeout so that it can be removed by the switch automatically due to inactivity.

3.3.3 Implement DIFANE with Wildcard Rules

All the data plane functions required in DIFANE can be expressed with three sets of wildcard rules of various granularity with simple actions, as shown in Figure 3.3(b).

Cache rules: The ingress switches cache rules so that most of the data traffic hits in the cache and is processed by the ingress switch. The cache rules are installed by the authority switches in the network.

[4]With encapsulation, the authority switch knows the address of the ingress switch from the packet header and sends the cache rules to the ingress switch.

[5]Here we assume that we cache flow rules only at the ingress switch. We discuss the design choices of where to cache flow rules in our Section 4.6.

Authority rules: Authority rules are only stored in authority switches. The controller installs and updates the authority rules for all the authority switches. When a packet matches an authority rule, it triggers a control-plane function to install rules in the ingress switch.

Partition rules: The controller installs partition rules in each switch. The partition rules are a set of *coarse-grained* rules. With these partition rules, we ensure a packet will always match at least one rule in the switch and thus always stay in the data plane.

The three sets of rules can be easily expressed as a single list of wildcard rules with different priorities. Priorities are naturally supported by TCAM. If a packet matches multiple rules, the packet is processed based on the rule that has the highest priority. The cached rules have highest priority because packets matching cache rules do not need to be directed to authority switches. In authority switches, authority rules have higher priority than partition rules, because packets matching authority rules should be processed based on these rules. The primary partition rules have higher priority than backup partition rules.

Since all functionalities in DIFANE are expressed with wildcard rules, DIFANE does not require any data-plane modifications to the switches and only needs minor software extensions in the control plane of the authority switches.

3.4 Handling Network Dynamics

In this section, we describe how DIFANE handles dynamics in different parts of the network: To handle rule changes at the controller, we need to update the authority rules in the authority switches and occasionally repartition the rules. To handle topology changes at the switches, we leverage link-state routing and focus on reducing the interruptions of authority switch failure and recovery. To handle host mobility, we dynamically update the rules for the host, and use redirection to handle changes in the routing rules.

3.4.1 Changes to the Rules

The rules change when administrators modify the policies, or network events (*e.g.*, topology changes) affect the mapping between policies and rules. The related authority rules, cache rules, and partition rules in the switches should be modified correspondingly.

Authority rules are modified by the controller directly: The controller changes the authority rules in the related authority switches. The controller can easily identify the related authority switches based on the partition it generates.

Cache rules expire automatically: Cached copies of the old rules may still exist in some ingress switches. These cache rules will expire after the timeout time. For critical changes (*e.g.*, preventing DoS attacks), the authority switches can get the list of all the ingress switches from the link-state routing and send them a message to evict the related TCAM entries.

Partition rules are recomputed occasionally: When the rules change, the number of authority rules in the authority switches may become unbalanced. If the difference in the number of rules among the authority switches exceeds a threshold, the controller recomputes the partition of the flow space. Once the new partition rules are generated, the controller notifies the switches of the new partition rules, and updates the authority rules in the authority switches.

The controller cannot update all the switches at exactly the same time, so the switches may not have a consistent view of the partition during the update, which may cause transient loops and packet loss in the network. To avoid packet loss, the controller simply updates the switches in a specific order. Assume the controller decides to move some authority rules from authority switch A to B. The controller first sends the authority rules to authority switch B, before sending the new partition rules for A and B to all the switches in the network. Meanwhile, switches can redirect the packets to either A or B for the authority rules. Finally, the controller deletes the authority rules in switch A. In this way, we can prevent packet loss during the change. The same staged update mechanism also applies to the partition change among multiple authority switches.

3.4.2 Topology Dynamics

Link-state routing enables the switches to learn about topology changes and adapt routing quickly. When authority switches fail or recover, DIFANE adapts the rules to reduce traffic interruption.

Authority switch failure: When an authority switch fails, packets directed through it are dropped. To minimize packet loss, we must react quickly to authority switch failures. We design a distributed authority switch takeover mechanism. As discussed in Section 3.3.1, the controller assigns the same group of authority rules to multiple authority switches to reduce stretch and failure-recovery time. Each ingress switch has primary partition rules directing traffic to their closest authority switch and backup partition rules with lower priority that directing traffic to another authority switch when the primary one fails.

The link-state routing protocol propagates a message about the switch failure throughout the network. Upon receiving this message, the switches invalidate their partition rules that direct traffic to the failed authority switch. As a result, the backup partition rule takes effect and automatically directs packets through the backup authority switch. For the switches that have not yet received the failure information, the packets may get sent towards the failed authority switch, but will finally get dropped by a switch who has updated its switch forwarding table.[6]

Authority switch addition/recovery: We use the controller to handle switches joining in the network, because it does not require fast reaction compared to authority switch failures. To minimize the change of the partition rules and authority rules, the controller randomly picks an authority switch, divides its flow range evenly into two parts. The controller then moves one part of the flow range to the new switch, and installs the authority rules in the new switch. Finally the controller updates the partition rules correspondingly in all the switches.

3.4.3 Host Mobility

In DIFANE, when a host moves, its MAC and perhaps IP address stays the same. The rules in the controller are defined based on these identifiers, and thus do not change as hosts move. As a result, the partition rules and the authority rules also stay the same.[7] Therefore we only need to consider the changes of cache rules.

Installing rules at the new ingress switch on demand: When a host connects to a new ingress switch, the switch may not have the cache rules for the packets sent by the hosts. So the packets are redirected to the responsible authority switch. The authority switch then caches rules at the new ingress switch.

Removing rules from old ingress switch by timeout: Today's flow-based switches usually have a timeout for removing the inactive rules. Since the host's old ingress switch no longer

[6] If the switch can decapsulate the packet and encapsulate it with the backup authority switch as the destination, we can avoid such packet loss.

[7] In some enterprises, a host changes its identifier when it moves. The rules also change correspondingly. We can use the techniques in Section 3.4.1 to handle the rule changes.

receives packets from the moved host, the cache rules at the switch are automatically removed once the timeout time expires.

Redirecting traffic from old ingress switch to the new one: The rules for routing packets to the host change when the host moves. When a host connects to a new switch, the controller gets notified through the link-state routing and constructs new routing rules that map the address of the host to the corresponding egress switch. The new routing rules are then installed in the authority switches.

The cached routing rules in some switches may be outdated. Suppose a host H moves from ingress switch S_{old} to S_{new}. The controller first gets notified about the host movement. It then installs new routing rules in the authority switch, and also installs a rule in S_{old} redirecting packets to S_{new}. If an ingress switch A receives a packet whose destination is H, A may still send the packet to the old egress point S_{old} if the cache rule has not expired. S_{old} then redirects packets to S_{new}. After the cache rule expires in switch A, A directs the packets to the authority switch for the correct egress point S_{new} and caches the new routing rule.

3.5 Handling Wildcard Rules

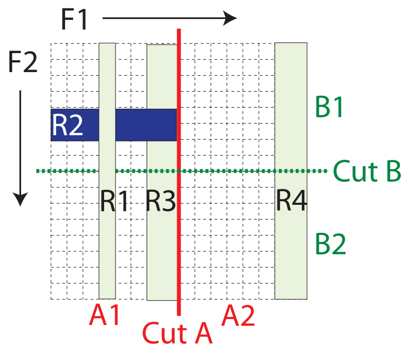
Most flow-management systems simply use microflow rules [13] or transform overlapping wildcard rules into a set of non-overlapping wildcard rules. However these methods significantly increase the number of rules in switches, as shown in our evaluation in Section 4.5. To the best of our knowledge, there is no systematic and efficient solution for handling overlapping wildcard rules in *network-wide* flow-management systems. In this section, we first propose a simple and efficient solution for multiple authority switches to independently insert cache rules in ingress switches. We then discuss the key ideas of partitioning overlapping wildcard rules

3.5.1 Caching Wildcard Rules

Wildcard rules complicate dynamic caching at ingress switches. In the context of access control, for example in Figure 3.4, the packet $(F_1 = 7, F_2 = \mathbf{0})$ matches an "accept" rule R_3 that overlaps with "deny" rule R_2 which has higher priority. Simply caching R_3 is not safe. If we just cache R_3 in the ingress switch, another packet $(F_1 = 7, F_2 = \mathbf{5})$ could incorrectly pass the cached rule

Rule	$\mathbf{F_1}$	$\mathbf{F_2}$	**Action**
R_1	4	0-15	Accept
R_2	0-7	5-6	Drop
R_3	6-7	0-15	Accept
R_4	14-15	0-15	Accept

(a) Wildcard rules listed in the decreasing order of priority. ($R_1 > R_2 > R_3 > R_4$)



(b) Graphical view and two partition solutions.

Figure 3.4: An illustration of wildcard rules.

R_3, because the ingress switch is not aware of the rule R_2. Thus, because rules can overlap with each other, the authority switch cannot *solely* cache the rule that a packet matches. This problem exists in all the flow management systems that cache wildcard rules and therefore it is not trivial to extend Ethane controllers [13] to support wildcards.

To address this problem, DIFANE constructs *one or more new wildcard rules that cover the largest flow range (*i.e.*, a hypercube in a flow space) in which all packets take the same action.* We use Figure 3.4 to illustrate the solution. Although the rules overlap, which means a packet may match multiple rules, the packet only takes the action of the rule with the highest priority. That is, each point in the flow space has a unique action (which is denoted by the shading in each spot in Figure 3.4(b)). As long as we cache a rule that covers packets with the same action (*i.e.*, spots with the same shading), we ensure that the caching preserves semantic correctness. For example, we cannot cache rule R_3 because the spots it covers have different shading. In contrast, we can safely cache R_1 because all the spots it covers has the same shading. For the packet ($F_1 = 7, F_2 = \mathbf{0}$), we construct and cache a *new* rule: $F_1 = [6..7], F_2 = [0..3]$.

The problem of constructing new wildcard rules for caching at a *single* switch was studied in [57]. Our contribution lies in extending this approach to *multiple* authority switches, each of which can independently install cache rules at ingress switches. Given such a setting, we must prevent authority switches from installing *conflicting* cache rules. To guarantee this, DIFANE ensures that the caching rules installed by different authority switches do not overlap. This is achieved by *allocating non-overlapping flow ranges to the authority switches, and only allowing the authority switch to install caching rules in its own flow range.* We later evaluate our caching scheme in Section 4.5.

3.5.2 Partitioning Wildcard Rules

Overlapping wildcard rules also introduce challenges in partitioning. We first formulate the partition problem: The controller needs to partition rules into M parts to minimize the total number of TCAM entries across all M authority switches with the constraint that the rules should not take more TCAM entries than are available in the switches. There are three key ideas in the partition algorithm:

Allocating non-overlapping flow ranges to authority switches: As discussed in the caching solution, we must ensure that the flow ranges of authority switches do not overlap with each other. To achieve this goal, DIFANE first partitions the entire flow space into M flow ranges and then stores rules in each flow range in an authority switch. For example, the "Cut A" shown in Figure 3.4(b) partitions the flow space on field F_1 into two equal flow ranges A_1 and A_2. We then assign R_1, R_2 and R_3 in A_1 to one authority switch, and R_4 to another.

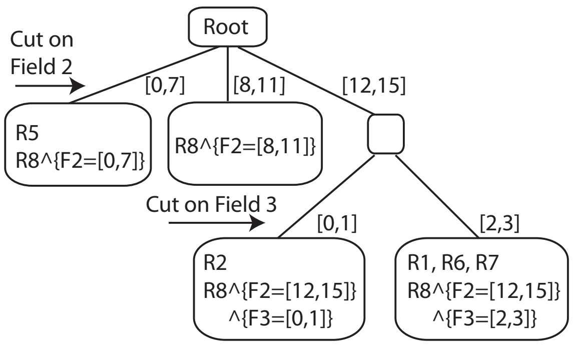
DIFANE splits the rules so that each rule only belongs to one authority switch. With the above partitioning approach, one rule may span multiple partitions. For example, "Cut B" partitions the flow space on field F_2, which results in the rules R_1, R_3, and R_4 spanning the two flow ranges B_1 and B_2. We split each rule into two independent rules by intersecting it with the two flow ranges. For example, the two new rules generated from rule R_4 are $F_1 = [14..15], F_2 = [0..7] \rightarrow$ *Accept* and $F_1 = [14..15], F_2 = [8..15] \rightarrow$ *Accept*. These two independent rules can then be stored in different authority switches. Splitting rules thus avoids the overlapping of rules among authority switches, but at the cost of increased TCAM usage.

To reduce TCAM usage, we prefer the cuts to align with rule boundaries. For example, "Cut A" is better than "Cut B" because "Cut A" does not break any rules. We also observe that cut on field F_1 is better than F_2 since we have more rule boundaries to choose.

Based on these observations, we design a decision-tree based partition algorithm. We first formulate the rule partition problem as follows: Assume that there are M candidate authority switches, each of which can store up to S TCAM entries. For a given set of N low-level rules of K dimensions, we would like to partition the flow space into n hypercubes ($n \leq M$), because hypercubes are easy to represent as wildcard partition rules. Each of the hypercubes is represented by a K-tuple of ranges, $[l_1..r_1], \ldots, [l_K..r_K]$, and is stored in an authority switch. The optimization objective is to minimize the total number of TCAM entries in all n authority switches. The flow partition problem is NP-hard for $K \geq 2$.[8] Therefore, we instead design a heuristic algorithm for partitioning rules.

Rule \ Field	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_5}$	**Action**
R_1	0-1	14-15	2	0-3	0	accept
R_2	0-1	14-15	1	2	0	accept
R_3	0-1	8-11	0-3	2	1	deny
R_4	0-1	8-11	2	3	1	deny
R_5	0-15	0-7	0-3	1	0	accept
R_6	0-15	14-15	2	1	0	accept
R_7	0-15	14-15	2	2	0	accept
R_8	0-15	0-15	0-3	0-3	0-1	deny

(a) A group of wildcard rules



(b) The decision tree for the rules

Figure 3.5: Construct decision tree for a group of rules.

[8]The proof of the NP-hardness of the partition problems is omitted due to lack of space.

```
Algorithm 1 Heuristic partition algorithm using decision tree
Initialization:
Step 1  $k = 0$. $T_0$ is the tree node which represents the entire flow range.
Split the flow range:
Step 2  Incrementk. Pick up a tree node $T_k$ to split.
        $T_k$ is a leaf node in the decision tree that contain more than $S$ TCAM entries to represent its
        hypercube $C_k$.
        If we cannot find such a node, stop.
Step 3  Select a flow dimension $i$ that have maximum number of unique components $u_i$.
Step 4  Pick $w$ boundaries of the unique components that minimizes $(\sum_{1 \le t \le w} f(C_k^t) - f(C_k))/w$.
Step 5  Put all the child node of $T_i$ in the tree.
Step 6  Goto Step 2.
```

The complete algorithm is shown in Algorithm 1, which consists of two key ideas:

Use a decision tree to represent the partition: The root node of the decision tree denotes the hypercube of the entire flow space. Similarly, each node in the decision tree denotes a hypercube of flow range and maintains all the rules intersecting with the hypercube. We start with the root node (**Step 1**). In each round of the splitting process, we pick a node in the decision tree which has more than S authority rules, and split it into a group of child nodes, each of which a subset flow range of its parent node (**Step 2**). The splitting process terminates when each leaf node has fewer than S authority rules. In the end, the authority rules in each leaf node will be assigned to an authority switch.

Cut based on rule boundaries: We first choose the dimension i that has the maximum number of unique components (*i.e.*, non-overlapping ranges) as the dimension to split (**Step 3**). This gives us a better chance to be able to split the flow range into balanced pieces. For example, in Figure 3.5, field F_2 has four unique components [0..7], [8..11], [12..13], [14..15].

The next challenge is to split the hypercube in the selected dimension. As we discussed earlier, partitioning the flow range equally may not be the best choice. It leads to many rules that span across multiple partitions, and hence yields more authority rules. Instead, we partition the flow space based on the boundaries of the unique components (**Step 4**). Let function $f(C)$ be the number of required TCAM entries for hypercube C. Assume that C_k is the hypercube we want to split in the k-th round of our algorithm. We select w boundaries of the unique components in dimension i ($b_1 \ldots b_w$) and the resulting sub-hypercubes $C_k^1 \ldots C_k^w$ such that the average increase of authority rules per cut is minimized: $U = (\sum_{1 \le t \le w} f(C_k^t) - f(C_k))/w$. We enumerate all the boundary selections and choose the one with minimal U.

Figure 3.5 illustrates an example of rule partitioning. Assume that we have $S = 4$ in each authority switch. Using our partition algorithm, we can construct a decision tree as shown in Figure 3.5(b) for the rules shown in Figure 3.5(a). In the first round, we choose the dimension on field F_2 to partition the root node, because it has the maximum number of unique ranges. We then divided the root node into three children nodes on field F_2: [0..7], [8..11], [12..15]. This yields $U=0$ because rules R_3, R_4, R_8 that fall in the second child of $F_2=[8..11]$ all take the deny actions. In the second round, since the third child node requires 5 authority rules, we further split it into two children nodes on field F_3.

Note that, although our partition algorithm is motivated by both HiCuts [58] and HyperCuts [59], which explored the efficient *software* processing of packet classification rules *in one switch* with the help of a decision tree, we differ in our optimization goals. Both HiCuts and HyperCuts sought to speed up software processing of the rules, while DIFANE aims at minimizing the TCAM usage in switches, because the partition and authority rules are all processed in hardware. This leads to two key design differences in our algorithm: (i) We choose to cut the selected dimension based on the rule boundaries, while HiCuts and HyperCuts cut the selected dimension equally for c cuts. (ii) We choose the cuts so that the number of TCAM entries per cut is minimized. In HiCuts and HyperCuts, they optimize on the number of cuts, in order to minimize the size of the tree (and especially its depth). DIFANE allows a slightly deeper decision tree if it reduces TCAM usage.

In summary, DIFANE partitions the entire rule space into M independent portions, and thus each authority switch is assigned a non-overlapping portion. However, *within* the portion managed by a single authority switch, DIFANE allows overlapping or nested rules. This substantially reduces the TCAM usage of authority switches (see Section 4.5).

Duplicating authority rules to reduce stretch: We can duplicate the rules in each partition on multiple authority switches to reduce stretch and to react quickly to authority switch failures. Due to host mobility, we cannot pre-locate authority switches to minimize stretch. Instead, we assume traffic that is related to one rule may come from any ingress switches and place replicated authority switches to reduce the average stretch. One simple method is to randomly place the replicated switches to reduce stretch. Alternatively, by leveraging an approximation algorithm for the "k-median problem" [60], we can place the replicated authority switches so that they have minimal average stretch to any pair of switches. Both schemes are evaluated in Section 4.5.

3.6 Design and Implementation

In this section, we present our design and implementation of DIFANE. First, we describe how our prototype handles multiple sets of rules from different kinds of high-level policies for different management functions. Second, we describe the prototype architecture, which just add a few control-plane functions for authority switches to today's flow-based switches.

3.6.1 Managing Multiple Sets of Rules

Different management functions such as access control, measurement and routing may have totally different kinds of policies. To make our prototype efficient and easy to implement, we generate different sets of rules for different policies, partition them using a single partition algorithm, and process them sequentially in the switch.

Generate multiple sets of low-level rules: Translating and combining different kinds of high-level policies into one set of rules is complicated and significantly increases TCAM usage. For example, if the policies are to monitor web traffic, and perform destination based routing, we have to provide rules for (*dst*, port 80) and (*dst*, other ports) for each destination *dst*. If the administrator changes the policy of monitoring port 21 rather than port 80, we must change the rules for every destination. In contrast, if we have different sets of rules, we only need one routing rule for each destination and a *single* measurement rule for port 80 traffic which is easy to change.

To distribute multiple sets of rules, the controller first partitions the flow space to minimize the total TCAM usage. It then assigns all the rules (of different management modules) in one flow range to one authority switch. We choose to use the same partition for different sets of low-level rules so that packets only need to be redirected to one authority switch to match all sets of authority rules.[9]

[9]One can also choose to provide a different partition of the flow space for different sets of low-level rules, but packets may be redirected to multiple authority switches to match the authority rules of different management modules.

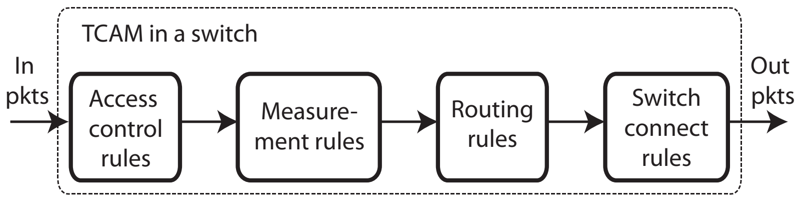


Figure 3.6: Rules for various management modules.

Processing packets through multiple sets of rules in switches: In switches we have one set of rules for each management module.[10] We process the flow rules sequentially through the rules for different modules as shown in Figure 3.6. We put access control rules first to block malicious traffic. Routing rules are placed later to identify the egress switch for the packets. Finally, the link-state routing constructs a set of *switch connection rules* to direct the packets to their egress switches.

To implement sequential processing in the switch where all the rules share the same TCAM, the controller sets a "module identifier" in the rules to indicate the module they belong to. The switch first initializes a module identifier in the packet. It then matches the packet with the rules that have the same module identifier. Next, the switch increments the module identifier in the packet and matches to the next set of rules. By processing the packet several times through the memory, the switch matches the packet to the rules for different modules sequentially. The administrator specifies the order of the modules by giving different module identifiers for the high-level policies in the controller.

A packet may be redirected to the authority switch when it is in the middle of the sequential processing (*e.g.*, while being processed in the measurement module). After redirection, the packet will be processed through the following modules in the authority switch based the module identifier in the packet.

3.6.2 DIFANE Switch Prototype

Figure 3.7 shows both the control and data plane of our DIFANE switch prototype.

[10]To make the rule processing simple, we duplicate the same set of partition rules in the management modules.

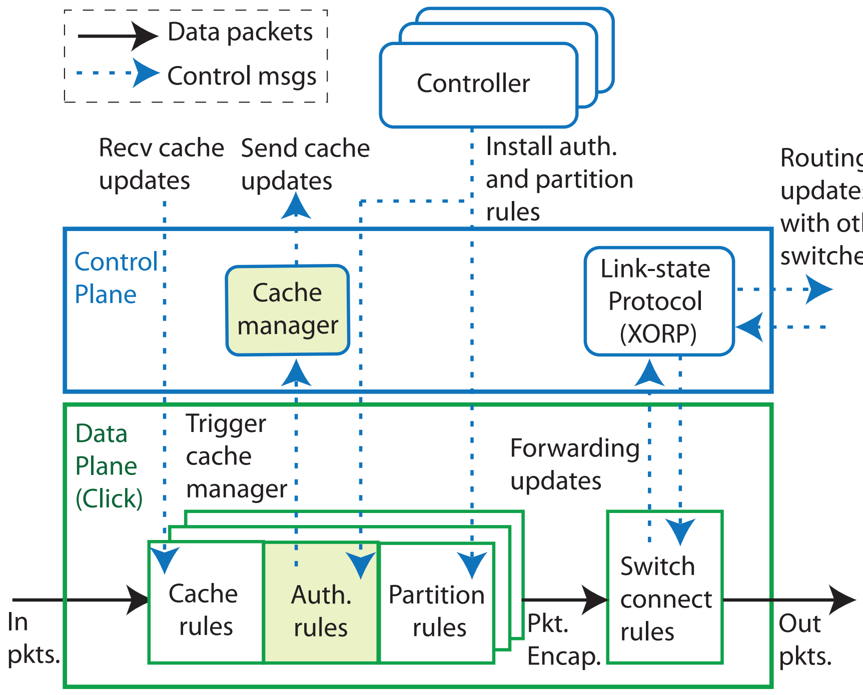


Figure 3.7: DIFANE prototype implementation. (Cache manager and authority rules (shaded boxes) only exist in authority switches.)

Control plane: We use XORP [61] to run the link-state routing protocol to maintain the switch-level connectivity, and keep track of topology changes. XORP also sends updates of the switch connection rules in the data plane.

The authority switch also runs a cache manager which installs cache rules in the ingress switch. If a packet misses the cache, and matches the authority rule in the authority switch, the cache manager is triggered to send a cache update to the ingress switch of the packet. The cache manager is implemented in software in the control plane because packets are not buffered and waiting for the cache rule in the ingress switch. The ingress switch continues to forward packets to the authority switch if the cache rules are not installed. The cache manager sends an update for *every* packet that matches the authority rule. This is because we can infer that the ingress switch does not have any related rules cached, otherwise it would forward the packets directly rather than sending them to the authority switch.[11]

[11]For UDP flows, they may be a few packets sent to the authority switch. The authority switch sends one feedback for each UDP flow, because it takes very low overhead to send a cache update message (just one UDP

Data plane: We run Click-based OpenFlow switch [53] in the kernel as the data plane of DIFANE. Click manages the rules for different management modules and encapsulates and forwards packets based on the switch connection rules. We implement the packet encapsulation function to enable tunneling in the Click OpenFlow element. We also modify the Click OpenFlow element to support the flow rule action "trigger the cache manager". If a packet matches the authority rules, Click generates a message to the cache manager through the kernel-level socket "netlink". Today's flow-based switches already support actions of sending messages to a *local controller* in order to communicate with the centralized controller [10]. We just add a new message type of "matching authority rules". In addition, today's flow-based switches already have interfaces for the centralized controller to install new rules. The cache manager then just leverages the same interfaces to install cache rules in the ingress switches.

3.7 Evaluation

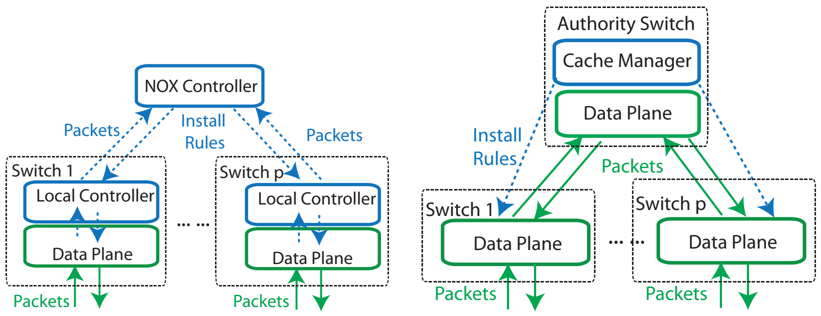
Ideally we would like to evaluate DIFANE based on policies, topology data, and user-mobility traces from real networks. Unfortunately, most networks today are still configured with rules that are tightly bound to their network configurations (*e.g.*, IP address assignment, routing, and VLANs). Therefore, we evaluated DIFANE's approach against the topology and access-control rules of a *variety* of different networks toexplore DIFANE's benefit across various settings. We also perform latency, throughput, and scalability micro-benchmarks of our DIFANE prototype and a trace-driven evaluation of our partition and caching algorithms.

To verify the design decisions in Section 3.2, we evaluate two central questions in this section: (1) How efficient and scalable is DIFANE? (2) How well do our partition and caching algorithms work in handling large sets of wildcard rules?

3.7.1 Performance of the DIFANE Prototype

We implemented the DIFANE prototype using a kernel-level Click-based OpenFlow switch and compared the delay and throughput of DIFANE with NOX [10], which is a centralized solution for flow management. For a fair comparison, we first evaluated DIFANE using only one authority

packet). In addition, we do not need to store the existing cached flow rules in the authority switch or fetch them from the ingress switch.



(a) NOX (b) DIFANE.

Figure 3.8: Experiment setup. (Dashed lines are control messages; straight lines are data traffic.)

switch. Then we evaluated the throughput of DIFANE with multiple authority switches. Finally we investigated how fast DIFANE reacts to the authority switch failures.

In the experiments, each sender sends the packets to a receiver through a single ingress switch, which is connected directly to either NOX or a DIFANE authority switch as shown in Figure 3.8. (In this way, the network delay from the ingress switch to NOX and the DIFANE authority switch is minimized. We evaluate the extra delay caused by redirecting through authority switches in Section 3.7.2.) With NOX, when a packet does not match a cached rule, the packet is buffered in the ingress switch before NOX controller installs a rule at the ingress switch. In contrast, in DIFANE the authority switch redirects the packet to the receiver in the data plane and installs a rule in the ingress switch at the same time. We generate flows with different port numbers, and use a separate rule for each flow. Since the difference between NOX and DIFANE lies in the processing of the first packet, we generate each flow as a single 64 Byte UDP packet. Based on the measurements, we also calculate the performance difference between NOX and DIFANE for flows of normal sizes. Switches, NOX, and traffic generators ("clients") run on separate 3.0 GHz 64-bit Intel Xeon machines to avoid interference between them.

(1) DIFANE achieves small delay for the first packet of a flow by always keeping packets in the fast path. In Figure 3.9, we send traffic at 100 single-packet flows/s and measure the round-trip time (RTT) of the each packet being sent through a switch to the receiver and an ACK packet being sent back. Although we put NOX near the switch, the packets still experience a RTT of 10 ms on average, which is not acceptable for those networks that have tight latency requirement such as data centers. In DIFANE, since the packets stay in the fast path (forwarded through an authority switch), the packets only experience 0.4 ms RTT on average. Since all the

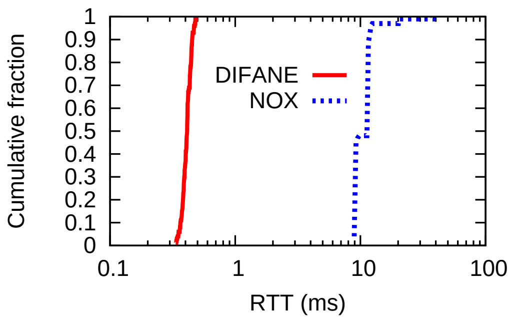


Figure 3.9: Delay comparison of DIFANE and NOX.

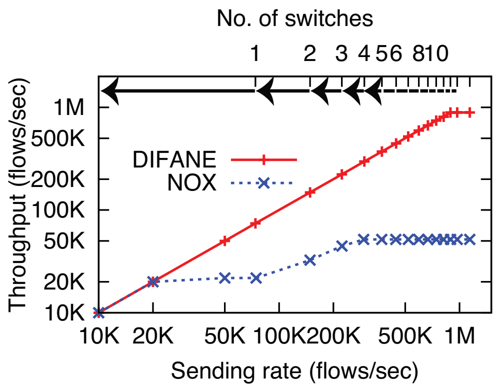


Figure 3.10: Throughput comparison of DIFANE and NOX.

following packets take 0.3 ms RTT for both DIFANE and NOX, we can easily calculate that to transfer a flow of a normal size 35 packets, which is based on the measurement in the paper [62]), the average packet transfer time is 0.3 ms (=(0.3*34+0.4)/35) transferring time for DIFANE but 0.58 ms (=(0.3*34+10)/35) for NOX. Others' tests of NOX with commercial OpenFlow switches observed similar delay [63].

(2) DIFANE achieves significantly higher throughput than NOX. We then increase the number of switches (p) to measure the throughput of NOX and DIFANE. In Figure 3.10, we show the maximum throughput of flow setup for one client using one switch. In DIFANE, the switch was able to achieve the client's maximum flow setup rate, 75K flows/s, while the NOX architecture was only able to achieve 20K flows/s. This is because, while all packets remain in the fast path (the software kernel) in DIFANE, the OpenFlow switch's local controller (implemented in user-space) becomes a bottleneck before NOX does. Today's commercial OpenFlow switches can only send

60-330 flows/s to the controller due to the limited CPU resources in the controller [64]. Section 3.8 discusses how to run DIFANE on today's commercial switches.

As we increase the number of ingress switches—each additional data-point represents an additional switch and client as shown in the upper x-axis—we see that the NOX controller soon becomes a bottleneck: With four switches, a single NOX controller achieves a peak throughput of 50K single-packet flows/s. Suppose a network has 1K switches, a single NOX controller can only support 50 new flows per second for each switch simultaneously. In comparison, the peak throughput of DIFANE with one authority switch is 800K single-packet flows/s.

Admittedly, this large difference exists because DIFANE handles packets in the kernel while NOX operates in user space. However, while it is possible to move NOX to the kernel, it would not be feasible to implement the entire NOX in today's switch hardware because the online rule generation too complex for hardware implementation. In contract, DIFANE is meant for precisely that — designed to install a set of rules in the data plane.[12]

We then evaluate the performance of the cache manager in authority switches. When there are 10 authority rules, the cache manager can handle 30K packets/s and generate one rule for each packet. When there are 9K authority rules, the cache manager can handle 12K packets/s. The CPU is the bottleneck of the cache manager.

(3) DIFANE scales with the number of authority switches. Our experiments show that DIFANE's throughput increases linearly with the number of authority switches. With four authority switches, the throughput of DIFANE reaches over 3M flows/s. Administrators can determine the number of authority switches according to the size and the throughput requirements of their networks.

(4) DIFANE recovers quickly from authority switch failure. Figure 3.11 shows the effect of an authority switch failure. We construct a diamond topology with two authority switches, both connecting to the ingress and the egress switches. We set the OSPF hello interval to 1 s, and the dead interval to 3 s. After the authority switch fails, OSPF notifies the ingress switch. It takes less than 10 ms for the ingress switch to change to another authority switch after the dead

[12]Today's TCAM with pipelined processing only takes 3–4 ns per lookup [65], which is more than three orders of magnitude faster than in software.

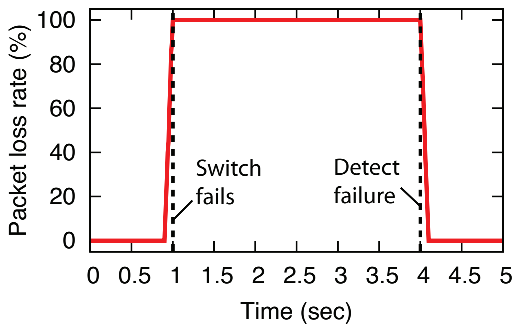


Figure 3.11: Authority switch failure.

Network	# switches/routers	# Rules
Campus	∼1700	30K
VPN	∼1500	59K
IPTV	∼3000	5M
IP	∼2000	8M

Table 3.1: Network characteristics.

interval, at which time the ingress switch sets the backup partition rule as the primary one, and thus connectivity is restored.

3.7.2 Evaluation of Partitioning and Caching

We now evaluate DIFANE's partitioning algorithm using the topologies and access-control rules from several sizable networks (as of Sept. 10, 2009) including a large-scale campus network [36] and three large backbone networks that are operated by a tier-1 ISP for its enterprise VPN, IPTV, and traditional IP services. The basic characteristics of these networks are shown in Table 3.1. In these networks, each access control rule has six fields: ingress interface, source IP/port, destination IP/port, and protocol. Access control rules are configured on the ingress switches/routers. It is highly likely that different sets of rules are configured at different switches and routers, hence one packet may be permitted in one ingress switch but denied at another. In addition, as Table 3.1 shows, there are a large number of access control rules configured in these networks. This is due to the large number of ingress routers that need to have access control rules configured and the potentially large number rules that need to be configured on even a single router. For example, ISPs often configure a set of rules on each ingress switch to protect their infrastructures and important servers from unauthorized customers. This would easily result in a large number of

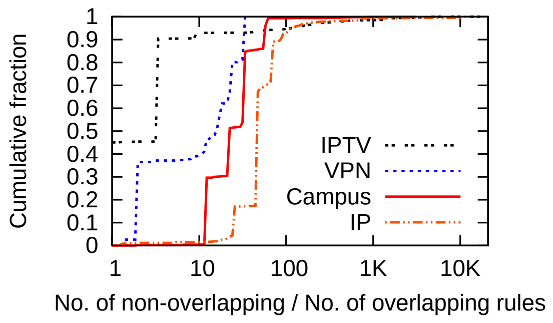


Figure 3.12: Comparison of overlapping and non-overlapping rules.

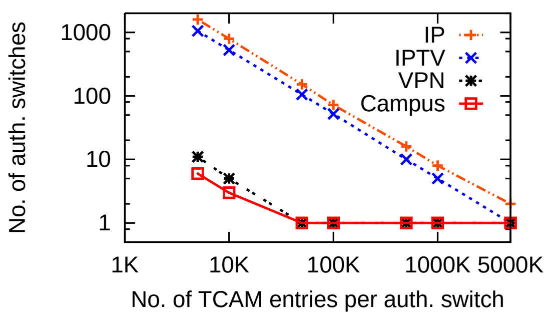


Figure 3.13: Evaluation on number of authority switches.

rules on an ingress switch if there are a large number of customers connected to the switch and/or these customers make use of a large amount of non-aggregatable address space.

In the rest of this section, we evaluate the effect of overlapping rules, the number of authority switches needed for different networks, the number of extra rules needed after the partitioning, the miss rate of caching wildcard rules, and the stretch experienced by packets that travel through an authority switch.

(1) Installing overlapping rules in an authority switch significantly reduces the memory requirement of switches: In our set of access control rules, the use of wildcard rules can result in overlapping rules. One straight forward way to handle these overlapping rules is to translate them into non-overlapping rules which represent the same semantics. However, this is not a good solution because Figure 3.12 shows that the resulting non-overlapping rules require one or two orders of magnitude more TCAM space than the original overlapping rules. This suggests that the use of overlapping wildcard rules can significantly reduce the number of TCAM entries.

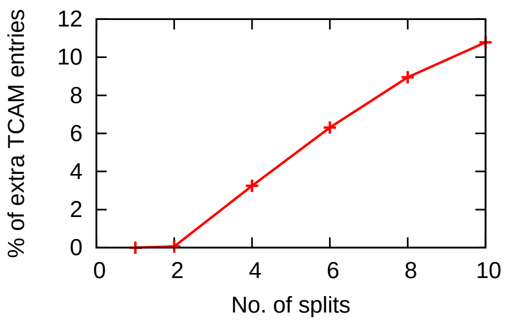


Figure 3.14: The overhead of rule partition.

(2) A small number of authority switches are needed for the large networks we evaluated. Figure 3.13 shows the number of authority switches needed under varying TCAM capacities. The number of authority switches needed decreases almost linearly with the increase of the switch memory. For networks with relatively few rules, such as the campus and VPN networks, we would require 5–6 authority switches with 10K TCAM in each (assuming we need 16B to store the six fields and action for a TCAM entry, we need about 160KB of TCAM in each authority switch). To handle networks with many rules, such as the IP and IPTV networks, we would need approximately 100 authority switches with 100K TCAM entries (1.6MB TCAM) each.[13] The number of the authority switches is still relatively small compared to the network size (2K - 3K switches).

(3) Our partition algorithm is efficient in reducing the TCAM usage in switches. As shown in Figure 3.4, depending on the rules, partitioning wildcard rules can increase the total number of rules and the TCAM usage for representing the rules. With the 6-tuple access-control rules in the IP network, the total number of TCAM entries increases only by 0.01% if we distribute the rules over 100 authority switches. This is because most of the cuts are on the ingress dimension and, in the data set, most rules differ between ingresses. To evaluate how the partition algorithm handles highly overlapping rules, we use our algorithm to partition the 1.6K rules in one ingress router in the IP network. Figure 3.14 shows that we only increase the number of TCAM entries by 10% with 10 splits (100–200 TCAM entries per split).

[13]Today's commercial switches are commonly equipped with 2 MB TCAM.

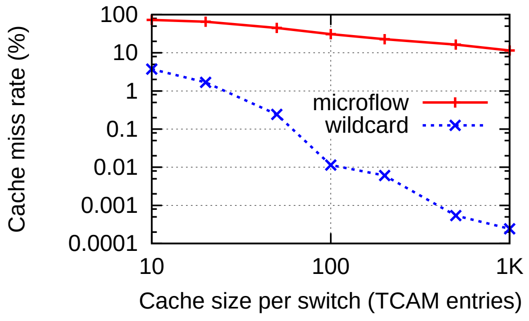


Figure 3.15: Cache miss rate.

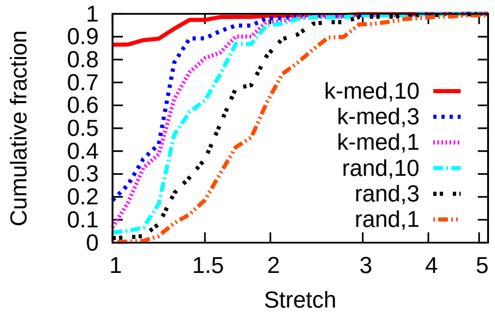


Figure 3.16: The stretch of campus network. (We place 1,3,10 authority switches for each set of the authority rules using k-median and random algorithms respectively.)

(4) Our wildcard caching solution is efficient in reducing cache misses and cache memory size. We evaluate our cache algorithm with packet-level traces of 10M packets collected in December 2008 and the corresponding access-control lists (9K rules) in a router in the IP network. Figure 3.15 shows that if we only cache micro-flow rules, the miss rate is 10% with 1K cache entries. In contrast, with wildcard rule caching in DIFANE, the cache miss rate is only 0.1% with 100 cache entries: 99.9% of packets are forwarded directly to the destination, while only 0.1% of the packets take a slightly longer path through an authority switch. Those ingress switches which are not authority switches only need to have 1K TCAM entries for cache rules (Figure 3.15) and 10 - 1000 TCAM entries for partition rules (Figure 3.13).

(5) The stretch caused by packet redirection is small. We evaluated the stretch of two authority switch placement schemes (random and k-median) discussed in Section 3.5. Figure 3.16 shows the distribution of stretch (delay normalized by that of the shortest path) among all source-destination pairs in the campus network. With only one authority switch for each set of authority rules, the average stretch is twice the delay of the shortest path length in the random scheme. Though some packets experience 10 times the delay of the shortest path, this usually happens to those pairs of nodes that are one or two hops away from each other; so the absolute delay of these paths is not large. If we store one set of rules at three random places, we can reduce stretch (the stretch is 1.8 on average). With 10 copies of rules, the stretch is reduced to 1.5. By placing the authority switches with the k-median scheme, we can further reduce the stretch (*e.g.*, with 10 copies of rules, the stretch is reduced to 1.07).

3.8 DIFANE Deployment Scenarios

DIFANE proposes to use authority switches to always keep packets in the data plane. However, today's commercial OpenFlow switches have resource constraints in the control plane. In this section, we describe how DIFANE can be deployed in today's switches with resource constraints and in future switches as a clean-slate solution. We provide three types of design choices: implementing tunneling with packet encapsulation or VLAN tags; performing rule caching in the authority switches or in the controller; choosing normal switches or dedicated switches as authority switches.

Deployment with today's switches: Today's commercial OpenFlow switches have resource constraints in the control plane. For example, they do not have enough CPU resources to generate caching rules or have hardware to encapsulate packets quickly. Therefore we use VLAN tags to implement tunneling and move the wildcard rule caching to the DIFANE controller. The ingress switch tags the "miss" packet and sends it to the authority switch. The authority switch tags the packet with a different VLAN tag and sends the packet to the corresponding egress switch. The ingress switch also sends the packet header to the controller, and the controller installs the cache rules in the ingress switch.

Performing rule caching in the controller resolves the limitations of today's commercial OpenFlow switches for two reasons. (i) Authority switches do not need to run caching functions. (ii) The authority switches do not need to know the addresses of the ingress switch. We can use VLAN tagging instead of packet encapsulation, which can be implemented in hardware in today's switches.

This deployment scenario is similar to Ethane [13] in that it also has the overhead of the ingress switch sending packets to the controller and the controller installing caching rules. However, the difference is that DIFANE always keeps the packet in the fast path. Since we need one VLAN tag for each authority switch (10 - 100 authority switches) and each egress switch (at most a few thousand switches), we have enough VLAN tags to support tunneling in DIFANE in today's networks.

Clean slate deployment with future switches: Future switches can have more CPU resources and hardware-based packet encapsulation techniques. In this case, we can have a clean slate design. The ingress switch encapsulates the "miss" packet with its address as the source address. The authority switch decapsulates the packet, gets the address of the ingress switch and re-encapsulates the packet with the egress switch address as the packet's destination address. The authority switch also installs cache rules to the ingress switch based on the address it gets from the packet header. In this scenario, we can avoid the overhead and single point of failure of the controller. Future switches should also have high bandwidth channel between the data plane and the control plane to improve the caching performance.

Deployment of authority switches: There are two deployment scenarios for authority switches: (i) The authority switches are just normal switches in the network that can be taken over by other switches when they fail; (ii) The authority switches are dedicated switches that have larger TCAM to store more authority rules and serve essentially as a distributed data plane of the centralized controller. All the other switches just need a small TCAM to store a few partition rules and the cache rules. In the second scenario, even when DIFANE has only one authority switch that serves as the data plane of the controller, DIFANE and Ethane [13] are still fundamentally different in that DIFANE *pre-installs* authority rules in TCAM and thus always keeps the packet in the fast path.

3.9 Flexible Management Support

Recently proposed flow-based management solutions [13, 49, 50, 51, 52] can easily run *on top* of DIFANE, by defining their own policies and translating them into rules, while capitalizing on our support for distributed rule processing for better scalability and performance. In addition, DIFANE is also flexible to support some other proposed techniques and management functions for managing how rules are handled.

Support scalable routing with packet redirection: SEATTLE [5] performs packet redirection based on the hash of a packet's destination MAC address, and reactively caching information about the host's current location. SEATTLE can be easily implemented *within* our architecture even when the flow-based switches do not support hashing. In particular, the DIFANE controller could generate routing rules that map a destination MAC address to the switch where the host is located. The partitioning algorithm could divide the space of rules by cutting along the destination address dimension to generate partition rules, with wildcards in some bit positions of the destination address, and with wildcards in all other header fields. When an ingress switch does not have a cached rule that matches an incoming packet, the partition rule directs the packet through the appropriate authority switch. This triggers the authority switch to install the rules for the destination address at the ingress switch.

Similarly, DIFANE could support ViAggre [56] by creating partition rules based on the destination IP address space (rather than MAC addresses), so ingress switches forward packets to authority switches with more-specific forwarding-table entries.

Handle measurement rules in both authority and ingress switches. In DIFANE packets may be processed in the ingress switch or in the authority switch. If we have a measurement rule that accumulates the statistics of a flow, we need to install it in both the ingress and the authority switch. The controller installs these measurement flow rules in the authority switches, which then installs the rules in the ingress switches. A packet matches one of the cached rule is counted at the ingress switch. Otherwise, it is redirected to an authority switch and counted there.

The controller can use either pull or push method to collect flow information from the authority switches and ingress switches. The cache rules in the ingress switch may be swapped out if the cache rule is not used for the timeout time or there is not enough memory. In this case, the ingress switch notifies the controller and sends the data of the cache rule to the controller, which is already supported by flow-based switches today.

Most network management functions can be achieved by caching rules only at the ingress switch. We choose to cache rules only at the ingress switch for lower overhead. One alternative is to cache rules at every switch on the path a packet transfers. The authority switch would have to determine all the switches on the packet's path and install the rule in them.

In fact, caching rules only at the ingress can meet most of the requirements of management modules. For example, access control rules are checked at the ingress switch to block the malicious traffic before they enter the network. Measurement rules are usually applied at the ingress switch to count the number of packets or the amount of traffic. Routing rules are used at the ingress switch to select the egress point for the packets. Packets are then encapsulated and forwarded directly to the egress switch without matching routing rules at other switches. Customized routing can also be supported by tagging the packets and selecting pre-computed paths at the ingress switch.

3.10 Summary

We design and implement DIFANE, a distributed flow management architecture that distributes rules to authority switches and handles all data traffic in the fast path. DIFANE can handle wildcard rules efficiently and react quickly to network dynamics such as policy changes, topology changes and host mobility. DIFANE can be easily implemented with today's flow-based switches. Our evaluation of the DIFANE prototype, various networks, and large sets of wildcard rules shows that DIFANE is scalable with networks with a large number of hosts, flows and rules.

Chapter 4

SNAP: Scaling Performance Diagnosis on Hosts

This chapter focuses on scaling the management task of performance diagnosis. Network performance problems are notoriously tricky to diagnose, and this is magnified when applications are often split into multiple tiers of application components spread across thousands of servers in a data center. Problems often arise in the communication between the tiers, where either the application or the network (or both!) could be to blame. In this chapter, we present SNAP, a scalable network-application profiler on hosts.

SNAP guides developers in identifying and fixing performance problems. SNAP passively collects TCP statistics and socket-call logs with low computation and storage overhead, and correlates across shared resources (*e.g.*, host, link, switch) and connections to pinpoint the location of the problem (*e.g.*, send buffer mismanagement, TCP/application conflicts, application-generated microbursts, or network congestion). Our one-week deployment of SNAP in a production data center (with over 8,000 servers and over 700 application components) has already helped developers uncover 15 major performance problems in application software, the network stack on the server, and the underlying network.

4.1 Introduction

Modern data-center applications, running over networks with unusually high bandwidth and low latency, should have great communication performance. Yet, these applications often experience low throughput and high delay *between* the front-end user-facing servers and the back-end servers that perform database, storage, and indexing operations. Troubleshooting network performance problems is hard. Existing solutions—like detailed application-level logs or fine-grain packet monitoring—are too expensive to run continuously, and still offer too little insight into where performance problems lie. Instead, we argue that data centers should perform continuous, lightweight profiling of the end-host network stack, coupled with algorithms for classifying and correlating performance problems.

4.1.1 Troubleshooting Network Performance

The nature of the data-center environment makes detecting and locating performance problems particularly challenging. Applications typically consist of tens to hundreds of application components, arranged in multiple tiers of front-ends and back-ends, and spread across hundreds to tens of thousands of servers. Application developers are continually updating their code to add features or fix bugs, so application components evolve independently and are updated while the application remains in operation. Human factors also enter into play: most developers do not have a deep understanding of how their design decisions interact with TCP or the network. There is a constant influx of new developers for whom the intricacies of Nagle's algorithm, delayed acknowledgments, and silly window syndrome remains a mystery.[1]

As a result, new network performance problems happen all the time. Compared to equipment failures that are relatively easy to detect, performance problems are tricky because they happen sporadically and many different components could be responsible. The developers sometimes blame "the network" for problems they cannot diagnose; in turn, the network operators blame the developers if the network shows no signs of equipment failures or persistent congestion. Often, they are both right, and the network stack or some subtle interaction between components is actually responsible [66, 67]. For example, an application sending small messages can trigger

[1]Some applications use UDP, and re-implement reliability, error detection, and flow control; however, these mechanisms can also introduce performance problems.

Nagle's algorithm in the TCP stack, causing transmission delays leading to terrible application throughput.

In the production data center we study, the process of actually detecting and locating even a single network performance problem typically requires tens to hundreds of hours of the developers' time. They collect detailed application logs (too heavy-weight to run continuously), deploy dedicated packet sniffers (too expensive to run ubiquitously), or sample the data (too coarse-grained to catch performance problems). They then pore over these logs and traces using a combination of manual inspection and custom-crafted analysis tools to attempt to track down the issue. Often the investigation fails or runs out of time, and some performance problems persist for months before they are finally caught and corrected.

4.1.2 Lightweight, Continuous Profiling

In this chapter, we argue that the data centers should *continuously* profile network performance, and analyze the data in real time to help pinpoint the source of the problems. Given the complexity of data-center applications, we cannot hope to fully automate the detection, diagnosis, and repair of network performance problems. Instead, our goal is dramatically reducing the demand for developer time by automatically identifying performance problems and narrowing them down to specific times and places (*e.g.*, send buffer, delayed ACK, or network congestion). Any viable solution must be

- **Lightweight:** Running everywhere, all the time, requires a solution that is very lightweight (in terms of CPU, storage, and network overhead), so as not to degrade application performance.
- **Generic:** Given the constantly changing nature of the applications, our solution must detect problems without depending on detailed knowledge of the application or its log formats.
- **Precise:** To provide meaningful insights, the solution must pinpoint the component causing network performance problems, and tease apart interactions between the application and the network.

Finally, the system should help two very different kinds of users: (i) a *developer* who needs to detect, diagnose, and fix performance problems in his particular application and (ii) a *data-center* *operator* who needs to understand performance problems with the underlying platform so that he can tune the network stack, change server placement, or upgrade network equipment. In this paper, we present *SNAP (Scalable Network-Application Profiler)*, a tool that enables application developers and data-center operators to detect and diagnose these performance problems. SNAP represents an "existence proof" that a tool meeting our three requirements can be built, deployed in a production data center, and provide valuable information to both kinds of users.

SNAP capitalizes on the unique properties of modern data centers:

- SNAP has *full knowledge* of the network topology, the network-stack configuration, and the mapping of applications to servers. This allows SNAP to use correlation to identify applications with frequent problems, as well as congested resources (*e.g.*, hosts or links) that affect multiple applications.

- SNAP can instrument the *network stack* to observe the evolution of TCP connections directly, rather than trying to infer TCP behavior from packet traces. In addition, SNAP can collect finer-grain information, compared to conventional SNMP statistics, without resorting to packet monitoring.

In addition, once the developers fix a problem (or the operator tunes the underlying platform), we can verify that the change truly did improve network performance.

4.1.3 SNAP Research Contributions

SNAP passively collects TCP statistics and socket-level logs in real time, classifies and correlates the data to pinpoint performance problems. The profiler quickly identifies the right location (end host, link, or switch), the right layer (application, network stack, or network), at the right time. Our major contributions of the chapter are:

Efficient, systematic profiling of network-application interactions: SNAP provides a simple, efficient way to detect performance problems through real-time analysis of passively-collected measurements of the network stack. We provide a systematic way to identify the component (*e.g.*, sender application, send buffer, network, or receiver) responsible for the performance problem. SNAP also correlates across connections that belong to the same application, or share underlying resources, to provide more insight into the sources of problems.

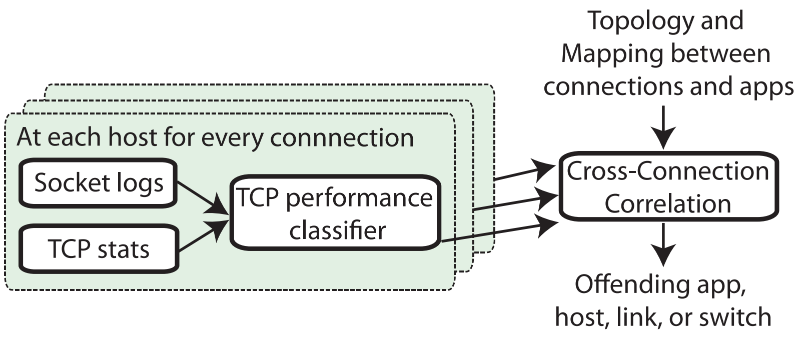


Figure 4.1: SNAP socket-level monitoring and analysis

Performance characterization of a production data center: We deployed SNAP in a data center with over 8,000 servers, and over 700 application components (including map-reduce, storage, database, and search services). We characterize the sources of performance problems, which helps data-center operators improve the underlying platform and better tune the network.

Case studies of performance bugs detected by SNAP: SNAP pinpointed 15 significant and unexpected problems in application software, the network stack, and the interaction between the two. SNAP saved the developers significant effort in locating and fixing these problems, leading to large performance improvements.

Section 4.2 presents the design and development of SNAP. Section 4.3 describes our data-center environment and how SNAP was deployed. Section 4.4 validates SNAP against both labeled data (*i.e.*, known performance problems) and detailed packet traces. Then, we present an evaluation of our one-week deployment of SNAP from the viewpoint of both the data-center operator (Section 4.5) and the application developer (Section 4.6). Section 4.7 shows how to reduce the overhead of SNAP through dynamic tuning of the polling rate. Section 4.8 discusses related work and Section 4.9 concludes the section.

4.2 Design of the SNAP Profiler

In this section, we describe how SNAP pinpoints performance problems. Figure 4.1 shows the main components of our system. First, we collect *TCP-connection statistics*, augmented by *socket-level logs* of application read and write operations, in real time with low overhead. Second, we run a

TCP classifier that identifies and categorizes periods of bad performance for each socket, and logs the diagnosis and a time sequence of the collected data. Finally, based on the logs, we have a *centralized correlator* that correlates across connections that share a common resource or belong to the same application to pinpoint the performance problems.

4.2.1 Socket-Level Monitoring of TCP

Data centers host a wide variety of applications that may use different communication methods and design patterns, so our techniques must be quite general in order to work across the space. The following three goals guided the design of our system, and led us *away* from using the SNMP statistics, packet traces, or application logs.

(i) Fine-grained profiling: The data should be fine-grained enough to indicate performance problems for individual applications on a small timescale (e.g, tens of milliseconds or seconds). Switches typically only capture link loads at a one-minute timescale, which is far too coarse-grained to detect many performance problems. For example, the TCP incast problem [67], caused by micro bursts of traffic at the timescale of tens of milliseconds, is not even visible in SNMP data.

(ii) Low overhead: Data centers can be huge, with hundreds of thousands of hosts and tens of thousands sockets on each host. Yet, the data collection must not degrade application performance. Packet traces are too expensive to capture in real time, to process at line speed, or to store on disk. In addition, capturing packet traces on high-speed links (*e.g.*, 1-10 Gbps in data centers) often leads to measurement errors including drops, additions, and resequencing of packets [68]. Thus it is impossible to capture packet trace everywhere, all the time to catch new performance problems.

(iii) Generic across applications: Individual applications often generate detailed logs, but these logs differ from one application to another. Instead, we focus on measurements that do not require application support so our tool can work across a variety of applications.

Through our work on SNAP, we found that the following two kinds of per-socket information can be collected cheaply enough to be used in analysis of large-scale data center applications, while still providing enough insight to diagnose where the performance problem lie (whether they are from the application software, from network issues, or from the interaction between the two).

Statistic	**Definition**
CurAppWQueue	# of bytes in the send buffer
MaxAppWQueue	Max # of bytes in send buffer over the entire socket lifetime
#FastRetrans	Total # of fast retransmissions
#Timeout	Total # of timeouts
#SampleRTT	Total # of RTT samples
#SumRTT	Sum of RTTs that TCP samples
RwinLimitTime	Cumulated time an application is receiver window limited
CwinLimitTime	Cumulated time an application is congestion window limited
SentBytes	Cumulated # of bytes the socket has sent over the entire lifetime
Cwin	Current congestion window
Rwin	Announced receiver window

Table 4.1: Key TCP-level statistics for each socket [1]

Locations	**Problems**	**App/Net**	**Detection method**
Sender app	Sender app limited	App	Not any other problems
Send buffer	Send buffer limited	App and Net	$CurAppWQueue \approx MaxAppWQueue$
Network	Fast retransmission	Net	*diff(#FastRetrans) > 0*
	Timeout	Net	*diff(#Timeout) > 0)*
Receiver	Delayed ACK	App and Net	*diff(SumRTT) > diff(SampleRTT) * MaxQueuingDelay*
	Receiver window limited	App and Net	*diff(#RwinLimitTime) > 0*

Table 4.2: Classes of network performance for a socket

TCP-level statistics: RFC 4898 [1] defines a mechanism for exposing the internal counters and variables of a TCP state-machine that is implemented in both Linux [69] and Windows [70]. We select and collect the statistics shown in Table 4.1 based on our diagnosis experience[2], which together expose the data-transfer performance of a socket. There are two types of statistics: (1) instantaneous snapshots (*e.g.*, *Cwin*) that show the current value of a variable in the TCP stack; and (2) cumulative counters (*e.g.*, *#FastRetrans*) that count the number of events (*e.g.*, the number of fast retransmissions) that happened over the lifetime of the socket. *#SampleRTT* and *SumRTT* are the cumulative values of the number of packets TCP sampled and the sum of the RTTs for these sampled packets. To calculate the retransmission timeout (RTO), TCP randomly

[2]There are a few other variables in the TCP stack such as the time TCP spends in SlowStart stage, which are also useful but we did not mention for simplicity.

samples one packet in each congestion window, and measures the time from the transmission of a packet to the time TCP receives the ACK for the packet as the RTT for this packet.

These statistics are updated by the TCP stack as individual packets are sent and received, making it too expensive to log every change of these values. Instead, we periodically poll these statistics. For the cumulative counters, we calculate the difference between two polls (*e.g.*, *diff(#FastRetrans)*). For snapshot values, we sample with a Poisson interval. According to the PASTA property (Poisson Arrivals See Time Averages), the samples are a representative view of the state of the system.

Socket-call logs: Event-tracing systems in Windows [71] and Linux [72] record the time and number of bytes (*ReadBytes* and *WriteBytes*) whenever the socket makes a read/write call. Socket-call logs show the applications' data-transfer behavior, such as how many connections they initiated, how long they maintain each connection, and how much data they read/write (as opposed to the data that TCP actually transfers, *i.e.*, *SentBytes*). These logs supplement the TCP statistics with application behavior to help developers diagnose problems. The socket-level logs are collected in an event-driven fashion, providing fine-grained information with low overhead. In comparison, the TCP statistics introduce a trade-off between accuracy and the polling overhead. For example, if SNAP polls TCP statistics once per second, a short burst of packet losses is hard to distinguish from a modest loss rate throughout the interval.

In summary, SNAP collects two types of data in the following formats: (i) timestamp, 4-tuples (source and destination address/port), *ReadBytes*, and *WriteBytes*; and (ii) timestamp, 4-tuples, TCP-level logs (Table 4.1). SNAP uses TCP-level logs to classify the performance problems and pinpoint the location of the problem, and then provides both the relevant TCP-level and socket-level logs for the affected connections for that period of time. Developers can use these logs to quickly find the root cause of performance problems.

4.2.2 Classifying Single-Socket Performance

Although it is difficult to determine the root cause of performance problems, we can pinpoint the component that is limiting performance. We classify performance problems in terms of the stages of data delivery, as summarized in the two columns of Table 4.2[3]:

1. Application generates the data: The sender application may not generate the data fast enough, either by design or because of bottlenecks elsewhere (*e.g.*, CPU, memory, or disk). For example, the sender may write a small amount of data, triggering Nagle's algorithm [73] which combines small writes together into larger packets for better network utilization, at the expense of delay.

2. Data are copied from the application buffer to the send buffer: Even when the network is not congested, a small send buffer can limit throughput by stalling application writes. The send buffer must keep data until acknowledgments arrive from the receiver, limiting the buffer space available for writing new data.

3. TCP sends the data to the network: A congested network may drop packets, leading to lower throughput or higher delay. The sender can detect packet loss by receiving three duplicate ACKs, leading to a fast retransmission. When packet losses do not trigger a triple duplicate ACK, the sender must wait for a retransmission timeout (RTO) to detect loss and retransmit the data.

4. Receiver receives the data and sends an acknowledgment: The receiver may not read data, or acknowledge their arrival, quickly enough. The receiver window can limit the throughput if the receiver is not reading the data quickly enough (*e.g.*, caused by a CPU starvation), allowing data to fill the receive buffer. A receiver delays sending acknowledgments in the hope of piggybacking the ACK on data in the reverse direction. The receiver acknowledges every other packet and waits up to 200 ms before sending an ACK.

The TCP statistics provide direct visibility into certain performance problems like packet loss and receiver-window limits, where cumulative counts (*e.g.*, *#Timeout*, *#FastRetrans*, and *RwinLimitTime*) indicate whether the problem occurred at any time during the polling interval. Detecting other problems relies on an instantaneous snapshot, such as comparing the current backlog of the send buffer (*CurAppWQueue*) to its maximum size (*MaxAppWQueue*); polling with

[3]The table only summarizes major performance problems and can be extended to cover other problems such as out-of-order packets.

a Poisson distribution allows SNAP to accurately estimate the fraction of time a connection is send-buffer limited. Pinpointing other latency problems requires some notion of expected delays. For example, the RTT should not be larger than the propagation delay plus the maximum queuing delay (*MaxQueuingDelay*) (whose value is measured in advance by operators), unless a problem like delayed ACK occurs. SNAP incorporates knowledge of the network configuration to identify these parameters.

SNAP detects send-buffer, network, and receiver problems using the rules listed in the last column of Table 4.2, where multiple problems may take place for the same socket during the same time interval. If any of these problems are detected, SNAP logs the diagnosis and all the variables in Table 4.1—as well as *WriteBytes* from the socket-call data—to provide the developers with detailed information to track down the problem. In the absence of any of the previous problems, we classify the connection as sender-application limited during the time interval, and log only the socket-call data to track application behavior. Being sender-application limited should be the most common scenario for a connection.

4.2.3 Correlation Across TCP Connections

Although SNAP can detect performance problems on individual connections in isolation, combining information across multiple connections helps pinpoint the location of the problem. As such, a central controller analyzes the results of the TCP performance classifier, as shown earlier in Figure 4.1. The central controller can associate each connection with a particular application and with shared resources like a host, links, and switches.

Pinpointing resource constraints (by correlating connections that share a host, link, or switch): Topology and routing data allow SNAP to identify which connections share resources such as a host, link, top-of-rack switch, or aggregator switch. SNAP checks if a performance problem (as identified by the algorithm in Table 4.2) occurs on many connections traversing the same resource at the same time. For example, packet losses (*i.e.*, $diff(\#FastRetrans) > 0$ or $diff(\#Timeout) > 0$) on multiple connections traversing the same link would indicate a congested link. This would detect congestion occurring on a much smaller timescale than SNMP could measure. As another example, send-buffer problems for many connections on the same host could indicate that the machine has insufficient memory or a low default configuration of the send-buffer size.

Pinpoint application problem (by correlating across connections in the same application): SNAP also receives a mapping of each socket (as identified by the four-tuple) to an application. SNAP checks if a performance problem occurs on many connections from the same application, across different machines and different times. If so, the application software may not interact well with the underlying TCP layer. With SNAP, we have found several application programs that have severe performance problems and are currently working with developers to address them, as discussed in Section 4.6.

The two kinds of correlation analysis are similar, except for (i) sets of connections to compare S (*i.e.*, connections sharing a resource vs. belonging to the same service) and (ii) the timescale for the comparison — correlation interval T (*i.e.*, transient resource constraining events taking a few minutes or hours vs. permanent service code problems that lasts for days).

We use a simple linear correlation heuristic that works well in our setting Given a set of connections S and a correlation interval T, the SNAP correlation algorithm outputs whether these connections have correlated performance problems, and provides a time sequence of SNAP logs for operators and developers to diagnose.

We construct a performance vector $\overrightarrow{P_T(c,t)} = (time_k(p_1,c), ..., time_k(p_5,c))_{k=1..\lceil T/t \rceil}$, where t is an aggregation time interval in T and $time_k(p_i)(i = 1..5)$ denotes the total time that connection c is having problem p_i during time period $[(k-1)t, kt]$.[4] We pick c_1 and c_2 in S, calculate the Pearson correlation coefficient, and check if the average across all pairs of connections (*Average Correlation Coefficient ACC*) is larger than a threshold α:

$$ACC = \underset{c_1,c_2 \in S, c_1 \neq c_2}{avg} (cor(\overrightarrow{P_T(c_1,t)}, \overrightarrow{P_T(c_2,t)}) > \alpha,$$

where

$$cor(\overrightarrow{x}, \overrightarrow{y}) = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 (y_i - \bar{y})^2}}.$$

[4] $p_i(i = 1..5)$ are the problems of send buffer limited, fast retransmission, timeout, delayed ACK and receiver window limited respectively. We do not include sender application limited because its time could be determined given the times of the first five problems.

Characteristic	**Value**
#Hosts	8K
#Applications	700
Operating systems	Win 2003,2008R2
Default send buffer	8 KB
Maximum segment size (MSS)	1460 Bytes
Minimum retrans. timeout	300 ms
Delayed ACK timeout	200 ms
Nagle's algorithm	mostly off
Slow start restart	off
Receiver window autotuning	off

Table 4.3: Characteristics in the production data center.

If the correlation coefficient is high, SNAP reports that the connections in S have a common problem. To extend this correlation for different classes of problems (*e.g.*, one connection's delayed ACK problem triggers the sender application limited problem on another connection), we can extend our solution to use other inference techniques [74, 75] or principal component analysis (PCA) [76].

In practice, we must choose t carefully. With a large value of t, SNAP only compares the coarse-grained performance between connections; for example, if $t = T$, we only check if two connections have the same performance problem with the same percentage of time. With a small t, SNAP can detect fine-grained performance problems (*e.g.*, two connections experiencing packet loss at almost the same time), but are susceptible to clock differences of the two machines and any differences in the polling rates for the two connections. The aggregation interval t should be large enough to mask the differences between the clocks and cannot be smaller than the least common multiple of the polling intervals of the connections.

4.3 Production Data Center Deployment

We deployed SNAP in a production data center. This section describes the characteristics of the data center and the configuration of SNAP, to set the stage for the following sections.

4.3.1 Data Center Environment

The data center consists of 8K hosts and runs 700 application components, with the configuration summarized in Table 4.3. The hosts run either Windows Server 2008 R2 or Windows Server 2003. The default send buffer size is 8K, and the maximum segment size is 1460 Bytes. The minimum retransmission timeout for packet loss is set to 300 ms, and the delayed-acknowledgment timeout is 200 ms. These values in Windows OS are configured for Internet traffic with long RTT.

While the OS enables Nagle's algorithm (which combines small writes into larger packets) by default, most delay-sensitive applications disable Nagle's algorithm using the *NO_DELAY* socket option.

Most applications in the data center use persistent connections to avoid establishing new TCP connections whenever they have data to transmit. Slow-start restart is disabled to reduce the delay arising when applications transfer a large amount of data after an idle period over a persistent connection.

Receiver-window autotuning—a feature in Windows Server 2008 that allows TCP to dynamically tune the receiver window size to maximize throughput—is disabled to avoid bugs in the TCP stack (*e.g.*, [77]). Windows Server 2003 does not support this feature.

4.3.2 SNAP Configuration

We ran SNAP continuously for a week in August 2010. The polling interval for TCP statistics follows the Poisson distribution with an average inter-arrival time of 500 ms. We collected the socket-call logs for all the connections from and to the servers running SNAP. Over the week, we collected less than 1 GB on each host per day and the total is just terabytes of logs for the week. This is a very small amount of data compared to packet traces which take more than 180 GB per host per day at a 1 Gbps link, even if we just keep packet header information.

To identify the connections sharing the same switch, link, and application, we collect the information about the topology, routing, and the mapping between sockets and applications in the data center. We collect topology and routing information from the data center configuration files. To identify the mapping between the sockets and applications, we first run a script at each machine to identify the process that created each socket. We then map the processes to the application based on the configuration file for the application deployment.

To correlate performance problems across connections using the correlation algorithm we proposed in Section 4.2.3, we chose two seconds as the aggregation interval t to summarize the time on each performance problems to mask time difference between machines. To pinpoint transient resource constraints which usually last for minutes or hours, we chose one hour as the correlation interval T. To pinpoint problems from application code which usually last for days, we chose 24 hours as the correlation interval T. We chose the correlation threshold $\alpha = 0.4$.[5]

4.4 SNAP Validation

To validate the design of SNAP in Section 4.2 and evaluate whether SNAP can pinpoint the performance problems at the right place and time, we take two approaches: First, we inject a few known problems in our production data center and check if SNAP correctly catches these problems; Second, to validate the decision methods that use inference to determine the performance class in Table 4.2 rather than observing from TCP statistics directly, we compare SNAP results against packet traces.

4.4.1 Validation by Injecting Known Bugs

To validate SNAP, we injected a few known data-center networking problems and verified if SNAP correctly classifies those problems for each connection. Next, running our correlation algorithm on the SNAP logs of these labeled problems together with the other logs from the data center, SNAP correctly pinpointed *all* the labeled problems. For brevity, we first discuss two representative problems in detail and then show how SNAP pinpoints problematic host for each of them.

Problems in receive-window autotuning: We first injected a receiver-window autotuning problem: This problem happens when a Windows Server 2008 R2 machine initiates a TCP connection to a Windows Server 2003 machine with a SYN packet that requests the receiver window autotuning feature. But due to a bug in the TCP stack of the Windows Server 2003[6], the 2003

[5]It is hard to determine the threshold α in practice. Operators can choose the top n shared resources/application code to investigate their performance problems.

[6]This bug is later fixed with a patch, but some machines do not have the latest patch.

server does not parse the request for the receiver window autotuning feature correctly, and returns the SYN ACK packet with a wrong format. As a result, the 2008 server tuned its receiver window to four Bytes, leading to low throughput and long delay.

To inject this problem, we picked ten hosts running Windows 2008 in the data center and turn on their receiver window autotuning feature. Each of the ten hosts initiated TCP connections to a HTTP server running Windows 2003 to fetch 20 files of 5KB each from a host running Windows 2003.[7] It took the Windows 2003 server more than 5 seconds to transfer each 5KB file. SNAP correctly reported that all these connections are receiver window limited all the time, and SNAP logs showed that the announced receiver window size (*RWin*) is 4 Byte.

TCP incast: TCP incast [67] is a common performance problem in data centers. It happens when an *aggregator* distributes a request to a group of *workers*, and after processing the requests, these workers send the responses back at almost the same time. These responses together overflow the switch on the path and experience significant packet losses.

We wrote an application that generates a TCP incast traffic pattern. To limit the effect of our experiment to the other applications in the production data center, we picked 36 hosts under the same top-of-rack switch (TOR), used one host as the *aggregator* to send requests to the remaining 35 hosts which serve as *workers*. These workers respond with 100KB data immediately after they receive the requests. After receiving all the responses, the aggregator sends another request to the workers. The aggregator sends 20 requests in total.

SNAP correctly identified that seven of the 35 connections have experienced a significant amount of packet loss causing retransmission timeouts. This is verified from our application logs which show that it takes much longer time to get the response through the seven connections than the rest of the connections.[8]

Correlation to pinpoint resource constraints for the two problems: We mixed the SNAP logs of the receiver window autotuning problem and TCP incast with the logs of an hour period

[7]We ran ten hosts to the same 2003 server to validate if the SNAP can correlate these connections and pinpoint the server.

[8]In this experiment, SNAP can only tell that the connections have correlated timeouts. If the same problem happens for different aggregators running the same application code, we can tell that the application code causes the timeouts. If SNAP reports all the connections have simultaneous small writes (identified from socket call logs) and correlated timeouts, we can infer that the application code has incast problems.

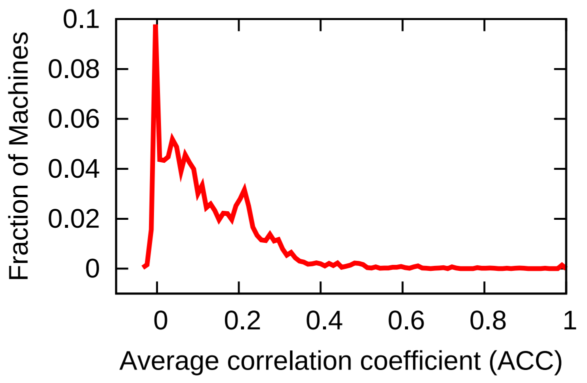


Figure 4.2: PDF of #Machines with different average correlation coefficient.

collected at all other machines in the data center. Then we ran SNAP correlation algorithm across the connections sharing the same machine.

SNAP correctly identified the Windows Server 2003 servers that have receiver-window limited problems across 5-10 connections with an average correlation coefficient (ACC) of 0.8. SNAP also correctly identified the aggregator machine because the ACC across all the connections that traverse the TOR is 0.45. Both are above the threshold $\alpha = 0.4$, which is chosen based on the discussion in Section 4.3.

Our correlation algorithm clearly distinguished the two injected problems with the performance of connections on the other machines in the data center. Figure 4.2 presents the probability density function (PDF) of the number of machines with different values of ACC. Only 2.7% of the machines have an ACC larger than 0.4. In addition to the two injected problems, the other machines with ACC > 0.4 may also indicate some problems that happen during our experiment, but we have not verified these problems yet.

4.4.2 Validation Against Packet Traces

We also need to validate the performance-classification algorithm defined in Table 4.2. The detection methods for the performance class of fast retransmissions, timeouts, receiver window limited is always accurate because these statistics are directly observed phenomena (*e.g.*, #Timeouts) from the TCP stack. The accuracy of identifying send buffer problems is closely related to the probability of detecting the moments when the send buffer is full in the Poisson sampling, which is well studied in [78].

There is a tradeoff between the overhead and accuracy of identifying delayed ACK. The accuracy of identifying the delayed ACK and small writes classes is closely related to the estimation of the RTT. However, we cannot get per-packet RTT from the TCP stack because it is a significant overhead to log data for each packet. Instead, we get the sum of estimated RTTs (*SumRTT*) and the number of sampled packets (*SampleRTT*) from the TCP stack.

We evaluate the accuracy of identifying delayed ACK in SNAP by comparing SNAP's results with the packet trace. We picked two real-world applications from the production data center for which SNAP detects delayed ACK problems: One connection serves as an aggregator distributing requests for a Web application that has the delayed ACK problems for 100% of the packets[9]. Another belongs to a configuration-file distribution service for various jobs running in the data center, which has 75% of the packets on average experiencing delayed ACK. While running SNAP with various polling rates, we captured packet traces simultaneously. We then compared the results of SNAP with the number of delayed-ACK incidents we identify from packet traces.

To estimate the number of packets that experience delayed ACK, SNAP should find a distribution of RTTs for the sampled packets that sum up to *SumRTT*. Those packets that experience delayed ACK have a RTT around *DelayedACKTimeout*. The rest of the packets all experience the maximum queuing delay. Therefore, we use the equation: *(diff(#SumRTT) − diff(#SampleRTT) * MaxQueuingDelay)/DelayedACKTimeout* to count the number of packets experiencing delayed ACK. We use *MaxQueuingDelay* = 10 ms and *DelayedACKTimeout* = 180 ms. The delayed timeout is set as 200 ms in TCP stack, but TCP timer is only accurate at 10 ms level and thus the real *DelayedACKTimeout* varies around 200 ms. So we use 180 ms to be conservative on the delayed ACK estimation.

Figure 4.3 shows the estimation error of SNAP's results which is defined by $(d_t - d_s)/d_t$, where d_s is the percentage of packets that experience delayed ACK reported by SNAP and d_t is the actual percentage of delayed ACK we get from the packet trace. For the application that always has delayed ACK, SNAP's estimation error is 0.006 on average. For the application that has 75%

[9]This application distributes requests whose size is smaller than MSS (*i.e.*, one packet), and waits more than the delayed ACK timeout 200 ms before sending out another request. So the receiver has to keep each packet for 200 ms before sending the ACK to the sender.

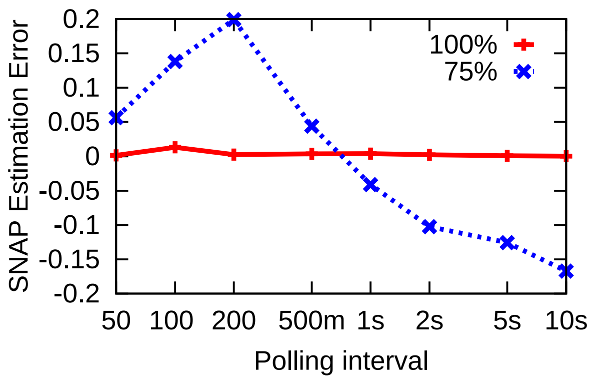


Figure 4.3: SNAP estimation error of identifying delayed ACK problems.

of packets experiencing delayed ACK, the estimation error is within 0.2 for the polling intervals that range from 500 ms to 10 sec.

Figure 4.3 shows that the estimation error drops from positive (underestimation) to negative (overestimation) with the increase of the polling interval. When the polling interval is smaller than 200 ms, there is at most one packet experiencing delayed ACK in one polling interval. If a few packets take less than *MaxQueuingDelay* to transfer, we would overestimate the part of *SumRTT* that is contributed by these packets, and thus the rest of RTT is less than *DelayedACKTimeout*. When the polling interval is large, there are more packets experiencing delayed ACK in the same time interval. Since we have use 180 ms instead of 200 ms to detect delayed ACK, we would underestimate those packets that take longer than 180 ms delayed ACK. Nine such packets would contribute enough RTT for SNAP to assume one more delayed ACK.

4.5 Profiling Data Center Performance

We deployed SNAP in the production data center to characterize different classes of performance problems, and provided information to the data-center operators about problems with the network stack, network congestion or the interference between services. We first characterize the frequency of each performance problem in the data center, and then discuss the key performance problems in our data center—packet loss and the TCP send buffer.

	% of conn. with prob. for >X% of time					**#Apps with prob. for >X% of time**	
Performance limitation	**>0**	**>25%**	**>50%**	**>75%**	**>99.9%**	**> 5%**	**> 50%**
Sender app limited	97.91%	96.62%	89.61%	59.21%	32.61%	561	557
Send buffer limited	0.45%	0.06%	0.02%	0.01%	0.01%	1	1
Congestion	1.90%	0.46%	0.22%	0.17%	0.15%	30	6
Receiver window limited	0.82%	0.36%	0.21%	0.15%	0.11%	22	8
Delayed ACK	65.71%	33.20%	10.10%	3.21%	1.82%	154	144
(belong to delay sensitive apps)	63.52%	32.82%	9.71%	3.01%	1.61%	136	129

Table 4.4: Percentage of connections and number of applications that have different TCP performance limitations.

4.5.1 Frequency of Performance Problems

Table 4.4 characterizes the frequency of the network performance problems (defined in Table 4.2) in our data center. Not surprisingly, the overall network performance of the data center is good. For example, only 0.82% of all the connections were receiver limited during their lifetimes. However, there are two key problems that the operators should address:

Operators should focus on the small fraction of applications suffering from significant performance problems. Several connections/applications have severe performance problems. For example, about 0.11% of the connections are receiver-window limited essentially all the time. Even though 0.11% sounds like a small number, when 8K machines are each running many connections, there is almost always some connection or application experiencing bad performance. These performance problems at the "tail" of the distribution also constrain the total load operators are willing to put in the data center. Operators should look at the SNAP logs of these connections and work with the developers to improve the performance of these connections so that they can safely "ramp up" the utilization of the data center.

Operators should disable delayed ACK, or significantly reduce Delayed ACK timeout: About two-thirds of the connections experienced delayed ACK problems. Nearly 2% of the connections suffer from delayed-ACKs for more than 99.9% of the time. We manually explore the delay-sensitive services, and count the percentage of connections that have delayed ACK. Unfortunately, about 136 delay-sensitive applications have experienced delayed ACKs. Packets that have delayed ACK would experience an unnecessary increase of latency by 200 ms, which is three orders of magnitude larger than the propagation delay in the data center and well exceeds the latency bounds for these applications. Since delayed ACK causes many problems for data-center applications, the operators are considering disabling delayed ACK or significantly reducing the de-

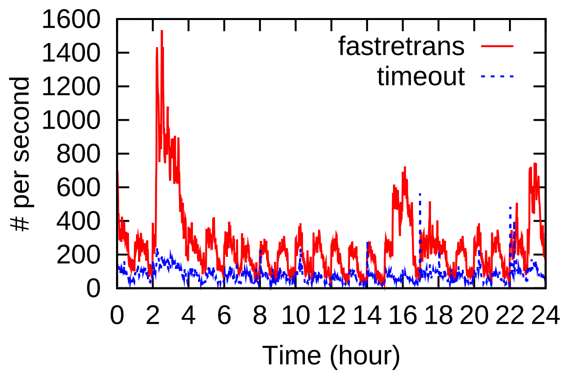


Figure 4.4: # of fast retransmissions and timeouts over time.

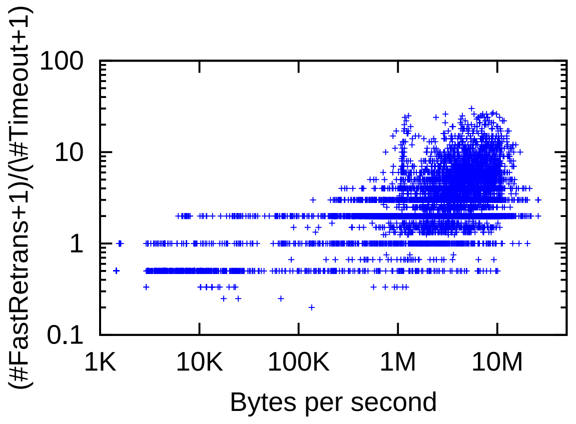


Figure 4.5: Comparing #FastRetrans and #Timeouts of flows with different throughput.

layed ACK timeout. The problems of delayed ACK for data center applications are also observed in [79].

4.5.2 Packet Loss

Operators should schedule backup jobs more carefully to avoid triggering network congestion: Figure 4.4 shows the number of fast retransmissions and timeouts per second over time. The percentage of retransmitted bytes increases between 2 am and 4 am. This is because most backup applications with large bulk transfers are initiated in this time period.

Operators should reduce the number and effect of packet loss (especially timeouts) for low-rate flows: SNAP data shows that about 99.8% of the connections have low throughput ($<$ 1 MB/sec). Although these low-rate flows do not consume much bandwidth and are usually not the cause of network congestion, they are significantly affected by network congestion. Figure 4.5 is a scatter plot that shows the ratio of of fast retransmissions to timeouts vs. the connection sending rate. Each point in the graph represents one polling interval of one connection. Low-rate flows usually experience more timeouts than fast retransmission because they do not have multiple packets in flight to trigger triple duplicate ACKs. Timeouts, in turn, limit the throughput of these flows. In contrast, high-rate flows experience more fast retransmission than timeouts and can quickly recover from packet losses achieving higher throughput (> 1 MB/sec).

4.5.3 Send Buffer and Receiver Window

Operators should allow the TCP stack to automatically tune the send buffer and receiver window sizes, and consider the following two factors:

More send buffer problems on machines with more connections: SNAP reports correlated send buffer problems on hosts with more than 200 connections. This is because the larger the send buffer for each connection, the more memory is required for the machine. As a result, the developers of different applications on the same machine are cautious it setting the size of the send buffer; most use the default size of 8K, which is far less than the delay-bandwidth product in the data center and thus is more likely to become the performance bottleneck.

Mismatch between send buffer and receiver window size: SNAP logs the announced receiver window size when the connection is receiver limited. From the log we see that 0.1% of the total time when the senders indicate that their connections are bottlenecked by the receiver window, the receiver actually announced a 64 KB window. This is because the send buffer is larger than the announced receiver size, so the sender is still bottlenecked by the receiver window.

To fix the send-buffer problems in the short term, SNAP could help developers to decide what send buffer size they should set in an online fashion. SNAP logs the congestion window size (*CWin*), the amount of data the application expect to send (*WriteBytes*), and the announced receiver window size (*RWin*) for all the connections. Developers can use this information to size the send buffer based on the total resources (*e.g.*, set the send buffer size to $Cwin_{thisconn} * TotalSendBufferMemory / \sum CWin$). They can also evaluate the effect of their change using SNAP. In the long term, operators should have the TCP stack automatically tune both the send-buffer and receiver-window sizes for all the connections (*e.g.*, [69]).

4.6 Performance Problems Caught by SNAP

In this section, we show a few examples of performance problems caught by SNAP. In each example, we first show how the performance problem is exposed by SNAP's analysis of socket and TCP logs into performance classifications and then correlation across connections. Next, we explain how SNAP's reports help guide developers to identify quickly the root causes. Finally, while outside of SNAP's scope, for interest value we discuss the developer's fix or proposed fix to these problems. For most examples, we spent a few hours or days discuss with developers to understand how their programs work and to discover how their programs cause the problems SNAP detects. It then took several days or weeks to iterate with developers and operators to find out the possible alternative ways to achieve their programing goals.

4.6.1 Sending Pattern/Packet Loss Issues

Spreading application writes over multiple connections lowers throughput: When correlating performance problems across connections from the same application, SNAP found one application whose connections always experienced more timeouts (*diff(#Timeout)*) than fast retransmission (*diff(#FastRetrans)*) especially when the *WriteBytes* is small. For example, SNAP reported repeated periods where one connection transferred an average of five requests per second with a size of 2 KB - 20 KB, while experiencing approximately ten timeouts but no fast retransmissions.

The developers were expecting to obtain far more than five requests per second from their system, and when this report showing small writes and timeouts was shown to them the cause became clear. The application sends requests to a server and waits for responses. Since some requests take longer to process than others and developers wanted to avoid having to implement request IDs while still avoiding head-of-line blocking, they open two connections to the server and place new requests on whichever connection is unused.

However, spreading the application writes over two connections meant that often there were not enough outstanding data on a connection to cause three duplicate ACKs and trigger fast retransmission when a packet was lost. Instead, TCP fell back to its slower timeout mechanism.

To fix the problem, the application could send all requests over a single connection, give requests a unique ID, and use pools of worker threads at each end.[10] This would improve the chances there is enough data in flight to trigger fast retransmission when packet loss occurs.

Congestion window failing to prevent sudden bursts: SNAP discovered that some connections belonging to an application frequently experience packet loss (*#FastRetrans* and *#Timeout* are both high, and correlate strongly to the application and across time). SNAP's logs expose a time sequence of socket write logs (*WriteBytes*) and TCP statistics (*Cwin*) showing that before/during the intervals where packet loss occurs, there is a single large socket write call after an idle period. TCP immediately sends out the data in one large chunk of packets because the congestion window is large, but it experiences packet losses. For example, one application makes a socket call with *WriteBytes* $>$ *100 MB* after an idle period of 3 seconds, the *Cwin* is 64 KB, and the traffic burst leads to a bunch of packet losses.

The developers told us they use a persistent connection to avoid three-way handshake for each data transfer. Since "slow start restart" is disabled, the congestion window size does not age out and remains constant until there is a packet loss. As a result, the congestion window no longer indicates the carrying capacity of the network, and losses are likely when the application suddenly sends a congestion window worth of data.

Interestingly, the developers are opposed to enabling slow start restart, and they intentionally manipulate the congestion window in an attempt to reduce latency. For example, if they send 64 KB data, and the congestion window is small (*e.g.*, 1 MSS), they need at multiple round-trip times to finish the data transfer. But if they keep the congestion window large, they can transfer the data with one RTT. In order to have a large congestion window, they first make a few small writes when they set up the persistent connection.

To reduce both the network congestion and delay, we need better scheduling of traffic across applications, allowing delay-sensitive applications to send traffic bursts when there is no network

[10]Note that the application should use a single connection because its requests are relatively small. For those applications that have a large amount of data to transfer for each request, they still have to use two connections to avoid head of line blocking during the network transfer.

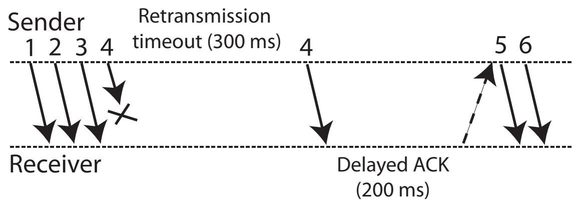


Figure 4.6: Delayed ACK after a retransmission timeout.

congestion, but pacing the traffic if the network is highly used. The feedback mechanism proposed in DCTCP [80] could be applied here.

Delayed ACK slows recovery after a retransmission timeout: SNAP found that one application frequently had two problems (timeout and delayed ACK) at almost the same time. As shown in Figure 4.6, when the fourth packet of the transferred data is lost, the TCP sender waits for a retransmission timeout (because there are not enough following packets to trigger triple-duplicate ACKs). However, the congestion window drops to one after the retransmission. As a result, TCP can only send a *single* packet, and the receiver waits for a delayed ACK timeout before acknowledging the packet. Meanwhile, the sender cannot increase its sending window until it receives the ACK from the receiver. To avoid this, developers are discussing the possibility of dropping the congestion window down to two packets when a retransmission timeout occurs. Disabling delayed ACK is another option.

4.6.2 Buffer Management and Delayed ACK

Some developers do not manage the application buffer and the socket send buffer appropriately, leading to bad interactions between buffer management and delayed ACK.

Delayed ACK caused by setting send buffer to zero: SNAP reports show that some applications have delayed ACK problems most of the time and these applications had set their send socket buffer length to 0. Investigation found that these applications set the size of the socket send buffer to zero in the expectation that it will decrease latency because data is not copied to a kernel socket buffer, but sent directly from the user space buffer. However, when send buffer is zero, the socket layer locks the application buffer until the data is ACK'd by the receiver so that the socket can retransmit the data in case a packet is lost. As a result, additional socket writes are blocked until the previous one has finished.

Whenever an application writes data that results in an odd number of packets being sent, the last packet is not ACK'd until the delayed ACK timer expires. This effectively blocks the sending application for 200 ms and can reduce application throughput to 5 writes per second. One team attempted to improve application performance by shrinking the size of their messages, but ended up creating an odd number of packets and triggering this issue — destroying the application's performance instead of helping it. After the developers increased the send buffer size, throughput returned to normal.

Delayed ACK affecting throughput: SNAP reports showed that an application was writing small amounts of data to the socket (*WriteBytes*) and its connections experienced both delayed ACK and sender application limited issues. For example, during 30 minutes, the application wrote 10K records at only five records per second and with a the record size of 20–100 Bytes.

The developers explained theirs is a logging application where the client uploads records to a server, and should be able generate far more than five records per second. Looking into the code with the developers, we found three key problems in the design: (i) *Blocking write:* to simplify the programming, the client does blocking writes and the server does blocking reads. (ii) *Small receive buffer:* The server calls recv() in a loop with a 200 bytes buffer in hopes that exactly one record is read in each receive call. (iii) *Send buffer is set to zero:* Since the application is delay-sensitive, the developer set send buffer size to zero. The application records are 20–100 Bytes — much less than the MSS of 1460 Bytes. Additionally, Nagle's algorithm forces the socket to wait for an ACK before it can send another packet (record).[11] As a result, the *single* packet containing each record always experience delayed ACK, leading to a throughput of only five records per second. To address this problem while still avoiding the buffer copying in memory, developers changed the sender code to write a group of requests each time. Throughput improved to 10K requests/sec after the change—a factor of 5000 improvement.

Delayed ACK affecting performance for pipelined applications: By correlating connections to the same machine, SNAP found two connections with performance problems that co-occur

[11]A similar performance problem caused by interactions between delayed ACK and Nagle is discussed in [73].

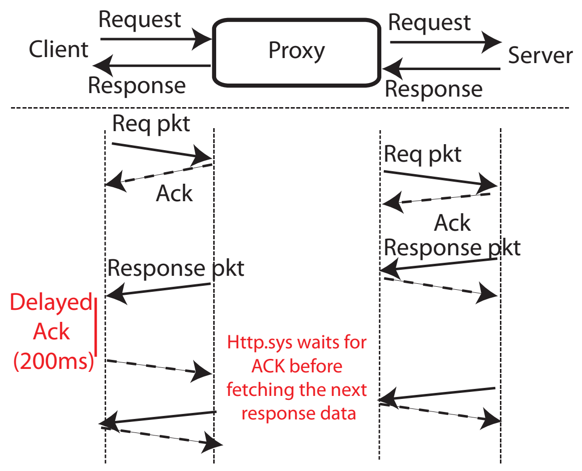


Figure 4.7: Performance problem in pipeline communication.

repeatedly: SNAP classified one connection as having a significant delayed ACK problem and the other as having sender application problems.

Developers told us that these two connections belong to the same application and form a pipeline pattern (Figure 4.7). There is a proxy that sits between the clients and servers and serves as a load balancer. The proxy passes requests from the client to the server, fetches a sequence of responses from the server, and passes them to the client. SNAP finds such a strong correlation between the delayed ACK problem and the receiver limited problem because both stem from the passing of the messages through the proxy.

After looking at the code, developers figured out that the proxy uses a single thread and a single buffer for both the client and the server. The proxy waits for the ACK of every transfer (one packet in each transfer most of the time) before fetching a new response data from the server.[12] When the developers changed the proxy to use two different threads with one fetching responses from the server and another sending responses to the client and a response queue between the two threads, the 99% tail of the request processing time drops from 200 ms to 10 ms.

[12]The proxy is using the HTTP.sys library without setting the HTTP_SEND_RESPONSE_FLAG_BUFFER_DATA flag [81], which waits for the ACK from the client before sending a “send complete” signal to the application. By waiting for the ACK, HTTP.sys can make sure the application send buffer is not overwritten until the data is successfully transferred.

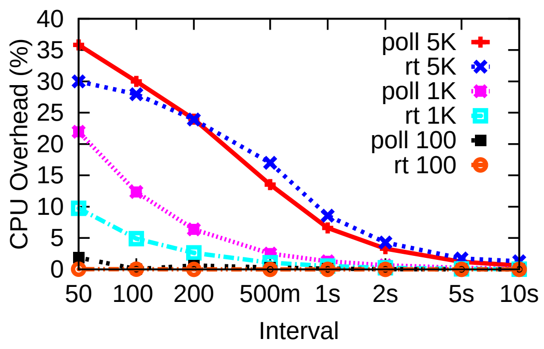


Figure 4.8: The CPU overhead of polling TCP statistics (*poll*) and reading TCP table (*rt*) with different number of connections (10, 100, 1K, 5K) and different intervals (from 50 ms to 10 sec).

4.6.3 Other Problems

SNAP has also detected other problems such as switch port failure (significant correlated packet losses across multiple connections sharing the same switch port), receiver window negotiation problems as reported in [77] (connections are always receiver window limited while receiver window size stays small), receiver not reading the data fast enough (receiver window limited), and poor latency caused by Nagle algorithm (sender application limited with small *WriteBytes*

4.7 Reducing SNAP CPU Overhead

To run in real time on all the hosts in the data center, SNAP must keep the CPU overhead and data volume low. The volume of data is small because (i) SNAP logs socket calls and TCP statistics instead of other high-overhead data such as packet traces and (ii) SNAP only logs the TCP statistics when there is a performance problem. To reduce CPU overhead, SNAP allows the operators to set the target percentage of CPU usage on each host. SNAP stays within a given CPU budget by dynamically tuning the polling rate for different connections.

CPU Overhead of Profiling Since SNAP collects logs for all the connections at the host, the overhead of SNAP consists of three parts: logging socket calls, reading the TCP table, and polling TCP statistics.

Logging socket calls: In our data center, the cost of turning on the event tracing for socket logging is a median of 1.6% of CPU capacity [82].

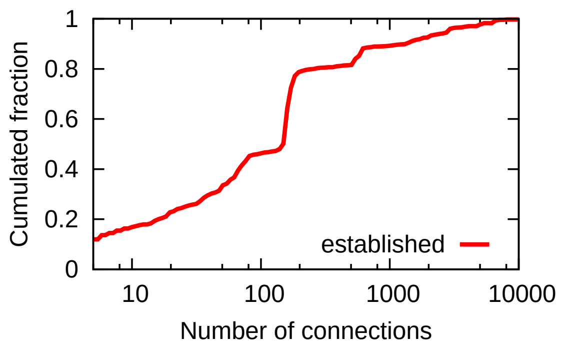


Figure 4.9: Number of connections per machine.

Polling CPU statistics and reading TCP table: The CPU overhead of polling TCP statistics and reading the TCP table depends on the polling frequency and the number of connections on the machine. Figure 4.8 plots the CPU overhead on a 2.5 GHz Intel Xeon machine. If we poll TCP statistics for 1K connections at 500 millisecond interval, the CPU overhead is less than 5%. The CPU overhead of reading the TCP table is similar.

The CPU overhead is closely related to the number of connections on each machine. Figure 4.9 takes a snapshot of the distribution of the number of established connections per machine. There are at most 10K established sockets and a median of 150. This means operators can configure the interval of reading TCP table in most machines to be 500 millisecond or one second to keep the CPU overhead lower than 5%.[13] Since most of the connections in our data center are long-lived connections (e.g., persistent HTTP connections), we can read the TCP table at a lower frequency compared to TCP statistics polling. For the machines with many connections, we need to carefully adjust the polling rate of TCP statistics for each connection to achieve a tradeoff between diagnosis accuracy and the CPU overhead.

Dynamic Polling Rate Tuning To achieve the best tradeoff between CPU overhead and accuracy, operators can first configure l_{CPU} (u_{CPU}) to be the lower (upper) bound of the CPU percentage used by SNAP. We then propose an algorithm to dynamically tune the polling rate for different connections to keep CPU overhead between the two bounds. The basic idea of the algorithm is to have high polling rate for those connections that are having performance issues and have low polling rate for the others.

[13]We read TCP tables at 500 millisecond interval in our data collection.

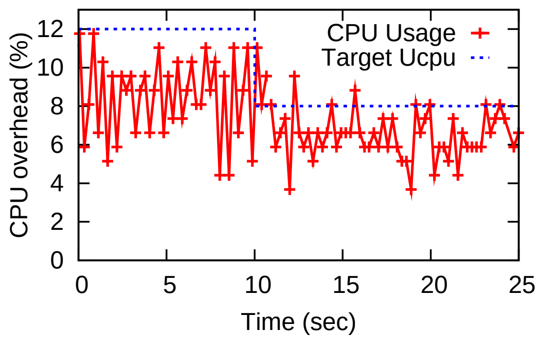


Figure 4.10: Adjust CPU overhead by dynamically tuning TCP polling rate on a machine with 500 connections ($\epsilon = 1$, $r = 2$).

We start polling all the connections on one host with the same rate (**Step 1**). Depending on the number of connections on the host, we set the polling rate pr_0 such that $\mathcal{F}(pr_0, \#Connections) < l_{CPU}$, where $\mathcal{F}(pr, \#Connections)$ is a function to model the polling overhead based on the measurements in Figure 4.8. For example, based on Figure 4.8, if a machine has 1K connections, we sample at 10 sec interval to keep the CPU overhead within 5%.

In **Step 2**, if the current CPU overhead is below l_{CPU}, we pick a connection that has the most performance problems in the past $T_{history}$ time, and increase its polling rate for more detailed data. we use $time_{[t_{now}-T_{history}, t_{now}]}(p, c)$ to record the time that connection c has problem p over the sliding time window of size $T_{history}$. Then we summarize all the $time(p, c)$ over all the performance classes p except sender application limited class that exist in the past $T_{history}$. Note that we also count the connections that closed in the past sliding window, to ensure we do not inadvertently "undercount" problems affecting short-lived flows. We pick the connection c with the largest $\sum_p(time_{[t_{now}-T_{history}, t_{now}]}(p, c))$ to increase its polling frequency by r.

Similarly in **Step 3**, we dynamically increase the polling rate by timing it with a constant factor r for those connections that do not have many problems. Figure 4.10 shows how fast the algorithm tunes the polling rate to keep the CPU overhead within the specified range of CPU overhead range. We pick a machine with 500 connections, let $l_{CPU} = 9\%$ and $u_{CPU} = 12\%$ at the beginning, and after 10 seconds switch the threshold to $l_{CPU} = 5\%$ and $u_{CPU} = 8\%$. It takes about 2.5 seconds for our algorithm to select the connections, increase the polling interval, and reduce the CPU overhead down to 8%.

4.8 Related Work

Previous work in diagnosing performance problems focuses on either the application layer or the network layer. SNAP addresses the interactions between them that cause particularly insidious performance issues.

In the application layer, prior work has taken several approaches: instrumenting application code [83, 84, 85] to find the causal path of problems, inferring the abnormal behaviors from history logs [74, 75], or identifying fingerprints of performance problems [86]. In contrast, SNAP focuses on profiling the interactions between applications and the network and diagnosing *network* performance problems, especially ones that arise from those interactions.

In the network layer, operators use network monitoring tools (e.g., switch counters) and active probing tools (ping, traceroute) to pinpoint network problems such as switch failures or congestion. To diagnose network performance problems, capture and analysis of packet traces remains the gold-standard. T-RAT [87] uses packet traces to diagnosis *throughput* bottlenecks in *Internet* traffic. Tcpanaly [68] uses packet traces to diagnose TCP stack problems. Others [88, 89] also infer the TCP performance and its problems from packet traces. In contrast, SNAP focuses on the multi-tier applications in data centers where it has access to the network stack, enabling us to create *simple* algorithms based on counters *far cheaper to collect than packet traces* to expose the network performance problems of the applications.

4.9 Summary

SNAP combines socket-call logs of the application's desired data-transfer behaviors with TCP statistics from the network stack that highlight the delivery of data. SNAP leverages the knowledge of topology, routing, and application deployment in the data center to correlate performance problems among connections, to pinpoint the congested resource or problematic software component.

Our experiences in the design, development, and deployment of SNAP demonstrate that it is practical to build a lightweight, generic profiling tool that runs continuously in the entire data center. Such a profiling tool can help both operators and developers in diagnosing network performance problems.

With applications in data centers getting more complex and more distributed, the challenges of diagnosing the performance problems between the applications and the network will only grow in importance in the years ahead. For future work, we hope to further automate the diagnosis process to save developers' efforts by exploring the appropriate variables to monitor in the stack, studying the dependencies between the variables SNAP collects, and combining SNAP reports with automatic analysis of application software.

Chapter 5

Conclusion

In conclusion, we propose a new management system that scales to many hosts, switches, and policies, which consists of three components: BUFFALO, DIFANE, and SNAP. In this chapter, we first discuss how these three components work together. Next, we summarize the key contributions of the thesis. The thesis ends with a discussion of open issues and future work.

5.1 The Overall Picture of the New Management Systems

In this section, we discuss combining BUFFALO, DIFANE, and SNAP into a single management system. As shown earlier in Figure 1.3 in Chapter 1, we have a centralized controller that configures the link weights for shortest path routing (BUFFALO), policies (DIFANE), and diagnoses performance problems (SNAP). BUFFALO and DIFANE are located at switches — with BUFFALO providing a scalable forwarding layer and DIFANE providing a way to scalably support policies. SNAP is located at each of the hosts collecting performance information for diagnosis. Now we will discuss in more details how the three key components work together.

The data plane behavior of BUFFALO and DIFANE are complementary: Our BUFFALO design provides a compact way to store the forwarding table (that maps the destination MAC addresses to the next hop) in the SRAM which performs exact match. In DIFANE, we store low-level rules for customized routing, access control, and measurement in the TCAM which performs wildcard matching.

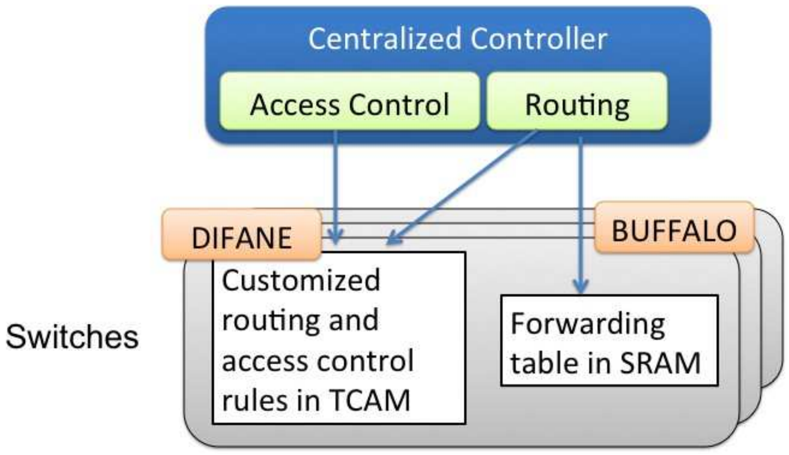


Figure 5.1: Combining BUFFALO and DIFANE functions in the same data plane.

When BUFFALO and DIFANE work together, we use DIFANE to handle customized forwarding rules that match the destination host address to the egress switch, and use BUFFALO to maintain the forwarding table that provides switch-to-switch reachability (Figure 5.1). We first match the packets based on the rules in TCAM, which defines whether the packets are to be dropped/forwarded and/or increments the counters. The routing rules in the TCAM of the ingress switch or the authority switch only decide the egress switch that a flow of packets should be forwarded. All the switches on the path match the packets based on the forwarding rules in SRAM to forward the packets towards the correct egress switch.

Redirection in DIFANE and BUFFALO do not affect the performance: Both BUFFALO and DIFANE leverage traffic redirection to overcome the limited memory at switches. With BUFFALO and DIFANE working together, a packet may be first redirected to an authority switch to hit the authority rules, and then redirected again at another switch in the path if it experiences a false positive in the compact forwarding table.

Such redirections would not be a problem because most enterprise and data center networks have a short end-to-end delay. Moreover, the chance of redirection is low because it only happens when a packet experiences false positives in the bloom filters in BUFFALO or it experiences a cache miss at the ingress switch in DIFANE. In general, it is a good tradeoff to save memory space at switches at the expense of slightly longer latency for enterprise and data center networks.

The centralized controller combines DIFANE and SNAP functions: In the new management system, the centralized controller is responsible for the interactions with operators. The controller performs two major tasks: In DIFANE, the controller will translate the high-level policies to low-level rules, partition the rules, and then enforce them in the TCAM of the switches; in SNAP, the controller collects the transport-layer statistics, socket-level logs, and the online diagnosis results from all the hosts, correlates the performance problems across connections with the knowledge of network topology, routing, and application deployment, and finally reports the potential problems to the operators and application developers.

5.2 Deployments in Enterprises and Data Centers Differ

Although the proposed management system works for both enterprise and data center networks, these two types of networks are different in topology, traffic pattern, and levels of control. As a result, we find that BUFFALO and DIFANE fit enterprise networks better and SNAP works better for data center networks. Here we discuss the differences between these two environments. We also discuss how to make BUFFALO and DIFANE work better for data centers and how to deploy SNAP in enterprises.

5.2.1 Differences Between Enterprise and Data Center Networks

There are three key differences between enterprise and data center networks:

Topology and traffic: Data centers usually have specific topologies such as a fat tree, torus, and clos, but enterprises can have any kind of topology. Therefore, for data centers much can be gained from leveraging the specific topology in the design of the network architecture, but for enterprises the design should work for any topology. Data centers can have a variety of traffic patterns depending on the applications. In contrast, enterprise networks are more overprovisioned than data centers. Therefore, it is fine to send traffic through longer paths but in data centers it is important to reduce the bandwidth overhead of traffic redirection. Enterprise traffic is also more sparse in traffic distribution, leading to more efficient caching of routing and access control rules.

Levels of control: In enterprises, hosts are heterogeneous and come and go frequently, so operators usually have little control over hosts. For example, in campuses and small companies, people can bring in different laptops and mobile devices every day, and it is hard to force the hosts to install any software or configuration.[1] However, hosts in data centers have similar types of hardware and are usually managed by a centralized controller. Therefore, enterprise networks sometimes cannot take advantage of end-host capabilities while data center networks can.

5.2.2 Customizing the Management System for Enterprises and for Data Centers

Now let us review the three components in our proposed management system and see how they fit in enterprises and data centers.

BUFFALO and DIFANE naturally fit enterprise networks because they do not assume specific topologies. Since the network is lightly loaded in enterprises, traffic indirection would not cause too much congestion in the network. Data centers have more severe scalability problems and require more scalable forwarding plane and policy enforcement schemes. In BUFFALO, we have shown that the bound of extra delay is much smaller in a tree topology, but we still need to prove a better bound for multi-rooted trees, which are common in data center. It is an interesting problem to see how we can redesign DIFANE to leverage the specific topologies in data centers.

SNAP was designed for data center applications so it can leverage the topology, routing, and the application deployment information to pinpoint the performance problems. We chose to deploy SNAP at hosts to make the monitoring system more scalable and to directly tell the interactions between applications and the network. In the enterprise environment, it is hard to deploy SNAP directly because operators may not have control on hosts to install SNAP, or may not trust the measurement results from the hosts. It would be interesting to find out ways to deploy SNAP at only the server end in enterprises, or just on proxies that understand TCP. It is also interesting to develop a secure way to run monitoring code at hosts so that hosts can trust the code and the operators can trust the measurement results.

[1]Large corporations have relatively more control on hosts because they can dictate what kinds of computers users use and what software they can run [28].

5.3 Summary of Contributions

In this section, we revisit the principles we introduced in Chapter 1, and summarize the major contributions we have in this thesis.

Placing the right functions at the right place: We revisit the placement of different network functions to achieve better scalability and flexibility. In DIFANE, we use the centralized controller to manage policies, but pull the function of rule processing back to the network, and distribute the rules among authority switches to achieve better performance and scalability. In SNAP, we collect diagnosis information and perform simple analysis at the hosts so that SNAP is more scalable than in-network measurements. We then use the centralized controller to perform correlation across the results from different hosts.

New algorithms and systems design: We propose new algorithms and data structures that fit in today's commodity switches and build new systems for enterprise and data center networks. In BUFFALO, we propose a Bloom filter based forwarding architecture that leverages the flat addresses and shortest path routing in edge networks. In DIFANE, we build a distributed directory service of rules to quickly match packets to rules without the involvement of the controller. We also design the wildcard rule partitioning algorithm which can efficiently handle a large number of rules. Finally, in SNAP, we build a simple diagnosis system at the hosts leveraging TCP-level statistics, and develop algorithms that correlate performance problems across connections sharing the same resources or code.

Prototypes and deployment: To validate our algorithms and systems design, we built prototypes of BUFFALO with the Click-based modular router, DIFANE with the Click implementation of OpenFlow, and SNAP with the Windows operating systems. SNAP has been deployed in a production data center and identified real performance problems there.

5.4 Concluding Remarks

It is an exciting time to study the design and management of edge networks. These networks are becoming more and more important to both people's lives and the daily operations of corporations. In addition, the edge networks are of high research interest because they are significantly different from the backbone networks. For example, these edge networks have different types of topology, different levels of control, and different types of traffic, which require us to rethink the design of networks from scratch in these new settings. Unlike the Internet, most of enterprise and datacenter networks are owned by a single entity and therefore are great opportunities for deploying these clean slate designs in practice.

For future work, we hope to further identify the operators' desires in edge networks and build better architectures to support them. We also hope to customize our proposed management system for enterprise and data center networks separately.

Bibliography

[1] http://www.ietf.org/rfc/rfc4898.txt.

[2] "Yankee group marketing reports," April 2008.

[3] T. Benson, A. Akella, and D. Maltz, "Unraveling the complexity of network management," in *NSDI*, 2009.

[4] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan, "A power benchmarking framework for network devices," in *IFIP Networking*, 2009.

[5] C. Kim, M. Caesar, and J. Rexford, "Floodless in SEATTLE: A scalable Ethernet architecture for large enterprises," in *Proc. ACM SIGCOMM*, 2008.

[6] M. Yu, X. Sun, N. Feamster, S. Rao, and J. Rexford, "A survey of VLAN usage in campus networks," in *IEEE Communications Magazine*, 2011.

[7] "IETF TRILL working group." http://www.ietf.org/html.charters/trill-charter.html.

[8] R. Perlman, "Rbridges: Transparent routing," in *Proc. IEEE INFOCOM*, 2004.

[9] http://www.ieee802.org/1/pages/802.1aq.html.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Toward an operating system for networks," *ACM Computer Communication Review*, July 2008.

[11] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM Computer Communication Review*, 2005.

[12] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai, "Tesseract: A 4D Network Control Plane," in *Proc. Networked Systems Design and Implementation*, Apr. 2007.

[13] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," in *IEEE/ACM Transactions on Networking*, 2009.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM Computer Communication Review*, Apr. 2008.

[15] "Anagran: Go with the flow." `http://anagran.com`.

[16] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations," in *CoNEXT*, 2009.

[17] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *SIGCOMM*, August 2010.

[18] "OpenFlow switch." `http://www.openflowswitch.org/`.

[19] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim, "Profiling network performance for multi-tier data center applications," in *NSDI*, 2011.

[20] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," in *Internet Mathematics*, vol. 1, pp. 485–509, 2005.

[21] S. C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *Proc. IEEE INFOCOM*, 2002.

[22] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest prefix matching using Bloom filters," in *Proc. ACM SIGCOMM*, 2003.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, Aug. 2000.

[24] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica, "ROFL: Routing on flat labels," in *Proc. ACM SIGCOMM*, 2006.

[25] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Accountable Internet protocol (AIP)," in *Proc. ACM SIGCOMM*, 2008.

[26] S. M. Bellovin, "Distributed firewalls," *;login:*, Nov. 1999.

[27] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," *ACM Conference on Computer and Communications Security*, 2000.

[28] "Introduction to server and domain isolation." `technet.microsoft.com/en-us/library/cc725770.aspx`.

[29] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica, "Achieving convergence-free routing using failure-carrying packets," in *Proc. ACM SIGCOMM*, 2007.

[30] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *IEEE/ACM Transactions on Networking*, 2000.

[31] D. Aldous and J. Fill, *Reversible Markov Chains and Random Walks on Graphs.* Book in preparation, 2001.

[32] R. Motwani and P. Raghavan, *Randomized Algorithms.* Cambridge University Press, 1995.

[33] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.

[34] J. L. Palacios, "Bounds on expected hitting times for random walk on a connected graph," *Linear Algebra and Its Applications*, vol. 141, pp. 241–252, 1990.

[35] E. Seneta, *Non-negative matrices and Markov chains.* Springer, 2nd ed., 1981.

[36] Y.-W. E. Sung, S. Rao, G. Xie, and D. Maltz, "Towards systematic design of enterprise networks," in *Proc. ACM CoNEXT*, 2008.

[37] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP topologies with Rocketfuel," in *IEEE/ACM Transactions on Networking*, 2004.

[38] M. Arregoces and M. Portolani, *Data Center Fundamentals.* Cisco Press, 2003.

[39] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proc. Internet Measurement Conference*, 2005.

[40] B. Dipert, "Special purpose SRAMs smooth the ride," *EDN*, 1999.

[41] "Interior point optimizer." `www.coin-or.org/Ipopt/`.

[42] S. Ratnasamy, A. Ermolinskiy, and S. Shenker, "Revisiting IP multicast," in *ACM SIGCOMM.*

[43] P. Jokela, A. Zahemszky, S. Arianfar, P. Nikander, and C. Esteve, "LIPSIN: Line speed publish/subscribe inter-networking," in *Proc. ACM SIGCOMM*, 2009.

[44] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proc. IEEE INFOCOM*, 2001.

[45] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: An aid to network processing," in *Proc. ACM SIGCOMM*, 2005.

[46] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: An efficient data structure for static support lookup tables.," in *Proc. Symposium on Discrete Algorithms*, 2004.

[47] C. Esteve, F. L. Verdi, and M. F. Magalhaes, "Towards a new generation of information-oriented internetworking architectures," in *ReArch Workshop*, 2008.

[48] M. Yu and J. Rexford, "Hash, don't cache: Fast packet forwarding for enterprise edge routers," in *Proc. ACM SIGCOMM Workshop on Research in Enterprise Networks*, 2009.

[49] A. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proc. Workshop on Research in Enterprise Networks*, 2009.

[50] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing Web traffic using OpenFlow," Aug. 2009. ACM SIGCOMM Demo.

[51] D. Erickson *et al.*, "A demonstration of virtual machine mobility in an OpenFlow network," Aug. 2008. ACM SIGCOMM Demo.

[52] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving energy in data center networks," in *Proc. Networked Systems Design and Implementation*, Apr. 2010.

[53] Y. Mundada, R. Sherwood, and N. Feamster, "An OpenFlow switch element for Click," in *Symposium on Click Modular Router*, 2009.

[54] A. Tootoocian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in *INM/WREN workshop*, 2010.

[55] S. Ray, R. Guerin, and R. Sofia, "A distributed hash table based address resolution scheme for large-scale Ethernet networks," in *Proc. International Conference on Communications*, June 2007.

[56] H. Ballani, P. Francis, T. Cao, and J. Wang, "Making routers last longer with ViAggre," in *Proc. NSDI*, 2009.

[57] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough," in *ACM SIGMETRICS*, 2007.

[58] P. Gupta, P. Gupta, and N. Mckeown, "Packet classification using hierarchical intelligent cuttings," in *Hot Interconnects VII*, 1999.

[59] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, 2003.

[60] J.-H. Lin and J. S. Vitter, "e-approximations with minimum packing constraint violation," in *ACM Symposium on Theory of Computing*, 1992.

[61] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *Proc. Networked Systems Design and Implementation*, Oct. 2005.

[62] N. Brownlee, "Some observations of Internet stream lifetimes," in *Passive and Active Measurement*, 2005.

[63] "Stanford OpenFlow network real time monitor." `http://yuba.stanford.edu/~masayosi/ofgates/`.

[64] Personal communication with Srini Seetharaman.

[65] "Netlogic microsystems." `www.netlogicmicro.com`.

[66] B. Krishnamurthy and J. Rexford, "HTTP/TCP Interaction," in *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*, Addison-Wesley, 2001.

[67] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *ACM SIGCOMM*, 2009.

[68] V. Paxson, "Automated packet trace analysis of TCP implementations," in *ACM SIGCOMM*, 1997.

[69] `www.web100.org`.

[70] `http://msdn.microsoft.com/en-us/library/bb427395%28VS.85%29.aspx`.

[71] `http://msdn.microsoft.com/en-us/library/bb968803%28VS.85%29.aspx`.

[72] `http://datatracker.ietf.org/wg/syslog/charter/`.

[73] "TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK." `www.stuartcheshire.org/papers/NagleDelayedAck`.

[74] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl, "Detailed diagnosis in computer networks," in *ACM SIGCOMM*, 2009.

[75] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM SIGCOMM*, 2007.

[76] I. Jolliffe, *Principal Component Analysis.* Springer-Verlag, 1986.

[77] `support.microsoft.com/kb/983528`.

[78] C. Sarndal, B. Swensson, and J. Wretman, *Model Assisted Survey Sampling.* Springer-Verlag, 1992.

[79] A. Diwan and R. L. Sites, "Clock alignment for large distributed services," *Unpublished report*, 2011.

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

[81] `http://blogs.msdn.com/b/wndp/archive/2006/08/15/http-sys-buffering.aspx`.

[82] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of datacenter traffic: Measurements and analysis," in *Proc. Internet Measurement Conference*, 2009.

[83] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *NSDI*, 2004.

[84] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *NSDI*, 2006.

[85] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *NSDI*, 2007.

[86] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *EuroSys*, 2010.

[87] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of Internet flow rates," in *ACM SIGCOMM*, 2002.

[88] Y.-C. Cheng, J. Bellardo, P. Benko, A. C. Snoeren, G. M. Voelker, and S. Savage, "Jigsaw: Solving the puzzle of enterprise 802.11 analysis," in *ACM SIGCOMM*, 2006.

[89] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with WISE," in *ACM SIGCOMM*, 2008.