UNIVERSITY OF CALIFORNIA, SAN DIEGO

Scalable Traffic Management for Data Centers and Logging Devices

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Vinh The Lam

Committee in charge:

Professor George Varghese, Chair
Professor Tara Javidi
Professor Bill Lin
Professor Amin Vahdat
Professor Geoffrey Voelker

2013

The Dissertation of Vinh The Lam is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2013

DEDICATION

To my parents

EPIGRAPH

Science is what we understand well enough
to explain to a computer.
Art is everything else we do.

Donald Knuth

Simplicity is prerequisite for reliability.

Edsger Dijkstra

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my PhD advisor, Professor George Varghese, for his inspirations and support during my PhD years at UCSD. George has been always generous in giving me help and guidance every time I need him. I have learned a great deal from him, not only his technical expertise, but more importantly, the thought process to approach research problems. He made the last six years a truly life-changing experience for me. I am also grateful to my other committee members: Tara Javidi, Bill Lin, Amin Vahdat, and Geoff Voelker for being in my doctoral committee and giving me invaluable feedbacks during my thesis proposal and preparation of this dissertation.

During my graduate studies at UCSD, I have had the fortune to collaborate with many smart and wonderful people, especially my co-authors: Tom Edsall, Andy Fingerhut, Erran Li, Michael Mitzenmacher, Rong Pan, Sivasankar Radhakrishnan, Yousuk Seung, Amin Vahdat, and Thomas Woo. Without them, my research would have taken much longer. I would never forget the long nights that they worked with me to complete multiple drafts of my papers.

My research internship opportunities at Bell Labs and Google were supplementary to my PhD research and prepared me well to work in the software industry. Especially I would like to thank Thomas Woo at Bell Labs, Jerry Chu, Nandita Dukkipati, and Abdul Kabbani at Google. My interactions with them during my internships have been truly eye-opening and helped me make up my career choice.

I would like to acknowledge Kostas Anagnostakis, my tech lead at the Institute for Infocomm Research, Singapore, where I had my first full-time job as a research engineer. It was during my time working with Kostas that I first picked up research skills and published my very first major research papers. Kostas was also a strong source of encouragement when I prepared my application package to PhD programs.

I have been extremely fortunate to have my fantastic friends: Christos Kozanitis for being a greatly helpful friend; Son Ngoc Duong for many times giving me valuable insights into modern mathematics; Son Kim Pham for being a source of academic musings; San Nguyen for constantly reminding me of mindfulness; Alan Pham for being an “extraordinary” landlord.

Finally, there are not enough words to describe how much I owe to my parents, my sister, and my brother. They have been always with me and always believed in me. All my achievements would never have been possible without their unconditional sacrifice and unbounded patience with me during my time in graduate school in the United States.

Chapter 2, in full, is a reprint of the material as it appears in “Carousel: Scalable Logging for

Intrusion Prevention Systems" in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lam, Vinh The; Mitzenmacher, Michael; Varghese, George. USENIX, 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in "Flame: Efficient and Robust Hardware Load Balancing for Data Center Routers" in *UCSD CSE Technical Report (CS2012-0980)*. Edsall, Tom; Fingerhut, Andy; Lam, Vinh The; Pan, Rong; Varghese, George. UCSD, 2012 The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in "NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers" in *Proceedings of ACM SIGCOMM Computer Communication Review (CCR)*. Lam, Vinh The; Radhakrishnan, Sivasankar; Pan, Rong; Vahdat, Amin; Varghese, George. ACM, 2012. The dissertation author was the primary investigator and author of this paper.

VITA

2004	B. E. in Electrical and Electronic Engineering, Nanyang Technological University, Singapore
2005	M. S. in Computer Science, National University of Singapore, Singapore
2006–2013	Graduate Student Researcher, University of California, San Diego, California, United States
2013	Ph. D. in Computer Science, University of California, San Diego, California, United States

PUBLICATIONS

Edsall, Tom; Fingerhut, Andy; Lam, Vinh The; Pan, Rong; Varghese, George. "Flame: Efficient and Robust Hardware Load Balancing for Data Center Routers", in *UCSD CSE Technical Report (CS2012-0980)* 2012

Lam,Vinh The; Radhakrishnan,Sivasankar; Pan,Rong; Vahdat,Amin; Varghese,George. "NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers", in *ACM SIGCOMM Computer Communication Review (CCR)* 2012.

Seung, Yousuk; Lam, Vinh The; Li, Li E.; Woo, Thomas. "CloudFlex: Seamless Scaling of Enterprise Applications into the Cloud", in *IEEE International Conference on Computer Communications (INFOCOM)* 2011.

Lam, Vinh The; Mitzenmacher, Michael; Varghese, George. "Carousel: Scalable Logging for Intrusion Prevention Systems", in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* 2010.

Akritidis, Periklis; Chin, Wee Yung; Lam, Vinh The; Sidiroglou, Stelios; Anagnostakis, Kostas. "Proximity Breeds Danger: Emerging Threats in Metro-area Wireless Networks", in *USENIX Security* 2007.

Lam, Vinh The; Antonatos, Spiros; Akritidis, Periklis; Anagnostakis, Kostas. "Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure", in *ACM Conference on Computer and Communications Security (CCS)* 2006.

ABSTRACT OF THE DISSERTATION

Scalable Traffic Management for Data Centers and Logging Devices

by

Vinh The Lam

Doctor of Philosophy in Computer Science

University of California, San Diego, 2013

Professor George Varghese, Chair

Traditional network resource allocation is not scalable because it requires per-flow state, large amount of memory in switches and routers, and control overhead. In this dissertation, we propose innovative and scalable mechanisms for network traffic management in three emerging contexts: network event loggers, network load balancing, and cloud services in data centers. First, we describe a probabilistic event logger called *Carousel* to collect unique items in a large stream of online events. By theoretical analysis, we prove that Carousel can collect almost all items with high probability. Our simulation and implementation prototype show an improvement factor of ten in event collection time. Second, we design a new load balancing algorithm called *Flame* that is implementable in high speed switches with small memory usage. Flame achieves fine granularity of load balancing at sub-flow level and binds flows to hash functions. Through trace simulation, we show that Flame can improve our load balancing performance metrics by an order of magnitude. Furthermore, Flame allows graceful degradation to the standard

ECMP in the worst case. Lastly, we propose a mechanism called *NetShare* to provide predictable network resource allocation for cloud services based on simple administrative weights. We describe mechanisms to implement and scale NetShare to a large number of services using a generalization of Stochastic Fair Queueing. We validate our NetShare design on a hardware testbed with MapReduce workloads.

Chapter 1

Introduction

Traffic management is a critical problem in high speed computer and communication networks. The ultimate goal of traffic management is to ensure that users achieve a desired quality of service. This problem is difficult due to several practical challenges. First, data traffic demands are highly unpredictable [31, 76]. Second, recent hardware developments have made high speed links common at the network edges (e.g., network adapters at 40 Gbps and 100 Gbps [2]), which creates high amounts of oversubscription to core network links [31] and exacerbates traffic management especially during periods of heavy loads. This is why traffic rate control, despite being only a part of traffic management, is a central aspect of traffic management [37].

Conceptually, the goal of rate control problems is to create *relations* among the rates of various data packet streams in a network subject to certain objectives. The objectives include fairness among traffic flows [1] (e.g., max-min fairness [35]), simplicity for implementation in networking devices (e.g., no maintenance of per-flow state, which led to the preference of rate-based congestion control over credit-based congestion control in ATM networks [37]), and good performance (e.g., high throughput, low latency, balanced resource utilization to reduce network infrastructure cost [31]).

Much research has been devoted to rate control since the early history of networks; despite this, rate control is still an open topic. For example, the classic rate control problem is congestion control. On end hosts, Transmission Control Protocol (TCP) [57] is designed to control the rate of the sender to fairly share the rate of the bottleneck link in the network path from the sender to the receiver. It was the design and deployment of TCP that saved the early Internet from congestion collapse [13]. As a second example, there are router mechanisms to aid congestion control such as queue management algorithms

[1] Depending on the context, a flow could mean a connection (e.g., TCP flow identified by TCP 5-tuple) or aggregated connections from a single source address.

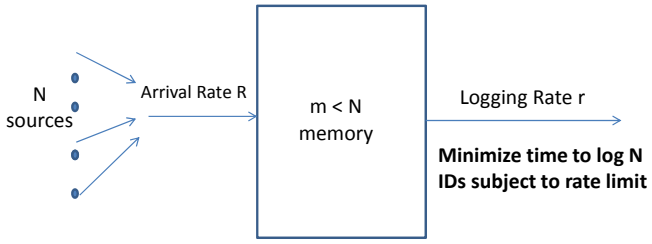


Figure 1.1. Logging problem in Chapter 2: logging of infected sources in an Intrusion Detection device. This is necessarily an open-loop control as sources are non-cooperative. *Thesis solution:* Carousel exploits source repetition and hashing to partition ID space into smaller chunks followed by iteration over the chunk space.

(e.g., Random Early Detection (RED) [26], CHOKe [55]) and traffic scheduling (e.g., Fair Queueing (FQ) [20], Stochastic Fair Queueing (SFQ) [46], Deficit Round Robin (DRR) [66]). More recent efforts attempt to find new ways to optimize for performance, especially for improving latency (e.g, TCP Fast Open (TFO) [59]) and bufferbloat resistance [30] (e.g., Controlled Delay (CoDel) [51]).

In this dissertation, we extend rate control problems to three emerging contexts: logging (techniques to log information at the rate of logger are described in Chapter 2), load balancing (techniques to spread load among available links are described in Chapter 3), and group Quality-of-Service (techniques to ensure bandwidth fairness for groups are described in Chapter 4).

Note that many rate control problems can be achieved with per-flow state (e.g., Resource Reservation Protocol (RSVP) [14] to reserve resources across a network by signalling for connection setup in advance). The goal of this thesis is to design simple, efficient, and scalable solutions to the rate control problems *without* using per-flow state to avoid the need for large amounts of high speed memory in networking devices such as switches and routers. Our solutions utilize novel mechanisms (e.g., hashing into chunks) and/or leverage assumptions (e.g., repeated sources) inherent to the specific problems. We now briefly describe the context of each problem and the thesis contributions.

Chapter 2 addresses the problem of collecting unique items in a large stream of information in the context of Intrusion Prevention Systems (IPSs) as shown in Figure 1.1. IPSs detect attacks at gigabit speeds and must log infected source IP addresses for remediation or forensics. An attack with millions of infected sources can result in hundreds of millions of log records when counting duplicates. If logging speeds are much slower than packet arrival rates and memory in the IPS is limited, *scalable logging* is a technical challenge. After showing that naïve approaches will not suffice, we solve the problem with a new algorithm we call Carousel. Carousel randomly partitions the set of sources into groups that

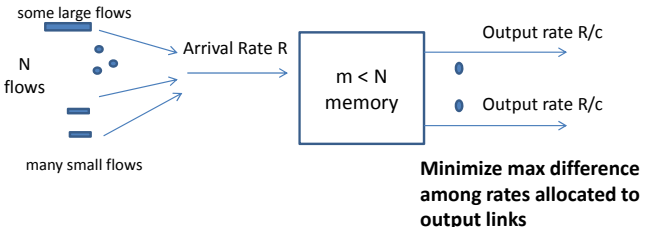


Figure 1.2. Load balancing problem in Chapter 3: going beyond ECMP for data centers because random assignment works badly when there are a few large flows. This is an online problem because the weight or size of a flow is only apparent over time and can change. *Thesis solution:* Flame uses a new online estimator, power of choice to reduce hardware complexity, and heavy-hitters & timeouts to reduce state.

can be logged without duplicates, and then cycles through the set of possible groups. We prove that Carousel collects almost all infected sources with high probability in close to optimal time as long as infected sources keep transmitting. We describe details of a Snort implementation and a hardware design. Simulations with worm propagation models show up to a factor of 10 improvement in collection times for practical scenarios. Our technique applies to *any* logging problem with non-cooperative sources as long as the information to be logged appears repeatedly.

Chapter 3 describes a new load balancing design, Flame, that is implementable at 480 Gbps with small memory and uses two novel mechanisms as shown in Figure 1.2. First, Flame uses a Discounting Rate Estimator (DRE); unlike exponential averaging, DRE quickly measures bursts and yet retains memory of recent bursts. Second, Flame binds flows to *hash functions* and not to *paths*. We show Flame is more resilient and efficient than the earlier Flare scheme, and provides better load balancing and is more deployable than Hedera. Flame also allows rebalancing of flows in hardware at rapid rates. This is interesting because we show TCP experiments at 1 and 10 Gbps that demonstrate that recent Linux stacks after 2.6.14 can tolerate rebalancing once every 10 packets with negligible loss of throughput. On the other hand, Windows 2008 stacks have degraded TCP throughput if rebalancing is done more often than 1 in 32,000 packets.

Chapter 4 studies the network isolation and virtualization in cloud data centers as shown in Figure 1.3. Service level agreements for cloud computing today specify network SLAs in terms of dollars per Gigabyte transferred and not in terms of network bandwidth. But application performance often depends crucially on network performance; a slow network can result in underutilized VMs. Chapter 4 describes a mechanism for Data Center networks called *NetShare* that requires no hardware changes to routers but allows bandwidth to be allocated predictably across services/users based on simple weights. Weights can

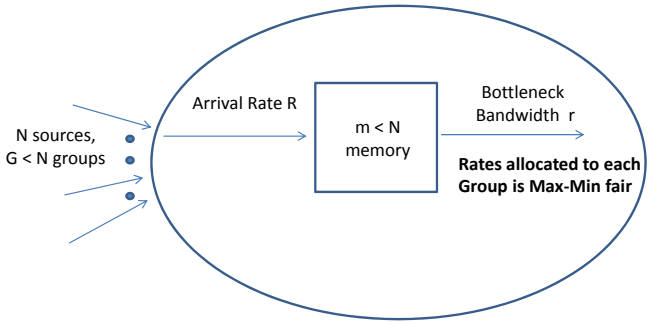


Figure 1.3. Group QoS problem in Chapter 4: sharing a data center among applications that can gain bandwidth by opening more TCP connections. *Thesis solution:* NetShare provides fairness on groups but relies on per-flow state and control at hosts (TCP) to achieve max-min fair rates.

be specified by a manager, or can be automatically assigned at each switch port based on virtual machines upstream and downstream of the port. Bandwidth unused by a service is shared proportionately by other services, providing weighted hierarchical max-min fair sharing. We present three mechanisms to implement NetShare including one that leverages TCP and requires only router configuration. We show how NetShare can scale to large numbers of users/services using a generalization of Stochastic Fair Queuing. On a testbed of Fulcrum switches, we show that *without* NetShare, performance of latency critical jobs can degrade by one order of magnitude in the presence of large Hadoop jobs. We also demonstrate that NetShare divides bandwidth proportional to weights despite the use of multipathing.

Chapter 1, in part, is a reprint of the material as it appears in "Carousel: Scalable Logging for Intrusion Prevention Systems" in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* 2010. Lam, V. T., Mitzenmacher, M., and Varghese, G., USENIX, 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material as it appears in "Flame: Efficient and Robust Hardware Load Balancing for Data Center Routers" in *UCSD CSE Technical Report (CS2012-0980)*. Edsall, Tom; Fingerhut, Andy; Lam, Vinh The; Pan, Rong; Varghese, George. UCSD, 2012 The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material as it appears in "NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers" in *Proceedings of ACM SIGCOMM Computer Communication Review (CCR)*. Lam, Vinh The; Radhakrishnan, Sivasankar; Pan, Rong; Vahdat, Amin; Varghese, George. ACM, 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Carousel: Scalable Logging for Intrusion Prevention Systems

2.1 Introduction

With a variety of networking devices reporting events at increasingly higher speeds, how can a network manager obtain a coherent and succinct view of this deluge of data? The classical approach uses a *sample* of traffic to make behavioral inferences. However, in many contexts the goal is *complete or near-complete collection* of information — MAC addresses on a LAN, infected computers, or members of a botnet. While our paper presents a solution to this abstract logging problem, we ground and motivate our approach in the context of Intrusion Prevention Systems.

Originally, Intrusion Detection Systems (IDSs) implemented in software worked at low speeds, but modern Intrusion Prevention Systems (IPSs) such as the Tipping Point Core Controller and the Juniper IDP 8200 [36] are implemented in hardware at 10 Gbps and are standard in many organizations. IPSs have also moved from being located only at the periphery of the organizational network to being placed throughout the organization. This allows IPSs to defend against internal attacks and provides finer granularity containment of infections. Widespread, cost-effective deployment of IPSs, however, requires using streamlined hardware, especially if the hardware is to be integrated into routers (as done by Cisco and Juniper) to further reduce packaging costs. By streamlined hardware, we mean ideally a single chip implementation (or a single board with few chips) and small amounts of high-speed memory (less than 10 Mbit).

Figure 2.1 depicts a logical model of an IPS for the purposes of this paper. A bad packet arrives carrying some key. Typically the key is simply the source address, but other fields such as the destination address may also be used. For the rest of the paper we assume the key is the IP source address. (We

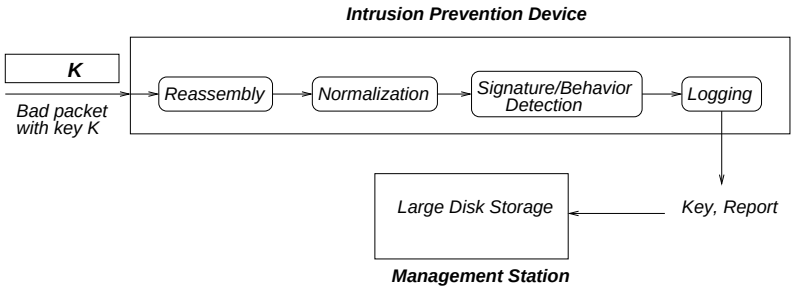


Figure 2.1. IPS logical model with logging component that is often implemented naïvely

assume the source information is not forged. Any attack that requires the victim to reply cannot use a forged source address.) The packet is coalesced with other packets for the same flow if it is a TCP packet, normalized [75] to guard against evasions, and then checked for whether the packet is indicative of an attack. The most common check is *signature-based* (e.g., Snort [69]) which determines whether the packet content matches a regular expression in a database of known attacks. However, the check could also be *behavior-based*. For example, a denial of service attack to a destination may be detected by some state accumulated across a set of past packets.

In either case, the bad packet is typically dropped, but the IPS is required to *log* the relevant information on disk at a remote management console for later analysis and reporting. The information sent is typically the key *K* plus a report indicating the detected attack. Earlier work has shown techniques for high speed implementations of reassembly [22], normalization [74, 75], and fast regular expression matching (e.g., [68]). However, to the best of our knowledge, there is no prior work in scalable logging for IPS systems or networking.

To see why logging may be a bottleneck, consider Figure 2.2, which depicts a physical model of a streamlined hardware IPS implementation, either stand-alone or packaged in a router line card. Packets arrive at high speed (say 10 Gbps) and are passed from a MAC chip to one or more IDS chips that implement detection by for example signature matching. A standard logging facility, such as in Snort, logs a report each time the source sends a packet that matches an attack signature and writes it to a memory buffer, from which it is written out later either to locally attached disk in software implementations or to a remote disk at a management station in hardware implementations. A problem arises because the logging speed is often much slower than the bandwidth of the network link. Logging speeds less than 100 Mbps are not uncommon, especially in 10 Gbps IDS line cards attached to routers. Logging speeds are limited by physical considerations such as control processor speeds and disk bandwidths. While logging speeds

can theoretically be increased by striping across multiple disks or using a network service, the increased costs may not be justified in practice.

In hardware implementations where the memory buffer is necessarily small for cost considerations, the memory can fill during a large attack and newly arriving logged records may be dropped. A typical current configuration might include only 20 Mbits of on-chip high speed SRAM of which the normalizer itself can take 18 Mbits [75]. Thus, we assume that the logger may be allocated only a small amount of high speed memory, say 1 Mbit. Note that the memory buffer may include duplicate records already in the buffer or previously sent to the remote device.

Under a standard naïve implementation, unless the logging rate matches the arrival rate of packets, there is no guarantee that all infected sources will be logged. It is easy to construct worst-case timing patterns where some set of sources A are never logged because another set of sources B always reaches the IDS before sources in the set A and fills the memory. Even in a random arrival model, intuitively as more and more sources are logged, it gets less and less probable that a new unique source will be logged. In Section 2.3 we show that, even with a fairly optimistic random model, a standard analysis based on the coupon collector's problem (e.g., [48]) shows that the expected time to collect all N sources is a *multiplicative* factor of $\ln N$ worse than the optimal time. For example, when N is in the millions, which is not unusual for a large worm, the expected time to collect all sources can be 15 times larger than optimal. We also show similar poor behavior of the naïve implementation, both through analysis and simulation, in more complex settings.

The main contribution of this paper, as shown in Figure 2.2, is a scalable logger module that interposes between the detection logic and the memory buffer. We refer to this module and the underlying algorithm as *Carousel*, for reasons that will become apparent. Our logger is scalable in that it can collect almost all N sources with high probability with very small memory buffers in close to optimal time, where here the optimal time is N/b with b being the logging speed. Further, Carousel is simple to implement in hardware even at very high speeds, adding only a few operations to the main processing path. We have implemented Carousel in software both in Snort as well as in simulation in order to evaluate its performance.

While we focus on the scalable logging problem for IPSs in this paper, we emphasize that the problem is a general one that can arise in a number of measurement settings. For example, suppose a network monitor placed in the core of an organizational network wishes to log all the IP sources that are using TCP Selective Acknowledgment option (SACK). In general, our mechanism applies to

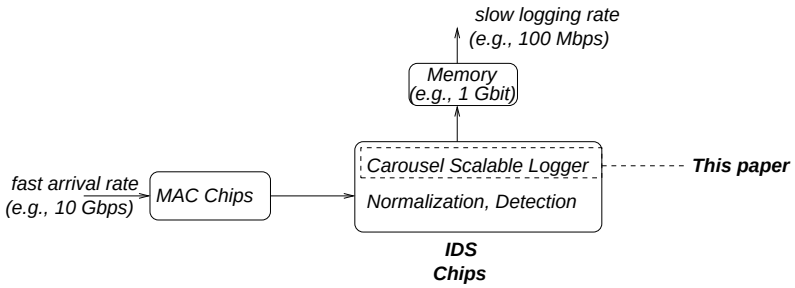


Figure 2.2. IPS hardware model in which we propose adding a scalable logger facility called Carousel. Carousel focuses on a small random subset of the set of keys at one time, thereby matching the available logging speed.

any monitoring setting where a source is identified by a predicate on a packet (e.g., the packet contains the SACK_PERMITTED option, or the packet matches the Slammer signature), memory is limited, and sources do not cooperate with the logging process. It does, however, require sources to keep transmitting packets with the predicate in order to be logged. Thus Carousel does not guarantee the logging of one-time events.

The rest of the paper is organized as follows. In Section 2.2 we describe a simple abstract model of the scalable logging problem that applies to many settings. In Section 2.3 we describe a simple analytical model that shows that even with an optimistic random model of packet arrivals, naïve logging can incur a multiplicative penalty of $\ln N$ in collection times. Indeed, we show this is the case even if naïve logging is enhanced with a Bloom filter in the straightforward way. In Section 2.4 we describe our new scalable logging algorithm Carousel, and in Section 2.5 we describe our Snort implementation. We evaluate Carousel using a simulator in Section 2.6 and using a Snort implementation in Section 2.7. Our evaluation tests both the setting of our basic analytical model, which assumes that all sources are sending at time 0, and a more realistic logistic worm propagation model, in which sources are infected gradually. Section 2.8 describes related work while Section 2.9 concludes the paper.

2.2 Model

The model shown in Figure 2.3 abstracts the scalable logging problem. First, there are N distinct keys that arrive repeatedly and with arbitrary timing frequency at a cumulative speed of B keys per second at the logger. There are *two* resources that are in scarce supply at the logger. First, there is a limited logging speed b (keys per second) that is much smaller than the bandwidth B at which keys arrive. Even this might not be problematic if the logger had a memory M large enough to hold all the distinct keys

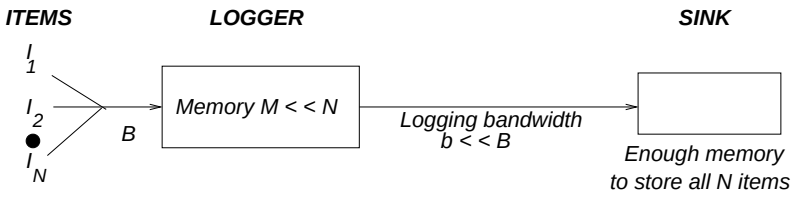


Figure 2.3. Abstract logging model: N keys to be logged enter the logging device repeatedly at a speed B that is much greater then the logging speed b and in a potentially adversarial timing pattern. At the same time, the amount of memory M is much less than the N, number of distinct keys to be logged. Source cooperation is *not* assumed.

N that needed to be logged (using methods we discuss below, such as Bloom filters [10, 15], to handle duplicates), but in our setting of large infections and hardware with limited memory, we must also assume that $N >> M$.

Eliminating all duplicates before transmitting to the sink is *not* a goal of a scalable logger. We assume that the sink has a hash table large enough to store all N unique sources (by contrast to the logger) and eliminate duplicates.

Instead, the ultimate goal of the scalable logger is *near-complete collection*: the logging of all N sources. We now adopt some of the terminology of competitive analysis [11] to describe the performance of practical logger systems. The best possible logging time $T_{optimal}$ for an omniscient algorithm is clearly N/b. We compare our algorithms against this omniscient algorithm as follows.

Definition 2.1. *We say that a logging algorithm is (ε, c)-scalable if the time to collect at least $(1-\varepsilon)N$ of the sources is at most $cT_{optimal}$. In the case of a randomized algorithm, we say that an algorithm is (ε, c)-scalable if in time $cT_{optimal}$ the expected number of sources collected is at least $(1-\varepsilon)N$.*

Note that in the case $\varepsilon = 0$ all sources are collected. While obviously collecting all sources is a desirable feature, some relaxation of this requirement can naturally lead to much simpler algorithms.

These definitions have some room for play. We could instead call a randomized algorithm (ε, c)-scalable if the expected time to collect at least $(1-\varepsilon)N$ is at most $cT_{optimal}$, and we may be concerned only with asymptotic algorithmic performance as either or both of N/M and B/b grow large. As our focus here is on practically efficient algorithms rather than subtle differences in the definitions we avoid such concerns where the meaning is clear.

The main goal of this paper is to provide an effective and practical (ε, c)-scalable randomized algorithm. To emphasize the value of this result, we first show that simple naïve approaches are not (ε, c)-scalable for any constants $\varepsilon, c > 0$. Our positive results will require the following additional assumption

for our model:

Persistent Source Assumption: We assume that any distinct key X to be logged will keep arriving at the logger.

For sources infected by worms this assumption is often reasonable until the source is "disinfected" because the source continues to attempt to infect other computers. The time for remediation (days) is also larger than the period in which the attack reaches its maximum intensity (hours). Further, if a source is no longer infected, then perhaps it matters less that the source is not logged. In fact, we conjecture that no algorithm can solve the scalable logging problem without the Persistent Source assumption.

The abstract logger model is a general one and applies to other settings. In the introduction, we mentioned one other possibility, logging sources using SACK. As another example, imagine a monitor that wishes to log all the sources in a network. The monitor issues a broadcast request to all sources asking them to send a reply with their ID. Such messages do exist, for example the SYSID message in 802.1. Unfortunately, if all sources reply at the same time, some set of sources can consistently be lost.

Of course, if the sources could randomize their replies, then better guarantees can be made. The problem can be viewed as one of congestion control: matching the speed of arrival of logged keys to the logging speed. Congestion control can be solved by standard methods like TCP slow start or Ethernet backoff *if* sources can be assumed to cooperate. However, in a security setting we cannot assume that sources will cooperate, and other approaches, such as the one we provide, are needed.

2.3 Analysis of a Naïve Logger

2.3.1 The Naïve Logger Alone

Before we describe our scalable logger and Snort implementation, we present a straw man naïve logger, and a theoretical analysis of the expected and worst-case times. The theoretical analysis makes some simplifications that only benefit the naïve logger, but still its performance is poor. The naïve logger motivates our approach.

We start with a model of the naïve logger shown in Figure 2.4. We assume that the naïve logger only has a memory buffer in the form of a queue. Keys, which again are usually source addresses, arrive at a rate of B per second. When the naïve logger receives a key, it is placed at the tail of the queue. If the queue is full, the key is dropped. The size of the queue is M. Periodically, at a smaller rate of b keys per second, the naïve logger sends the key (and any associated report) at the head of the queue to a disk log. Let L_D denote the set of keys logged to disk, and L_M the set of keys that are in the memory.

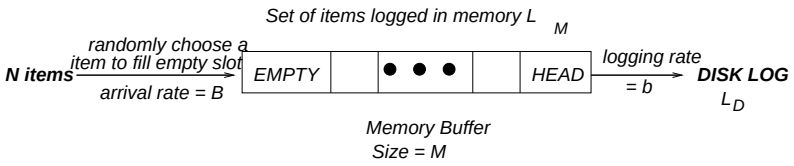


Figure 2.4. Model of naïve logging using an optimistic random model. When space opens up in the memory log, a source is picked uniformly and randomly from the set of all possible N sources. Unfortunately, that source may already be in the memory log (L_M) or in the disk log (L_D). Thus as more sources are logged it gets increasing less probable that a new unique source will be logged, leading to a logarithmic increase in collection time over optimal

The naïve logger works very poorly in an adversarial setting. In an adversarial model, after the queue is full of M keys, and when an empty slot opens up at the tail, the adversary picks a duplicate key that is part of the M keys already logged. When the queue is full, the adversary cycles through the remaining unique sources to pick them to arrive and be dropped, thus fulfilling the persistent source assumption in which every source must arrive periodically. It is then easy to see the following result.

Theorem 2.2. Worst-case time for naïve logger: *The worst-case time to collect all N keys is infinity. In fact, the worst-case time to collect more than M keys is infinite.*

We believe the adversarial models can occur in real situations especially in a security setting. Sources can be synchronized by design or accident so that certain sources always transmit at certain times when the logger buffers are full. While we believe that resilience to adversarial models is one of the strengths of Carousel, we will show that even in the most optimistic random models, Carousel significantly outperforms a naïve logger.

The simplest random model for key arrival is one in which the next key to arrive is randomly chosen from the N possible keys, and we can find the expected collection time of the naïve logger in this setting.

Let us assume that $M < B/b$, so that initially the queue fills entirely before the first departure. (The analysis is easily modified if this is not the case.) Figure 2.5 is a timeline which shows that the dynamics of the system evolve in cycles of length T seconds, where $T = 1/b$. Every T seconds the current head of the memory queue leaves for the disk log, and within the smaller time $t = 1/B$, a new randomly selected key arrives to the tail of the queue. In other words, the queue will always be full except when a key leaves from the head, leaving a single empty slot at the tail as shown in Figure 2.4. The very next key to be selected will then be chosen to fill that empty slot as shown in Figure 2.5.

The analysis of this naïve setting now follows from a standard analysis of the coupon collector's

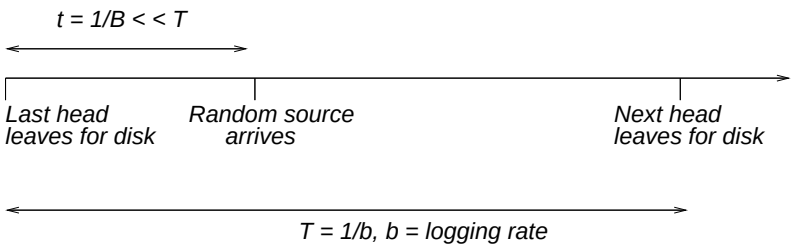


Figure 2.5. Portion of timeline for random model shown in Figure 2.4. We divide time into cycles of time T where T is the time to send one piece of logged information at the logging rate b. The time for a new randomly chosen source to first arrive is much smaller $t = 1/B$, where B is the faster packet arrival rate.

problem [48]. Let $L = L_M \cup L_D$ denote the set of unique keys logged in either memory or disk. Let T_i denote the time for L to grow from size $i-1$ to i (in other words, the time for the i-th new key to be logged). If we optimistically assume that the first M keys that arrive are distinct, we have $T_i = T$ for $1 \leq i \leq M$, as the queue initially fills. Subsequently, since the newly arriving key is chosen randomly from the set of N keys, it will get increasingly probable (as i gets larger) that the chosen key already belongs to the logged set L.

The probability that a new key will not be a duplicate of $i-1$ previously logged keys is is $P_i = (N-i+1)/N$. If a key is a duplicate the naïve logger simply wastes a cycle of time T. (Technically, it might be $T-t$ where $t = 1/B$, but this distinction is not meaningful and we ignore it.) The expected number of cycles before the i-th key is not a duplicate is the reciprocal of the probability or $1/P_i$. Hence for $i > M, i \leq N$ the expected value of T_i is

$$\mathbf{E}(T_i) = \frac{N}{b(N-i+1)}.$$

Using the linearity of expectation, the collection time for the last $N-M$ keys is

$$\sum_{i=M+1}^{N} \frac{N}{b(N-i+1)} = \frac{N}{b} \sum_{j=1}^{N-M} \frac{1}{j} = \frac{N}{b} \left(\ln(N-M) + O(1) \right),$$

using the well-known result for the sum of the harmonic series. Hence if we let $T_{collect}^{naive}$ be the time to collect all N keys for the naïve collector then $T_{collect}^{naive} > \frac{N}{b} \ln(N-M)$, and so the naïve logger is a multiplicative factor of $\ln(N-M)$ worse than the optimal algorithm.

It might be objected that it is not clear that N/b is in fact the optimal time in this random model, and that this $\ln N$ factor is due entirely to the embedded coupon collector's problem arising from the

random model. For example, if $B = b = 1$, then you cannot collect the N keys in time N, since they will not all appear until after approximately $N \ln N$ keys have passed [48]. However, as long as $B/b > \ln N$ (and $M > 1$), for any $\gamma > 0$, with high probability an omniscient algorithm will be able to collect all keys after at most $(1+\gamma)NB/b$ keys have passed in this random model, so the optimal collection time can be made arbitrarily close to N/b. Hence, this algorithm is indeed not truly scalable in the sense we desire, namely in a comparison with the optimal omniscient algorithm.

Even if we seek only to obtain $(1-\varepsilon)N$ keys, by the same argument we have the collection time is

$$\frac{N}{b}\left(\ln((1-\varepsilon)N - M) + O(1)\right).$$

Hence when $M = o(N)$, the logger is still not (ε, c)-scalable for any constants ε and c. We can summarize the result as follows:

Theorem 2.3. Expected time for naïve logger: *The expected time to collect $(1-\varepsilon)N$ keys is at least a multiplicative factor of $\ln((1-\varepsilon)N - M)$ worse than the optimal time for sufficiently large N, M, and ratios B/b.*

As stated in the introduction, for large worm outbreaks, the naïve logger can be prohibitively slow. For example, as $\ln 1,000,000$ is almost 14, if the optimal time to log 1 million sources is 1 hour, the naïve logger will take almost 14 hours.

The results for the random model can be extended to situations that naturally occur in practice and appear somewhere between the random model and an adversarial model. For example, suppose that we have two sets of sources, of sizes N_1 and N_2, but the first source sends at a speed that is j times the second. This captures, at a high level, the issue that sources may be sending at different rates. We assume each source individually behaves according to the random model. Let T_1 be the expected time to collect all the keys in the fast set, and T_2 the expected time for the slow set. Then clear the expected time to collect all sources is at least $\max(T_1, T_2)$, and indeed this lower bound will be quite tight when T_1 and T_2 are not close. As an example, suppose $N_1 = N_2 = N/2$, and $j > 1$. Then T_2 is approximately

$$\frac{N(j+1)}{2b} \ln\left(\frac{N}{2} - \frac{M}{j+1}\right).$$

The time to collect in this case is dominated by the slow sources, and is still a logarithmic factor from optimal.

2.3.2 The Naïve Logger with a Bloom Filter

A possible objection is that our naïve logger is far too naïve. It may be apparent to many readers that additional data structures, such as a Bloom filter, could be used to prevent logging duplicate sources and improve performance. This is true, and we shall use such measures in our scalable approaches. However, we point out that as the Bloom filter of limited size, it cannot by itself prevent the problems of the naïve logger, as we now explain.

To frame the discussion, consider 1 million infected sources that keep sending to an IPS. The solution to the problem may appear simple. First, since all the sources may arrive at a very fast rate of B before even a few are logged, the scheme must have a memory buffer that can hold keys waiting to be logged. Second, we need a method of avoiding sending duplicates to the logger, specifically one that takes small space, in order to make efficient use of the small speed of the logger.

To avoid sending duplicates, one naturally would think of a solution based on Bloom filters or hashed fingerprints. (We assume familiarity with Bloom filters, a simple small-space randomized data structure for answering queries of the form "Is this an item in set X" for a given set X. See [15] for details.) For example, we could employ a Bloom filter as follows. For concreteness, assume that a source address is 32 bits, the report associated with a source is 68 bits, and that we use a Bloom filter [10] of 10 bits per source.[1] Thus we need a total of 100 bits of memory for each source waiting to be logged, and 10 bits for each source that has been logged. (Instead of a Bloom filter, we could keep a table of hash-based fingerprints of the sources, with different tradeoffs but similar results, as we discuss in Section 2.4.2.)

Unfortunately, the memory buffer and Bloom filter have to operate at Gigabit speeds. Assume that the amount of IDS high speed memory is limited to storing say 1 Mbit. Then, assuming 100 bits per source, the IPS can only store information about a burst of 10,000 sources pending their transmission to a remote disk. This does not include the size of the Bloom filter, which can only store around 100,000 sources if scaled to 1 Mbit of size; after this point, the false positive rate starts increasing significantly. In practice one has to share the memory between the sources and the Bloom filter.

The inclination would be to clear the Bloom filter after it became full and start a second phase of logging. One concern is that timing synchronization could result in the same sources that were logged in phase 1 being logged and filling up the Bloom filter again, and this could happen repeatedly, leading to missing several sources. Even without this potential problem, there is danger in using a Bloom filter, as

[1] This is optimistic because many algorithms would require not just a Bloom filter but instead a counting Bloom filter [25] to support deletions, which would require more than 10 bits per entry.

we can see by again considering the random model.

Consider enhancing the naïve logger with a Bloom filter to prevent the sending of duplicates. We assume the Bloom filter has a counter to track the number of items placed in the filter, and the filter is cleared when the counter reaches a threshold F to prevent too many false positives. Between each clearing, we obtain a group of F distinct random keys, but keys may be appear in multiple groups. Effectively, this generalizes the naïve logger, which simply used groups of size $F = 1$.

Not surprisingly, this variation of the coupon collector's problem has been studied; it is know as the coupon subset collection problem, and exact results for the problem are known [61, 70]. Details can be examined by the interested reader. A simple analysis, however, shows that for reasonable filter sizes F, there will be little or no gain over the naïve logger. Specifically, suppose $F = o(\sqrt{N})$. Then in the random model, the well-known birthday paradox implies that with high probability the first F keys to be placed in the Bloom filter will be distinct. While there may still be false positives from the Bloom filter, for such F the filter fills without detecting any true duplicates with high probability. Hence, in the random case, the expected collection time even using a Bloom filter of this size is still $\frac{N}{b}\ln(N-M)+O(1)$. With larger filters, some true duplicates will be suppressed, but one needs very large filters to obtain a noticeable gain. The essential point of this argument remains true even in the setting considered above where different sets of sources arrive at different speeds.

The key problem here is that we cannot supply the IDS with the list of all the sources that have been logged, even using a Bloom filter or a hashed set of fingerprints. Indeed, when $M \ll N$ no data structure can track a meaningful fraction of the keys that have already been stored to disk. Our solution to this problem is to partition the population of keys to be recorded into subsets of the right size, so that the logger can handle each subset without problem. The logger then iterates through all subsets in *phases*, as we now describe. This repeated cycling through the keys is reminiscent of a Carousel, yielding our name for our algorithm.

2.4 Scalable logging using Carousel

2.4.1 Partitioning and logging

Our goal is to partition the keys into subsets of the right size, so that during each phase we can concentrate on a single subset. The question is how to perform the partitioning. We want the size of each partition to be the right size for our logger memory, that is approximately size M. We suggest using a randomized partition of the sources into subsets using a hash function that uses very little memory and

processing. This randomized partitioning would be simple if we initially knew the population size N, but that generally will not be the case; our system must find the current population size N, and indeed should react as the population size changes.

We choose a hash-based partition scheme that is particularly memory and time-efficient. Let $H(X)$ be a hash function that maps a source key X to an r-bit integer. Let $H_k(X)$ be the lower order k bits of $H(X)$. The size of the partition can be controlled by adjusting k.

For example, if $k = 1$, we divide the sources into two subsets, one subset whose low order bit (after hashing) is 1, and one whose lower order bit is a 0. If the hash function is well-behaved, these two sets will be approximately half the original size N. Similarly, $k = 2$ partitions the sources approximately into four equally sized subsets whose hash values have low order bits 00, 01, 10, and 11 respectively. This allows only very coarse-grained partitioning, but that is generally suitable for our purposes, and the simplicity of using the lower order k bits of $H(X)$ is particularly compelling for implementation and analysis. To begin we will assume the population size is stable but unknown, in which case the basic Carousel algorithm can be outlined as follows:

- *Partition:* Partition the population into groups of size 2^k by placing all sources which have the same value of $H_k(X)$ in the same partition.

- *Iterate:* A phase is assigned time $T_{phase} = M/b$ which is the time to log M sources, where M is the available memory in keys and b is the logging time. The i-th phase is defined by logging only sources such that $H_k(s) = i$. Other sources are automatically dropped during this phase. The algorithm must also utilize some means of preventing the same source from being logged multiple times in the phase, such as a Bloom filter or hash fingerprints.

- *Monitor:* If during phase i, the number of keys that match $H_k() = i$ exceeds a high threshold, then we return to the Partition step and increase k. While our algorithms typically use $k = k+1$, higher jumps can allow faster response. If the number of number of keys that match $H_k() = i$ falls below a low threshold, then we return to the Partition step and decrease k.

In other words, Carousel initially tries to log all sources without hash partitioning. If that fails because of memory overflow, the algorithm then works on half the possible sources in a phase. If that fails, it works on a quarter of the possible sources, and so on. Once it determines the appropriate partition size, the algorithm iterates through all subsets to log all sources.

As described, we could in the monitoring stage change k by more than 1 if our estimate of the number of keys seen during that phase suggests that would be an appropriate choice. Also, of course, we can choose to decrease k if our estimate of the keys in that phase is quite small, as would happen if we are logging suspected virus sources and these sources are stopped. There are many variations and optimizations we could make, and some will be explored in our experiments. The important idea of Carousel, however, is to partition the set of keys to match the logger memory size, updating the partition as needed.

2.4.2 Collection Times for Carousel

We assume that the memory includes, for each key to be recorded, the space for the key itself, the corresponding report, and some number of bits for a Bloom filter. This requires slightly more memory space that we assumed when analyzing the random model, where we did not use the Bloom filter. The discrepancy is small, as we expect the Bloom filter to be less than 10% of the total memory space (on the order of 10 bits or less per item, against 100 or more bits for the key and report). This would not effectively change the lower bounds on performance of the naïve logger. We generally ignore the issue henceforth; it should be understood that the Bloom filter takes a small amount of additional space.

Recall that Carousel has 3 components: partition, iterate, and monitor. Faced with an unknown population N, the scalable logger will keep increasing the number of bits chosen k until each subset is less than size M, the memory size available for buffering logged keys.

We sketch an optimistic analysis, and then correct for the optimistic assumptions. Let us assume that all N keys are present at the start of time, that our hash function splits the keys perfectly equally, and that there is no failed recording of keys due to false positives from the Bloom filter (or whatever structure suppresses duplicates). In that case it will take at most $\lceil \log_2 \frac{N}{M} \rceil$ partition steps for Carousel to get the right number of subsets. Each such step required time for a single logging phase, $T_{phase} = M/b$. The logger then reaches the right subset size, so that k is the smallest value such that $N/2^k \leq M$. The collector then goes through 2^k phases to collect all N sources. Note that $2^k \leq 2N/M$, or else k would not be the smallest value with $N/2^k \leq M$. Hence, after the initial phases to find the right value of k, the additional collection time required is just $2N/b$, or a factor of two more than optimal. The total time is thus at most

$$\frac{M\lceil \log_2(N/M) \rceil}{b} + \frac{2N}{b},$$

and the generally the second term will dominate the first. Asymptotically, when $N \gg M$, we are roughly

within a factor of 2 of the optimal collection time.

Note that the factor of 2 in the $2N/b$ term could in fact be replaced in theory by any constant $a > 1$, by increasing the number of sets in the partition by a factor of a rather than 2 at each partition step. This would increase the number of partition steps to $\lceil \log_a \frac{N}{M} \rceil$. In practice we would not want to choose a value of a too close to 1, because keys will not be partitioned equally into sets, as we describe in the next subsection. Also, as we have described a factor of 2 is convenient in terms of partitioning via the low order bits of a hash. In what follows we continue to use the factor 2 in describing our algorithm, although it should be understood smaller constants (with other tradeoffs) are possible.

In some ways our analysis is actually pessimistic. Early phases that fail can still log some items, and we have assumed that we could partition to require $2N/M$ phases, when generally the number of phases required will be smaller. However, we have also made some optimistic assumptions that we now revisit more carefully.

Unequal Partitioning: Maximum Subset Analysis

If the logger uses k bits to partition keys, then there are $K = 2^k$ subsets. While the expected number of sources in a subset is $\frac{N}{K}$, even assuming a perfectly random hash function, there may be deviations in the set sizes. Our algorithm will actually choose the value of k such that the biggest partition is fit in our memory budget M, not the average partition, and we need to take this into account. That is, we need to analyze the *maximum* number of keys being assigned to a subset at each phase interval.

In general, this can be handled using standard Chernoff bound analysis [48]. In this specific case, for example, [58] proves that with very high probability, the maximum number of sources in any subset is less than $\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}}$. Therefore we can assume that the smallest integer k satisfying

$$\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}} \leq M, \tag{2.1}$$

where $K = 2^k$, is greater than or equal to the k eventually found by the algorithm.

Note that the difference between our optimistic analysis, where we required the smallest k such that $N/K \leq M$, and this analysis is generally very small, as $\sqrt{\frac{2N \ln K}{K}}$ is generally much less than N/K. That is, suppose that $N/K \leq M$, but $\frac{N}{K} + \sqrt{\frac{2N \ln K}{K}} > M$, so that at some point we might increase the value k to more than the smallest value such that $N/K \leq M$, because we unluckily have a subset in our partition

that is bigger than the memory size. The key here is that in this case $N/K \approx M$, or more specifically

$$M \geq \frac{N}{K} > M - \sqrt{\frac{2N \ln K}{K}},$$

so that our collection time is now

$$\frac{2KM}{b} < \frac{2N}{b} + \frac{2}{b}\sqrt{\frac{2N \ln K}{K}}.$$

That is, the collection time is still, at most, very close to $2N/b$, with the addition of a smaller order term that contributes negligibly compared to $2N/b$ for large N. Hence, asymptotically, we are still with a factor of c of the optimal collection time, for any $c > 2$.

Effects of False Positives

So far, our analysis has not taken into account our method of suppressing duplicates. One natural approach is to use a Bloom filter, in which case false positives can lead to a source not being logged in a particular phase. This explains our definition of an (ε, c)-scalable logger. We have already seen that c can be upper bounded by any number larger than 2 asymptotically. Here ε can be bounded by the false positive rate of the corresponding Bloom filter. As long as the number of elements per phase is no more than $M' = \frac{N}{K} + \sqrt{\frac{2N \ln K}{K}}$ with high probability, then given the number of bits used for our Bloom filter, we can bound the false positive rate. For example, using $10M'$ bits in the Bloom filter, the false positive rate is less than 1%, so our logger asymptotically converges to a $(0.01, 2)$-scalable logger.

We make note of some additions one can make to improve the analysis. First, this analysis assumes only a single *major cycle* that logs each subset in the partition once. If one rerandomized the chosen hash functions each major cycle, then the probability a persistent source is missed each major cycle is independently at most ε each time. Hence, after two such cycles, the probability of a source being missed is at most ε^2, and so on.

Second, this analysis is pessimistic, in that in this setting, items are gradually added to an empty Bloom filter each phase; the Bloom filter is not in its full state at all times, so the false positive probability bound for the full filter is a large overestimate. For completeness we offer the following more refined analysis (which is standard) to obtain the expected false positive rate. (As usual, the actual rate is concentrated around its expectation with high probability.)

Assume the Bloom filter has m bits and uses h hash functions. Consider whether the $(i+1)$st item added to the filter causes a false positive. First consider a particular bit in the Bloom filter. The

probability that it is not set to 1 by one of the hi hash functions thus far is $(1-\frac{1}{m})^{hi}$. Therefore the probability of a false positive at this stage is $(1-(1-\frac{1}{m})^{hi})^h \approx (1-e^{-\frac{hi}{m}})^h$.

Suppose M' items are added into the Bloom filter within a phase interval. The expected fraction of false positives is then (approximately) $\sum_{i=0}^{M'-1}(1-e^{-\frac{hi}{m}})^h$, compared to the $(1-e^{-\frac{hM'}{m}})^h$ given by the standard analysis for the false positive rate after M' elements have been added. As an example, with $M'=312$, $h=5$, and $m=5000$, the standard analysis gives a false positive rate of $1.4\cdot 10^{-3}$, while our improved analysis gives a false positive rate of $2.5\cdot 10^{-4}$.

Third, if collecting all or nearly all sources is truly paramount, instead of using a Bloom filter, one can use hash-based fingerprints of the sources instead. This requires more space than a Bloom filter ($\Theta(\log M')$ bits per source if there are M' per phase) but can reduce the probability of a false positive to inverse polynomial in M'; that is, with high probability, all sources can be collected. We omit the standard analysis.

Carousel and Dynamic Adaptation

Under our persistent source assumption, any distinct key keeps arriving at the logger. In fact, for our algorithm as described, we need an even stronger assumption: each key must appear during the phase in which it is recorded, which means each key should arrive every N/b steps. Keys that do not appear this frequently may miss their phase and not be recorded. In most settings, we do not expect this to be a problem; any key that does not persist and appear this frequently does not likely represent a problematic source in terms of, for example, virus outbreaks. Our algorithm could be modified for this situation in various ways, which we leave as future work. One approach, for example, would be to sample keys in order to estimate the 95% percentile for average interarrival times between keys, and set the time interval for the phase time to gather a subset of keys accordingly.

A more pressing issue is that the persistent source assumption may not hold because external actions may shut down infected sources, effectively changing the size of the set of keys to record dynamically. For example, during a worm outbreak, the number of infected sources rises rapidly at first but then they can go down due to external actions (for example, network congestion, users shutting down slow machines due to infection, and firewalling traffic or blocking a part of the network). In that case, the scalable logger may pick a large number of sampling bits k at first due to large outbreak traffic. However, the logger should correspondingly increase the value of k subsequently as the number of sources to record declines, to avoid inefficient logging based on too large a number of phases.

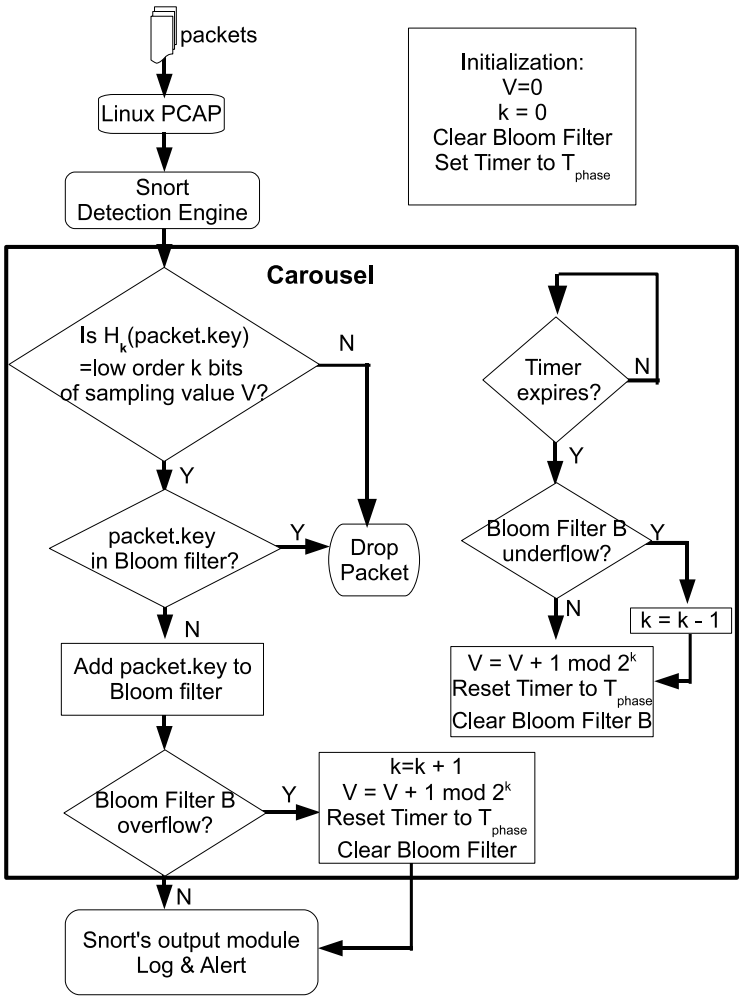


Figure 2.6. Flowchart of Carousel within Snort packet flow

2.5 Carousel Implementations

We describe our Snort evaluation in Section 2.5.1 and a sketch of a hardware implementation in Section 2.5.2.

2.5.1 Snort Implementation

In this section, we describe our implementation of Carousel integrated into the Snort [69] IDS. We need to first understand the packet processing flow within Snort to see where we can interpose the Carousel scalable logger scheme. As in Figure 2.6, incoming packets are captured by *libpcap*, queued in a kernel buffer, and then processed by the callback function *ProcessPacket*.

ProcessPacket first passes the packet to preprocessors, which are components or plug-ins serving to filter out suspicious activity and prepare the packet to be further analyzed. The detection engine then matches the packet against the rules loaded during Snort initialization. Finally, the Snort output module performs appropriate actions such as logging to files or generating alerts. Note that Snort is designed to be strictly single-threaded for multiplatform portability.

The logical choice is to place Carousel module between the detection engine and output module so that the traffic can either go directly to the output plugin or get diverted through the Carousel module. We cannot place the logger module before the detection engine because we need to log only after a rule (e.g., a detected worm) is matched. Similarly, we cannot place the logger after the output module because by then it is too late to affect which information is logged. Our implementation also allows a rule to bypass Carousel if needed and go directly to the output module.

Figure 2.6 is a flowchart of Carousel module for Snort interposed between the detection engine and the output model. The module uses the variables $T_{phase} = M/b$ (time for each phase) and k (number of sampling bits) described in Section 2.4.1. M is the number of keys that can be logged in a partition and b is the logging rate; in our experiments we use $M = 500$. The module also uses a 32-bit integer V that represents the hash value corresponding to the current partition. Initially, $k = 0$, $V = 0$, the Bloom filter is empty, and a timer T is set to fire after T_{phase}. The Bloom filter uses 5000 bits, or 10 bits per key that can fit in M, and employs 5 hash functions (SDBM, DJP, DEK, JS, PJW) taken from [56].

The Carousel scalable logger first compares the low-order k bits of the hash of the packet key (we use the IP source address in all our experiments) to the low order k bits of V. If they do not match, the packet is not in the current partition and is not passed to the output logging. If the value matches but the key yields a positive from the Bloom filter (so it is either already logged, or a false positive), again the packet is not passed to the output module. If the value matches and the key does not yield a positive from the Bloom filter, then the module adds the key to the Bloom filter. If the Bloom filter overflows (the number of insertions exceeds M), then k is incremented by 1, to create smaller size partitions.

When the timer T expires, a phase ends. We first check for underflow by testing whether the number of insertions is less than M/x. We found empirically that a factor $x = 2.3$ worked well without causing oscillations. (A value slightly larger than 2 is sensible, to prevent oscillating because of the variance in partition sizes.) If there is no underflow, then the sampling value V is increased by 1 mod 2^k to move to the next partition.

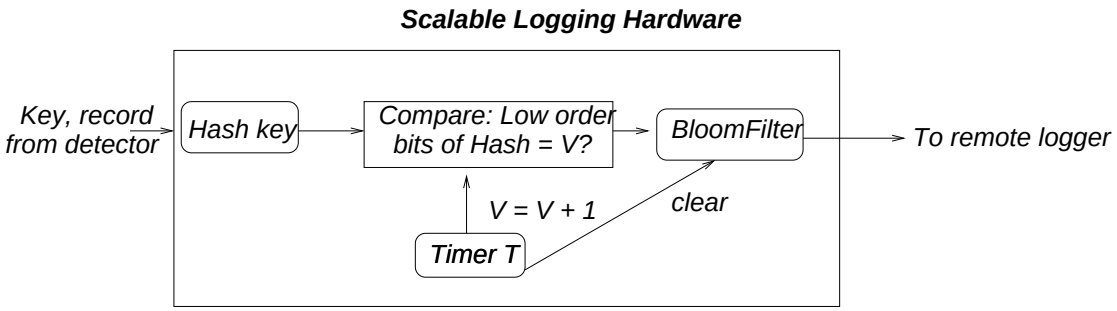


Figure 2.7. Schematic of the Carousel Logger logic as part of an IPS Chip.

2.5.2 Hardware Implementation

Figure 2.7 shows a schematic of the base logic that can be inserted between the detector and the memory buffer used to store log records in an IPS ASIC. Using 1 Mbit for the Bloom filter, we estimate that the logic takes less than 5% of a low-end 10mm by 10 mm networking ASIC. All results are reported for a standard 400 Mhz 65 nm process currently being used by networking vendors. The logic is flow-through: in other words, it can inserted between the detector and logging logic without changing any other logic. This allows the hardware to be incrementally deployed *within* an IPS without changing existing chip sets.

We assume the detector passes a key (e.g., a source IP address) and a detection record (e.g., signature that matched) to the first block. The hash blocks computes a 64-bit hash of the key. Our estimates use a Rabin hash whose loop is unrolled to run at 40 Gbps using 20K gates.

The hash output supplies a 64-bit number which is passed to the Compare block. This block masks out the low-order k bits of the hash (a simple XOR) and then compares it (comparator) to a register value V that denotes the current hash value for this phase. If the comparison fails, the log attempt is dropped. If it succeeds, the key and record are passed to the Bloom filter logic. This is the most expensive part of the logic. Using 1 Mbit of SRAM to store the Bloom filter and 3 parallel hash functions (these can be found by taking bits 1-20, 21-40, 41-60 etc of the first 64-bit hash computed without any further hash computations), the Bloom filter logic takes less than a few percent of a standard ASIC.

As in the Snort implementation, a periodic timer module fires every $T_{phase} = M/b$ time and causes the value V to be incremented. Thus the remaining logic other than the Bloom filter (and to a smaller extent the hash computation) is very small. We use two copies of the Bloom filter and clear one copy while the other copy is used in a phase. The Bloom filter should be able to store a number of keys equal to the number of keys that can be stored in the memory buffer. Assuming 10 bits per entry, a 1

Mbit Bloom filter allows approximately 100,000 keys to be handled in each phase with the targeted false positive probability. Other details (underflow, overflow etc.) are similar to the Snort implementation and are not described here.

2.6 Simulation Evaluation

To evaluate Carousel under more realistic settings in which the population grows, we simulate the logger behavior when faced with a typical worm outbreak as modeled by a logistic equation. We used a discrete event simulation engine that is a stripped down (for efficiency) version of the engine found in ns-2. We implement the Carousel scalable logger as described in Section 2.4. The simulated logger maintains the sampling bit count k and only increases k when the Bloom filter overflows; k stabilizes when all sources sampled during T_{phase} fit the into memory budget M with logging speed b. Simulation allows us to investigate the effect of various input parameters such as varying worm speed and whether the worm uses a hit list. Again, in all the simulations below, the Bloom filter uses 5000 bits and 5 hash functions (SDBM, DJP, DEK, JS, PJW) taken from [56]. For each experiment, we plot the average of 50 runs of simulation.

We start by confirming the theory with a baseline experiment in Section 2.6.1 when all sources are present at time 0. We examine the performance of our logger with the logistic model in Section 2.6.2. We evaluate the impact of non-uniform source arrivals in Section 2.6.3. In Section 2.6.4, we examine a tradeoff between using a smaller number of bits per Bloom filter element and taking more more major cycles to collect all sources. Finally, in Section 2.6.5, we demonstrate the benefit of reducing k in the presence of worm remediation.

2.6.1 Baseline Experiment

In Figure 2.8, we verify the underlying theory of Carousel in Section 2.4 assuming all sources are present at time 0. We consider various starting populations $N = 10000$ to 80000 sources, a memory budget of $M = 500$ items, and a logging speed $b = 100$ items per second.

Figure 2.8 shows that the Carousel scalable logger collects *almost all* (at least 99.9%) items by $t = 189, 354, 679$ and 1324 seconds for $N = 10000, 20000, 40000$ and 80000 respectively. This is no more than $\frac{2N}{b}$ in all cases, matching the predictions of our optimistic analysis in Section 2.4.

With these settings, the $10,000$ sources will be partitioned into 32 subsets, each of size approximately 312 (in expectation). In fact, our experiment trace shows that the number of sources per phase is

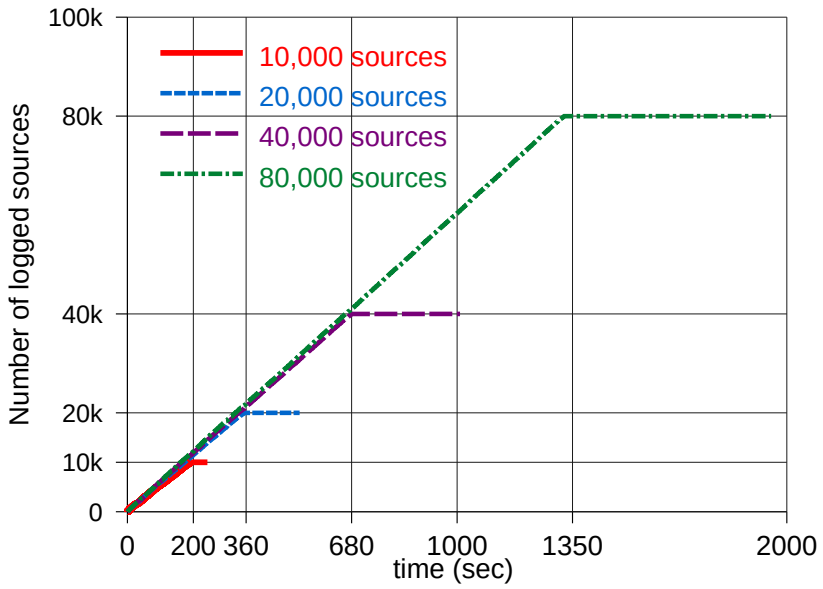


Figure 2.8. Performance of Carousel with different logging populations

in the range of 280 to 340. Since the Bloom filter uses 5000 bits, essentially we have more than 10 bits per item once the right number of partitions is found. As we calculated previously (in Section 2.4.2), the accumulated false positive rate of 312 sources in a 5000-bit Bloom filter with 5 hash functions is $2.5 \cdot 10^{-4}$. We also verified that most phases have no false positives. However, the Carousel algorithm may need additional major cycles to collect these remaining sources. Since a major cycle is 2^k iterations, the theory predicts that Carousel requires more time to collect missed false positives for larger k and hence for larger N. We observe that the length of horizontal segment of each curve in Figure 2.8, which represents the collection time of all sources missed in the first major cycle, is longer for larger populations N.

2.6.2 Logger Performance with Logistic Model

In the logistic model, a worm is characterized by H, the size of the initial hit list, the scanning rate, and a probability p of a scan infecting a vulnerable node. In our simulations below, we use a population of $N = 10,000$, a memory size $M = 500$ with Bloom filter and $M = 550$ without Bloom filter, and logging speed $b = 100$ packets/sec; the best possible logging time to collect all sources is $N/b = 100$ seconds.

For our first 3 experiments, shown in Figures 2.9, 2.10 and 2.11, we use an initial hit list of

$H = 10,000$. Since the hit list is the entire population, as in the baseline, all sources are infected at time $t = 0$. We use these simulations to see the effect of increasing the scan rate and monitoring ability assuming all sources are infected. Our subsequent experiments will assume a much smaller hit list, more closely aligned with a real worm outbreak.

For the first experiment, shown in Figure 2.9 we use 6 scans per second (to model a worm outbreak that matches the Code Red scan rate [78]) and $p = 0.01$. Figure 2.9 shows that Carousel needs 200 seconds to collect the $N = 10,000$ sources whereas the naïve logger takes $4,000$ seconds. Further, the difference between Carousel and the naïve logger increases with the fraction of sources logged. For example, Carousel is 6 times faster at logging 90% level of all sources but 20 times faster to log 100% of all sources. This is consistent with the analysis in Section 2.3.1.

In Figure 2.10 we keep all the same parameters but increase the scan rate ten times to 60 scans/sec. The higher scan rate allows naïve logging a greater chance to randomly sample packets and so the difference between scalable and naïve logging is less pronounced. Figure 2.11 uses the same parameters as Figure 2.9 but assumes that only 50% of the scanning packets are seen by the IPS. This models the fact that a given IPS may not see all worm traffic. Notice again that the difference between naïve and Carousel logging decreases when the amount of traffic seen by the IPS decreases.

The remaining simulations assume a logistic model of worm growth starting with a hit list of $H = 10$ infected sources when the logging process starts. The innermost curve illustrates the infected population versus time, which obeys the well-known logistic curve. Even under this propagation model, Carousel still outperforms naïve logging by a factor of almost 5. Carousel takes around 400 seconds to collect all sources while naïve logger takes 2000 seconds.

Figure 2.13 shows a slower worm. A slower worm can be modeled in many ways, such using a lower initial hit list, a lower scan rate, or a lower victim hitting probability. In Figure 2.13, we used a smaller hitting probability of 0.001. Intuitively, the faster the propagation dynamics, the better the performance of the Carousel scalable logger when compared to the naïve logger. Thus the difference is less pronounced.

Figure 2.14 demonstrates the scalability of Carousel, as we scale up N from $10,000$ to $100,000$ with all other parameters staying the same (i.e., 6 scans per second and $p = 0.01$). Carousel takes around 9,000 seconds to collect all sources, while the naïve logger takes 40,000 seconds. Note also that in all simulations with the logistic model (and indeed in all our experiments) the performance of the naïve logger with a Bloom filter is indistinguishable from that of the naïve logger by itself — as the theory

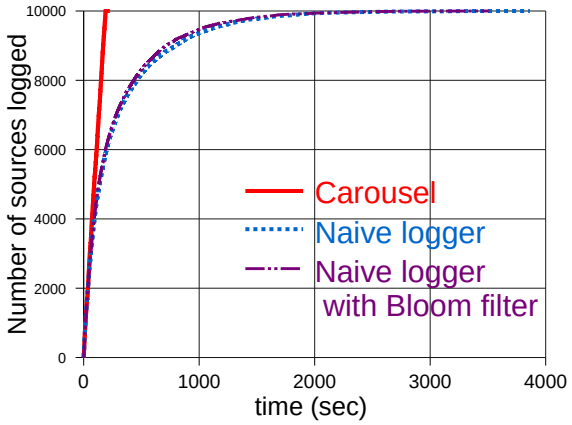


Figure 2.9. Performance of the Carousel scalable logger. Scan rate = 6/s, victim hit=1%, $M = 500$, $N = 10,000$, $b = 100$

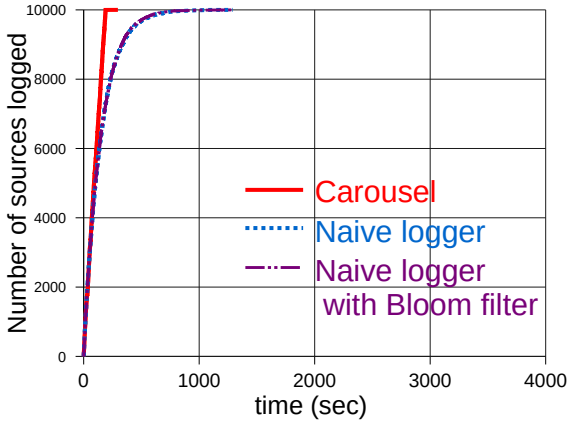


Figure 2.10. High scan rate (60 scans/s)

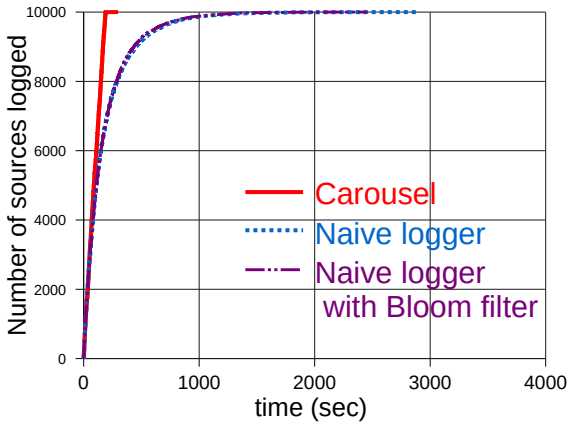


Figure 2.11. Reduced monitoring space (50%)

predicts.

2.6.3 Non-uniform source arrivals

In this section, we study logging performance when the sources arrive at different rates as described in Section 2.3.1. In particular, we experiment with two equal sets of sources in which one set sends at ten times as fast as the other set. Figure 2.15b shows the result for the naïve logger. We observe that the naïve logger has a significant problem in logging the slow sources, which are responsible for dragging down the overall performance. As predicted by our model, the times taken to log all slow sources is ten times slower than the time taken to log all fast sources. The times to log all and almost all sources are $8,000$ and $4,000$ seconds respectively.

Simply adding a Bloom filter only slightly increases the performance of the naïve logger as predicted by the theory . On the other hand, Carousel is able to consistently log all sources as shown in Figure 2.15a. Carousel is not susceptible to source arrival rates: sources from both the fast and slow sets are logged equally in each minor cycle once the appropriate number of sampling bits has been determined.

2.6.4 Effect of Changing Hash Functions

In this section, we study the effect of randomly changing the hash functions for the Bloom filter on each major cycle (that is, each pass through all of the sets of the partition). Recall that this prevents similar arrival patterns between major cycles from causing the same source to be missed repeatedly.

Figure 2.17abc compares the performance in Carousel of using fixed hash functions throughout and changing the hash functions each major cycle with 1-bit, 5-bit and 10-bit Bloom filters respectively. We changed the hash functions randomly by simply XORing each hash value with a new random number after each major cycle. In these experiments, a major cycle is approximately 160 seconds. For the 1-bit results, one can clearly see knees in the curves at $t = 160$, 320, and 480 corresponding to each major cycle in which the logger collects sources missed in previous cycles.

Carousel instrumented with changing hash functions is much faster in collecting *all sources* across several major cycles. For example, for the 1-bit case, with changing hash functions each major cycle, it takes 1500 seconds to log all sources while using fixed hash functions takes 2500 seconds to log all sources.

Should one prefer using a smaller number of bits per Bloom filter element and a greater number of major cycles or using a larger number of Bloom filter elements? This depends on the exact goals; for a fixed amount of memory, using a smaller number of Bloom filter bits per element allows the logger to log

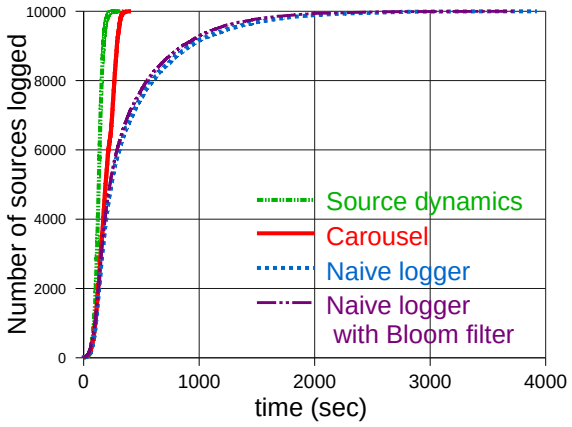


Figure 2.12. Logistic model of propagation - fast worm

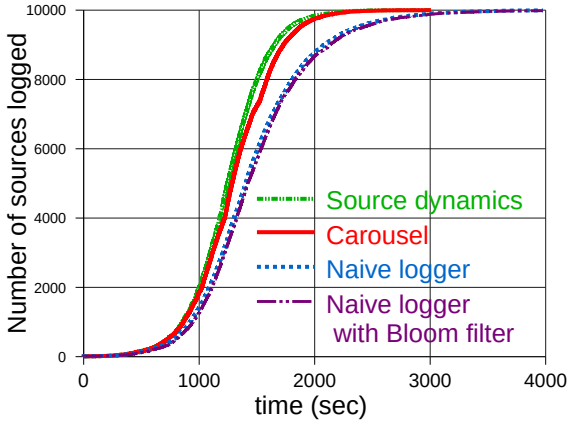


Figure 2.13. Logistic model of propagation - slow worm

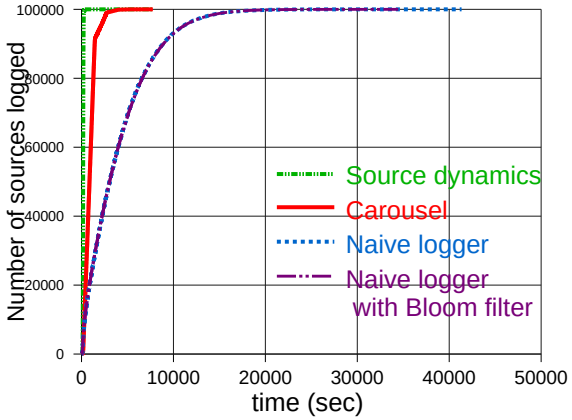
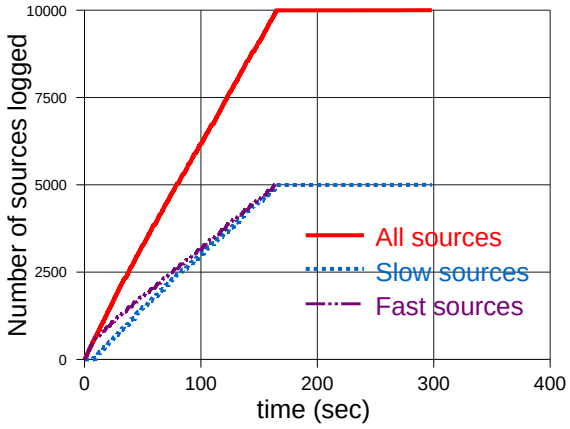
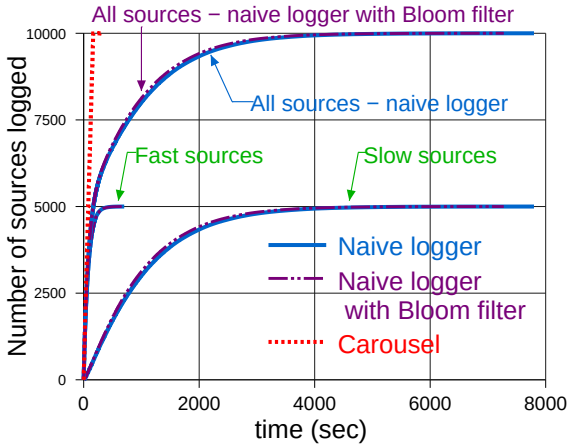


Figure 2.14. Scaling up the vulnerable population



(a) Carousel



(b) Naive logger

Figure 2.15. Logger performance under non-uniform source arrivals

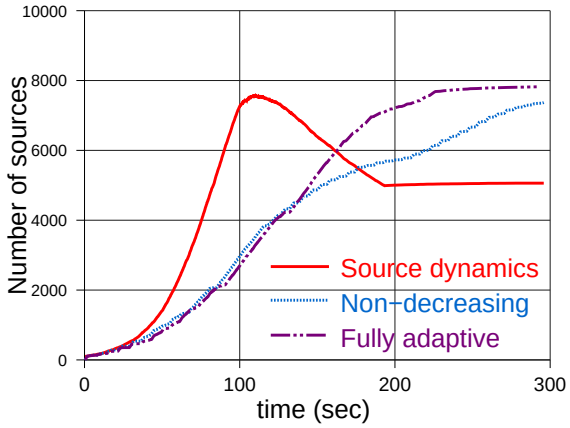
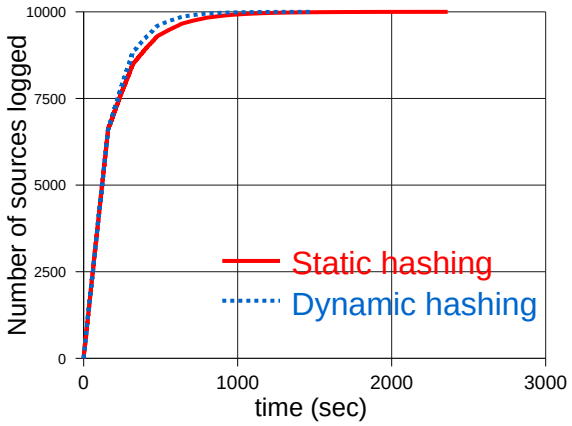
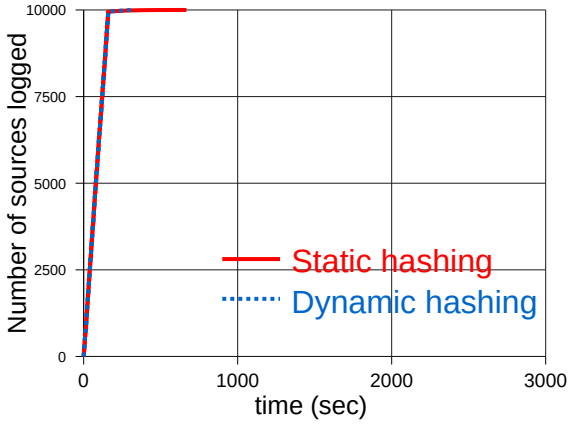


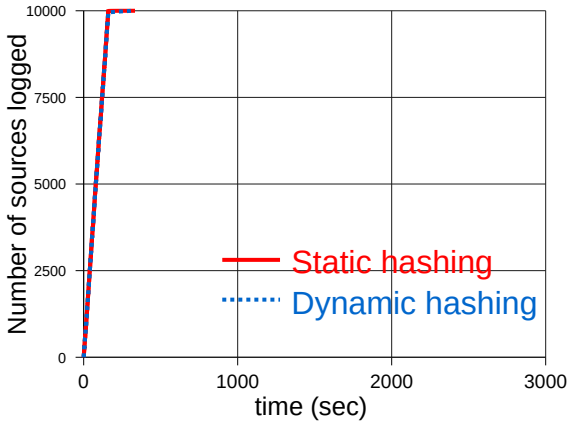
Figure 2.16. Dynamic source sampling in Carousel



(a) 1-bit Bloom filter



(b) 5-bit Bloom filter



(c) 10-bit Bloom filter

Figure 2.17. Comparison of fixed vs. changing hash functions in Carousel

slightly more keys in every phase at the cost of a somewhat increased false positive probability. Based on our experiments, we believe using 5 bits per element provides excellent performance, although our Snort implementation (built before this experiment) currently uses 10 bits per element.

2.6.5 Adaptively Adjusting Sampling Bits

As described in Section 2.4.2, an optimization for Carousel is to dynamically adapt the number of sampling bits k to match the currently active source population. In a worm outbreak, the value of k needs to be large as the when the population of infected sources is large, but it should be decreased when the scope of the outbreak declines.

To study this effect, we use the *two-factor worm model* [78] to model the dynamic process of worm propagation coexisting with worm remediation. The two-factor worm model augments the standard worm model with two realistic factors: dynamic countermeasures by network administrators/users (such as node immunization and traffic firewalls) and additional congestion due to worm traffic that makes scan rates reduce when the worm grows. The model was validated using measurements of actual Internet worms (see [78]).

In Figure 2.16, we apply the two-factor worm model. The curve labeled "Source dynamics" records the number of infected sources as time progresses. Observe the exponential increase in the number of infected sources prior to $t = 100$. However, the infected population then starts to decline.

If we let the two-factor model run to completion, the number of infected sources will eventually drop to zero, which makes logging sources less meaningful. In practice, however, it is the logging that makes remediation possible. Thus to illustrate the efficacy of using fully adaptive sampling within the logger, we only apply the two-factor model until the infectious population drops to half of the initial vulnerable tally. We then look at the time to collect the final infected population. Note that a non-decreasing logger will choose a sampling factor based on the peak population and thus may take unnecessarily long to collect the final population of infected sources.

Figure 2.16 shows that the fully adaptive scheme (increment k on overflow, decrement on underflow) enhances performance in terms of logging time and also the capability to collect more sources before they are immunized. In particular, the fully adaptive scheme collects almost all sources at 220 seconds while the non-decreasing scheme (only increments k on overflow, no decrements) takes more than 300 seconds to collect all sources. Examining the simulation results more closely, we found the non-decreasing scheme adapted to $k = 5$ (32 partitions) and stayed there, while the fully adaptive scheme

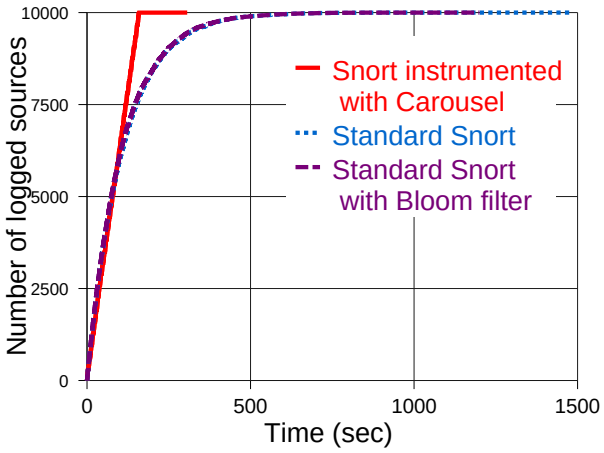


Figure 2.18. Logging performance of Snort instrumented with Carousel under a random traffic pattern

eventually reduced to $k = 4$ (16 partitions) at time $t = 130$.

2.7 Snort Evaluation

We evaluate our implementation of Carousel in Snort using a testbed of two fast servers (Intel Xeon 2.8 GHz, 8 cores, 8 GB RAM) connected by a 10 Gbps link. The first server sends simulated packets to be logged according to a specified model while the second server runs Snort, with and without Carousel, to log packets.

We set the timer period $T_{phase} = 5$ seconds. The vulnerable population is $N = 10,000$ sources and the memory buffer has $M = 500$ entries. In the first experiment, the pattern of traffic arrival is random: each incoming packet is assigned a source that is uniformly and randomly picked from the population of N sources.

Figure 2.18 shows the logging performance of Snort instrumented with Carousel. Traffic arrives at the rate (B) of 100 Mbps. All packets have a fixed size of 1000 bytes. The logging rate is $b = 100$ events per second, i.e., $b \approx 1$ Mbps and $\frac{B}{b} = 100$. Figure 2.18 shows the improvements in logging from our modifications. Specifically, our scalable implementation is able to log all sources within 300 seconds while standard Snort needs 1500 seconds. Also, adding a Bloom filter does not significantly improve the performance of Snort, matching our previous theory.

Figure 2.19 shows the logging performance when the sources are perpetually dispatched in a periodic pattern 1, 2, ..., N, 1, 2..., N, ... Such highly regular traffic patterns are common in a number of

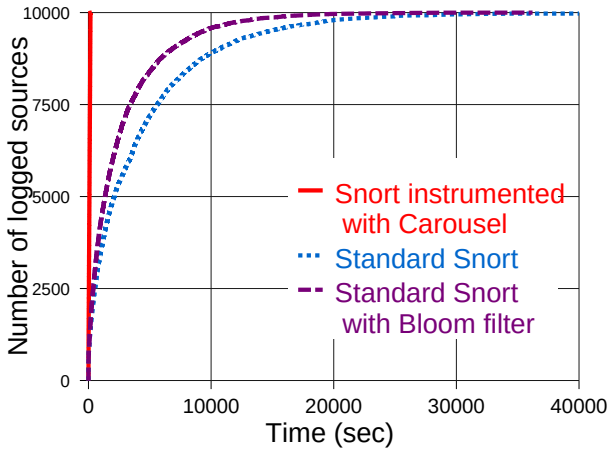


Figure 2.19. Logging performance of Snort instrumented with Carousel under a periodic traffic pattern

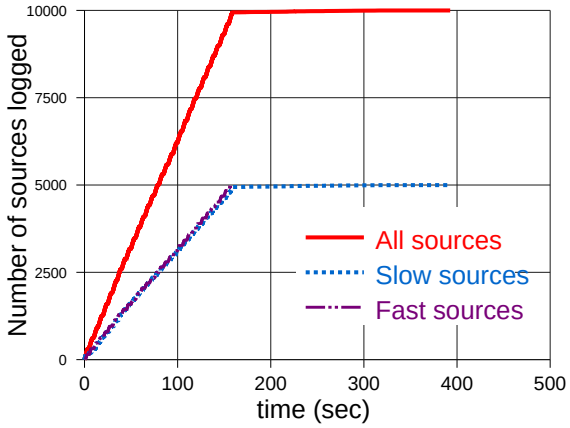
practical scenarios, such as synchronized attacks or periodic broadcasts of messages in the communication fabric of large distributed systems. We observe that the performance of standard Snort degrades by one order of magnitude as compared to the random pattern shown in Figure 2.18. Further examination shows that the naïve logger keeps missing certain sources due to the regular timing of the source arrivals. On the other hand, Carousel performance remains consistent in this setting.

We also performed an experiment with two equally sized sets of sources arriving at different rates, with fast sources arriving at 1 Gbps and slow sources at 100 Mbps, as shown in Figure 2.20. Our observations are consistent with the simulation results in Section 2.6.3. Note that in this setting standard Snort takes about 20 times longer to collect all sources than Snort with Carousel (300 seconds versus 6000 seconds); in contrast, Snort took only about 5 times longer in our experiment with random arrivals.

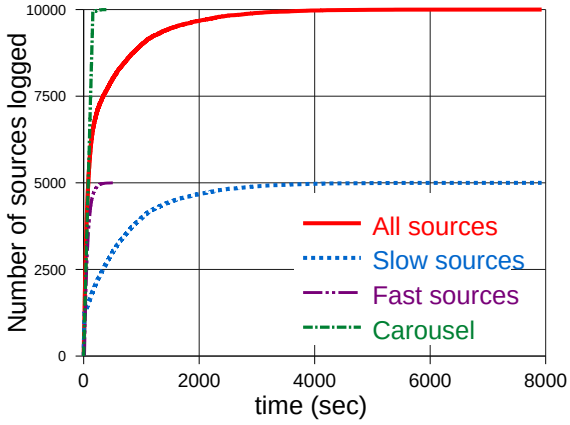
2.8 Related Work

A number of recent papers have focused on high speed implementations of IPS devices. These include papers on fast reassembly [22], fast normalization [74, 75], and fast regular expression matching (e.g., [68]). To the best of our knowledge, we have not seen prior work in network security that focuses on the problem of scalable logging. However, network managers are not just interested in detecting whether an attack has occurred but also in determining which of their computers is already infected for the purposes of remediation and forensics.

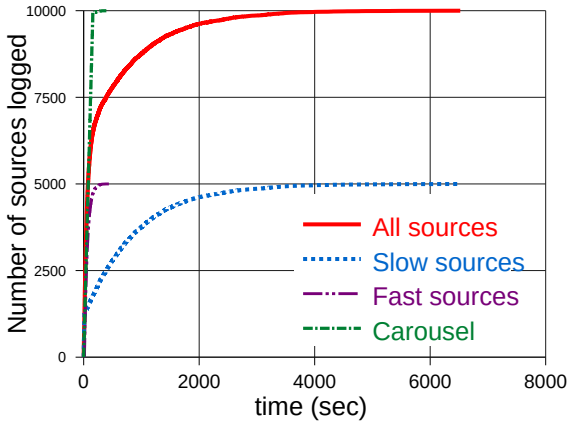
The use of random partitions, where the size is adjusted dynamically, is probably used in other



(a) Carousel



(b) Standard Snort



(c) Standard Snort with Bloom filter

Figure 2.20. Snort under non-uniform source arrivals

contexts. We have found a reference to the Alto file system [42], where if the file system is too large to fit into memory (but is on disk), then the system resorts to a random partition strategy to rebuild the file index after a crash. Files are partitioned randomly into subsets until the subsets are small enough to fit in main memory. While the basic algorithm is similar, there are differences: we have *two* scarce resources (logging speed and memory) while the Alto algorithm only has one (memory). We have duplicates while the Alto algorithm has no duplicate files; we have an analysis, the Alto algorithm has none.

2.9 Summary

In the face of internal attacks and the need to isolate parts of an organization, IPS devices must be implementable cheaply in high speed hardware. IPS devices have successfully tackled hardware reassembly, normalization, and even Reg-Ex and behavior matching. However, when an attack is detected it is also crucial to also detect who the attacker was for potential remediation. While standard IPS devices can log source information, the slow speed of logging can result in lost information. We showed a naïve logger can take a multiplicative factor of $\ln N$ more time than needed, where N is the infected population size, for small values of memory M required for affordable hardware.

We then described the Carousel scalable logger that is easy to implement in software or hardware. Carousel collects nearly all sources, assuming they send persistently, in nearly optimal time. While large attacks such as worms and DoS attacks may be infrequent, the ability to collect a list of infected sources and bots without duplicates and loss seems like a useful addition to the repertoire of functions available to security managers.

While we have described Carousel in a security setting, the ideas applies to other monitoring tasks where the sources of all packets that match a predicate must be logged in the face of high incoming speeds, low memory, and small logging speeds. The situation is akin to congestion control in networks; the classical solution, as found in say TCP or Ethernet, is for sources to reduce their rate. However, a passive logger cannot expect the sources to cooperate, especially when the sources are attackers. Thus, the Carousel scalable logger can be viewed as a form of randomized admission control where a random group of sources is admitted and logged in each phase. Another useful interpretation of our work is that while a Bloom filter of size M cannot usefully remove duplicates in a population of $N >> M$, the Carousel algorithm provides a way of recycling a small Bloom filter in a principled fashion to weed out duplicates in a very large population.

Chapter 2, in full, is a reprint of the material as it appears in “Carousel: Scalable Logging for

Intrusion Prevention Systems" in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* 2010. Lam, V. T., Mitzenmacher, M., and Varghese, G., USENIX, 2010. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Flame: Efficient and Robust Hardware Load Balancing for Data Center Routers

3.1 Introduction

To support growth in cloud applications, data centers offer higher aggregate bandwidth by utilizing multiple paths in the network [31, 4, 33]. For example, the standard data center network topology is a fat-tree where edge switches load balance across a set of paths to core switches. To fully exploit the aggregate bandwidth of these abundant multipaths, effective network load balancing is crucial in order to allow core network bandwidth beyond that allowable by link technology. For example, today 10 Gbps core links are reasonably priced and 40 Gbps links are expensive. Therefore, as the size of network grows with a growing number of edge links, the only way to economically scale large data centers is to load balance traffic across multiple 10 Gbps core links.

While this is a classic trend in networks, what makes the problem more difficult today is the presence of potentially high bandwidth edge flows. 10 Gbps has reached the edge; with fast CPUs and adaptors, it is not unreasonable for a single TCP flow to go beyond 5 Gbps. As the number of high-bandwidth edge flows increases, customers are increasingly finding that load balancing performance at core links is unsatisfactory for reasons we explain below. Indeed, this problem is far from an academic curiosity, router vendors are actively looking to improve the state-of-the-art.

In this paper, we investigate this load balancing problem. In particular, our goal is to spread network load across all available paths [1] in order to realize bandwidth equal to the sum of the paths.

[1]While port aggregation and multi-pathing are distinct switch features, the forwarding hardware is nearly the same. In this paper, we will refer to them both as *multi-pathing*. A *path* refers to a physical port in port aggregation and a physical path in true multi-pathing.

However, it is traditionally required that packets within a flow [2] be delivered to the TCP stack in order. If they are not, performance of that connection can suffer, due to the additional processing required to handle the re-ordered packets or due to TCP sender congestion window reduction, re-transmissions and timeouts that are triggered by the re-ordering as we quantify in Section 3.6.4.

There are two general methods the above requirements are addressed and they both use hashing. The first method is a common load balancing algorithm used in routers, called *equal-cost multi-path* routing (ECMP), using a *static* hash. ECMP implies that load balancing is done only over equal cost paths, while static hash assigns a flow to a path by hashing the *TCP/IP 5-tuple* with a *single* hash function [3]. In particular, ECMP hashes the 5-tuple of each packet and a modulus operation is then applied to the result to get a path number and the packet is sent out the corresponding path. The modulus applied is the number of possible paths. Note that static hashing is computationally fast and requires no state. It also guarantees no reordering of TCP flows as long as paths do not change. However, it has a drawback that is specific to this approach in addition to the drawbacks that both common approaches have. If the number of paths changes, either up or down, the modulus changes. This means that the result of the hash and modulus for any given flow is likely to change which results in packet reordering. This potentially can happen to all active flows.

The second method creates the same hash, but, rather than applying a modulus, it uses the hash result to index into a relatively small table. Each table entry then has the path number to use and the packet is sent out that path. For example, a 256-entry table might be used and the maximum number of potential paths might be 16. In this case, each path ID would be loaded into the table 16 times. However, if the number of paths is 15, then 14 of the paths will be in the table 17 times and one will be in the table 18 times which leads to an inherent imbalance. One nice feature of this approach is that when the number of available paths changes, only the traffic that needs to be moved is affected unlike the previous approach.

While the above approaches are universally implemented, they have poor load balancing performance when there are large edge flows, as the following examples demonstrate.

Example 1: Assume we wish to balance 4 flows each with 6 Gbps of bandwidth across 4 equal cost paths of 10 Gbps. The offered load (24 Gbps) is smaller than the network capacity (40 Gbps). Assuming a static hash that distributes uniformly, the probability that all 4 flows will pick distinct paths is

[2] A *flow* refers to all the packets of a single TCP connection and is identified by a unique 5-tuple.

[3] *5-tuple* is IP source and destination addresses, TCP source and destination port numbers, and the protocol field.

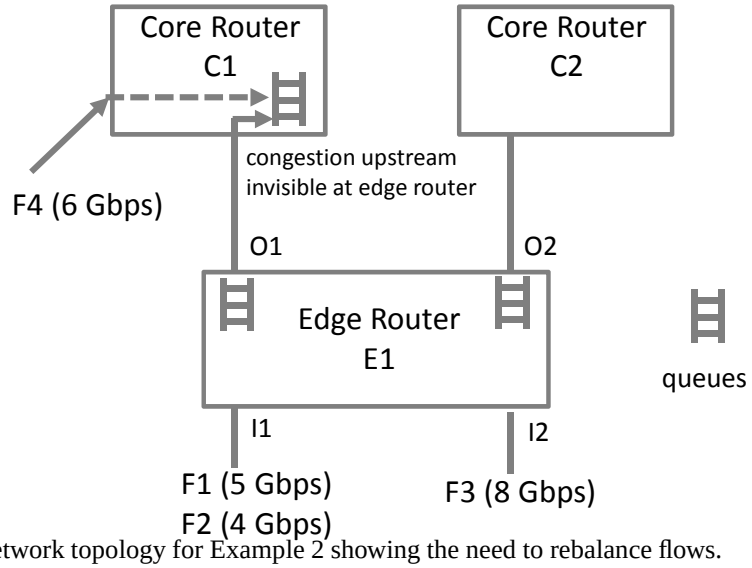


Figure 3.1. Network topology for Example 2 showing the need to rebalance flows.

only 24/256, less than 10%. Thus with 90% probability, at least two 6 Gbps flows will be assigned to the same 10 Gbps link and thus will be throttled to 5 Gbps each even though there is a completely unassigned 10 Gbps link. There is also significant probability that three flows pick the same link. It is difficult to explain to customers why expensive 10 Gbps links remain unused!

One can quantify the problem with static hash by computing the standard deviation of the number of flows per path compared to the mean. If n flows are uniformly randomly assigned to p paths, then the number of flows follows a Bernoulli distribution (each flow is assigned a path with probability $\frac{1}{p}$). The mean number of flows per path is $\frac{n}{p}$ and the standard deviation is $\sqrt{\frac{n}{p}(1-\frac{1}{p})}$. As n grows large, the deviation grows as the square root while the mean grows linearly; thus the deviation becomes insignificant as the number of flows increases. But the deviation *is significant* when there are a small number of large flows. For example, consider two TCP flows being hashed on to two paths. The mean number of flows per path is $\frac{1}{p} = 1$ but the standard deviation is also very close to 1. Intuitively, if there are two flows, there is a probability of 50% that they choose the same path. Thus with 50% probability we get no load balancing; with 50% probability we get perfect balancing, resulting in a large average deviation.

Note that the second approach described above does not explicitly address this problem. Indeed, the tables today are statically configured so a 4 path solution will have 1/4 of the table entries statically assigned to each path. This becomes equivalent to using a modulus, but allows a more graceful changing of the number of paths the traffic is balanced across.

Example 2: Beside random assignment, a second culprit is fixing a flow assignment indefinitely.

Assume an edge router (Figure 3.1) with two input links $I1$ and $I2$ of 10 Gbps, and paths to two different core routers $C1$ and $C2$ via output links $O1$ and $O2$ of 10 Gbps each. Consider three flows, $F1$, $F2$, and $F3$ where $F1$ arrives first on $I1$ and sends at 5 Gbps. A short time later $F2$ arrives on $I1$ and sends at 4 Gbps. Some time later, $F3$ arrives on $I2$ and sends at 8 Gbps. This is a feasible traffic pattern because there is no more than 10 Gbps arriving on any input link. Assume that static hash ECMP gets "lucky" and assigns $F1$ to $O1$ and $F2$ to $O2$. At this point in time, traffic is well balanced. However, when $F3$ arrives, static hash can only assign $F3$ to either $O1$ or $O2$. In either case, we have at most 5 Gbps on one output link and at least 13 Gbps on the other output link. For instance, if $F3$ is assigned to $O1$, then 13 Gbps cannot be sustained on a 10 Gbps output and so queues will build on $O2$ or downstream in core router $C1$. The "right packing" would be to move $F2$ back to $O1$ along with $F2$ and then to assign $F3$ to $O2$. This would assign 9 Gbps to one link and 8 Gbps to another link.

We propose Flame, an efficient dynamic load balancer to address these challenges. In particular, we turn away from static flow-to-path assignment but instead attempt to do the assignment dynamically in an intelligent manner to avoid the imbalance as described above. Our contribution includes the following key ideas.

a. *New bandwidth estimator:* We propose a new Discounting Rate Estimator (DRE) to accurately measure path loads without relying on queues (which may be empty at this router) (Section 3.3.1). DRE responds much faster to new bursts than an exponential weighted moving average (EWMA) while retaining memory of past bursts. By relying on such instantaneous load information of each path to make decision on path assignment of the next new flow, we can achieve significant improvement over static assignment such as ECMP.

b. *Remembering hash functions not paths:* We devise a robust and implementable path choice technique by using standard power of choice hashing to pick the least loaded link which reduces hardware comparisons in real-time (Section 3.3.2). Unlike Flare, however, we remember the *hash* corresponding to the least loaded link and not the *path*. Remembering hash functions is more robust when there is limited memory and two flows collide in the same bucket because it is no worse than ECMP. This allows a simpler hardware implementation (by k-way comparison). Further, it takes a few bits to remember a hash function as opposed to nearly 128 bits to remember a flow. We present both mathematical analysis and experimental results to show that our scheme never degrades below static hashing regardless of the memory capacity.

c. *Hardware for 48-port 10 Gbps switch:* We introduce a number of simple techniques to make the implementation feasible (Section 3.4). In particular, we propose a hash table instead of a per-flow state table to deal with memory overflow gracefully; we integrate heavy-hitter detection to maximize the efficiency of the hash table; and we show how to incorporate periodic load balancing at any parameterized value (from say 1 in 10 packets to 1 in 100,000) in *hardware* (Figure 3.4).

d. *Periodic rebalancing:* We show how periodic rebalancing can be implemented in our framework (Section 3.3.2). Note that heavy hitter detection is even more important with regard to rebalancing because the heavy-hitters send more packets and can hence afford to have periodic load balancing for the global good without greatly impact their own TCP throughput.

e. *Load balancing metrics:* We present a framework and analytical measures for characterizing the goodness of load balancing schemes (Section 3.5.2 and 3.6.1). While this is implicit in earlier works, we make this explicit. We also discuss guidance for setting the parameters in our scheme.

f. *Updated experiments on the effect of rebalancing on TCP:* In Section 3.6.4, we describe new experiments with various combinations of Windows and Linux stacks to show the effect of rebalancing on TCP. As expected, Windows stacks degrade considerably in throughput when rebalancing more frequently than 1 in 100,000 (because out of order packets lead to DupAcks that cause the congestion window to fall). More surprisingly we show that the latest Linux stacks (after 2.6.14) allow load balancing as often as 1 in 10 packets with at most 10% loss in throughput.

3.2 Related Work

The classic way to load balancing has been to do random load balancing via hashing as in ECMP. Such a hashing scheme comes with many advantages. First, it is inexpensive because it requires no state per TCP flow and a small amount of logic to do the hash. Second, it is compatible with TCP because it does not reorder packets. It is well known that because of the fast retransmit option, more than three packets out of order can be interpreted as loss which can be followed by unnecessary retransmission, reduction of the congestion window, and overall loss of throughput. For example, [43] shows experiments where reordering in the order of more than 0.1% affects TCP throughput and reordering of more than 10% drastically reduces throughput. Third, ECMP does a reasonable job of balancing flows when the number of TCP flows being balanced is large.

While ECMP has been a standard for every router, there are several recent alternative proposals in the academic literature. Flare [39] goes beyond static hash using two ideas. First, long flows are broken into multiple flowlets based on a packet gap timeout. Second, the first packet of each flowlet is allocated to the least loaded link and the result is stored in a flowlet table and used to route all subsequent packets in the flowlet. If the flowlet timeout is larger than network latency then no reordering should occur despite reassigning the flow across flowlet boundaries. The Flare paper uses wide area traces to claim that the flowlet table is small because the number of concurrent flowlets is much smaller. In Example 2, if $F1$ and $F2$ are sufficiently spaced apart, $F2$ is *guaranteed* to be assigned to a different link by Flare unlike static hash. Flare does no repacking and will not address the imbalance in Example 2 when $F3$ arrives — unless $F1$ or $F2$ have a sufficiently long gap which allows their respective flowlets to be reassigned.

Hedera [5], on the other hand, does not attempt to optimally place flows when they first arrive but instead waits until flows are measured as "heavy-hitters" and then reassigns flows based on a heuristic packing algorithm implemented in software on a centralized switch controller. Doing this assumes Open Flow [47] environment or MPLS to control flow routes via software. Hedera allows entire paths to be rebalanced which goes beyond link-by-link balancing as in Flare; for example, in Figure 3.1 it allows flow $F1$ to be assigned to the $E1, C1, E3$ path and flow $F3$ to be assigned to the $E1, C2, E4$ path. However, it not immediately deployable in today's networks by changes to single routers. Further, Hedera only allows flows to be repacked every few seconds (a good choice as we show later); but this implies a few seconds worth of imbalance when a new flow such as $F3$ arrives in Figure 3.1.

Finally, note that if one could invent a reordering-resilient TCP or simply reorder packets in the destination network adaptor, then one could simply spray packets of a flow across links to get near-optimal load balancing. A more approximate alternative is to realize that the problem is caused by a few large TCP flows and could be mitigated by splitting large TCP flows into multiple TCP subflows at the source as in Multipath TCP [60]. However, both Hedera and Multipath TCP are clean slate approaches while Flare only requires implementation changes within a single switch. We choose to work in the same setting but we point out the following problems with those previous works.

a. Memory: Flare indicates that the memory can be small but this is for one trace and for one particular flowlet timeout parameter. We show real traces where the number of flowlets can be quite high. We also show that when the memory is smaller than the number of flowlets, hashing must be used and hash collisions can cause Flare to perform worse than ECMP.

b. Implementation: previous works assume that one can easily compute the least loaded link in hardware. In reality, doing this computation at high speeds across say 32 links is prohibitive at 20 Gbps and higher. With 48 ports of 10Gbps Ethernet and a lookup rate of approximately 750 million lookups per second and a clock rate of 750MHz gives us 1.3nsec per clock cycle. This makes it hard to minimize (across up to 32 registers) in a clock cycle. The computation can also not be done in the background because different destinations may use different subsets of the links for equal paths and the number of possible subsets (2^{32}) is too large to precompute.

c. Load balancing effectiveness: the current norm is that a flow is rebalanced just once when it is created. However, as we have seen in Example 2, a heavy flow may never be timed-out and yet the path loads may change leading to more optimal load balancing if the flow is reassigned. Hedera [5] suggests periodic load balancing across routes but it is unclear how often this can be done without harming TCP throughput and does so in software without integrating into the hardware. Furthermore, it appears that one can do better if periodic reassignment is done (at a rate less than say the 0.1% threshold that reduces TCP throughput [43]) even in flowlet like hop-by-hop load balancing schemes.

3.3 Mechanisms

In this section, we describe the essential ingredients of our dynamic load balancing scheme. We first describe the concept of a *Discounting Rate Estimator* to accurately measure link loads. We then show our design of flow state table to enforce packet order in the same flow. Finally, we discuss our handling of heavy-hitter flows to reduce memory in hardware implementation of the state table and also to perform periodic rebalancing. These mechanisms are combined in Figure 3.4.

3.3.1 Discounting Rate Estimator (DRE)

In this section, we design a bandwidth estimator for link bandwidths to assign new flows to the least loaded link. Two requirements for a bandwidth estimator to do:

Quick reaction to new bursts: In Example 2, if $F2$ arrives a short time after $F1$ and $F1$ has been assigned to output link $O1$, we would like the link estimator for $O1$ to quickly ramp up so that $O1$ looks "more loaded" than $O2$ and $F2$ is (correctly) assigned to $O2$.

Remembering old bursts: In Example 2 again, suppose that $F2$ arrives 100 usec after $F1$ has finished sending at 5 Gbps for a few seconds. At this moment assume that $O1$'s queue is empty and so is

Algorithm 1. Discounting Rate Estimator (DRE)

```
Parameters:
  T_P: DRE timer period
  R_P: DRE discount ratio
for each path i do
  initialize shallow counter Q[i] = 0
end for
loop
  if packet D sent to path i then
    Q[i] = Q[i] + D.size
  end if
  if proxy queue timer T_P expires then
    for each path i do
      Q[i] = Q[i] − Q[i] · R_P
    end for
  end if
  if f is new flow then
    assign f to path of smallest Q
  end if
end loop
```

that of $O2$. However, the effect of $F1$'s burst may still remain downstream at core router $C1$ in Figure 3.1. Thus, we would like the path estimator at the edge router to "remember" the fact that $F1$ has sent a burst for a small period equal to the network latency (say 300 usec in a data center). Otherwise, $F2$ could wrongly be assigned to $O1$ causing unnecessary congestion at core router $C1$.

Let us see how four standard estimators do with respect to these requirements:

1. Epoch estimator: Bandwidth is traditionally measured in epochs such as 1s, 1ms or 1us. We count how many bytes are sent in one epoch interval and obtain one measurement point. Then we reset the counter value, count the bytes in the next epoch interval to get another measurement point. We send a packet to the link with the smallest current epoch counter and update that links epoch counter. In other words, this is a time-averaged and memoryless approach. The problem with the epoch estimator is that it keeps no memory after a burst which ends close to the end of an epoch. For example, in Example 2, if $F1$ ends just before the end of an epoch, and $F2$ arrives soon after the epoch ends, there will be no memory of $F1$'s burst and $F2$ could be wrongly be assigned to $O2$. One could also modify the least loaded choice to use the epoch estimator of the last epoch; but that will be even worse, because it will not react fast if $F1$ and $F2$ start in the same epoch.

2. Token estimator: The Flare paper [39] uses a token counting approach of subtracting the ideal bytes to be sent on each link from the actual bytes and then sends packets to the link with the least tokens. Finally, to avoid keeping memory forever, periodically, the token counters are reset to zero. While this

generalization is useful when doing fractional load balancing across links, it is identical to the epoch estimator in the case of whole flow load balancing. Thus, again it keeps no memory of past bursts across measurement intervals.

3. Exponentially weighted moving average (EWMA): The simplest way to keep memory of past bursts is to use an EWMA estimator on the epoch bandwidth measurements using small epoch periods. If the EWMA estimator uses a small weight for new information and a large weight for past information, past information will die down gradually but not instantaneously. But in this case, the EWMA will react quite slowly to new arrivals such as $F1$ and this may not be fast enough if $F2$ follows close on $F1$s heels. On the other hand, if we make the weight for recent information to be high and the weight for past information to be low, then the past is forgotten very quickly.

4. Physical queue size: In Figure 3.1, can we use the physical queue size of output link $O1$ and output link $O2$ to determine the least loaded link? Unfortunately, this does not work. After flow $F1$ has been assigned to output link $O1$, the physical queue at the edge router is likely to be zero because $F2$ is 5 Gbps and the link is 10 Gbps. But $F2$ can cause congestion at the upper core router $C1$ because of a fourth flow $F4$ that wishes to go on the same path. This congestion is invisible at the edge router but can be avoided by the estimator by measuring that a large number of bytes have been assigned to output link $O1$ which suggests that flow $F2$ be assigned to output link $O2$. Fundamentally, physical queue size does not work because it does not reflect past traffic sent at a rate smaller than the link bandwidth.

It is to reconcile the simultaneous demands of fast reaction to new bursts with memory of old bursts that we were led to design a new rate estimator called Discounting Rate Estimator (DRE). Pseudocode for DRE is shown in Algorithm 1. DRE keeps a counter Q_i for each switch port output link i which is incremented by the packet size when a packet is transmitted on that output link. However, the counter for path i is not periodically reset to zero. Instead, every period say T_P, the counter is decreased by an amount *proportional* to the current counter value. We call the proportionality factor the *discount factor* R_P. If R_P is chosen to be a power of 2, discounting can be implemented in hardware using a shifter and a subtractor. For example, in Section 3.6 we use very small values of DRE parameters such as $T_P = 100$us and $R_P = 1/512$.

Intuitively, we expect that the proxy queue gives good indication of traffic bursts, but also accumulate some data irrespective of the link utilization (within some limit). It drains slower as it gets near empty so when the real utilization is low, we still get a reading.

As we will show in our analysis in Section 3.5.1, DRE quickly reacts to new bursts because it

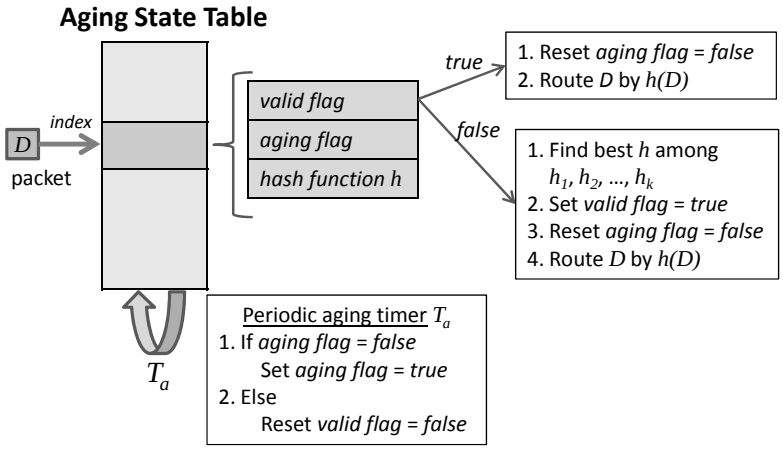


Figure 3.2. Overview of Flame state table design. Packet D is indexed into a fixed-size table by hashing (Section 3.3.3) or exact-matching (Section 3.3.4). A new table entry is set up by comparing k paths as specified by k independent hash functions $h_1, h_2, \ldots, h_k$. The period aging timer is triggered every time interval T_a to age out inactive table entries.

simply adds the packet bytes. DRE also remember old bursts because every period, the DRE counter is merely discounted by R_P and not reset. Note that a small discount ratio R_P translates to a large weight to older events and vice versa. For example, with $R_P = 1$, we reduce back to the epoch-based approach. The DRE counter will also not diverge to infinity because the higher the counter, the greater the discounting effect. We prove formally in Section 3.5.1 that the DRE counter stays bounded, is a scaled rate estimator, and balances rise and fall times for new and old bursts. DRE is almost identical to EWMA except that while EWMA weights both old and new information, DRE only weights past information. While this is a simple change, it makes a great difference to rate estimation.

3.3.2 Choosing the least loaded link

In Section 3.1, we discussed several examples showing the unsatisfactory performance of a static hash-based approaches such as ECMP to do link assignment to flows. Instead, we opt for using instantaneous traffic utilization (DRE in Section 3.3.1) on each link to make decision on path assignment of the next new flow. To do this, we wish to examine traffic utilization on all links to pick the least-loaded one to assign to the new path and then save that choice for subsequent packets of the same flow in order to prevent reordering. In this section, we describe how to choose the least loaded link. At first glance, this seems straightforward. When a new flow F arrives, the forwarding table yields the set of equal cost paths P for F. Next, simply read the DRE counters of all links in P and assign F to the path with the smallest DRE counter. Unfortunately, this is non-trivial for three reasons:

1. Large number of potential paths: In data centers today, 8 and 16-way multipathing are common but there is growing interest in multi-pathing as high as 32 or even 64. More concretely, consider a fat-tree topology that is maximal in diameter and the top-of-rack switch that has 96 ports. With 40 servers in the rack, there will be 40 uplinks which results in 40 ECMP paths (or 64 by rounding up to the next power of 2).

2. Rising traffic rates leading to small time budget: In the market today there are already 48 ports of 10 Gbps Ethernet each. These require a lookup rate of approximately 750 million lookups per second. This means a clock rate of 750MHz. This gives us 1.3nsec per clock cycle which is near the limit of ASIC technology (doubling the clock frequency to get more time is a non-starter). This allows for only a small number of register reads, not 40.

3. Exponential numbers of potential ECMP path sets: If there were 64 ECMP paths used by *all* flows, one could do incremental computation by keeping a pointer to the least loaded link; when a DRE counter is updated, if it is lower than the current lowest the pointer is updated. Unfortunately, each flow can use a different subset of the 40 output links, leading to 2^{40} possible subsets, too many to keep state for, let alone update. Consider the case when an edge router E has 32 outlinks to 32 core routers. One of the core routers, say C, has a failed downlink to an edge router E'. Then, flows from E to E' cannot be routed by the output link to C but the remaining flows can. Similar patterns of failure can result in every possible subset of paths being chosen by some flow.

We cope with the small time budget and the small number of register reads possible using power-of-choice hashing [49] as an intermediate approach between all paths minimization and pure static hash-based computation as follows. When a flow first starts, we hash it with k independent hash functions h_1, h_2, …, h_k function to get k paths, say P_1, P_2, …, P_k. If P_i is the least loaded path, we assign the new flow to P_i. So far this is standard power of choice for load balancing as has been proposed for server load balancing [49].

What is new in our setting is the need to maintain flow order to avoid TCP throughput degradation. Instead of remembering the *path* P_i in a hash table, we remember the *hash* h_i that generated the least loaded path index. This is a good idea for two reasons. First, we can remember more flows if the state is smaller: the state needed to remember a hash is $\log_2 k$ (2 bits for 4 hash functions!) is much smaller than the 128 bits required to remember a TCP flow. Second, since we do not store the flow ID, then we have to deal with hash collisions. Remembering a hash function is *more robust than remembering a path* because if two flows collide, the second flow will not use the path of the earlier flow but the hash of the earlier

flow. Thus the collided flow has a significant chance of being assigned to a different link, no worse than static hash ECMP; on the other hand, we show examples later where remembering paths is much worse than ECMP. Thus remembering hashes is more economical in memory and more robust.

Another advantage is the ability to handle the following problem that occurs frequently in practice. When many heavy flows start at about the same time, the considering all-paths approach would assign all of them to the same least loaded path. On the other hand, since our hybrid approach uses only a small number of hash functions (k), there is a high probability that this problem will not happen. We have a formal analysis for this problem in Section 3.5.2.

Note that in the special case $k = 2$, we further enhance the computation of two-hash choices so as to guarantee no hash collision as follows. (Otherwise, done independently, there is a $1/4$ probability that the two will pick the same link to sample which is wasteful.) Despite this coupling, the two hash functions are "sufficiently independent" to guarantee good sampling, based on some recent unpublished work by Mitzenmacher.

Let p be the number of equal cost paths. Then given a flow f, we compute its two path choices by the following formulas.

$$P_1 = h_1(f) \mod p \tag{3.1}$$
$$P_2 = (h_2(f) \mod (p-1) + 1 + h_1(f)) \mod p$$

3.3.3 State table design

Section 3.3.2 discussed a practical technique to dynamically determine an optimal path for the next new flow that can improve significantly over static assignment such as ECMP. However, since TCP congestion control requires that packets in the same flow should arrive in the same order, we want to preserve the packet order by dispatching all packets in the same flow to the same path. We need a mechanism to store flow states so as to preserve the packet order in the same TCP flow. In this section, we describe our design of this state table, which is a core module of our Flame load balancing scheme.

Figure 3.2 demonstrates our approach to keeping flow state and shows how state is updated on packet arrival. In particular, when a packet arrives, if the packet's flow is not in the state table, we use the power-of-choice method in Section 3.3.2 to assign it to the optimal path and insert the chosen hash into the table. The packet is also sent to the path just selected. On the other hand, if the packet's flow is already in the table, then we just send the packet to the path indicated by the stored hash applied to the

packet's flow ID.

Intuitively, this dynamic path assignment approach is better than a static assignment such as ECMP because of the highly dynamic nature of flow arrival and departure in real networks. In practice, even large flows come and go and assigning that flow to a path when the flow starts will give the best possible assignment at that instant. The choice may turn out to be poor based on the behavior of the new flow in the future and the other, already assigned flows. behavior in the future, but subsequent flows will continually correct any such mistakes. This is especially true if the behavior of a flow is fairly constant, i.e. high bandwidth flows tend to remain high bandwidth and low bandwidth flows remain low.

While [39] indicates that the number of concurrent flowlets is small, their traces were wide-area traces. For large data center traces, we see no reason why the number of concurrent flowlets cannot be much larger. One challenge is that a forwarding ASIC cannot afford to keep memory equal to the number of flows/flowlets. In order to reduce the amount of state to a more affordable level, we use a hash of the flow (instead of exact-matching on the flow ID) to index into the state table. If the flows are hashed into a state table of, say, 1024 buckets, then each bucket can be dynamically assigned a path just as a flow was assigned a path dynamically.

When a flow dies out, we need to reclaim state. We will do this with an aging algorithm in this section. Alternatively, we will discuss a technique to keep flow state for a much smaller subset comprising of only significant flows in Section 3.3.4. To age out and free up state entries as soon as possible, we use a simple and efficient aging mechanism akin to LRU page eviction. We associate an aging field (typically a one-bit flag, but could be larger for finer granularity) with each state table entry and update it when a packet arrives. A complete design of the Flame state table with the aging mechanism is shown in Figure 3.2. Each table entry is a record with three fields: *1. valid flag*: indicates whether the table entry is valid and contains the state of an active flow; *2. aging flag*: marks inactive or idle flows. *3. hash function*: points to one of the k hash functions $h_1, h_2, \ldots, h_k$.

When a packet arrives, its flowID is indexed into the state table, which maps the packet to one entry in the state table. The valid flag is inspected to check the validity of table entry. If the table entry is valid, the packet is dispatched according to the *hash function h* stored in the state entry. The aging flag is also cleared to indicate that the flow entry is active. On the other hand, if the state entry is invalid, we set up a new valid table entry by comparing k paths as specified by k independent hash functions $h_1, h_2, \ldots, h_k$ as in Section 3.3.2. The hash function resulting in the optimal path is saved in the table entry h. So subsequent packets of the same flow are guaranteed to use the same hash function and be dispatched to

the same path to prevent packet reordering.

Further, a timer process visits every table entry every *aging* timeout T_a. When it visits a table entry, it either turns on the aging flag or invalidates the entry if the aging flag is already on. In other words, T_a is the timeout threshold to age out inactive flows. Note that with this aging timer process, the aging interval is in the range between T_a and $2T_a$.

Note that as shown in a previous study [39], network traffic is inherently bursty. This timing mechanism allows us to leverage the burtiness by essentially splitting a long flow into several much smaller flowlets (i.e. burst of nearby packets). The flowlets are determined by a flowlet timeout, the idle interval between two successive flowlets. Note that if the time between two successive packets is larger than the path latency difference, the two packets can be routed on two different paths without changing their arrival order at the receiver. By setting $2T_a$ to be larger than the maximum latency difference between two paths in the network, flowlets belonging to the same flow can be independently switched into different paths without causing packet reordering. While [39] suggests a flowlet timeout of 60 ms based on wide area traces, we suggest that this be a parameter and even numbers like 100 usec may be reasonable in a modern data center.

Therefore, the aging timeout T_a is an important design turning knob. If T_a is too large, the table entries may never be idle long enough to be aged out and thus re-assigned. Or, more likely, they do get aged out, but too infrequently to be effective. Furthermore, it is likely that those entries that have high-bandwidth flows would not be aged out since they, by definition, have smaller gaps between packets. On the other hand, if T_a is too small, we increase the likelihood of packet reordering in the same flow. So in practice, T_a should be related to the round-trip-time of the network. For example, that could be tens or hundreds of microseconds in a data center environment. Since an entry in the state table is flagged idle after the first aging timeout and eventually deleted after the second aging timeout, the effective flowlet timeout is between one to two times of T_a.

As we discussed earlier, designing the state table as a hash table allows two or more flows to hash to the same bucket with a subsequent flow $F2$ using the same state that was established by the previous flow that hashed into the bucket say $F1$. This is essentially the same problem encountered in the current implementations, however possibly to a lesser degree since the number of hash buckets can be larger than what is typically in use today. Note that a collision is undesirable for two reasons. First, it reuses the state of the first collided flow that has not timed out in that entry. Since the entry was previously optimized to that different flow, reusing its state is suboptimal. Second, it virtually bridges the flow/flowlet gap

by holding on the aging flag, and hence reduces the number of flows/flowlets that could have been split and routed independently. However, in our example, while this collision results in $F2$ not using optimal information (the state of the links seen by $F1$ may be outdated when $F2$ arrives), since $F2$ only applies the hash stored by $F1$, $F2$ will quite likely be assigned to a different link from $F1$. Storing the path would not have this property. While this may not be as optimal as assigning $F2$ to the least loaded link when $F2$ arrives, it is no worse than ECMP. This property of *graceful degradation* to static hash ECMP when the number of flows is too large seems an essential property of load balancers that keep state.

3.3.4 Handling heavy-hitters

In Section 3.3.3, we discussed the aging algorithm with fixed-size hash table. When a packet arrives, it is hashed into the table and routed according to the valid table entry. If the table entry is invalid, a new path is determined according to a greedy heuristic such as least loaded path of DRE counters (Section 3.3.1). Each table entry is aged out if it is idle beyond a certain threshold. We noted that collisions of distinct flows into the same table entry (e.g. by existence of spoilers as we will discuss in Section 3.5.2) can have significant adverse impact on the performance of the algorithm. Concretely, since a flow $F2$ can reuse the state set up by an earlier flow $F1$, if $F1$ is a low rate flow, while $F2$ sends at 5 Gbps, then $F2$ will be routed by a static hash but this is not optimal. If $F1$ or $F2$ keep sending, $F1$'s entry will never be timed out and $F2$ will keep using essentially ECMP instead of a more optimal assignment. In this section, we seek to combat this problem by applying the load-balancing algorithm only to *heavy-hitters* (i.e. *elephant* flows) since only a small number of heavy-hitters are responsible for a large fraction of traffic in the network [31, 6]. Non-heavy-hitters (i.e. mice flows) can be treated with the standard ECMP. However, since the hardware logic for the hash table is inexpensive, we can opt for a hybrid approach by having both an exact-matching state table for heavy-hitters and a hash-based state table in Section 3.3.3 for non-heavy-hitters.

As illustrated in Figure 3.3, every flow is treated with ECMP or the hash table approach until it is established as a heavy-hitter. Upon being classified as a heavy-hitter, it is assigned to a new path and its state is stored in the Flame heavy-hitter table. Then subsequent packets of the same flow are routed by exact matching to the respective entry in the heavy-hitter table. Each entry in the heavy-hitter table also has an aging flag so that it is periodically aged out at every flowlet expiry timeout.

Next, we describe our heavy-hitter filter module so that only true heavy-hitters are assigned to the heavy-hitter table with high probability. We also discuss the process to reclaim unused state by

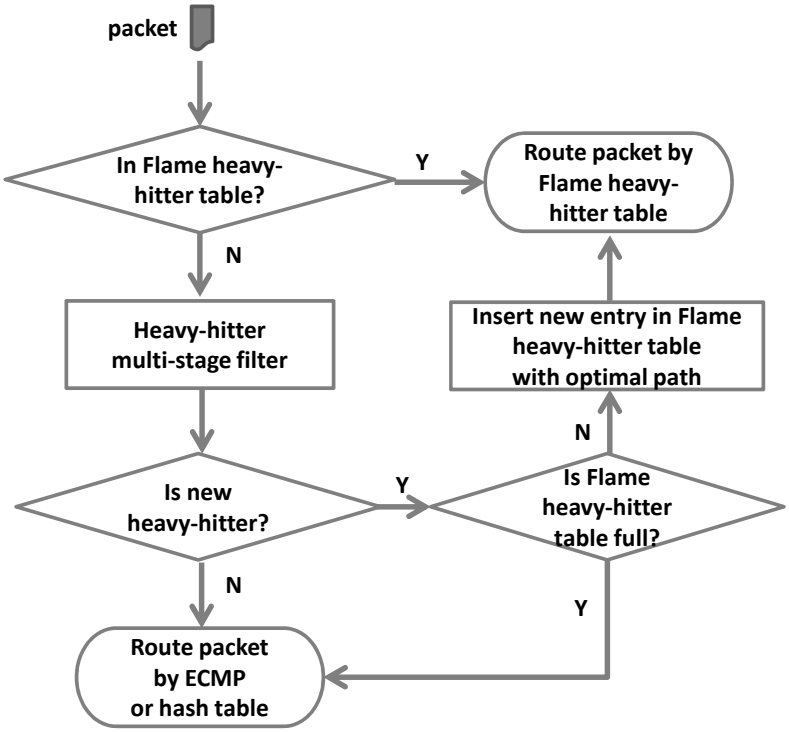


Figure 3.3. Overview of Flame scheme with an exact-matching heavy-hitter table. A flow is routed according to the Flame heavy-hitter table once it is classified as heavy-hitter. Each entry in the Flame heavy-hitter table is also periodically aged out with an aging flag and aging timeout. Non heavy-hitters can use hash state table in Section 3.3.3.

evicting flows out of the heavy-hitter table when they become mice or terminate. Due to hardware memory constraints, this is necessary to make room for new heavy-hitters.

Since heavy-hitter admission and eviction can introduce another form of TCP packet reordering, we need a parameter F (reordering parameter) to decide how often a flow can be reordered without causing undue harm to TCP. For example, a prior experimental study [43] suggests that TCP performance will not be adversely effected if the number of reorderings is less than 0.1%, i.e. $F = 1000$. Our updated study reported in Section 3.6.4 suggests that F can be much worse $(32,000)$ for Windows Server destinations and much better (even $F = 10$) for recent Linux versions. Thus we leave F as a parameter.

Thus we retain a heavy-hitter for F packets before applying any change in load balancing policy. We keep a packet counter for each heavy-hitter in the heavy-hitter table. Every time the packet counter reaches multiple of F, we set up a rebalancing flag. The aging algorithm would be then invoked to assign the heavy-hitter to another path, which is likely to be better.

With this preliminary, we now describe our heavy-hitter filter module so that only true heavy-

hitters are assigned to the heavy-hitter table with high probability. We also discuss the process to reclaim unused state by evicting heavy-hitters.

We employ a heavy-hitter multistage filter algorithm with conservative update of counters as in [24]. There are four heavy-hitter detection tables. Each table consists of 1024 counters. A packet is indexed into these tables by four different hash functions. The counters are updated with the packet size according to the conservative update rule. If the counters at *all* four tables exceed a threshold B_H, the flow is classified as a heavy-hitter. Typically $B_H = 3$ KBytes in our experiments. The counters are reset to zero every T_H interval. Typically $T_H = 30$ msec in our experiments.

If a flow passes the heavy-hitter filter, it will be admitted into the heavy-hitter table and switch the load balancing policy from ECMP/hash-table to heavy-hitter based Flame. One approach is to allow graceful switching from ECMP/hashtable to heavy-hitter policy. That is, at the beginning we set the valid flag to true and the path selection to the same as ECMP/hash-table (i.e. skipping computing least-loaded path). This guarantees that the switching from ECMP/hashtable to heavy-hitter based Flame occurs when either (immediately whichever happens first) the current heavy-hitter flowlet aging out and beginning a new flowlet, or after the first F packets.

However in our experiments, we opt for doing the first reordering early and then subsequent ones being paced at F packets. In other words, once a flow is classified as heavy-hitter, we do an abrupt admission into the heavy-hitter table, and so the flow is immediately switched from ECMP/hashtable to the heavy-hitter scheme. Such *abrupt insertion* can cause a reordering when the the heavy-hitter is first detected while graceful insertion will not.

We now turn to eviction. If the bandwidth of a heavy-hitter falls below some threshold, the flow should be removed from the heavy-hitter table. Our goal is that with memory cleanup, the memory-constrained Flame scheme is better than ECMP and almost as good as the case with infinite memory. For example, we suppose the heavy-hitter table contains only 2048 entries in our experiments.

As a first principle, we let the eviction policy be less stringent than the insertion policy. We propose the following eviction policy: evict a flow if it sends less than B_H bytes in k_e consecutive periods of T_H. In our experiments, typically $k_e = 3$.

When a heavy-hitter is evicted, its state can be immediately deleted from the heavy-hitter table (called *abrupt eviction*), and then subsequent packets of that flow are routed under ECMP. Clearly abrupt eviction can result in packet reordering. Instead, we propose a different approach, called *graceful eviction*, which exploits both the flowlet aging expiry timeout and the “reorder no frequently than every F packets”

rule. In particular, when the heavy-hitter traffic is below a threshold, we turn on an *eviction-ready* flag and start counting the remaining packets. Then we only physically delete it from the heavy-hitter table upon either *i)* next flowlet aging expiry or *ii)* packet count $> F$. The two conditions make graceful eviction effective against both inactive and slow heavy-hitters. This facilitates our goal of evicting heavy-hitters very soon after they become idle to minimize the heavy-hitter table size and accommodate new heavy-hitters.

In practice, we have a predetermined budget on the size of the heavy-hitter table and want to tune the heavy-hitter admission and eviction parameters so that the heavy-hitter table is always near full. However, network traffic can vary widely from time to time and exceed the heavy-hitter table capacity. With our design for heavy-hitter admission and eviction, we can tune to accept fewer heavy-hitters by increasing B_H and/or decreasing T_H. Note that by reducing T_H, we evict more frequently.

3.3.5 Profile-based rebalancing

Rebalancing a heavy-hitter by greedily selecting the least utilized path is vulnerable to the following synchronized *greedy flash crowd* effect: when many flows are rebalanced almost simultaneously, all of them most likely make the same path choice. Such situations occur in our experiments of bandwidth-intensive synthetic flows in Section 3.6.3, which are rebalanced frequently by the one in F packet rule. We observe that the synchronized greedy flash crowd effect can lead to severe oscillation in path assignment and poor load balancing performance.

As an example, consider ten heavy-hitter flows and three paths P_1, P_2, and P_3. The best path assignment is by having three flows to path P_1 and P_2 each and four flows to path P_3. Let's denote this path assignment by a triple $(3,3,4)$, i.e. each number in the triple denotes the number of heavy-hitters being assigned to the respective path. Since our greedy method in Section 3.3.2 is not perfect, typically we only get a near-optimal assignment such as $(2,4,4)$. Now if the heavy-hitters are rebalanced frequently, with the initial path assignment $(2,4,4)$, they all will be reassigned to path P_1, leading to the subsequent path assignment $(10,0,0)$. Next, they all will move away from path P_1, leading to the path assignment $(0,10,0)$, and so on. Clearly, such oscillation is undesirable.

We propose the following traffic profiling approach to mitigate this problem. First, we profile heavy-hitter traffic by having a counter per heavy-hitter that counts heavy-hitter bytes in the *previous* epoch. Second, we also maintain a path profile, which is initialized to the path traffic in the *previous* epoch. Third, if we need to rebalance a heavy-hitter, we will rely on both the heavy-hitter profile and the

path profile. We ensure that the heavy-hitter reassignment only *improves* the path profile. Finally, we also adjust the path profile by the amount of the flow profile at the rebalancing moment. This avoids moving multiple heavy-hitters to the same new path.

As an example, suppose we have three paths with path profile (P_1, P_2, P_3) = (8 KBytes, 9 KBytes, 11 KBytes). If the heavy-hitter were from path P_3 with heavy-hitter profile 2 KBytes, we would rebalance it to path P_1 and update the path profile to (10 KBytes, 9 KBytes, 9 KBytes). However, if the heavy-hitter were from path P_2 with heavy-hitter profile 2 KBytes, we would still keep it at path P_2 since moving to path P_1 would lead to an even worse state.

More concretely, we define *profile deviation* to be the difference between max path profile and min path profile. Then rebalancing is aborted if new profile deviation is larger than a factor k_p of the old profile deviation. Intuitively it is natural to pick $k_p = 1$. However, our experimental results showed that a larger and hence more relaxed value (e.g. $k_p = 2$) is sufficient to prevent the greedy flash crowd effect, but at the same time does not hurt the performance by being less restrictive and allowing more rebalancing opportunity. Note that in theory, setting $k_p > 1$ is always prone to oscillation under certain network traffic behavior since that condition allows path profile to get worsened continually with time. One way to bound the profile deviation is by instituting a tiered value for k_p. For example, we can use the following simple scheme:

- $k_p = 2$ if path profile deviation $\leq L_1$
- $k_p = 1.5$ if path profile deviation $> L_1$ and $< L_2$
- $k_p = 1$ if path profile deviation $\geq L_2$

where L_1 and L_2 ($L_2 > L_1$) are two thresholds depending on the profile epoch and network bandwidth.

3.4 Hardware implementation

Figure 3.4 describes a hardware block diagram for a chip we are building that puts together all the mechanisms we described in the last section including the DRE estimation, the Flame table, and hardware rebalancing parameterized by the parameter F that can do rebalancing as often as once every 10 packets (feasible with Linux receivers) or as infrequently as once every 100,000 packets (needed for Windows receivers, see Section 3.6.4).

Start at the top of Figure 3.4. The forwarding logic provides a base address into the path table and the number of paths p. The flow ID f is hashed using a hash function $F1$ but that is modified (see

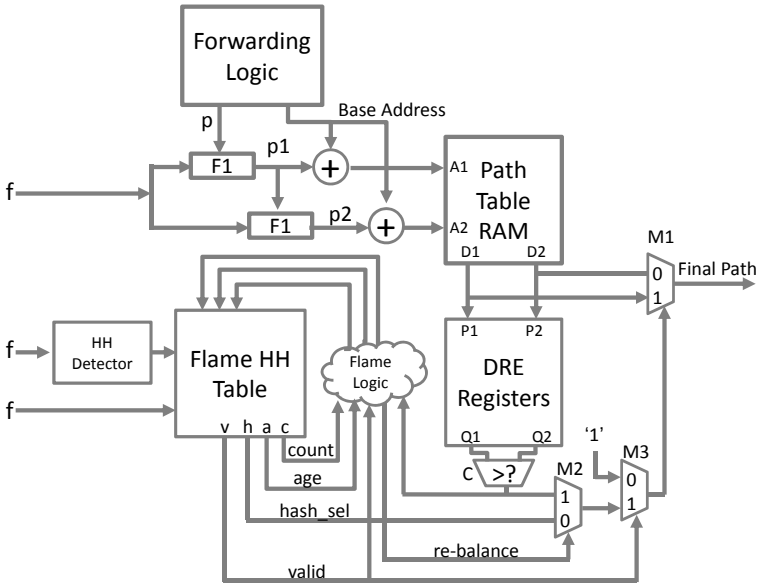


Figure 3.4. Flame hardware schematic

Equation 3.1) in the lower path to essentially compute a second hash function. This produces two offsets $p1$ and $p2$ that are added to the base address and used to index in parallel into a dual-ported memory using addresses $A1$ and $A2$. Note that a dual-ported memory has two read ports and is almost as expensive in gates as two independent memories. Thus using more than two hash functions could be expensive. The path table stores the DRE counters. The two addresses $A1$ and $A2$ yield two link IDs $D1$ and $D2$ from the path table. (This level of indirection allows graceful handling of path failures). The two link IDs are used to index into the DRE registers to produce two DRE values $Q1$ and $Q2$. Comparator C picks the least loaded link of the two and outputs the result to multiplexor (mux) $M2$.

Now move to the bottom of the figure. Concurrently, the flow ID f is also fed to the Flame table whose output is four values: a valid flag v, a hash select flag h (since we use only two hashes for power of choice, 1-bit suffices), and age bit a, and a count c (an integer of at least 17 bits capable of counting to $32,000$). If the valid flag is "false" (the flow has no valid entry), the mux $M3$ will select the input 1 and pass it as the selection value for mux $M1$, (note that when the selection bit shown at the bottom of a mux is 1, the output corresponds to the input labeled 1, and vice versa). In this case, mux $M1$ picks the output link $D1$ and the forwarding is exactly as in static hash ECMP. This is correct because when there is no state for f we should use ECMP.

If the valid flag is "true" (the flow entry is valid), the mux $M3$ will select the input 0 which is the

output of mux $M2$. This value from $M2$ is then fed to $M1$ to select the proper output link, either $D1$ or $D2$.

When the "re-balance" signal is 0, the $M2$ mux will select as its output the hash select signal coming from the Flame table. On the other hand, if the re-balance signal is a 1, the mux $M2$ selects as output the least loaded link from the output of comparator C as we described above. This is fed via mux $M3$ to set the select signal for $M1$ which now actually selects the least loaded path as the output of mux $M1$, The rebalance signal is computed by the (simple) Flame logic. If either the age is 0 or the $count > F$, the rebalance signal is asserted. At the same time, the least loaded link output of comparator $C1$ is fed back via the Flame logic to be stored in the Flame Table.

Note that unlike Hedera in which software periodically identifies heavy-hitters and moves flows to paths, the entire rebalancing process is done in hardware. This is essential if one wishes to do fine-grain rebalancing say once every 10 or 100 packets which is possible without significant TCP degradation for Linux servers as we show in Section 3.6.4.

3.5 Analysis

In this section, we analyze the components of our Flame load balancing scheme to investigate its performance guarantee. We first develop an analytical model for DRE design in Section 3.5.1. Then we present theorems about the robustness of Flame in Section 3.5.2.

3.5.1 DRE analysis

We analyze the DRE proposal in Section 3.3.1 to show that it reacts quickly to new bursts, is robust and is independent of arrival rates. In particular, we show that the stabilized of the DRE counter value scales inverse proportionally to the DRE decay rate, which can be easily converted back to the arrival rate.

Let T_P and R_P be the timer period and discount ratio parameters for DRE. Let $q(t)$ denote the value of the DRE counter at time t. Our DRE model is described by the following differential equations:

$$\frac{dq(t)}{dt} = \alpha - \kappa \cdot q(t) \tag{3.2}$$

where α is the instantaneous traffic arrival rate and κ is the instantaneous DRE decay rate.

Note that with very small value of T_P, we compute the instantaneous decay rate as $\kappa \approx R_P/T_P$.

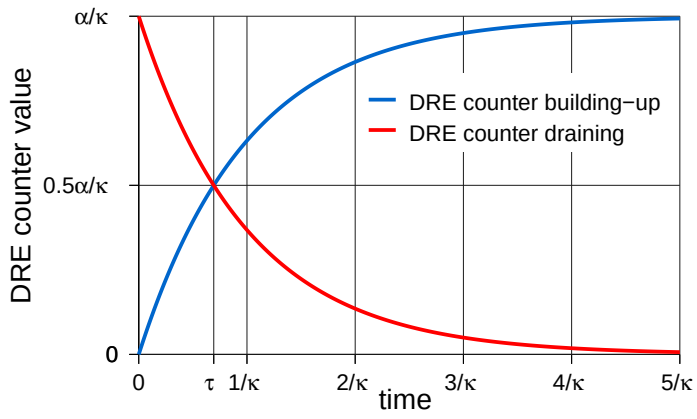


Figure 3.5. Convergence of DRE counter under constant traffic arrival rate α followed by an abrupt stop to traffic. κ denotes the instantaneous DRE decay rate. τ is the intersection point of the two scenarios.

Suppose α is a constant for all time t, we can solve equation (3.2) as follows.

$$\begin{aligned} \frac{d(\alpha-\kappa q(t))}{\alpha-\kappa q(t)} &= -\kappa dt \\ \ln(\alpha-\kappa q(t)) &= -\kappa t+\gamma' \\ q(t) &= \frac{\alpha-\gamma e^{-\kappa t}}{\kappa} \end{aligned} \tag{3.3}$$

where γ and γ' are constants determined by initial conditions

Equation (3.3) indicates that when a flow has been sending at rate α for a while and stops sending, its DRE counter starts from the stabilized value α/κ and then decays exponentially to 0. On the other hand, when a fresh flow starts sending at rate α, its DRE counter starts from 0 and increases exponentially to the stabilized value α/κ. The exact equations for these two scenarios are as follows.

DRE counter building-up: Suppose a flow has not been sending before time 0^- and then starts sending at time 0^+ with a rate α. Then the boundary conditions are $q(0)=0$ and $\gamma=\alpha$. Hence, its DRE counter value can be described by:

$$q_1(t) = \frac{\alpha-\alpha e^{-\kappa t}}{\kappa} \tag{3.4}$$

DRE counter draining: Suppose a flow has been sending at rate α up to time 0^- and then it stops at time 0^+. The boundary conditions are $q(0)=\alpha/\kappa$ and $\gamma=-\alpha$. Hence, its DRE counter can be described by:

$$q_2(t) = \frac{\alpha e^{-\kappa t}}{\kappa} \tag{3.5}$$

Stabilizing point of DRE counter: Figure 3.5 illustrates the DRE counter building up and draining scenarios. Let τ denote the intersection time. By solving $q_1(\tau) = q_2(\tau)$, we get

$$\tau = \ln(2)/\kappa \tag{3.6}$$

Since network traffic is not continuous but consists of discrete datagram packets, we also verified our model using Matlab simulations with practical data center settings (e.g. bandwidth 10 Gbps and 20 Gbps). In particular, we validated two important properties of the DRE design: the cross point τ is independent of the arrival rate α and $\tau = \ln(2)/\kappa$. From Figure 3.5, we observe that DRE counters are bounded and eventually converge as long as the arrival rate is bounded. Further, the DRE counters converge quickly as measured by the metric τ and so the DRE timer period should be larger than τ.

Note the important property that the parameter κ itself defines where the cross point is, and is independent of the arrival rate α. In the context of load balancing flows, we believe that τ should be in the order of the queuing delay in the network. In particular, we can set the DRE parameters according to the formula $T_P \times R_P = d$ where d is the network delay.

3.5.2 Analysis of Flame state table design

In this section, we analyze Flame, particularly the hash-table based approach of Section 3.3.3. We argue that Flame is robust and outperforms Flare in general and yet there are certain circumstances in which Flame degenerates to ECMP but in which Flare does poorly.

Our notation is as follows. Let k be the number of hash functions in Flame (Section 3.3.2). Let p be an upper bound on the number of equal paths individually denoted as P_1, P_2, ..., P_p. Let n be the number of heavy-hitters, individually denoted as H_1, H_2, ..., H_n. Let m be the number of entries in the state table. Let T_a be the aging timeout. We assume the finite memory versions of Flame and Flare. Recall that Flame remembers one of k hash functions, while Flare one of p paths. Any state table entry is timed out after at most $2T_a$ because of the LRU approximation. When Flare or Flame insert a flow into a hitherto invalid entry, the algorithm measures the current state of all paths and places the flow in the least loaded path: we sometimes refer to this as "sensing" in what follows.

First, observe trivially that if $k = 1$, Flame behaves exactly like ECMP. Next, our first theorem shows that while sensing and assigning to the least loaded path appears to be a good idea it can sometimes backfire when there are bursts. That means least loaded path schemes do not always outperform ECMP.

Theorem 1. (Burst vulnerability) *Under bursty arrivals of heavy-hitters, any scheme that allocates a new flow to the least loaded path and preserves flow packet order can perform much worse than ECMP for arbitrary time periods.*

Proof. Consider the following traffic scenario for Flare. Suppose the amount of memory m is much larger than the number of heavy-hitters n so that all heavy-hitters hash to distinct entries in the hash table, with no hash collisions. Without loss of generality, suppose the first heavy-hitter H_1 is assigned to path P_1. Then bring on the second heavy-hitter H_2 after enough time for our load measuring algorithm to "sense" H_1. Without loss of generality, suppose H_2 is assigned to path P_2. Repeat until H_1 through H_{p-1} have been assigned to paths P_1 to P_{p-1} respectively. Now bring on simultaneously H_p , H_{p+1}, ..., H_n. Since at the start the new heavy-hitters have not sent any traffic, this last burst of heavy-hitters will be assigned to path P_p. Now suppose each of the heavy-hitters continue to send traffic for arbitrary time. Then none of the entries assigned to the heavy-hitters will time out. Thus all will have valid entries, and the system will never sense the links for the least loaded link and reassign because no new flows arrive. However, that means for an unbounded period of time, $(n-p+1)$ heavy-hitters are assigned to path P_p and one heavy-hitter apiece of P_1 through P_{p-1}, leading to an unbounded load discrepancy over any time scale. $|n-p|$ can be made arbitrarily large by increasing n).

Flame is susceptible to the same "flash crowd" scenario but its imperfect sensing actually makes it somewhat more likely to spread flows out better. For example, if $k=2$ and p is large, when a later heavy-hitter comes, the probability that neither of the two hash functions picks path P_p is $(1-\frac{1}{p})(1-\frac{1}{p-1}) = 1-\frac{2}{p}$, i.e. $=75\%$ with $p=8$. So Flame places 25% of the heavy-hitters on P_p. On the other hand, ECMP would put roughly $1/8 = 12.5\%$. □

The flash crowd scenario of this theorem has two implications. First, it shows all sensing schemes are vulnerable to flash crowds where flows arrive simultaneously and can do worse than ECMP though Flame is better than Flare. This and Example 2 in the introduction suggests that periodic rebalancing is not merely a desirable but a requirement. Second, it shows why heavy-hitter detection can help even in the case of flash crowds if there is sufficient memory to keep state for each heavy-hitter. Note that if there are more than 1000 flows, static hash ECMP should work well because the standard deviation falls with the square root of the number of flows. Thus keeping state for around 1000 heavy-hitters should alleviate this scenario for either Flare or Flame.

Next, we show that remembering paths in Flare can cause robustness problems for Flare but not

for Flame. The problem can arise due to *spoilers*, small flows that capture hash table entries early.

Theorem 2. (Spoiler resilience) *Flame is resilient to the presence of spoilers and is no worse than ECMP. However, Flare can be arbitrarily worse than ECMP.*

Proof. Flame resorts to one more level of hashing within a table entry, so degrades gracefully to ECMP in the presence of spoilers. Now consider Flare in the following scenario. We first bring on the first $p-1$ heavy-hitters, well-spaced out in time so as to get assignment in paths P_1 through P_{p-1}. Then, we bring on $O(m)$ spoilers that capture all remaining entries that are left invalid. Since the spoilers are small, they are all assigned to path P_p. Thus, all state table entries are marked as valid, in which $p-1$ entries are assigned to paths P_1 through P_{p-1} and the remaining $m-p+1$ cells are assigned to path P_p. In fact, the system is reasonably load balanced at this time.

Next, we bring on the remaining heavy-hitters H_p, H_{p+1}, ..., H_n nicely spaced in time so that none is bursty. If m is large, the majority of the state table is filled with path P_p and valid bit set. Thus, the later heavy-hitter will likely pick such a "spoiler entry" and be then assigned to path P_p. Thus with high probability, all later heavy-hitters will be assigned to path P_p. If all heavy-hitters continue sending for an arbitrary period of time, the situation will persist and no further sensing will take place because no entry times out. In other words, we now have $(n-p+1)$ heavy-hitters to path P_p and only one each assigned to paths P_1 through P_{p-1}. So by increasing n without bound, we have arbitrarily bad average and worst case load discrepancy. Even if the spoilers stop sending traffic completely after a heavy-hitter arrives in their cell, the heavy-hitter will keep the cell from timing out even though the recorded path information is prehistoric. □

If m is small, Flare gets into the following memory starvation problems. First, if $m < p$, Flare can only use m out of p paths. Such starvation seems unlikely as the paths are often quite small ($p \leq 128$ in large data centers) and if state memory is static RAM or even registers, it is possible to get $m \gg 100$, e.g. in 1000s. A second starvation regime occurs when the memory $m < n$ and $n > p$. By pigeonhole principle, two heavy-hitters will fall in the same bucket and be willy-nilly be treated as one bigger heavy-hitter and Flare cannot distinguish them.

Theorem 3. (Small memory) *If $m \leq p$, then Flare does not use $(p-m)$ paths while Flame still uses all paths. If $m \leq n$, then the load discrepancy in Flare can be $O(\log n)$.*

Proof. The first part is obvious. The second part is equivalent to throwing n balls in m bins and determining the worst case discrepancy in balls between the least loaded and heaviest loaded. From balls and bins

theory we have $O(\log n/\log\log n)$ discrepancy in the load if $m = n$. □

Theorem 4. (Greedy bin packing) *If there are no heavy-hitter bursts and no spoilers, and $m \gg n$, then Flare will behave like a greedy bin packing algorithm that can outperform ECMP.*

Proof. Without bursts, the heavy-hitters come in spaced widely apart. Then with sufficient memory, each heavy-hitter can get its own entry with high probability. Consider a heavy-hitter H with its own table entry. Without spoilers, the last regular flow F entered the same entry would be quite recent (at most T_a old). Since F is not a heavy-hitter, it senses the paths at a time after the last heavy-hitter H' that preceded H has "settled" (the measurement algorithm has reliably sensed its rate). Thus this is equivalent to a greedy algorithm that simply places the heavy-hitters in sequence in the least loaded path at each iteration. As a simple example, suppose we have three heavy-hitters and three paths. ECMP will not use one path with probability $(1-1/3)^3 = 8/27$, i.e. 30% chance of discrepancy at least $2B$, where B is the bandwidth of each path, while the greedy algorithm will place them on separate path. □

Theorem 5. (Squatter susceptibility) *If the total number of flows is much larger than the state table capacity m, even if $m \gg n$, then Flame will do no better than ECMP.*

Proof. If the next heavy-hitter H comes in sufficiently spaced apart and there are other flows come in between, then it is very likely that the entry H hashes to is already occupied not by a spoiler but a squatter. The squatter has indeed picked its own hash but this hash has as the same chance of assigning a future heavy-hitter to the least loaded link as ECMP. □

Doing no worse than ECMP is a good robustness guarantee for Flame. However, since we aspire to *do better* than ECMP, it is better to avoid collisions for large flows (as far as possible) using heavy-hitter filters. One way to make Flame behave like the greedy algorithm is to change the hash when a heavy-hitter arrives but that can cause reordering for the squatter and the heavy-hitter. However, this could be mitigated in practice as follows. If a heavy-hitter sends a lot more packets than other flows that are not heavy-hitters, it is perhaps much more likely to be best first packet than squats in an entry after an entry times out or is invalid.

3.6 Evaluation

In this section, we experimentally study how Flame does with respect to ECMP and Flare. We begin with a discussion on the appropriate metrics to quantify load balancing goodness. Then we describe

our implementation to simulate several load balancing schemes on a realistic network trace. Due to the lack of public data center traces, we enhance realistic Internet traces with data center-like heavy-hitters, the number, duration, and intensity of which we can control for. Finally, we present the experimental results showing the efficacy of the Flame scheme. We also show results of benchmarking TCP performance under packet reordering at 1 and 10 Gbps which inform the choice of the hardware rebalancing parameter F in Figure 3.4.

3.6.1 Load balancing goodness metrics

In this section, we describe a diverse set of metrics for quantifying load balancing effectiveness that are abstract and independent of particular algorithms. Denote by T_s the measurement time scale parameter. First, fix a value of T_s and divide the traffic trace into disjoint and contiguous time intervals of length T_s. Then for each interval, measure the *path traffic vector* accumulated during the interval and compute the *balancing quality* within the interval using a load balancing *goodness metric* as shown below.

Let p be the number of paths and $P_1, \ldots, P_p$ be paths. For path P_i ($1 \leq i \leq p$), denote $P_i.load$ as the network traffic on path P_i during the current time interval. We denote average load on all paths as $\overline{P}.load = \frac{1}{p}\sum_{i=1}^{p} P_i.load$. We consider $\overline{P}.load$ to be the ideal balance in the current time interval. Then we propose three goodness metrics:

1. Absolute deviation: worst case bandwidth difference of one path from the ideal balance

$$G_d \triangleq \max_{P \in \{P_1,\ldots,P_p\}} \frac{|P.load - \overline{P}.load|}{T_s}$$

2. Normalized deviation: percent of bandwidth difference from the ideal

$$G_n \triangleq \max_{P \in \{P_1,\ldots,P_p\}} \frac{|P.load - \overline{P}.load|}{\overline{P}.load}$$

3. Jain's fairness index: standard fairness metric for a set of p load values

$$G_{\mathscr{J}} \triangleq \frac{(\sum_{i=1}^{p} P_i.load)^2}{p \cdot \sum_{i=1}^{p} (P_i.load)^2}$$

Note that load balancing quality is better with smaller deviation value and higher Jain's fairness index. Next, the goodness values in all intervals of size T_s form a time series for which we can calculate statistics such as max, average, and $99th$ percentile. One final complication remains: what is a good

choice of T_s?

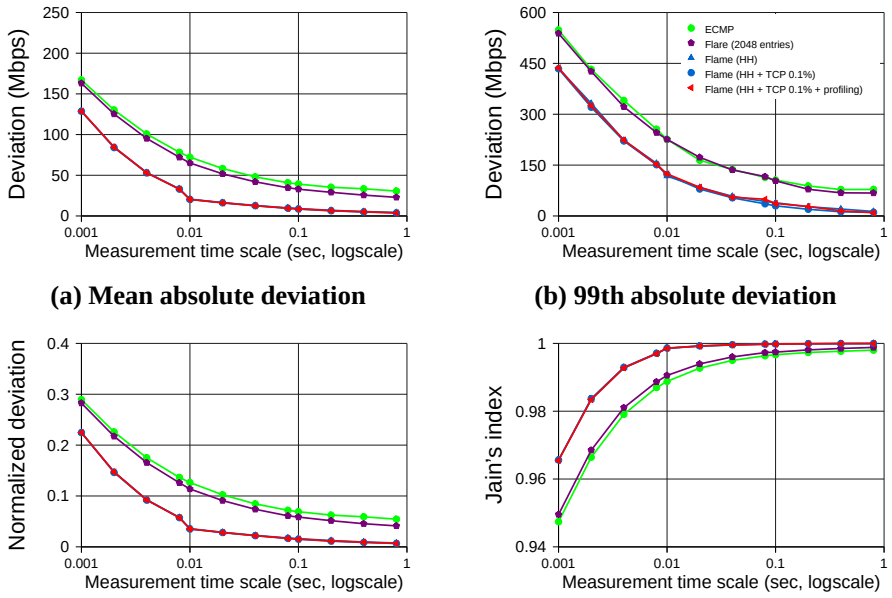
Flare [39] picks $T_s = 300$ms since that is about the amount of data that a router can buffer. However, in recent data center routers at 10G, even 10 ms worth of buffering is large and the amount of buffering per link may actually decrease further at 40 Gbps. Further, load balancing goodness metrics appear better with larger T_s. As an example, suppose with $T_s = 1$ ms and $p = 3$ paths, we have the following path traffic vectors for (P_1, P_2, P_3) in three successive measurement intervals (100, 0, 0) KBytes, (0, 100, 0) KBytes, and (0, 0, 100) KBytes. Clearly, load balancing performance is poor with absolute deviation = 67 MBytes/s. However, by enlarging T_s to 3ms, we have a single path traffic vector (100, 100, 100) KBytes, which apparently has perfect performance — absolute deviation = 0, normalized deviation = 0, and Jains fairness index = 1.

Therefore, we visualize the load balancing quality across *all* measurement time scales recognizing that only values of T_s above some implementation-dependent floor (which depends on buffering) are interesting. We do so by plotting a graph with the goodness statistic on the *y*-axis versus time scale choice on the *x*-axis. Since computing goodness for all choices of T_s is infeasible computationally, we limit T_s to powers of 2 beyond one packet transmission. In particular, in our experiments we only use $T_s = 1, 2, 4, 8, 10, 20, 40, 80$ msec.

3.6.2 Simulation setup

We use an Internet backbone trace provided by CAIDA [62]. The trace is collected at a San Jose monitor point in 2008 with bandwidth 1.8 Gbps and length one minute. We simulate load balancing schemes (ECMP, Flare, and Flame). in Perl using the CoralReef software suite [17]. To impose synthetic traffic on top of any real *pcap* trace, we augment the software to support synthetic events (such as enqueuing and dequeuing synthetic packets at synthetic timestamps) The synthetic events are managed by an efficient implementation of a heap-based discrete-event simulation engine.

The number of heavy-hitters is an input parameter. Each heavy hitter comprises of start time, end time, and traffic pattern. We simulate a heavy-hitter as a large constant-bit-rate FTP file transfer of 50-128 MB. based on the default Hadoop block size [6, 31]. The start time of each parameter is a parameter that can be controlled. For example, we can simulate simultaneous arrival of flows to or have the start time be sampled from a specified distribution.. The end time is either determined by fixing the duration of heavy-hitter (say, 10 seconds) or by randomizing either the duration (e.g. as a Gaussian distribution with mean 10 seconds) or the rate of a heavy-hitter.



(a) Mean absolute deviation **(b) 99th absolute deviation**

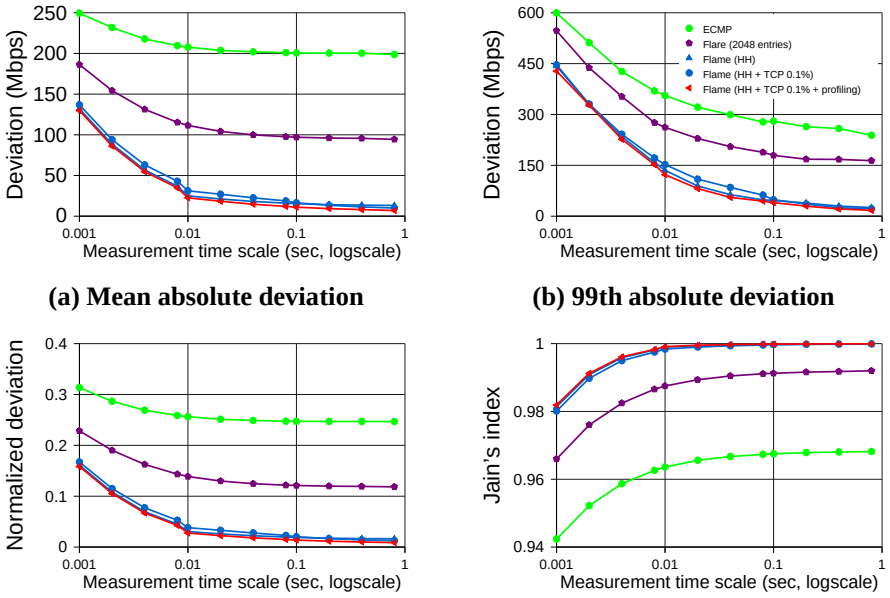
(c) Mean normalized deviation **(d) Mean Jain's fairness index**

Figure 3.6. Load balancing performance on CAIDA trace across all measurement time scales. *HH* denotes inclusion of a heavy-hitter table

Each heavy-hitter is represented by a random TCP 5-tuple. Static hash ECMP does badly when the heavy-hitters have the same source and destination IP address. Finally, our framework allows the simulation of sophisticated heavy-hitter traffic patterns to exhibit several burstiness and flowlet behaviors including ON-OFF heavy-hitters with a Pareto distribution for the OFF period. Such patterns exercise the load balancing algorithms ability to continually admit and evict flows. The flowlet behavior can be controlled; even when a heavy-hitter is OFF, it can send at least one packet every flowlet timeout so that eviction only occurs under the "once every F packets rule". We also allow the introduction of spoilers that capture a hash table bucket and send at a slow rate.

3.6.3 Simulation results

In the following set of experiments, we impose 8 synthetic flows on the CAIDA Internet trace. The full CAIDA trace lasts for 52 seconds. Each synthetic flow sends at 100 Mbps by dispatching packets of size 1250 bytes at 100 us intervals. The starting times are staggered at 1 second apart, but added random noise up to $\pm$100 ms. We let the synthetic flows run until experiment completion so that we can observe their full impact. We limit the Flame table to 2048 and overflow to ECMP if the table is full. Our Flame scheme use the heavy-hitter *abrupt admission* and *graceful eviction* policies as discussed in Section 3.3.4. Unless otherwise stated, the default parameters are as follows: number of paths $p = 3$, heavy-hitter filter



(a) Mean absolute deviation **(b) 99th absolute deviation**

(c) Mean normalized deviation **(d) Mean Jain's fairness index**

Figure 3.7. Load balancing performance with synthetic data center-like traffic across all measurement time scales. *HH* denotes inclusion of a heavy-hitter table

threshold $B_H = 3$ KBytes, heavy-hitter filter timeout $T_H = 30$ msec, flowlet aging timeout $T_a = 30$ msec.

Figure 3.6 and 3.7 compare the load balancing performance of ECMP, Flare, and Flame load balancing schemes. Note that higher Jain's fairness index means better fairness quality, which is opposite to other deviation metrics. We also tease apart the effect of each individual Flame mechanisms. For example, we illustrate how the performance changes with the addition of each Flame mechanisms, i.e. a heavy-hitter filter to track heavy-hitters, periodic rebalancing with "1 every 1000 packets rule" and profiling to prevent greedily rebalancing.

We observe that the synthetic heavy-hitters have a significant effect on ECMP and Flare but much less on Flame with a heavy-hitter filter. To be sure, Flare would also improve with a heavy-hitter filter: the real reason for Flame over Flare is the robustness and memory efficiency caused by remembering the hash and not the path. In the figures, we call the "1 in 1000" packets rule the TCP 0.1% rule. Note that Flame with the "HH + TCP 0.1%" curves can be worse than the"HH" only curves because of rebalancing oscillation which is removed by the "HH + TCP 0.1% + profiling" rule.

In Figure 3.6, there is little difference between static hash and Flare while the results of Flame algorithm are better at all time scales. The relatively poor performance of the Flare algorithm is likely due to its state table becoming saturated. Note that while our analytical results went further and suggested that Flare can do *worse* than ECMP, the experimental scenario seems more plausible. Again, the difference is

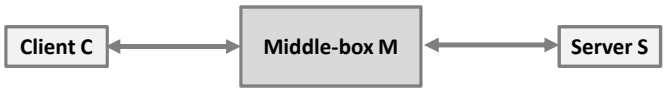


Figure 3.8. Testbed for TCP packet reordering

the heavy-hitter filter which, to be fair, would improve Flare as well. However, the experiments do point to the crucial need for robustness and graceful degradation when the memory does not suffice.

Figure 3.7 shows performance when synthetic traffic is added to the real trace. Here we see a large difference at all time scales between static hash, Flare and Flame with static hash much worse than shown in Figure 3.6. The Flare results are somewhat worse than without the synthetic traffic although noticeably better than static hash.

While Flare (without heavy-hitters) performs better than static hash because it is breaking the synthetic flows into flowlets and balancing each one, by separating heavy-hitters Flame does not suffer from table saturation the way Flare does. This allows Flame to intelligently balance a much larger fraction of traffic. These results clearly show that Flame outperforms both static hash and Flare with the amount of state being about the same as Flare. We also evaluated other synthetic heavy-hitter pattern (larger number of heavy-hitters, simultaneous heavy-hitter launch, and rate-varying heavy-hitters) which showed that Flame is resilient to all such combinations. We do not include these graphs due to space constraints.

3.6.4 Impact of packet reordering on TCP

Recall that Figure 3.4 has a parameter F that controls the frequency of reordering. While earlier studies have suggested $F = 1000$ we felt it was essential to update these studies to see the effects of operating system changes, higher link speeds, and the subtle difference in reordering patterns caused by load balancing compared to arbitrary reordering. To evaluate the impact of packet reordering on TCP performance at 1 Gbps and 10 Gbps, we set up two hardware testbeds, each consisting of three nodes connected serially as shown in Figure 3.8. The middle-box M controls the forwarding of all traffic between the client C and the server S.

Results from the 1 Gbps testbed:

In the 1 Gbps experiments, the middle-box M was a 2-processor Intel Xeon 2.4 GHz machine running Ubuntu 10.10 32-bit server with Linux kernel 2.6.35. The kernel was recompiled after applying the Trace Control for Netem patch [50, 72] which enables the flexible addition of latency to packets needed for our experiments. The client and server used the same model of machines as the middle-box.

For Linux client and server experiments, they were running Ubuntu 8.10 32-bit server with Linux kernel 2.6.27. For Windows experiments, they were running Windows Server Standard 2008, 32-bit, SP2. All machines had two Intel PRO/1000 MT Desktop Ethernet interfaces, but only one was enabled on the client and server machines.

In the 1 Gbps experiments, the middle-box was configured to reorder packets by selecting a number of packets F. The first $F/2$ packets sent out of a network interface had 0.9 msec of extra latency added to the normal time required to forward the packet. The next $F/2$ packets sent had 1.1 msec of extra latency added. This pattern repeats every F packets, so when the latency changes from 1.1 msec to 0.9 msec, one or more packets can be transmitted out of order. This emulates the situation where a flow's packets are switched from a low to high latency path, then switched from a high to low latency path, every F packets.

The exact pattern in which the packets are reordered by this method varies with the timing that packets arrive at, and are processed by, the middle-box's kernel. From examination of the netem code, the packets are time stamped when they begin their processing in the kernel. These time stamps are maintained with a resolution of 64 nanoseconds. The latency of 0.9 msec or 1.1 msec is added to the packet's arrival time to get its scheduled departure time. The packet is then placed in an output queue for the target network interface, inserted at the appropriate place so that the queue is maintained in order of scheduled departure time.

The typical pattern of reordering seen during experiments that achieve high throughput is the same as if N consecutive in-order packets arrive at the middle-box and are buffered, then the next M packets are allowed through, passing the N buffered packets on their way to the receiver. Then the N buffered packets are forwarded. For example, with $N = 9$ and $M = 7$, if the sender sent packets D_1 through D_{19} in that order, the packets would arrive in the order $D_1, D_{11}, D_{12}, \ldots D_{17}, D_2, D_3, \ldots D_{10}, D_{18}, D_{19}, \ldots$ at the receiver, and would then be in order until the next time the latency went from high to low. Examples of pairs of value (N,M) observed in actual packet traces recorded at the receiver are $(10,12)$, $(4,18)$, $(6,20)$, and $(17,3)$.

This pattern of adding latency was done for packets in the forward (i.e. client-to-server) and backward directions, with an independent state machine counting packets in each direction.

Figure 3.9 shows the throughput achieved by a single TCP connection for a Linux client and server, and for a Windows Server 2008 client and server. The throughput measurement was made using *iperf* with data transmission only in the client-to-server direction, and is the average rate over 10 seconds.

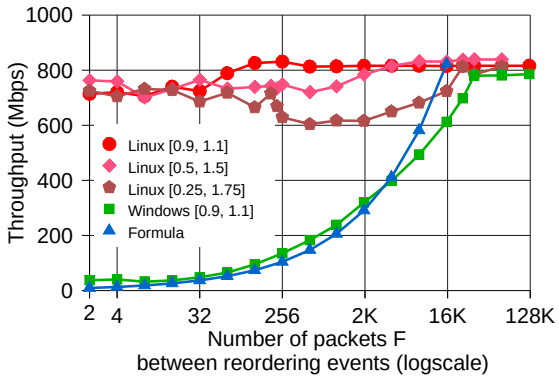


Figure 3.9. TCP throughput experiments at 1 Gbps. Latency differentiation in msec.

Each data point plotted is the median throughput of 11 runs with the same conditions. We also repeat the Linux experiments with two other pairs of latency values, $[0.5, 1.5]$ msec (i.e. latency drops by 1.0 msec when it decreases) and $[0.25, 1.75]$ msec (i.e. latency drops by 1.5 msec). With wider latency difference range, we expect worse throughput because there is more reordering that can be introduced when the drop from high latency to low latency is by a larger quantity of latency. Figure 3.9 shows that the Linux throughput is still high in all cases, especially when F is no less than about 128.

The "Formula" curve is the TCP throughput predicted by the $1/\sqrt{p}$ model of TCP performance [45]. It is the value of $(MSS \cdot C)/(RTT\sqrt{p})$ for $p = 1/F$, $MSS = 1460$ bytes, $RTT = 2.2$ msec, and $C = 1.22$.

Linux achieves remarkably good throughput even with very frequent reordering. Wu et al [77] ran similar experiments with a middle-box that added normally-distributed random latencies to each forwarded packet. They found similarly good throughput, as long as the standard deviation of the normal distribution was small compared to the mean. They attribute this resilience against reordering to: "an adaptive TCP reordering threshold mechanism. Under Linux, *dupthresh* is adaptively adjusted in the sender to reduce unnecessary retransmissions and spurious congestion window reductions. Some network stacks [...] still implement a static TCP reordering threshold mechanism with a default *dupthresh* value of 3."

Our experiments confirm this. We ran additional experiments where the middle-box dropped a single packet every F packets, and added 1 msec of latency to all packets (thus no reordering). Under these conditions the throughput graphs for both Windows and Linux were nearly identical to the Windows throughput graph of Figure 3.9, where reordering but no loss is introduced.

We also recorded packet traces on the sender in a few experiments (not used to create the graphs)

and used *tcptrace* [73] to estimate the sender's congestion window. This estimate is called "outstanding data" in tcptrace. It is calculated as the largest data sequence number transmitted by the sender, minus the largest cumulative ACK sequence number it has received so far. These graphs showed the Linux sender's congestion window increasing steadily despite packet reordering events, whereas the Windows sender's congestion window dropped every time it received 3 or more duplicate ACKs in a row. Thus the Windows sender is misinterpreting the kinds of reordering we introduce as packet loss, as was also the case with older versions of Linux (circa kernel version 2.6.14 and earlier).

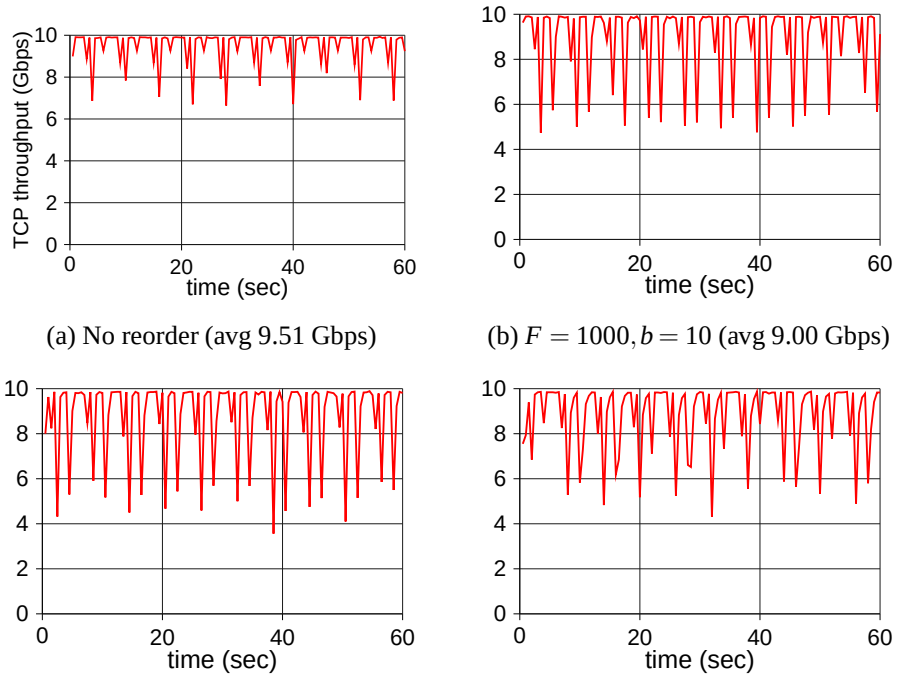
We also ran Linux client to Windows server experiments, and vice versa. Although not identical, they are substantially similar to the Windows throughput graph presented here. It is not enough that Linux is the sender in order to achieve high throughput during reordering. The receiver TCP implementation is also important.

The results here make a strong case that for Linux-to-Linux TCP traffic, per-packet load balancing such as DRR may be acceptable to increase overall application throughput, if the gain in throughput from the more even load balancing is greater than the small throughput reduction caused by packet reordering. For the common case where multiple TCP connections share the network capacity, their competition for bandwidth is likely to be the limiting factor before packet reordering effects are noticeable. For TCP traffic that is not Linux-to-Linux, reordering that causes the sender to react as if there were a packet loss (i.e. 3 or more duplicate ACKs in a row) as often as once every 1024 packets cuts throughput by a factor of 4, according to the results in Figure 3.9.

Results from the 10 Gbps testbed:

NetBump [3] allows modification to packets between C and S at 10 Gbps by software (e.g. changing packet headers, adding delay, dropping packets, and crafting packet reordering). We set up a NetBump testbed with fast machines equipped with Myricom 10 GigE so that packet processing can be done in real time at full 10 Gbps by NetBump techniques to offload work on multiple CPU cores [3]. Since our NetBump testbed is only Linux-based , we show only the result for Linux TCP in this part.

We consider a common pattern of packet reordering caused by load balancing in data center networks. As an example, assume packets D_1 through D_5 go on path P_1 and then packets D_6 onward switch to path P_2 with *lower* latency. Thus although D_1 through D_5 have left the switch, packet D_6 may overtake some of them. A possible scenario at the receiver (in terms of received packets, assume no loss) is $\mathrm{D}_1, \mathrm{D}_2, \mathbf{D_6}, \mathrm{D}_3, \mathbf{D_7}, \mathrm{D}_4, \mathbf{D_8}, \mathrm{D}_5, \mathbf{D_9}, \mathbf{D_{10}}, \mathbf{D_{11}}, \ldots$ back to normal. In other words, it is not a complete burst of packets that are delayed but the delayed burst *interleaves* with the burst switched onto the lower



(a) No reorder (avg 9.51 Gbps)

(b) $F = 1000, b = 10$ (avg 9.00 Gbps)

(c) $F = 1000, b = 50$ (avg 8.82 Gbps)

(d) $F = 1000, b = 100$ (avg 8.81 Gbps)

Figure 3.10. Throughput experiments with one TCP flow at 10 Gbps with interleaving reordering burst by load balancing. F is reordering frequency and b is length of interleaving burst.

latency path.

We emulate this reordering situation according to the following two parameters.

- Reordering frequency F: we do packet reordering once every F dispatched packets.
- Interleaving burst b: the amount of interleaved packets ($b = 3$ in our example)

Our method to craft the interleaving burst is by buffering upto b packets and alternating them with the subsequent packets accordingly. To avoid infinite packet delay (especially for SYN packets), we hold each packet in the interleaving buffer for at most 1 ms.

Figure 3.10 plots our TCP throughput benchmark with the *iperf* tool for one minute. Note that we use the default TCP implementation of the Linux operating system installed on the servers (64-bit Debian on Linux kernel 2.6.32). The bandwidth measurement granularity is 0.5 sec. In our testbed, the round-trip-time (RTT) between the client and server varies in the range $0.2 - 0.5$ ms. From Figure 3.10, we conclude that the TCP stack on our Linux kernel is highly tolerant to packet reordering. Indeed, we also get consistent results with Figure 3.10 for several other patterns such as interleaving of short bursts

(instead of packets) and per-packet delay differentiation (not shown due to space limit).

3.7 Summary

Our testbed results surprised us a great deal. They suggest that rather than deploy new transport protocols such as Multipath TCP [60], the simple TCP modifications to recent Linux stacks may allow packet-by-packet rebalancing with only small loss in performance. The essential idea is to not to blindly reduce the congestion window on getting a duplicate ack, or at least to increase rapidly again if the situation is reordering and not loss.

In the interim, at least for Windows machines that are very common in data centers, the situation is neatly reversed. We cannot afford to rebalance often without incurring severe throughput degradation: a value of once every $32,000$ packets seems to work well. Given the uncertainty, load balancing chips today would be wise to have a controllable parameter F. We assert that hardware load balancing such as shown in Figure 3.4 will be crucial. Software load balancing such as [5] (where the optimal flow assignments for heavy flows is computed by software) will be too slow to allow values of F below 1000 and hence miss balancing opportunities in the future. The Flame hardware described in Figure 3.4 can also be deployed one router at a time compared to the deployment issues for Hedera [5]. While Flame does not provide path optimality, DRE estimation goes beyond local optimality based on local physical queues and attempts to reduce downstream congestion.

Unlike Hedera, Flame also attempts to initially do a good flow assignment by stealing the basic ”place new flow on least loaded link” paradigm from Flare. However, Flame goes beyond Flare by having a more robust link bandwidth estimator (DRE), a more resilient and memory-efficient method to remember flow state by memorizing one of multiple hash functions, and by integrating periodic rebalancing in hardware. In conclusion, while Flame is deeply influenced by Hedera and Flare, we believe it adds significant new mechanisms (summarized in Figure 3.4) that will be essential for *deployable* and *robust* data center routers at 10 Gbps and beyond. While our paper appears to be narrowly about ”load balancing”, the broader issue at stake is cheaply providing bandwidth for cluster computation in data centers, which in turn underlies the promise and effectiveness of cloud computing.

Chapter 3, in part, is a reprint of the material as it appears in “Flame: Efficient and Robust Hardware Load Balancing for Data Center Routers” in *UCSD CSE Technical Report (CS2012-0980)*. Edsall, Tom; Fingerhut, Andy; Lam, Vinh The; Pan, Rong; Varghese, George. UCSD, 2012 The dissertation author was the primary investigator and author of this paper.

Chapter 4

NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers

4.1 Introduction

Cloud services and enterprise networks are hosted by data centers that concurrently support many distinct services — e.g., search and email for cloud services, and say accounting and engineering for an enterprise data center. The services use a shared data center because the physical equipment is expensive, costing over 100 M a year to maintain [31] and because statistical multiplexing using Virtual Machines (VMs) is effective. However, the economics also require two other characteristics of the *network*, both of which are imperfectly provided today. First, to be profitable, the networks must have high *utilization*. Second, many services have stringent performance SLAs that must be met to keep customers satisfied: thus the network should also ideally provide *bandwidth guarantees* to each service. Any new mechanism to provide these should not require hardware changes to existing switches so that providers do not have to retrofit their networks.

Service level agreements today specify network SLAs in terms of dollars per Gigabyte transferred and not in terms of network bandwidth. But the performance of frameworks such as Map Reduce depends greatly on network performance. With current SLAs, a user may pay for 10 hours for a number of VMs only to find that the VMs are mostly idle waiting for slow network transfers. The user job may complete in one hour with a faster network rate and the user may be willing to pay more for the higher bandwidth. In addition, as cloud computing and shared data centers gain momentum, there is a growing demand to provide performance isolation between different services and tenants. While isolation can be achieved by strict rate limits, this leads to inefficient use of the expensive data center network because traffic is often bursty.

We propose a new mechanism for data center networks called *NetShare* that provides predictable bandwidth allocation, bandwidth isolation, and high utilization — and can be implemented without any changes to existing switches. NetShare does so using *hierarchical weighted max-min fair sharing* in which the bisection bandwidth of the network is *i)* first allocated to services according to weights and *ii)* the bandwidth of each service is then allocated equally among its TCP connections. Hierarchical max-min fair sharing generalizes hierarchical fair sharing of *links* [27] to *networks*. We also generalize stochastic fair queuing [46] to stochastic weighted max-min fair queuing. While the ideas in NetShare are extremely simple and can be viewed as a repackaging of existing ideas, the fact remains that no such mechanism exists in the market today.

We view "fairness" only as a mechanism for providing predictable and tunable application performance. Section 4.6.1 for instance, shows that without NetShare, an FTP transfer of a 1 GB file slows down by a factor of 10 if it happens to be concurrent with the sort phase of a Hadoop application. While better latencies for the FTP transfer could be provided by rate-limiting the Hadoop application, in that case the throughput of the Hadoop application halves. In contrast, NetShare allows both low latency for FTP and high throughput for Hadoop.

If Internet QoS did not succeed, why hope for data center QoS? First, Internet QoS issues are often solved by overprovisioning, but overprovisioning core links in data centers from say 10 Gbps to 40 Gbps is very expensive. Current core links are indeed oversubscribed [31]. Second, users have begun to notice latency degradation when VM traffic from different applications[1] interfere [32]. Third, a reason for the failure of QoS was that there was no simple policy for setting QoS parameters. NetShare uses a simple set of per-service weights which can be set automatically based on VM placement, or set manually by a manager based on the revenue or cost of each service analogous to VMware's ESX server shares [32].

We present three simple mechanisms to implement NetShare including one that relies on TCP and fair queuing, only requires configuration changes, and responds to changes in a few round trip delays. We also show how this mechanism can scale to a larger number of applications using what we call *Stochastic NetShare*. Our second mechanism handles UDP, and our third mechanism uses centralized allocation to provide more general bandwidth allocations.

NetShare can also be viewed as a way to virtualize (i.e., statistically multiplex) a data center network among multiple services. Together with virtualized CPUs and disks, it allows managers to create "virtual data centers" with performance isolation. While one can argue whether our definition of network

[1]We use the terms *application* and *service* interchangeably.

virtualization is right, NetShare is perhaps the simplest starting point. Our contributions are:

- A specification of what it means to share data center bandwidth across services using hierarchical weighted max-min fair sharing (Section 4.2).
- Three mechanisms to implement NetShare (Section 4.3) with tradeoffs (Table 4.1). They require *no* hardware changes to existing routers and work with multipathing — earlier multipath Max-Min allocations used complex and hard to implement mechanisms [41].
- An implementation using Fulcrum switches and using ns-2 (Section 4.5).
- Analysis (Section 4.4) and implementations (Section 4.6) that show the benefits and scalability of NetShare.
- A scheme for automatic weight assignment (Section 4.7).

4.2 NetShare Specification

The generalization of fair sharing to multiple resources such as a network is called Pareto Optimality (in economics) or max-min fair sharing (in networks). While max-min fair sharing at the TCP level is an old goal, we argue this is insufficient. A large corporation may wish to split bandwidth between a parallel CAD application, SAP, and Microsoft Exchange. Max-min fair sharing at the TCP level is not the appropriate model for two reasons. First, services that open up multiple connections get an unfair share of bandwidth. Second, the manager cannot allocate more bandwidth to certain services based on their importance.

Thus we are led to: *weighted hierarchical max-min fair sharing*. First, the manager specifies services with weights manually or automatically assigned (see Section 4.7 for a simple technique to assign weights automatically). Next, there is a mechanism that allocates network bandwidth in weighted max-min fair fashion among these services. The bandwidth assigned to each service is then recursively divided (again in max-min fair fashion) among the individual flows for that service. In addition, each application can be limited to some maximum bandwidth if needed using token bucket limiters though we do not consider this further in the paper. We reiterate that fairness is not a primary goal; our foremost concern is controlled allocation of network resources to maintain latency guarantees and high utilization.

Example: Figure 4.1 shows a simple data center topology consisting of four edge switches $E1$, $E2$, $E3$, and $E4$ and 1 core switch $C1$. Each edge switch is connected by a 10 Gbps link to the core.

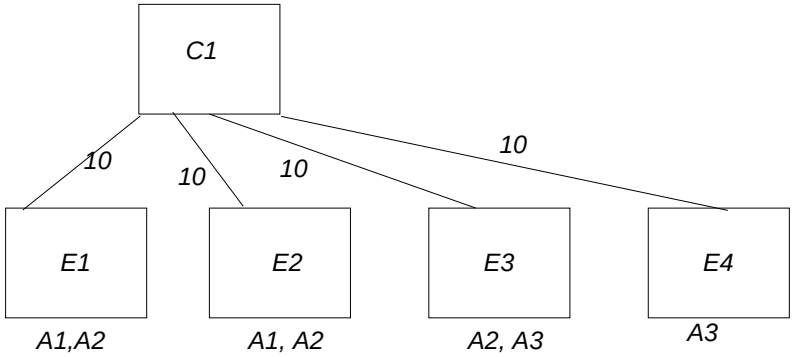


Figure 4.1. Example of a data center network shared between three services A1, A2, and A3.

Assume that there are three services $A1$, $A2$, and $A3$. Further, assume $A1$'s traffic needs to be sent from switch $E1$ to $E2$. Service $A2$ needs to send traffic from $E1$ to $E2$, and from $E1$ to $E3$. Service $A3$ needs to send traffic from $E4$ to $E3$. For this simple example, assume the manager has *manually* set global weights with only Service $A1$ having weight 4, while the $A2$, and $A3$ have weight 1.

The link from $E1$ to $C1$ is shared by applications $A1$ and $A2$. Assuming weights of 4:1, $A1$ should be assigned 8 Gbps while $A2$ should be assigned 2 Gbps. However, $A2$ has traffic from $E1$ to $E2$ and from $E1$ to $E3$. Assuming equal sharing of each edge-to-edge traffic flow *within* a given service, service $A2$ is allocated a bandwidth of 1 Gbps for traffic from $E1$ to $E2$, and 1 Gbps for traffic from $E1$ to $E3$.

But this allows service $A3$ to be assigned 9 Gbps for its traffic from switch $E4$ to $E3$ even though it has only the same weight as service $A2$ with which it shares a link. This happens because $A2$ is bottlenecked because of another link (the link from $E1$ to $C1$). This kind of calculation where a bottleneck limits the bandwidth of a flow, which then affects the bandwidth available to another flow, and so on iteratively, is formalized in the so-called Weighted max-min fair share calculation.

Now assume that service $A1$ reduces its bandwidth need to 6 Gbps. After some amount of time (measured by the responsiveness of the algorithm) NetShare can allocate 2 Gbps to $A2$'s traffic on the link from $C1$ to $E3$. This in turn reduces $A3$'s share to 8 Gbps. We can formalize this allocation as follows.

Definitions: A feasible bandwidth allocation of a set of flows is *max-min fair* if and only if a rate increase of one flow must come at the cost of a rate decrease of another flow with a smaller or equal rate. A feasible bandwidth allocation of a set of flows is *weighted max-min fair* if and only if a weighted rate increase of one flow must be at the cost of a weighted rate decrease of another flow with a smaller or equal weighted rate.

We extend to hierarchical weighted max-min sharing: A feasible bandwidth allocation to a set of

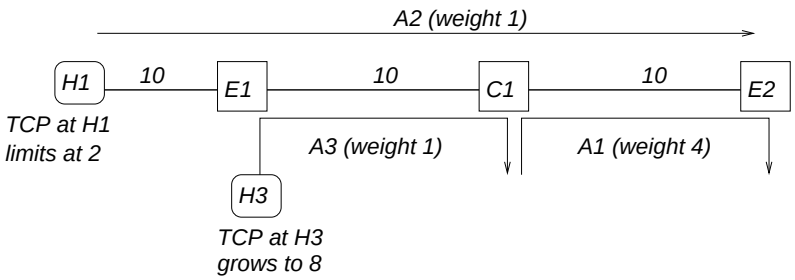


Figure 4.2. Simple fair queuing at switches together with TCP implements max-min fair sharing of TCP flows [35].

applications is *hierarchical max-min fair* if and only if a rate increase of a flow within one application must be at the cost of a rate decrease of some other flow either (i) within the same application with a smaller or equal flow rate or (ii) within some other application with a smaller or equal weighted application bandwidth.

For this paper, we assume that in hierarchical sharing, weights can be specified at the service level, but all TCP connections within the same service have equal weights.

4.3 NetShare Algorithms

In this section, we describe how NetShare can be implemented. Section 4.3.1 describes *group allocation* which relies on TCP. Section 4.3.2 describes *Stochastic NetShare* approach to address the scalability of weighted fair queuing. Section 4.3.3 describes *rate throttling* for UDP hosts. Finally, Section 4.3.4 describes a centralized bandwidth allocator that can implement more general allocation policies.

4.3.1 Group Allocation Leveraging TCP

Our starting point is a classic result by Hahne [35] which is paraphrased as follows in [12].

Proposition 1: [35]: A large sliding window at sender plus fair queuing achieves max-min allocation.

The intuition is illustrated in Figure 4.2. Assume 3 competing TCP flows: a first from service $A1$ that traverses bottlenecked link from $C1$ to $E2$; a second from service $A2$ starts at host $H1$ and goes from $E1$ to $C1$ and from $C1$ to $E2$; finally, a third flow from service $A3$ that traverses the link from $E1$ to $C1$. Assume that $A1$'s flow is configured to have a fair queuing weight of 4 at core switch $C1$ while other flows are assigned weight 1.

Thus fair queuing at $C1$ will assign $1/5$-th of the bandwidth of the $C1, E2$ link to the $A2$ flow

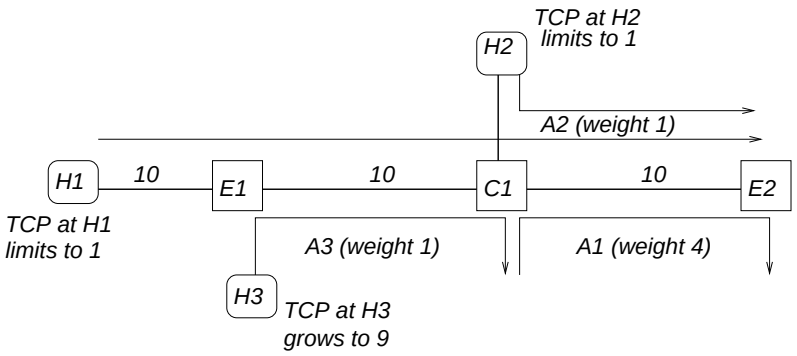


Figure 4.3. Simple fair queuing at switches at the *service* level together with TCP achieves *hierarchical* max-min fair sharing of services.

because the $A1$ flow has 4 times the weight. In a few round trip delays, TCP at $H1$ will adjust its rate to 2 Gbps. But this allows TCP at H3 to grow to 8 Gbps because only 2 Gbps is used on the link from $E1$ to $C1$. Hahne's result formalizes this intuition but has a number of caveats. For example, the proof [35] applies to only some arrival distributions such as Bernoulli arrivals and to single path topologies.

However, in NetShare we wish to allocate in hierarchical max-min fashion first at the service level and only then at the TCP connection level. So consider Figure 4.3 which adds one more host $H2$ that also belongs to service $A2$ with weight 1 and shares the link from $C1$ to $E2$ with $A2$'s other TCP flow from $H1$ and $A1$'s flow. Fair queuing at the *TCP connection level does not* achieve hierarchical max-min fair sharing. The TCP connection from $A1$ is allocated $4/6$th of the bandwidth and thus gets only 6.6 Gbps instead of 8 Gbps.

However, if we do fair queuing at the *service* level, then both connections belonging to service $A2$ are treated identically at core router $C1$ (i.e., mapped to the same queue). Assuming the fair mechanism gives both the TCP connections from $H1$ and $H2$ equal bandwidth, both limit themselves to 1 Gbps, which then allows TCP at $H3$ to grow to 9 Gbps. Thus we state the following proposition:

Proposition 2: Window flow control plus fair queuing at service level achieves hierarchical max-min allocation.

An informal argument is as follows. It follows the standard water-filling algorithm described in [9] when modified to do hierarchical allocation. It starts by finding the *weighted bottleneck*. NetShare will emulate this by DRR at the bottleneck link to give each application its weighted share. Next, we assume that the TCP flows of each application share the bottleneck link equally. While this is not strictly true if the TCPs have very different RTTs, we assume this is true in the data center. We assume each of

these TCP flows cannot increase any further. Just as in the standard water-filling algorithm, we remove these TCP flows and their bandwidths, and recurse to find the new bottleneck.

ECMP multipathing can also be handled as follows. We assume each path used by each application flow is independent in the following sense. For example, we assume there are no cases where one application uses two flows $F1$ and $F2$ on disjoint paths and a second application uses a third flow $F3$ which overlaps partly with the path of $F1$ and with the path of $F2$. Such dependencies will cause the dynamics of the two flows to be coupled. However, the use of multipathing in common data center topologies typically results in flows being independent due to symmetric structure. If the flow dynamics are independent, then the above argument applies independently to each flow sent between the same pair of hosts on multiple paths. By contrast, max-min allocation with path splitting in general topologies requires complex optimization algorithms [41, 19].

Our argument above makes a number of simplifications. Despite this, we have found in our experiments with real switches and ns-2 experiments on data center topologies that Proposition 2 holds even with multipath topologies. Proposition 2 suggests an extremely simple mechanism that requires no software or hardware changes to endnodes or switches.

Group Allocation Mechanism: For every switch and every outbound link configure separate fair queuing queues for each service/application class with weights specified by the manager.

For example, in Fulcrum switches [1] we have used DRR [66] to configure fair queuing and TOS bits to distinguish services. The queue weights are then set (on all outbound links) to the NetShare weights specified by the manager. Note that this is not the same as reservation. If a service is inactive or is routed on a different path it will not consume bandwidth on this link.

Current switches typically support a small number of DRR queues such as 16. While this often suffices for the enterprise, we wish to handle a large number of services, especially in cloud data centers. First, note that Approximate Fair Dropping (AFD)[54] is a replacement for DRR. Cisco routers will appear within the next year with a few thousand AFD queues for 16 DRR queues. Note that AFD scales better because it uses a counter for each class as opposed to a queue. We describe a temporary measure, however, to deal with existing routers with small queues.

4.3.2 Stochastic NetShare

Large enterprises or web service providers like Amazon EC2 or Google App Engine could have several customers or classes that they wish to isolate. One technique to scale NetShare to several appli-

cation classes is what we call *Stochastic Weighted Max Min Fair Sharing*. Applications are randomly hashed to specific DRR queues at each switch port. Each DRR group is assigned weight equal to the sum of weights of the individual applications that are hashed on to the DRR queue. The DRR grouping of applications in each switch can be different and can also be different at each switch port. Also, the grouping of applications on DRR queues is changed periodically to avoid any intermittent hashing imbalances.

Stochastic Max-Min Fair sharing is a generalization of Stochastic Fair Queuing [46]. We, however, have to deal with three more issues. First, current routers do not support hashing to queues based on packet headers. However, they do support mapping from header fields to queues via ACLs. Thus, a way to simulate hashing is to have a central allocator provide labels to services; these labels can be random labels and can be changed periodically. Second, note that McKenney described his scheme at a single router. When we do this across the data center, deviations at a single hop can cascade across the network (think of errors at each stage of the water-filling algorithm). Third, McKenney's scheme assumed all weights were equal.

Clearly, pure random assignment regardless of weights will not work well because a high weight service may be assigned to the same queue as a low weight service. Since allocation within a queue is done by TCP which allocates bandwidth regardless of weights, a small weight service can steal more bandwidth than its share. We propose a mixture of random and weight-based allocation as follows.

First, group services based on weight classes (say all services of weight 1, all of weight 2, all of weight 4, etc.). Then map each weight class into a set of queues and randomly assign services to a specific queue within each set. In practice, given the small number of existing queues, we suggest grouping large weight services and low weight services into 2 classes, and randomly assigning within each set of 8 queues. Clearly, this introduces errors due to weight aggregation and these errors can cascade but it provides a solution to the difficult problem of combining scalability together with tunability. In a theoretical sense, if there are S services, M misbehaving applications, and W possible weights, our mechanism requires $O(M \log_2 W)$ queues instead of $O(S)$ queues.

The mixture of deterministic (weight-classes) and random assignment to queues can also be achieved on existing routers by a centralized label allocator. For example, using 16 values of the IP TOS Bits (as supported by many routers including Fulcrum and Cisco): the allocator could assign values 0 through 7 to low priority applications, and values 8 through 15 to high priority applications. In all routers and switches, value I is mapped to queue I using say ACLs.

Next, the allocator assigns each high priority service a random value between 8 and 15, and each

low priority service a random value between 0 and 7. The label allocator periodically changes these at intervals of a few seconds to avoid cases where one service persistently gets assigned to the same queue as a misbehaving service. Note that the sources are using random labels directly instead of router rehashing.

4.3.3 Rate Throttling for UDP

Group allocation relies on TCP. However, many important applications including Tibco's multicast protocols, Veritas, and Oracle [8] do not use TCP. Buggy or erroneous settings can cause such traffic to flood the network. While we could use existing TCP-friendly UDP congestion control protocols, this would require modifying all UDP applications (not easy to find) and change them to send packets via the TCP friendly code, which requires finding the calls in the code. Instead, we show a very simple scheme that can be implemented as a kernel patch (some effort but easier than modifying applications) that is layered *below* UDP and is thus transparent to existing applications.

Further, most TCP-friendly congestion protocols measure drops and use the TCP equation [52] unlike our simple scheme. In particular, the simplicity of our scheme also allows it to be implemented in switches which can be useful in cloud environments and multi-tenanted data centers where host software cannot be trusted.

First, what goes wrong in Figure 4.3 if $H1$ and $H2$ use UDP? In that case, $H1$ could continue to send at 10 Gbps on the link to $E1$ and the fair queuing mechanism at $E1$ will assign it 5 Gbps on the link to $C1$ (dropping the remaining traffic) providing only 5 Gbps to $A3$'s traffic from $H3$. This is unfortunate because the fair queuing mechanism at $C1$ will only allocate 1 Gbps to the traffic from $H1$. Thus if $H1$ sends at 10 Gbps, 5 Gbps of traffic is dropped at $E1$ and 4 Gbps is dropped at $C1$. There is no congestion collapse but the allocation is far from optimal. Instead, using our UDP rate throttler, the host $H1$ will send at 1 Gbps and the extra traffic will be buffered at the host.

We propose a simple idea as follows. Assume that each host has a *rate throttling shim layer* just below UDP. For example, in Figure 4.3 suppose that $H1$ sends at 10 Gbps to some other host $H4$ as shown in Figure 4.4. The shim layer at $H4$ measures received traffic of 1 Gbps from $H1$. This is sent back to the corresponding rate throttling layer at $H1$ which rate limits the traffic at close to 1 Gbps.

Unfortunately, $H1$ cannot rate limit exactly to 1 Gbps. This is because if say the flow from $H2$ disappears, $H1$ could grow to 2 Gbps. But the rate limit at $H1$ will prevent $H1$ from ever finding that it can grow. Thus we need to set the throttled rate to somewhat more (say x%) than the measured rate to allow ramp up. Higher values of x allow faster ramp-up but increase the amount by which a flow can

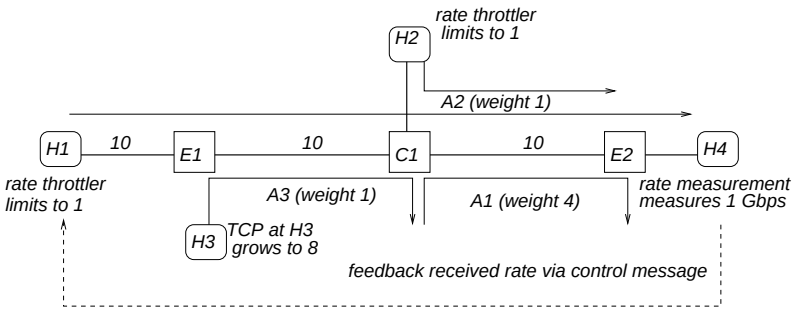


Figure 4.4. Simple fair queuing at switches at the *service* level together with rate measurement and rate throttling implements hierarchical max-min fair even with UDP.

overshoot its allocation. We chose a value of $x = 20\%$ as a compromise.

The throttling code we use is slightly more complicated and is described in Algorithm 2. Note that Rate Throttling requires weighted fair queuing at each router as well. We assume the receiver measures received throughput in some period T (we use 50 msec in our experiments) and sends a control (e.g., ICMP) message to the sender with the current measured rate C every T msec. The sender then executes Algorithm 2 to set the throttled rate R.

First, the code adds some hysteresis to prevent changes when the difference between the last measured rate (stored in L) and the current rate is too small (say, less than 5%). Next, if the current measured rate is greater than the last measured rate, the new sending rate is set to an additional factor r_I of the measured rate C. If the current measured rate is smaller than the last measured rate, the new sending rate is set to an additional factor r_D of the measured rate C. The values r_I and r_D are performance tuning knobs, indicating how much network bandwidth can be wasted.

For example, if we use $r_D = 10\%$, we could have upto 10% of the bandwidth wasted since we set the rate to 10% more than current measured rate. So this motivates us to reduce this value. On the other hand, if the values r_I or r_D are larger, then any newly freed up bandwidth can be acquired fast by the UDP traffic class. So this is an essential tradeoff. We note that very large differences between r_I and r_D would also cause instability and would make it harder for the rate to stabilize. In our experiments, we achieved good performance with $r_I = 20\%$ and $r_D = 10\%$.

There are two final subtleties. First, even if the difference is too small, if the sender had increased on the last iteration (this is kept track of by flag f), the sending rate is set to a factor r_O (say, 10%) higher than the measured rate C. This limits the final overshoot to 10%. For example, suppose the target max-min rate is 100 and the last measured rate is 94 and the current measured rate is 100. The sender goes

Algorithm 2. Compute NetShare Rates at Rate Throttler

```
Performance tuning knobs:
  d: threshold of rate difference
  r_I: factor for increasing flows
  r_D: factor for decreasing flows
  r_O: factor for overshooting flows
Measurement parameters:
  L: last measured rate
  C: current measured rate at receiver
  f: flag indicating that the flow increased on last iteration.
  R: current rate limit
if (|(L−C)/L| ≥ d) then {is rate change substantial?}
  if C−L > 0 then {increasing, allow overshoot by r_I}
    R ← C∗(1+r_I)
    f ← true
  end if
  if C−L < 0 then {decreasing, allow overshoot by r_D}
    R ← C∗(1+r_D)
  end if
else
  if f = true then {limit overshoot by r_O}
    R ← C∗(1+r_O)
    f ← false
  end if
end if
if R < Th then {do not lower below threshold}
  R ← Th
end if
L ← C
```

20% higher in the next iteration to roughly 120. However, if the next measured rate is also 100, the next iteration will set the sending rate to 110. Thus after a brief overshoot of at most 20% the final overshoot will be 10%. Finally, we do not let the rate to fall below a threshold, because if a flow's rate becomes too small it will take too long to ramp up.

Implementation Choices: Rate throttling can be implemented entirely in the hosts through a kernel patch which is gradually deployed. It can also be implemented in the network at switches with open APIs or OpenFlow [47] switches. For our experiments, we used the Fulcrum switch API to implement rate throttling with just under 100 lines of code.

4.3.4 Centralized Bandwidth Allocator

While NetShare based on fair queuing is efficient, it can only calculate a hierarchical max-min allocation. A more general policy would allow some connections between important servers to be allocated higher bandwidth. As a more complex example of a useful allocation policy, consider the single

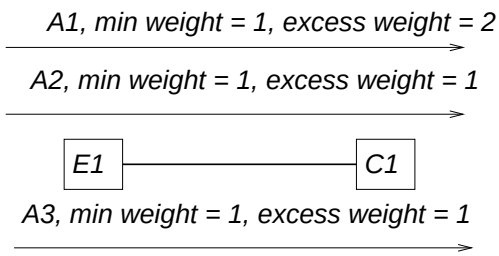


Figure 4.5. Allocation mechanisms that divide excess bandwidth using different weights.

link example shown in Figure 4.5 with 3 services $A1, A2, A3$ sharing a congested link from an edge to a core switch.

With 10 Gbps link bandwidth and equal weights, each service is guaranteed 3.33 Gbps. However, if $A3$ idles, then $A1$ and $A2$ get 5 Gbps each. Suppose, however, we wish to limit the "excess" bandwidth that $A2$ gets. Then we can define a second set of weights for sharing the excess bandwidth. For example, if the excess bandwidth weight of $A1$ is twice that of $A2$, then if $A3$ idles, the excess is allocated among $A1$ and $A2$ in the ratio 2:1, so that $A1$ gets 2.22 Gbps of the excess and $A2$ gets 1.1 Gbps. Thus, in sum $A1$ gets 5.55 Gbps and $A2$ gets 4.4 Gbps. By setting the excess weight to zero we can prevent a service from getting any excess bandwidth. While such an allocation may seem contrived, the ability to do such general allocations for a single storage resource (we do it for the entire network) is a key motivation for MCLOCK [32] implemented in VMWare's ESX server.

However, such an allocation is impossible using fair queuing at switches. Instead, inspired by centralized routing schemes like RCP [16] or [18] we propose the use of a centralized bandwidth allocator based on four simple mechanisms.

1. Rate Measurement: The rate of each flow (TCP or UDP) for each service is measured at either the switches (using ACLs) or at the hosts (using a shim layer) in intervals of T seconds and used to predict a demand for the next interval.

2. Rate Reporting: The predicted rates are sent to a centralized bandwidth allocator (implemented on a PC in the network) that is also supplied with the service weights and the topology via routing updates.

3. Centralized Calculation: The centralized allocator calculates rates for each flow and each service and sends back rate updates to the switches or hosts.

4. Rate Enforcement: Token bucket rate limiters are used at the hosts or ingress switch ports to limit the rates to the calculated rates. As in rate-throttling, each flow (especially TCP flows) must be

Table 4.1. Comparison of different NetShare mechanisms

	Deployment	Responsiveness	Generality
Group Allocation	Configuration at routers	$<$ 1 msec	Only TCP flows Only Hierarchical max-min
Rate Throttling	Configuration at routers Added endnode *or* router software	10-50 msec	Only Hierarchical max-min
Centralized Allocation	Centralized allocation software Added router software	10 - 100 msec	More general allocation policies

allocated say 10% higher than its optimal centralized allocation to allow it to grow.

We have designed and implemented such a centralized allocator. The predictor in Step 1 is a standard least squares predictor using the last 5 measurements of offered traffic. The algorithm in Step 3 is a variant of the standard water-filling algorithm [9] which starts by finding the *weighted bottleneck*. We implemented the centralized algorithm on several large simulated 2-tier data center topologies. On a simulated topology with 16 cores, 128 edge switches and 128 million flows, the algorithm took less than 100 msec on a standard Intel Core2Duo 3GHz desktop. Smaller and more common topologies took less than 10 msec.

Table 4.1 shows the tradeoffs between the three NetShare algorithms: group allocation, rate throttling, and centralized allocation. Note that increasing generality must be paid for by smaller responsiveness and more software deployment.

4.4 Analysis

Section 4.4.1 gives a bandwidth model for NetShare with Stochastic DRR. Section 4.4.2 proves the stability of the central bandwidth allocator in a control theoretic framework.

4.4.1 Stochastic NetShare Model

Assume we have N applications (as large as several thousands). Let M be number of misbehaving applications. Initially, assume $M = 1$. Let Q be number of queues per switch port ($Q = 16$ in a standard commercial switch). For scaling analysis, assume $Q \ll N$.

Since one misbehaving application is hashed to one of Q queues, with probability $1/Q$ a well-behaved application will hash onto the same queue and get no bandwidth (in the worst case). On the other hand, with probability $(Q-1)/Q$ it gets the normal bandwidth of a queue, which is B/Q (the queue

bandwidth assuming perfect weighted fair queuing with DRR) divided by average number of applications in the queue (N/Q). Thus the average bandwidth for a good application is $0 \cdot 1/Q + B/N \cdot (Q-1)/Q = (B/N) \cdot (Q-1)/Q$.

In other words, if there is one bad application, the average bandwidth is slightly degraded by a factor $(Q-1)/Q$. So if $Q \gg M$, the degradation is small (1/16-th for 16 queues). The upshot is that it is feasible to *scale the queues with the number of bad applications, instead of the number of active applications*.

4.4.2 Stability of Centralized Allocation

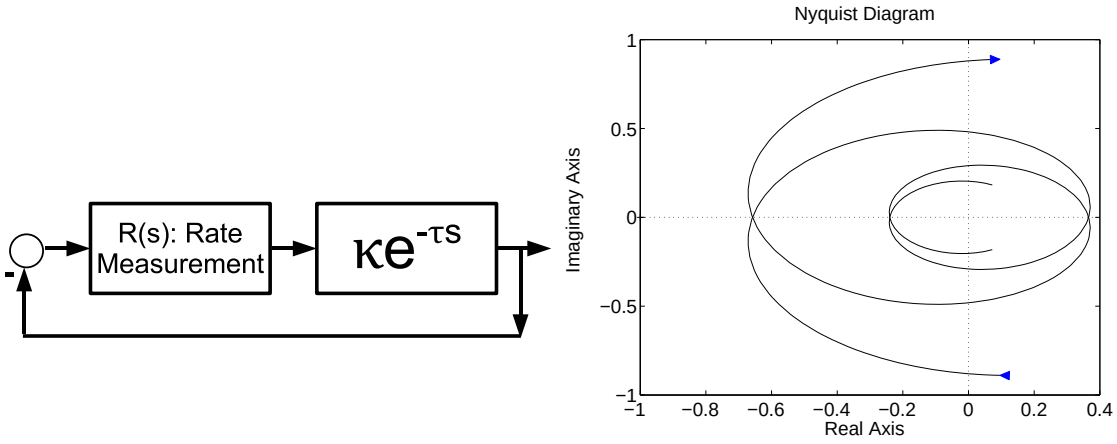
While the stability and equilibrium behaviors of TCP and active queue management schemes (Schemes 1 and 2) have been studied before [44], our centralized algorithm (Scheme 3) also involves a feedback loop for which we need to guarantee its stability and convergence. We analyze its properties in this section.

Following the general structure in [53], our design can be modeled as the feedback control system shown in Figure 4.6a. $R(s)$ represents the transfer function of rate measurement with an interval of T seconds. It is shown in [29] that a time averaging system with an interval T can be approximated with a transfer function $R(s) = \frac{2/T}{s+2/T}$.

We model the other steps of our algorithm (rate reporting, desired rate calculation and enforcement) as adding an aggregated system delay, τ, together with a general proportional gain factor κ. Hence, our overall loop transfer function is $L(s) = \frac{2\kappa/T}{s+2/T}e^{-\tau s}$. This equation indicates two key factors that would determine the system's response time: T and τ. As $T < \tau$ would mean more rate reporting traffic is introduced into the system, which is not desirable, we study the case when $\tau < T$. When $\tau \ll$ the measurement window T, it means that the delay caused by rate reporting, calculation and enforcing is much smaller than the rate measurement window. The system is reduced to a single pole system that is guaranteed to be stable. This implies that if we measure rates in an interval of a few seconds, the centralized scheme with a few milliseconds delay would certainly not be a concern.

Now let's look at the interesting case when T and τ are of similar order. Figure 4.6b shows the Nyquist plot of the system: the system would be stable if the curve touches the negative real axis to the right of the critical point -1. As the first order lag can be used to approximate $\frac{1}{T/2s+1}$ [29], the transfer function can be rewritten as $L(s) \approx \kappa e^{-(\tau+T/2)s}$.

Therefore, if we set $\kappa < 1$, then our loop can reach stability. This indicates that even when the



(a) Feedback diagram (b) Nyquist plot

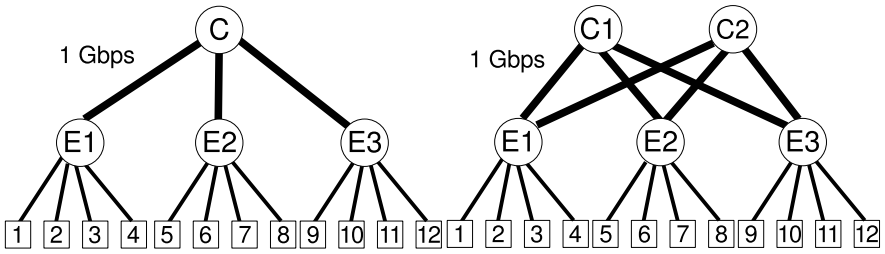
Figure 4.6. Feedback control model for the Centralized Bandwidth Allocator

system delay τ gets close to the rate measurement window T, we can still choose the system gain in such a way that guarantees stability. Note that our stability analysis only gives guidance regarding the order of magnitude of the time intervals involved; we leave detailed analysis for future work. The result indicates that with a properly designed measurement window, T, the system is stable.

4.5 Implementation

In this section, we show the effectiveness of NetShare in sharing real data center applications, providing both bandwidth isolation and statistical multiplexing. For simplicity, we model each application as a Hadoop instance. We implemented NetShare on a small scale data center testbed consisting of a 24-port Fulcrum Monaco 10GigE switch[1], a commercial switch with an extensive programming API for advanced customization. Twelve switch ports are directly connected to servers. Each server has 2 quad core Intel Xeon E5520 2.26GHz processors, 24 GB of RAM, and 16 local hard disks with 8 TB of total capacity. The remaining twelve ports are all connected to a Glimmerglass optical MEMS switch which is used like a patch panel to setup loopbacks between these twelve ports on the Fulcrum switch. This gives us the flexibility to partition the 24 port physical switch into virtual switches using VLANs and create interesting multi-switch data center topologies through the loopbacks.

We configured two data center topologies shown in Figure 4.7. Multipathing on the edge switches to utilize both core switches in Figure 4.7b is based on Equal Cost Multipath (ECMP). End to end RTT between any two nodes was less than 100us. Also, in our topologies, the term pod corresponds



(a) Single path (1 core) (b) Multipath (2 cores)

Figure 4.7. NetShare testbed topologies

to an edge switch and there are four servers connected to each pod.

We implemented Group Allocation by configuring Deficit Round Robin (DRR) at a service level. DRR already existed on the switch but we had to implement UDP rate throttling in the switches; we did not modify servers. To classify application traffic, we marked the application ID in the Type of Service (ToS) field in the packet header.

4.6 Evaluation

We describe experiments using a single path topology in Section 4.6.1, and using a multipath topology in Section 4.6.2. We evaluate the effectiveness of rate throttling in Section 4.6.3. We examine NetShare scalability to a large network topology in Section 4.6.4, and show how Stochastic NetShare scales to smaller number of queues and large number of applications in Section 4.6.5. All experiments were conducted on the Fulcrum testbed except the scalability experiments which used ns-2.

4.6.1 Single Path Experiments

We evaluated the performance of one latency critical application (modeled with FTP) in the presence of a large Hadoop Sort application with and without NetShare. We used a single-path topology with a single core switch as shown in Figure 4.7a. HDFS was configured with a default replication factor of 3 and the HDFS block size was set to 256 MB. For Hadoop, one of the servers was configured as a master while all the servers were configured as slaves. The Hadoop application was configured to use 8 disks (4 for HDFS and 4 for the task tracker) on each server.

Before the experiment began, we generated 56GB of data using the Hadoop RandomWriter application. To start the experiment, we ran a Hadoop Sort job on the 56 GB of random data using 36 maps (3 per slave) and 36 reducers (3 per slave). During the sort, we introduced an FTP job of various

Table 4.2. Completion time of the latency-sensitive FTP job for different file sizes

File size	NetShare (sec)	Without NetShare (sec)
1 MB	0.02	0.3
10 MB	0.2	1.9
100 MB	1.7	31
1 GB	18	240

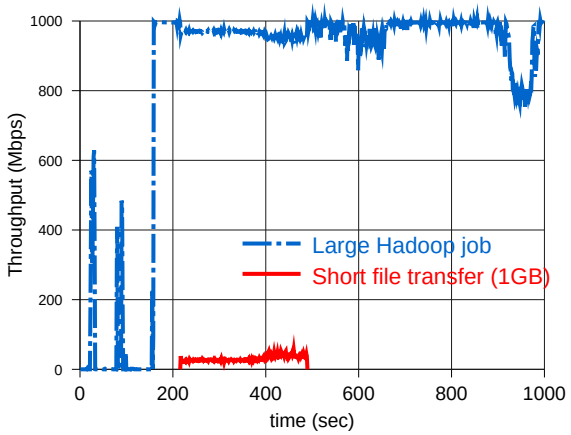
file sizes ranging from 1 MB to 1 GB from Host 1 to Host 5 at different times.

In the map phase of the Hadoop application, there is minimal network utilization and the jobs are mainly CPU/disk bottlenecked. During the reduce phase, there is considerable data shuffling and the network is highly utilized. In particular, the Hadoop application opens up a large number of TCP connections between its reducers. Thus one would hypothesize that during the reduce phase, the latency an FTP job of a few Gbytes or less could greatly increase without ”protection” from NetShare because of contention for bandwidth on the core links.

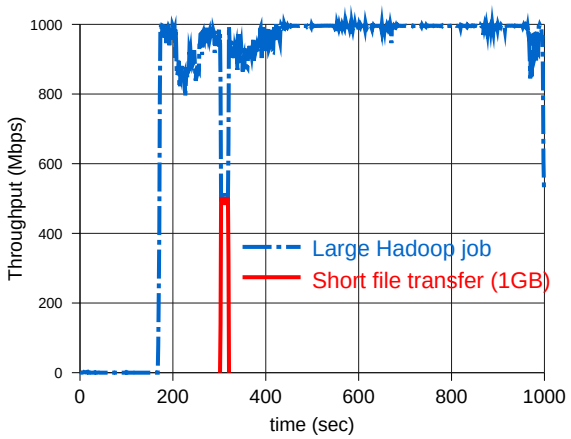
Figure 4.8a shows the bandwidth obtained by the FTP and Hadoop jobs on one of the core links without NetShare where the FTP job was started after time t = 200 when the map phase of the Sort job finished. Note that the FTP job could only acquire less than 50 Mbps of the core bandwidth of 1 Gbps and completed in 240 seconds. This is because the Hadoop application often has 20 concurrent TCP connections competing with the single TCP connection of the FTP for bandwidth on the core link. By contrast, using NetShare with equal weights for both applications, the FTP job immediately acquired 500 Mbps, i.e. the fair share, and completed in 18 seconds (Figure 4.8b). With NetShare the 1 Gbyte FTP job completed in 18 seconds (a speed of slightly less than 0.5 Gbps possibly because the speed of writing to disk is also a factor).

Note also that before t = 200 s, the map phase of the Hadoop application uses very little core bandwidth. Thus one would suspect that if the FTP application was started during the map phase, the FTP would complete much faster. Figure 4.8c shows this is indeed the case by showing the core bandwidth usage when the FTP job was started in the map phase. In this case, the FTP job acquires the full link capacity of 1 Gbps and completes in 9 seconds.

To show that these results scale down to small file sizes, Table 4.2 shows the completion time for the FTP job using file sizes as small as 1 MB. Each FTP was started in the reduce phase as described above and the completion times are described with and without NetShare. It is easy to see that with NetShare, the FTP application gets a guaranteed bandwidth of roughly 0.5 Gbps regardless of network



(a) Without NetShare



(b) With NetShare

(c) No job overlapping

Figure 4.8. Competition for bandwidth between a short latency sensitive (1GB file transfer) job and a long running Hadoop job on a core link

activity. Without NetShare, FTP performance is 10 to 15 times worse, depending on the aggressiveness of the other job. To see how these results are affected by the number of reducers, we repeated the experiment with a file size of 100 MB and changed the Hadoop application to use 14 instead of 36 reducers. The FTP transfer time without Hadoop improved from 31 to 12 seconds (corresponding to a factor of 2.5 reduction in number of connections). The FTP time with NetShare remained unchanged.

The transfer of a small file is a representative of a latency critical application. Without NetShare, the latency of such an application can vary from feast (during the Map phase) to famine (during the Reduce phase). With NetShare, on the other hand, a minimum level of performance for SLAs is possible regardless of timing. While the same *latency* guarantees can be obtained by rate-limiting the Hadoop application to 0.5 Gbps, in that case the Sort phase for the Hadoop application doubles from 800 sec to 1600 seconds. With NetShare, the Hadoop application gets 0.5 Gbps during the 18 seconds it is concurrent with the FTP but gets 1 Gbps during the remaining 780 seconds of the Sort and finishes in around 810 seconds. It is precisely this ability to provide maximal *throughput* for big jobs together with predictable latency guarantees for smaller jobs that makes mechanisms such as NetShare essential for data centers.

4.6.2 Multipath Experiments

We have seen NetShare's ability to divide the network bandwidth fairly on demand for a *single path* topology. We now show that the isolation extends to a multipath topology where NetShare truly divides the bisection bandwidth (both core links) on demand. We configured the Fulcrum switch to use the two-core switch topology shown in Figure 4.7b. Each edge/pod switch performs ECMP to hash flows onto the two paths for interpod flows. Again the core switches were the bottlenecks with an oversubscription factor of 2:1 for interpod traffic. Instead of a single FTP application and a single Hadoop application, we used *two* Hadoop Sort applications $A1$ with 96 maps and 96 reducers, and $A2$ with 96 maps and 48 reducers. Note that since ECMP only divides TCP flows across multiple paths, we need two applications, each of which opens up multiple flows.

Concretely, we first generated 96GB of data for each instance using the Hadoop RandomWriter application (8 maps per slave $\times$ 12 slaves). We subsequently ran two Hadoop Sort jobs in the two Hadoop instances $A1$ and $A2$. $A1$ used a total of 96 maps (8 per slave) and 96 reducers (8 per slave) while $A2$ used 96 maps (8 per slave) and 48 reducers (4 per slave). All other Hadoop parameters were as described in Section 4.6.1.

First we ran the sort jobs without NetShare in the network. In this case, $A1$ used twice the

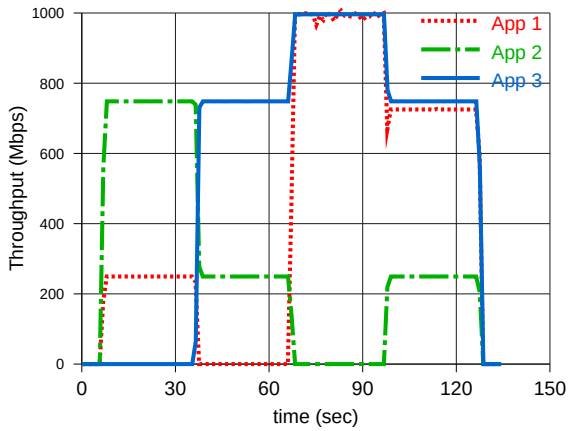


Figure 4.9. NetShare with Group Allocation (DRR) + Rate Throttling

bandwidth (summed over all core links, the "bisection bandwidth") when compared to $A2$ because it opens up nearly twice the connections. Next, we set up NetShare by configuring DRR with equal weights for the 2 applications. Note that the bandwidths on the various core links are not shared as uniformly because of hashing effects and because the sort does not saturate all links consistently.

Using NetShare, $A1$ completed sorting in 1633s while $A2$ completed sorting the data in 1810s. To show that NetShare is sharing the bisection bandwidth, we ran $A1$ and $A2$ in exactly the same way except using the single core topology. Using NetShare in the single core topology we found that $A1$ and $A2$ finished sorting in 3070s and 3212s. After factoring out the 500s for the map phase (that is unaffected by the extra bandwidth), the bisection bandwidth appears to be nearly equally shared between the two "services" and both have been sped up by nearly a factor of 2. Some difference is not surprising because $A1$ has more connections, and thus its use of ECMP load balancing is likely more effective than $A2$.

4.6.3 How Effective is Rate Throttling?

We deploy three applications with the testbed in Figure 4.7a: $A1$ generates a TCP flow from host $H1$ to host H5; $A2$ generates a UDP flow from host $H2$ to host $H9$; and $A3$ generates a UDP flow from host $H6$ to host $H10$. The weights of the applications A1, $A2$, A3 were set to 1:3:9 respectively.

Table 4.3 shows the traffic pattern. During the time 5-35s, $A3$ is inactive and thus the TCP flow $A1$ (weight 1) contends with the UDP flow $A2$ (weight 3) for the core link $E1,C$. From time 35-65, the two UDP applications $A2$ and $A3$ (with weights 3 and 9) contend for the core link $C,E3$. From time 65-95, the TCP application $A1$ contends with the high weight UDP application but only on the link from edge

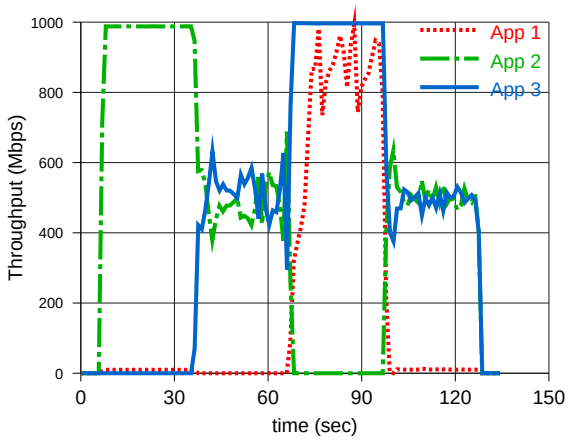


Figure 4.10. No NetShare mechanisms

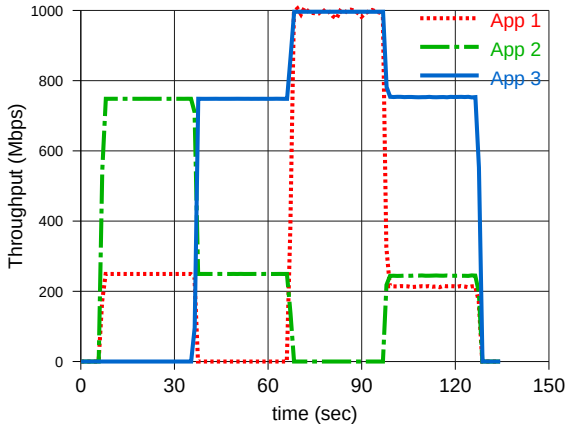


Figure 4.11. NetShare with Group Allocation Alone

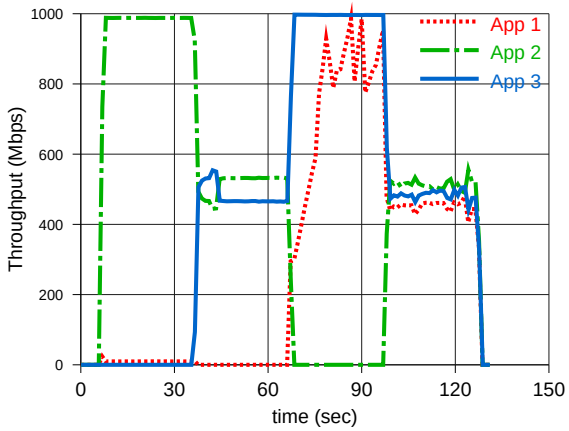


Figure 4.12. NetShare with Rate Throttling Alone

Table 4.3. Traffic pattern that indicates times during which different flows are active.

Time(s)	$A1$	$A2$	$A3$	Bottlenecks
5-35	✓	✓	X	$E1, C$
35-65	X	✓	✓	$C, E3$
65-95	✓	X	✓	$E2, C$
95-125	✓	✓	✓	All of the above

router $E2$ to core router C. Thus the UDP application can only interfere with TCP *acknowledgements* for $A1$ destined to Host $H1$.

We evaluate the following scenarios.

1. Group Allocation and Rate Throttling: As shown in Figure 4.9, each application receives its weighted share of the network resources. For instance, during the period 5-35s, $A2$ gets 750 Mbps and $A1$ gets 250 Mbps as they are sharing the bottleneck $E1, C$ in the ratio 3:1 of their weights. However, from t=95-125s $A1$'s TCP flow gets close to 725Mbps, which exceeds the share allocated by its application weight, but since $A2$'s UDP flow has a downstream bottleneck on the link $C, E3$ only 250 Mbps of the UDP flow is "useful" (that is the throughput of the UDP flow that actually reaches the receiver $H9$). So in this case, $A2$ gets rate limited at the ingress switch to 275 Mbps (250 * 1.1) which results in $A1$ getting close to 725 Mbps. Without rate throttling we will see that $A1$ will send at much higher rates and get dropped at C.

2. No NetShare: As shown in Figure 4.10, when $A1$ and $A2$ are both active in time 5-35s, $A1$'s TCP flow is overwhelmed by $A2$'s UDP flow and receives zero throughput. Note that from t=65-95s, $A1$'s throughput does not reach 1 Gbps although its path from $H1$ to $H5$ is not affected by $A3$'s UDP flow. However, the ACKs from $H5$ to $H1$ share a link with $A3$'s UDP flow; some of the ACKs get dropped, this results in $A1$'s throughput dropping to sometimes as low as 750 Mbps.

3. Group Allocation Only: Figure 4.11 shows the impact of omitting Rate Throttling. In the period t=95-125s, $A2$ and $A3$ share the bandwidth of their shared bottleneck link in the ratio of their application weights (3:9). Thus $A2$ only receives 250Mbps. Unfortunately, $A1$ also receives only 250Mbps because $A2$ continues to send greedily at 750Mbps on the $E1, C$ link of which 500Mbps gets dropped at C.

4. Rate Throttling Only: In Figure 4.12, the behavior is similar Case 1 from t=5-95s. However from t=95-125s, $A1$ only achieves nearly 450-500Mbps. This is because $A2$ gets rate limited at $E1$ to a little over 500Mbps, so $A1$ is able to use the remaining bandwidth on the $E1, C$ link. Thus rate throttling

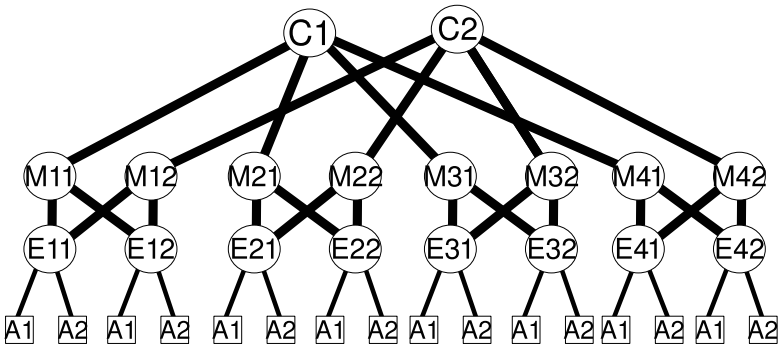


Figure 4.13. Three-tiered data center topology used for scalability experiments

and fair queuing are orthogonal and complementary mechanisms.

4.6.4 Scaling to Larger Topologies

Due to the constraints of our hardware testbed, we explore the scalability of NetShare to larger topologies and more applications by ns-2 simulation. As shown in Figure 4.13, each application has its own dedicated node at each edge switch. Two different applications such as $A1$ and $A2$ are assigned to nodes alternately. All links between the switches are 10 Mbps. We vary number of applications and TCP connections per application.

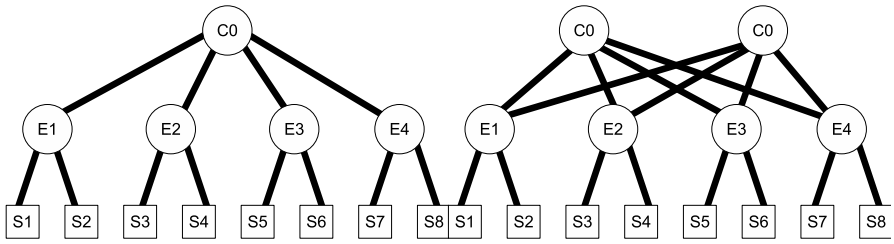
Within each application, the communication pattern is all-pairs. Furthermore, each pair of nodes open up to C connections in parallel, where C is a parameter. We explore the parameter space of the number of applications (N), application connections (C) and policies with and without NetShare.

We vary the parameters of the first application A_1 and keep same configurations for the remaining $N-1$ applications A_2, ..., A_N. We report only the maximum and minimum bandwidths for the applications in set A_2, ..., A_N. W denotes the DRR weights.

We observe that NetShare with Group Allocation via DRR comes close to achieving the desired network sharing independent of the number of connections that each individual application makes. For example, as shown in Table 4.4, each of the four applications with two connections between any pair of nodes get 36 Mbps. Without NetShare, one heavyweight application can acquire more bandwidth by increasing its connections per node pair (e.g. upto 84 Mbps with 8 connections and 106 Mbps with 16 connections). On the other hand, NetShare always prevents that application from getting more that 40 Mbps. Note that this is slightly above its fair share (36 Mbps) but independent of the connections made by other applications.

Table 4.4. Application bisection bandwidth under several traffic parameters and with and without NetShare (DRR only).

N	C	DRR?	W	Bandwidth (Mbps)		
				A_1	max $A_{2..N}$	min $A_{2..N}$
1	2	-	-		131.7	
1	8	-	-		141.7	
4	2, 2	Y	1, 1	35.6	35.1	37.0
4	2, 2	Y	1, 2	22.6	40.0	41.5
4	8, 2	Y	1, 1	40.2	34.0	36.0
4	8, 2	N	-	83.8	19.4	20.6
4	8, 2	Y	1, 2	24.7	38.5	41.9
4	16, 2	Y	1, 1	40.1	35.1	35.8
4	16, 2	N	-	105.5	12.2	14.0
4	16, 2	Y	1, 2	25.2	39.4	40.7
8	2, 2	Y	1, 1	18.3	17.7	19.0
8	2, 2	Y	1, 2	10.4	18.9	20.0
8	8, 2	Y	1, 1	20.4	17.1	18.5
8	8, 2	Y	1, 2	11.6	18.3	19.8
8	8, 2	N	-	53.1	11.6	13.9
8	16, 2	Y	1, 1	20.2	17.1	18.4
8	16, 2	Y	1, 2	11.7	19.0	20.2
8	16, 2	N	-	77.9	9.1	10.4



(a) Single path (1 core) (b) Multipath (2 cores)

Figure 4.14. Topologies for Stochastic DRR experiments

Furthermore, bisection bandwidths also reflect NetShare administrative weights. For example, the bisection bandwidths for the four applications, one of which having weight 1 and the remaining having weight 2, are 22 Mbps and 41 Mbps respectively. Also, the application with smaller weight cannot increase its share by increasing the number of connections between its nodes. Finally, we scale the experiments from 2 to 4 to 8 applications and observe similar effects.

4.6.5 Scalability of Stochastic NetShare

A concern with Group Allocation is that it requires a number of queues equal to to the number of applications. To scale beyond the 16 queues available today and the 1000's available shortly with AFD-based routers[54], we proposed stochastic NetShare. We now evaluate this scheme and compare it with the analysis in Section 4.4.1.

Figure 4.14a shows our experimental setup with one core switch $C0$, four edge switches $E1$ to $E4$, and eight servers $S1$ to $S8$ (two servers per edge switch). Note that all links have equal capacity with an oversubscription factor of 2 at core links. There is an instance of each application on all servers and the traffic pattern is all-to-all. We evaluate $N = 32$ applications, in which one application is "bad", i.e. with low priority weight and competing aggressively for bandwidth by opening ten times the number of connections. Link capacity is $B = 100$ Mbps. We evaluate the scalability of Stochastic DRR by varying the number of DRR queues per switch port $Q = 4, 8, 16$.

Stochastic DRR with equal weights Table 4.5 shows the application bandwidth at one typical server. All DRR queues are assigned the same weights and independent of the number of applications being hashed into them. The rates fluctuate but the bandwidth mean and variance are consistent among all applications. As captured by our model in Section 4.4.1, the impact of the bad application declines with

Table 4.5. Scalability of Stochastic DRR: application bandwidth at one typical server in (mean, stddev) over time. All queues have equal weights. $\bar{B} = \frac{B}{N} \cdot \frac{Q-1}{Q}$ is the expected bandwidth per application. Ideal bandwidth is $\frac{B}{N} = 3.1$ Mbps. T is rehashing period (in seconds).

		T=5	T=10	T=20
Q=4	Bad app	(13.6, 3.3)	(14.3, 2.6)	(12.4, 2.8)
$\bar{B} = 2.3$	Good app	(2.2, 1.5)	(2.2, 1.3)	(2.4, 1.4)
Q=8	Bad app	(8.9, 2.1)	(8.9, 1.9)	(9.0, 2.0)
$\bar{B} = 2.7$	Good app	(2.8, 2.2)	(2.3, 1.7)	(2.5, 1.8)
Q=16	Bad app	(6.7, 1.5)	(6.6, 1.6)	(6.7, 1.7)
$\bar{B} = 2.9$	Good app	(2.8, 1.9)	(3.1, 2.2)	(3.3, 2.5)

additional queues in the system. The mean is close to our prediction (the ideal bandwidth is B/N which around 3.1 Mbps, together with a degradation of $\frac{Q-1}{Q}$, where Q is the number of queues). Note that while we have simulated only 32 applications because of ns-2 limitation, the model, which is validated here, should scale to larger numbers of applications.

Note that periodic rehashing of applications onto DRR queues reduces variance. Clearly the rehashing period T should neither be too small (for good stability and minimizing out-of-order packets) nor too large (for good bias correction). 10 seconds appears to be a good compromise.

Priorities and Stochastic NetShare In this experiment, we divided 32 applications into two buckets: 16 of low priority (weight 1) and 16 of high priority (weight 2). All the low priority applications are more bandwidth-aggressive. We omit the details but note that there was isolation between low priority and high priority applications. However, the final bandwidth ratios averaged across all 8 connections of each service show a ratio of approximately 1:1.4 between low weight and high weight services, which is less than the ideal ratio of 1:2.

Note that if there is only one single hop, DRR guarantees bandwidth per application being divided exactly according to weight assignment. However, in a general topology with with multiple hops, bandwidth per application is only approximately related to its weight because, as we said earlier, errors can cascade across hops. If no high priority traffic is available at a core router because it is queued at an edge switch, the core router can send more of the low priority queue traffic. We conclude that Stochastic NetShare works well with equal weights; with unequal weights, it only works approximately. It will work best with routers that should be entering the market soon with 1000's of AFD [54] classes.

4.7 Automatic Weight Assignment

We propose a simple scheme for automatic weight assignment for applications or services. In an enterprise or cloud data center, when resources are provisioned for an application or customer, the customer usually requests some number of servers or VMs (instances) each with some number of CPUs, RAM and disk. Besides this, each instance must also be allocated some units of network bandwidth. For example, if each server has a 10Gbps NIC, we could place upto 10 VMs on the server each allocated 1Gbps of bandwidth.

We leverage two fundamental ideas. First, we use per switch-port weights, i.e. weights per application can vary from switch to switch, and even from one switch port to another. Second, we assign weights based on VM placement. In particular, we compute both the *downstream* and *upstream* sums of the bandwidths assigned to all VMs allocated to application A with respect to switch port P. Then the weight assigned to A at P is the smaller of the two.

As an example, suppose there is an accounting application with 2 servers connected to an edge switch and each server has 4 instances of an accounting application. The uplink of the edge switch is connected to a core switch and from there to other servers with 8 instances of the accounting application. Assume each VM instance is allocated 1 Gbps. Then we set the accounting application's weight at each of the 2 downlink ports of the edge switch to be 4 (smaller of 4 and 12), while we set its weight at the uplink port to be 8 (smaller of 8 and 8). Note that taking the minimum makes sense because even if there are 8 VMs upstream that can transmit at 8 Gbps, there are only 4 VMs downstream that can receive only at an aggregate capacity of 4 Gbps.

We make the following assumptions. First, VM bandwidths at servers are enforced using mechanisms like Linux HTB qdisc. Second, we have knowledge of the complete topology and placement of each VM instance. Third, in a multirooted tree network, forwarding is based on ECMP. We assume that each egress port in a switch either forwards traffic upwards from a server towards the core layer (up facing ports) and the rest of the fabric or forwards traffic down towards a server (down facing ports). This is a technique that simplifies the routing while also avoiding routing loops. Finally, in this setup, each egress port on the switch has a definite role in terms of the which server's traffic flows through it. For example in a two level multirooted tree network, a down facing port on a core switch can forward traffic to servers in a particular edge switch from all other edge switches while an upfacing port on an edge switch can forward traffic from the servers on that edge switch to all servers in other edge switches. Servers to which

the particular port forwards traffic are called downstream servers and the servers from which this traffic could be coming are called upstream servers of that port.

While we have used global weights for the bulk of this paper for simplicity, we note that extending the definitions to per-port weights is straightforward. For example, the standard water-filling algorithm [9] must be modified to use the weight of each application/service at the current bottleneck link as opposed to a global weight.

4.8 Related Work

The need for QoS in data centers has become apparent in several recent papers. Seawall[65] performs isolation by enforcing VM-to-VM rates for VMs belong to one application/customer in the hypervisor using congestion feedback. Seawall mechanisms are more complex but they are more granular (VM to VM) and can handle large numbers of applications/customers. AF-QCN [38] is an approximation to the standard QCN Layer congestion control scheme that can use different drop rates for each application. However, the scheme requires router hardware changes and currently provides guarantees only for a single link. mClock [32] proposes an algorithm specific to I/O resource allocation in the hypervisor at end hosts. SecondNet[34] proposes a heuristic to map *virtual data center* specifications into the physical data center infrastructure with constraints on resource demands. SecondNet uses reservations and hence is complementary to NetShare. Flowvisor [63] virtualizes a testbed network to allow multiple experiments to run concurrently but does so using suboptimal hop-by-hop allocation. The HP QoS Framework [40] allows network QoS to be implemented centrally but is only a framework that can, in fact, be used to implement NetShare. [7] discusses a VM placement policy based on the network requirements for each customer. Bandwidth is reserved for each customer's VMs and the fair share for each flow is computed by a centralized controller for that customer. Multiplexing across customers requires coordination among controllers of different customers or a single central controller similar to NetShare.

Fair queuing [21] and Core-stateless fair queuing(CSFQ) [71] do not guarantee max-min allocation. [67] and [64] extend CSFQ to obtain max-min allocations but require header changes. DiffServ allows statistical multiplexing by marking traffic exceeding the allocated share and dropping marked packets if needed. For lack of space, we omit experiments that show that DiffServ dropping reduces efficiency compared to NetShare. NetShare differs from DRL which controls bandwidth *in and out* of the cloud as opposed to *within* the cloud.

Many papers (e.g., [41, 19]) show that max-min allocation in the presence of load balancing or

multipath is, in general, NP-complete. However, the hardness result does not apply to regular data center topologies where routes are fixed. Duffield et al introduce a hose model [23] to specify aggregate demand but requires complex algorithms which decrease responsiveness. NetShare assumes that routing is fixed by a routing protocol such as OSPF. Thorup and Rexford show that there is considerable flexibility to change routes by changing OSPF weights [28] without pinning every route using MPLS. Traffic engineering such as OSPF-TE (RFC 3630) can be used to efficiently route traffic but does not allocate bandwidth across services. RFC 3630 (Traffic Engineering Extensions to OSPF) only supports static reservation.

4.9 Summary

NetShare allows managers to use weights to tune the relative bandwidth allocation for different services, providing allocation, isolation and statistical multiplexing without changing routers. Managers can use NetShare with Virtual Disks and Virtual Machines to create Virtual Data Centers. While NetShare is based on a simple packaging of existing ideas (max-min fair share, stochastic fair queuing, UDP rate throttling) no such mechanism exists today.

Without NetShare, latency critical jobs can be slowed down by Hadoop jobs. With NetShare, latency critical jobs can be protected without artificially slowing down large jobs. Group allocation works well with only configuration changes at routers; it can be extended to scale to more applications than the number of DRR queues available today either using AFD [54] or stochastic methods. Rate throttling protects against UDP application misbehavior and may be simpler than deploying TCP-friendly UDP. Finally, centralized allocation can implement arbitrary bandwidth allocation policies, and can provide stability. We suggest a simple automatic weight assignment algorithm based on finding the number of VMs upstream and downstream from a port.

Chapter 4, in part, is a reprint of the material as it appears in “NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers” in *Proceedings of ACM SIGCOMM Computer Communication Review (CCR)* 2012. Lam, Vinh The; Radhakrishnan, Sivasankar; Pan, Rong; Vahdat, Amin; Varghese, George. ACM, 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Conclusions

In this dissertation, we have discussed novel designs for scalable bandwidth allocation on horizon of emerging technologies: event loggers, load balancing, and cloud services. Our solutions are based on probabilistic algorithms and designs because these approaches are inexpensive to implement, require constant memory and processing time, and are applicable to a wide range of contexts. In Chapter 2, we showed that our Carousel scalable logger can collect nearly all sources, assuming they send persistently, in nearly optimal time while it is easy to implement in both software and hardware. Furthermore, Carousel is applicable to other monitoring tasks where the events must be logged at high speed, but with low logging memory and small logging speeds. In Chapter 3, we have described Flame, a system to do dynamic load balancing in data center networks. Our key techniques include the proxy queue to measure traffic load accurately, the power of choice to select the optimal path in a hardware-friendly manner, and the aging mechanism and heavy-hitter filter to optimize memory usage. We demonstrated the efficacy of Flame through analytical studies as well as simulations on realistic network traces and synthetic data center workloads as inspired by recent studies of production data centers. In Chapter 4, we showed that the notion of a virtual data center requires *both* computing and bandwidth guarantees. NetShare allows managers to use weights to tune the relative bandwidth allocation of different services. Without NetShare, a service can be held hostage by other services that either open multiple connections or use non-compliant congestion control protocols. We introduced three simple techniques for implementing the NetShare abstraction ranging from group allocation per link to centralized allocation that trade decreasing responsiveness for more general allocation policies.

Bibliography

[1] Fulcrum Monaco http://www.fulcrummicro.com/.

[2] . IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force. http://www.ieee802.org/3/ba/.

[3] Mohammad Al-Fares, Rishi Kapoor, George Porter, Sambit Das, Hakim Weatherspoon, Balaji Prabhakar, and Amin Vahdat. User-extensible Active Queue Management with Bumps on the Wire. In *ANCS*, 2012.

[4] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2010.

[5] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[7] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable data center networks. In *Prc. SIGCOMM'11*.

[8] D. Bergamasco and Rong Pan. Backward Congestion Notification (BCN) Version 2.0. *IEEE 802.1 Meeting, 2005*.

[9] D. Bertsekas and R. Gallager. *Data Networks*. P. H., 1992.

[10] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, 1970.

[11] A. Borodin and R. El-Yaniv. Online computation and competitive analysis. In *Cambridge University Press*, 1998.

[12] J. L. Boudec. Rate adaptation, congestion control and fairness: A tutorial. 2008.

[13] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Pamidge, L. Pererson, K. Ramakrisbnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the Internet. In *IETF RFC (Informational) 2309, April 1998*.

[14] R. Braden and L. Zhang et al. Resource reservation protocol (rsvp) – version 1, function specifica-

tion, rfc 2205. In *http://www.rfc-editor.org/rfc/rfc2205.txt*, 1997.

[15] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Math*, 1(4), 2003.

[16] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 15–28, 2005.

[17] CAIDA. CoralReef Software. http://www.caida.org/tools/measurement/coralreef/.

[18] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *Proc. SIGCOMM '07*.

[19] J. Chou and B. Lin. Optimal multi-path routing and bandwidth allocation under utility max-min fairness. In *IWQoS '09*.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proc. SIGCOMM '89*.

[21] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proc. SIGCOMM '89*.

[22] S. Dharmapurikar and V. Paxson. Robust tcp stream reassembly in the presence of adversaries. In *14th USENIX Security Symposium*, pages 5–5, 2005.

[23] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. Van Der Merwe. Resource management with hoses: point-to-cloud services for virtual private networks. *IEEE/ACM Trans. Netw. 2002*.

[24] Cristian Estan and George Varghese. New Directions in Traffic Measurement and Accounting. In *SIGCOMM*, 2002.

[25] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3), 2000.

[26] S. Floyd and V. Jacobson. Random Early Detechon Gateways for Congestion Avoidance. In *IEEE/ACM Transaction on Networking, 1993*.

[27] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. Netw.'95*.

[28] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional ip routing protocols. *IEEE Comm.*, 2002.

[29] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. Feedback Control of Dynamic Systems. In *3rd ed, Addison Wesley*.

[30] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. In *Communications of the ACM, January 2012*.

[31] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[32] Ajay Gulati, Arif Merchant, and Peter Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proc. OSDI '10*.

[33] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[34] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *Proc. ACM CoNEXT '10*.

[35] Ellen L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE J. Comms*, 1991.

[36] S. Hogg. Security at 10 Gbps: http://www.networkworld.com/community/node/39071. In *Network World*, 2009.

[37] Raj Jain. Congestion Control and Traffic Management in ATM Networks: Recent Advances and A Survey. In *Computer Networks and ISDN Systems, 1996*.

[38] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. AF-QCN: Approximate Fairness with Quantized Congestion Notification for Multi-tenanted Data Centers. In *Proc. Hot Interconnects '10*.

[39] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Flare: Responsive Load Balancing Without Packet Reordering. In *ACM CCR*, 2007.

[40] Wonho Kim, Puneet Sharma, Jeongkeun Lee, Sujata Banerjee, Jean Tourrilhes, Sung-Ju Lee, and Praveen Yalagandula. Automated and scalable qos control for network convergence. In *USENIX INM/WREN*, 2010.

[41] J. Kleinberg, Y. Rabani, and E. Tardos. Fairness in routing and load balancing. In *J. Comput. Syst. Sci*, pages 568–578, 1999.

[42] B Lampson. Alto: A personal computer. In *Computer Structures: Principles and Examples*, 1979.

[43] Michael Laor and Lior Gendel. The Effect of Packet Reordering in a Backbone Link on Application Throughput. In *IEEE Network*, 2002.

[44] Steven H. Low. A duality model of TCP and Queue Management Algorithms. In *IEEE/ACM Trans. Net. Vol 11, 2003*.

[45] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. In *SIGCOMM CCR*, 1997.

[46] P. McKenney. Stochastic fairness queueing. In *Internetworking,1991*.

[47] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR'08*.

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

[49] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. In *PhD thesis*, 1996.

[50] netem. The Linux Foundation. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[51] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. In *ACM Queue, 2012*.

[52] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A simple model and its empirical validation. In *ACM SIGCOMM'98*.

[53] Fernando Paganini, Zhikui Wang, John C. Doyle, and Steven H. Low. Congestion control for high performance, stability and fairness in general networks. In *IEEE/ACM Trans. Net. Vol 13, 2005*.

[54] R. Pan, B. Prabhakar, F. Bonomi, and B. Olsen. Approximate Fair Bandwidth Allocation: A Method for Simple and Flexible Traffic Management. In *46th Allerton Conf., 2008*.

[55] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation. In *IEEE Infocom, 2000*.

[56] Arash Partow. General purpose hash functions: http://www.partow.net/programming/hashfunctions/.

[57] J. B. Postel. Transmission control protocol. Technical Report Technical Report RFC 793, Information Sciences Institute, September 1981.

[58] M. Raab and A. Steger. Balls into bins: a simple and tight analysis. In *Workshop on Randomization and Approximation Techniques in Computer Science*, 1998.

[59] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. Tcp fast open. In *ACM CoNEXT, 2011*.

[60] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[61] A. Ross. The coupon subset collection problem. In *Journal of Applied Probability*, 2001.

[62] Colleen Shannon, Emile Aben, kc claffy, and Dan Andersen. The CAIDA Anonymized 2008 Internet Traces. http://www.caida.org/data/passive/passive_2008_dataset.xml.

[63] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. 2009. Technical report.

[64] Z. Shi. Token-based congestion control: Achieving fair resource allocations in P2P networks. In

K-INGN'08.

[65] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance Isolation in Cloud Datacenter Networks. In *Proc. HotCloud'10*.

[66] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM'95*.

[67] Raghupathy Sivakumar, Tae-Eun Kim, Narayanan Venkitaraman, Jia-Ru Li, and Vaduvur Bharghavan. Achieving per-flow weighted rate fairness in a core stateless network. In *Proc. ICDCS'00*.

[68] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *SIGCOMM CCR'08*.

[69] Snort. Snort ids: http://www.snort.org.

[70] W. Stadje. The collector's problem with group drawings. In *Advances Applied Probability*, 1990.

[71] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Trans. Netw. '03*.

[72] tcn. Trace Control for Netem. http://tcn.hypert.net.

[73] tcptrace. . http://www.tcptrace.org.

[74] G. Varghese, J. Fingerhut, and F. Bonomi. Detecting evasion attacks at high speeds without reassembly. *SIGCOMM*, 36(4), 2006.

[75] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and robust tcp stream normalization. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 96–110, 2008.

[76] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert G. Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *Proc. SIGCOMM '06*.

[77] W. Wu, P. Demar, and M. Crawford. Sorting Reordered Packets with Interrupt Coalescing. In *Comput. Netw.*, 2009.

[78] C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *ACM CCS '02*.