

# UC San Diego

## UC San Diego Electronic Theses and Dissertations

### Title

Network Performance Improvements For Web Services : : An End-to-End View

### Permalink

<https://escholarship.org/uc/item/7h6975zs>

### Author

Radhakrishnan, Sivasankar

### Publication Date

2014

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Network Performance Improvements For Web Services – An End-to-End View

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

in

Computer Science

by

Sivasankar Radhakrishnan

Committee in charge:

Amin Vahdat, Chair  
George Papen  
George Porter  
Stefan Savage  
Geoffrey M. Voelker

2014

Copyright

Sivasankar Radhakrishnan, 2014

All rights reserved.

The Dissertation of Sivasankar Radhakrishnan is approved and is acceptable in quality and form for publication on microfilm and electronically:

---

---

---

---

---

Chair

University of California, San Diego

2014

## TABLE OF CONTENTS

Signature Page .....	iii
Table of Contents .....	iv
List of Figures .....	vii
List of Tables .....	ix
Acknowledgements .....	x
Vita .....	xiii
Abstract of the Dissertation .....	xv
Chapter 1 Introduction .....	1
1.1 Challenges .....	2
1.1.1 Wide Area Network Latency .....	2
1.1.2 Data Center Network Infrastructure .....	3
1.1.3 Co-tenancy and Network Performance Isolation .....	3
1.2 Efficient Networking for Modern Web Services .....	4
1.2.1 TCP Fast Open .....	4
1.2.2 Dahu .....	6
1.2.3 SENIC .....	7
1.3 Organization .....	7
Chapter 2 Design and Implementation Principles .....	8
2.1 Principles .....	8
2.1.1 State Management Principle .....	8
2.1.2 Short Circuiting Principle .....	9
2.1.3 Inheritance Principle .....	9
2.2 Implications of the Principles .....	10
2.2.1 State Management Principle .....	10
2.2.2 Short Circuiting Principle .....	12
2.2.3 Inheritance Principle .....	14
2.3 Summary .....	15
Chapter 3 TCP Fast Open .....	16
3.1 Introduction .....	16
3.2 Motivation .....	18
3.2.1 Google Server Logs Analysis .....	19
3.2.2 Chrome Browser Statistics .....	20
3.3 Design .....	22

3.3.1	Context and Assumptions .....	22
3.3.2	Design Overview .....	24
3.3.3	Cookie Design .....	26
3.3.4	Security Considerations .....	26
3.3.5	Handling Duplicate SYN Segments .....	29
3.3.6	API Changes .....	30
3.4	Deployability .....	32
3.4.1	New TCP Options / Data in SYN .....	32
3.4.2	Server Farms .....	33
3.4.3	Network Address Translation (NAT) .....	34
3.4.4	TCP Option Space .....	34
3.5	Implementation .....	34
3.5.1	Kernel Support .....	35
3.5.2	Application Support .....	36
3.6	Evaluation .....	36
3.6.1	Whole Page Download Performance .....	36
3.6.2	Server Performance .....	39
3.7	Discussion .....	40
3.7.1	One Time Cookies .....	40
3.7.2	Data After SYN .....	42
3.7.3	Server-side TFO Cache .....	43
3.7.4	TCP Fast Open in Low Latency Networks .....	43
3.7.5	Cookie-less TCP Fast Open .....	44
3.8	Related Work .....	45
3.9	Summary .....	47
3.10	Acknowledgments .....	48
Chapter 4	Dahu: Commodity Switches for Direct Connect Data Center Networks	50
4.1	Introduction .....	51
4.2	Motivation and Requirements .....	53
4.2.1	Challenges .....	54
4.2.2	Dahu Requirements and Design Decisions .....	56
4.3	Switch Hardware Primitives .....	57
4.3.1	Port Groups With Virtual Ports .....	58
4.3.2	Allowed Port Bitmaps .....	60
4.3.3	Eliminating Forwarding Loops .....	61
4.4	Switch software .....	62
4.4.1	Background on HyperX Topology .....	63
4.4.2	Non-Minimal Routing .....	64
4.4.3	Traffic Load Balancing .....	68
4.4.4	Load Balancing Algorithm .....	71
4.4.5	Load Balancing Heuristic .....	72

4.4.6	Fault Tolerance .....	73
4.5	Deployability .....	73
4.6	Evaluation .....	75
4.6.1	Simulator .....	75
4.6.2	HyperX Networks .....	77
4.6.3	Fat-Tree Networks .....	82
4.6.4	MPTCP in HyperX Networks .....	83
4.7	Discussion .....	85
4.8	Related Work .....	86
4.9	Summary .....	87
4.10	Acknowledgments .....	88
Chapter 5	SENIC: Scalable NIC for End-Host Rate Limiting .....	89
5.1	Introduction .....	90
5.2	Motivation .....	92
5.2.1	The Need For Scalable Rate Limiting .....	92
5.2.2	Limitations of Current Systems .....	93
5.3	Design .....	96
5.3.1	Service Model .....	96
5.3.2	CPU and NIC Responsibilities .....	96
5.4	Packet Scheduling in SENIC .....	100
5.4.1	SENIC Packet Scheduling Algorithm .....	100
5.4.2	Hierarchical Bandwidth Sharing .....	103
5.5	Advanced NIC features .....	104
5.5.1	OS and Hypervisor Bypass .....	105
5.5.2	Other Features .....	107
5.6	Implementation .....	108
5.6.1	Software Prototype .....	108
5.6.2	NetFPGA Prototype .....	109
5.7	Evaluation .....	112
5.7.1	Hardware Microbenchmarks .....	113
5.7.2	Software Macrobenchmarks .....	116
5.8	Practical Considerations .....	121
5.9	Related Work .....	123
5.10	Summary .....	124
5.11	Acknowledgments .....	125
Chapter 6	Conclusion .....	126
	Bibliography .....	129

## LIST OF FIGURES

Figure 1.1.	Overview of key networking challenges addressed in this dissertation	5
Figure 3.1.	TCP handshake overhead for Google services	19
Figure 3.2.	TCP handshake overhead for all web services accessed by Chrome users on Windows	21
Figure 3.3.	TCP Fast Open connection overview	24
Figure 3.4.	Server application sample code	31
Figure 3.5.	Client application sample code	32
Figure 3.6.	CPU utilization vs. web server load	39
Figure 4.1.	Illustration of shortest and non-shortest path routing	54
Figure 4.2.	Datapath pipeline for packet forwarding in Dahu	57
Figure 4.3.	HyperX topology	63
Figure 4.4.	Restricting non-minimal forwarding	66
Figure 4.5.	Port group rebalancing algorithm	70
Figure 4.6.	Load balancing heuristic	72
Figure 4.7.	Simulator throughput vs. theoretical maximum	76
Figure 4.8.	Throughput gain with Clique traffic pattern	79
Figure 4.9.	Average hop count with Clique traffic pattern	79
Figure 4.10.	Link utilization with Clique traffic pattern	80
Figure 4.11.	Dahu performance with Mixed Traffic Pattern	81
Figure 4.12.	Throughput gain for $k = 32$ Fat-tree with Stride and Random (Rnd) traffic patterns	83
Figure 4.13.	Dahu and MPTCP performance comparison	84
Figure 5.1.	Comparison of CPU overhead and accuracy of current software and hardware rate limiting approaches	94



Figure 5.2.	SENIC— “Schedule and Pull” model . . . . .	98
Figure 5.3.	SENIC hardware design . . . . .	99
Figure 5.4.	Transmit schedule example . . . . .	102
Figure 5.5.	SENIC architecture with NIC virtualization . . . . .	105
Figure 5.6.	SENIC hardware prototype packet scheduler overview . . . . .	110
Figure 5.7.	Maximum throughput per traffic class . . . . .	115
Figure 5.8.	CDF of memcached latency at different loads . . . . .	117
Figure 5.9.	SENIC memcached latency . . . . .	118
Figure 5.10.	SENIC memcached latency with UDP background traffic . . . . .	119
Figure 5.11.	Throughput achieved by UDP background traffic . . . . .	120

## LIST OF TABLES

Table 2.1.	Summary of implications of the principles .....	10
Table 3.1.	Average page load time for various websites for an emulated residential broadband user .....	37
Table 5.1.	Pros and cons of current hardware and software approaches to rate limiting .....	91
Table 5.2.	Per-class metadata in NIC SRAM .....	97
Table 5.3.	Rate limiting accuracy as we vary the number of traffic classes, and the rate limit per class .....	113

## ACKNOWLEDGEMENTS

My graduate school career has been an extremely memorable and satisfying experience. I have had the pleasure of collaborating with several incredibly smart people and learning from them. I would like to take this opportunity to thank them.

First, I wish to express my gratitude to my advisor, Amin Vahdat who has been a constant source of inspiration for me. He has relentlessly encouraged me to push further and think beyond my limits. He has always inspired his students to aspire higher and set ambitious goals, and instilled in them the confidence to achieve the goals. He pushed me to think critically and deeply about problems, while persevering to solve them. I treasure the values he inculcated in students such as humility, integrity, the importance of being a team player, and the ability to lead and inspire others. Amin is a visionary and a great researcher. It has been an honor to be his student.

Special thanks to George Porter for being a great mentor, and for working closely with me on several projects. I enjoyed numerous thought provoking and insightful discussions with George while brainstorming on various ideas. He spurred me to approach problems systematically and rigorously to explore multiple alternatives while designing solutions.

I had the privilege of collaborating with and learning from George Varghese during my PhD. His immense enthusiasm and passion to discuss new ideas no matter how simple or complex, how mundane or creative, made me always look forward to the exciting conversations with him. He encouraged me to look at ideas with an unbiased mind, and look at the potentially big impact that simple ideas can have.

I'd like to thank the sysnet faculty Geoffrey Voelker, Stefan Savage, and Alex Snoeren who readily offered valuable advice and feedback on several of my projects. I have learned immensely from all of them, and appreciate the deep, thoughtful, and often hard questions that they brought up while brainstorming ideas. I am also thankful to

George Papen for serving on my committee and offering feedback on my dissertation.

I was fortunate to get to know several brilliant people at Google, and collaborate with them during my PhD. My co-authors on different works Yuchung Cheng, Jerry Chu, Arvind Jain, Barath Raghavan, and Abdul Kabbani have contributed immensely to my learning. Numerous insightful discussions with them led me to get a deeper understanding of several networking challenges, and better appreciate the importance of designing practical and deployable systems.

During the course of my PhD, I have been lucky to work closely with several fellow researchers, colleagues, and friends whose immense efforts helped materialize several ideas into reality. I will always cherish the research discussions and stimulating work days that I had with Malveeka Tewari, Rishi Kapoor, Vimalkumar Jeyakumar, Yilong Geng, Mohammad Al-Fares, Radhika Niranjana, Nathan Farrington, Nelson Huang, and Terry Lam.

I wish to thank Brian Kantor, and Cindy Moore for their help in setting up various systems, for keeping them up and running, and for being so accommodating through various deadlines. I would also like to thank the ever helpful UC San Diego academic and administrative staff, particularly Julie Conner, Nadyne Nawar, Jennifer Folkestad, and Kathy Krane.

Finally, I would like to thank my parents, for their unconditional support and encouragement through the highs and lows of my Ph.D. This would definitely not have been possible without them. This achievement is as much theirs as it is mine.

Chapter 3, in part, contains material as it appears in the Proceedings of the 7th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '11), Tokyo, Japan, December 2011. "TCP Fast Open". Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, contains material as it appears in the IETF Internet Drafts, March 2011 – February 2014. “TCP Fast Open”. Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain.

Chapter 4, in part, contains material as it appears in the Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13), San Jose, CA, October 2013. “Dahu: Commodity Switches for Direct Connect Data Center Networks”. Sivasankar Radhakrishnan, Malveeka Tewari, Rishi Kapoor, George Porter, and Amin Vahdat. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, contains material as it appears in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14), Seattle, WA, April 2014. “SENIC: Scalable NIC for End-Host Rate Limiting”. Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, contains material as it appears in the Proceedings of the 5th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '13), San Jose, CA, June 2013. “NicPic: Scalable and Accurate End-Host Rate Limiting”. Sivasankar Radhakrishnan, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. The dissertation author was the primary investigator and author of this paper.

## VITA

- 2008        B.Tech., Indian Institute of Technology Madras, India
- 2010        M.S., University of California, San Diego
- 2014        Ph.D., University of California, San Diego

## PUBLICATIONS

Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. “SENIC: Scalable NIC for End-Host Rate Limiting”. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI’14)*, Seattle, WA, April 2014.

Sivasankar Radhakrishnan, Malveeka Tewari, Rishi Kapoor, George Porter, and Amin Vahdat. “Dahu: Commodity Switches for Direct Connect Data Center Networks”. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS ’13)*, San Jose, CA, October 2013.

Sivasankar Radhakrishnan, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. “NicPic: Scalable and Accurate End-Host Rate Limiting”. In *Proceedings of the 5th USENIX Conference on Hot Topics in Cloud Computing (HotCloud ’13)*, San Jose, CA, June 2013.

Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. “NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers”. In *ACM SIGCOMM Computer Communication Review*, pp. 5-11, July 2012.

Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. “TCP Fast Open”. In *Proceedings of the 7th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT ’11)*, Tokyo, Japan, December 2011.

Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. “TCP Fast Open”. *IETF Internet Drafts*, March 2011 – February 2014.

Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. “Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers”. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM ’10)*, New Delhi,

India, August 2010.

Amin Vahdat, Mohammad Al-Fares, Nathan Farrington, Radhika Niranjana Mysore, George Porter, Sivasankar Radhakrishnan, “Scale-Out Networking in the Data Center”. *IEEE Micro*, pp. 29-41, July/August, 2010.

Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. “Hedera: Dynamic Flow Scheduling for Data Center Networks”. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, San Jose, CA, April 2010.

Radhika Niranjana Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. “PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric”. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '09)*, Barcelona, Spain, August 2009.

## ABSTRACT OF THE DISSERTATION

Network Performance Improvements For Web Services – An End-to-End View

by

Sivasankar Radhakrishnan

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Amin Vahdat, Chair

Modern web services are complex systems with several components that impose stringent performance requirements on the network. The networking subsystem in turn consists of several pieces, such as the wide area and data center networks, different devices, and protocols involved in a user's interaction with a web service. In this dissertation we take a holistic view of the network and improve efficiency and functionality across the stack. We identify three important networking challenges faced by web services in the wide area network, the data center network, and the host network stack, and present solutions.



First, web services are dominated by short TCP flows that terminate in as few as 2-3 round trips. Thus, an additional round trip for TCP’s connection handshake adds a significant latency overhead. We present TCP Fast Open, a transport protocol enhancement, that enables safe data exchange during TCP’s initial handshake, thereby reducing application network latency by a full round trip time. TCP Fast Open uses a security token to verify client IP address ownership, and mitigates the security considerations that arise from allowing data exchange during the handshake. TCP Fast Open is widely deployed and is available as part of the Linux Kernel since version 3.6.

Second, provisioning network bandwidth for hundreds of thousands of servers in the data center is expensive. Traditional shortest path based routing protocols are unable to effectively utilize the underlying topology’s capacity to maximize network utilization. We present Dahu, a commodity switch design targeted at data centers, that avoids congestion hot-spots by dynamically spreading traffic uniformly across links, and actively leverages non-shortest paths for traffic forwarding.

Third, scalable rate limiting is an important primitive for managing server network resources in the data center. Unfortunately, software-based rate limiting suffers from limited accuracy and high CPU overhead at high link speeds, whereas current NICs only support few tens of hardware rate limiters. We present SENIC, a NIC design that natively supports tens of thousands of rate limiters — 100x to 1000x the number available in NICs today — to meet the needs of network performance isolation and congestion control in data centers.

# Chapter 1

## Introduction

The Internet has seen immense growth over the last two decades and everyday, more users are coming online. In 2012, an estimated 2.7 billion people used the Internet [41]. People around the world are spending an increasing amount of time online. There are a number of online services that people use almost on a daily basis such as email, news, web search, online shopping, social networking, video streaming, entertainment, and media services. The world is interconnected, making the network a critical piece of worldwide infrastructure.

Web services are typically architected as multi-tiered applications, with many different components, a number of stages, and many layers in the network stack alone. They touch various aspects of networking, such as different kinds of networks involved in a user's interaction with a web service, different types of network devices, and protocols operating across them. The network stack itself involves several components, and in this dissertation, we will look at a few principles guiding the design and implementation of different components of the network stack.

With the rise of mobile devices, web services are accessed from a diverse variety of devices through diverse networks. The growth of cloud computing has led to many services that once used to reside on individual user's computers being moving to the cloud. For example, users today leverage online services such as Google Docs and

Microsoft Office 365 to access and edit documents that are stored in the cloud and are accessible from any device. These services also allow multiple users to collaboratively edit and share documents, further increasing the appeal over their offline counterparts. Internet users have been spending more and more time online, and web services have become an integral part of how people access and consume information today. Further, user expectations of web service performance has been increasing—web services are expected to be fast, responsive, and provide relevant and fresh information.

In this dissertation, we take a holistic view of the network and improve efficiency and functionality across the network stack. We identify three key challenges in networking touched by web services and present solutions.

## **1.1 Challenges**

If we look at how modern web services are used, Internet users connect to front-ends or servers located in data centers. Data centers have complicated networks of their own which have different characteristics from the Internet. With cloud based services and server consolidation, many web services might be hosted within a single data center. The data center network is shared, thereby necessitating performance isolation between services sharing the network. Modern web services are complex systems whose design and performance is deeply influenced by the network subsystem. In this section, we describe three key networking challenges that impact the performance of web services.

### **1.1.1 Wide Area Network Latency**

When a user accesses a web service by connecting to a server, Internet latency (or wide area latency) is a challenge. Most web pages are composed of small web objects that can be fetched in 2-3 round trips. With such short transfers, the web transfer latency is primarily determined by the round trip time (RTT), or latency between the user and

the server, and the number of round trips to complete the transfer. The web transfer latency has a strong impact on the responsiveness of the web service to the user, which in turn heavily influences the user experience [66]. With RTT largely being constrained by the speed of light and distance between the user and server (typical wide area RTTs being 10s to 100s of milliseconds), reducing network transfer latency mainly comes down to reducing the number of RTTs required to complete the transfer. Even a few 10s of milliseconds improvement in transfer latency causes a noticeable increase in user satisfaction [98, 99].

### **1.1.2 Data Center Network Infrastructure**

When a user request reaches the data center, it enters a network whose characteristics are quite different from the wide area network. The data center network has low latency (few 10s of microseconds) and high bandwidth (on the order of 10Gb/s or more between any pair of servers). The network topology is generally well structured unlike the Internet, and the network is managed by a single organization. Today, data centers house anywhere from tens of thousands to hundreds of thousands of servers. These servers communicate with each other not just to service the incoming user requests in real time, but also to run other tasks such as indexing the web, data analysis etc. All these tasks require a lot of bandwidth. In fact, the traffic within the data center far exceeds the user generated traffic [42]. Designing a network that can provide the necessary bandwidth at such a massive scale to hundreds of thousands of servers is expensive.

### **1.1.3 Co-tenancy and Network Performance Isolation**

Within the data center, a server is usually shared by many different applications. The network bandwidth on the server has to be shared by all these applications. With the growth of cloud computing and services such as Amazon EC2, Microsoft Azure, Google

Compute Engine, several web services are hosted by these cloud service providers. The use of shared infrastructure requires a means to provide isolation between different tenants or services. With the rise of virtualized infrastructures and consolidation of servers, there is an need to scale the mechanisms for providing isolation in the network. Current server Network Interface Cards (NICs) are only capable of isolating few tens of traffic classes [87], which is orders of magnitude less than the thousands of traffic classes required to provide network performance isolation in a modern virtualized data center shared by multiple tenants.

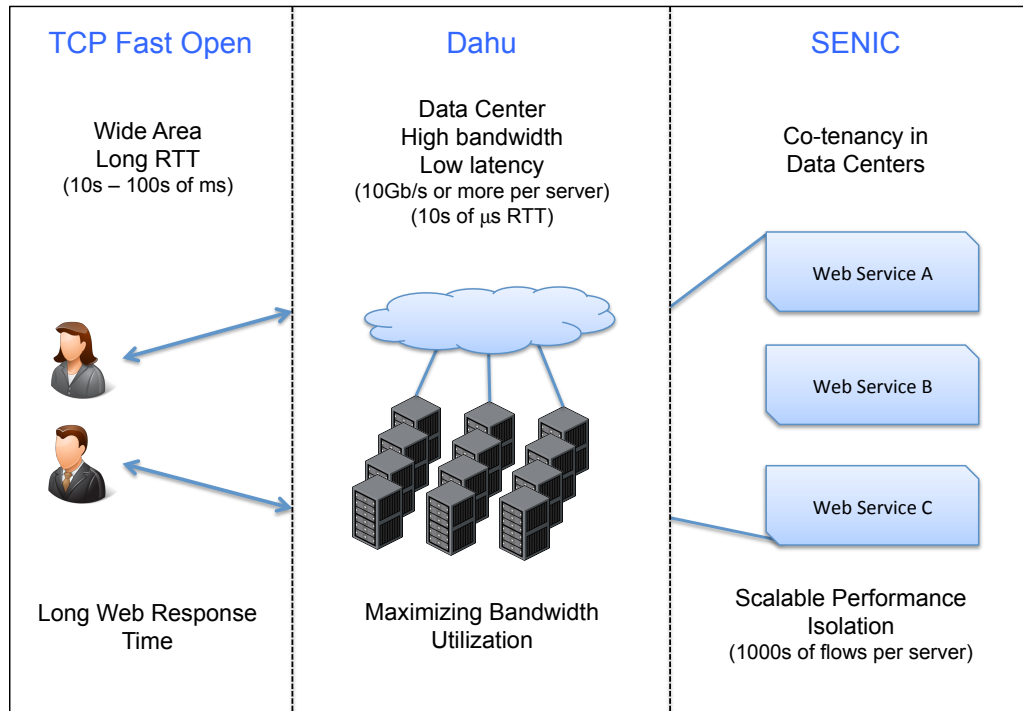
In summary, wide area network latency, efficient data center bandwidth utilization, and scalable mechanisms for network performance isolation in the data center are challenges that come in the way of designing high performance web services. We now look at our contributions towards addressing these challenges.

## **1.2 Efficient Networking for Modern Web Services**

In this dissertation, we take a holistic approach to improve efficiency or functionality across the board in different networking aspects that influence web services. TCP Fast Open addresses the wide area network latency between users and data centers and helps speed up short web transfers. Dahu looks at maximizing bandwidth utilization in the data center network. SENIC offers a scalable rate limiting substrate, that is required to provide network performance isolation when multiple services share the physical network. Figure 1.1 illustrates the key networking challenges presented in this chapter and the proposed solutions. We now present a brief overview of each.

### **1.2.1 TCP Fast Open**

A typical webpage is composed of several web objects (44 objects on average [76]), each of which is small. A vast majority of HTTP requests made by users result



**Figure 1.1.** Overview of key networking challenges addressed in this dissertation

in short transfers, that are just a few KB in size. TCP is the de facto protocol used for HTTP transfers. First, a TCP connection is established from the user’s computer to the server, second an HTTP request is sent to the server, and finally the server sends a HTTP response back to the client. HTTP responses have an average size of 7.3KB [76], and are usually transferred within 2-3 round trips to the client. The TCP connection setup phase requires one full round trip time (RTT), before the client can send the HTTP request to the server. This connection setup is a significant overhead for short web transfers.

TCP Fast Open is a transport layer mechanism that enables transfer of data during TCP’s connection setup phase. It eliminates the connection setup overhead of TCP, allowing data transfer to begin concurrently with TCP’s connection setup. This saves one RTT for each connection, and for short web transfers, this has a significant impact on the total response time. TCP Fast Open retains reliability, congestion control, and other features offered by traditional TCP, while eliminating the connection setup overhead.

### 1.2.2 Dahu

Traditionally, large scale high bandwidth networks have taken one of two design approaches. Supercomputers have built what are known as *direct network topologies* (e.g. torus, hypercube) [26]. They rely on proprietary protocols running in switches and NICs, along with application logic for making forwarding and routing decisions. These networks are resource efficient, and offer sufficient capacity for common communication patterns. Like most large scale networks, they offer multiple paths between servers to meet the high bandwidth requirements, however these paths are of differing lengths, and the routing scheme must fundamentally take advantage of these non-shortest paths to leverage the full capacity offered by the network.

Data centers on the other hand, have adopted *indirect networks*, and embraced commodity switches, with simple schemes for multipath routing such as Equal Cost Multi Path (ECMP) routing. The applications are decoupled from network forwarding, making the developer's task easier. Topologies such as folded-clos and fat trees that are deployed in data centers are great for worst case communication patterns. However for most common workloads, they are over-provisioned, and result in high capital expenditure (CAPEX). The limitations imposed by commodity switches and shortest path routing in data centers, along with the need to decouple application logic from the network have restricted data centers from adopting direct networks.

Dahu is a commodity switch design that bridges the benefits of the two approaches. Dahu uses non-shortest path routing without coupling application logic to the network topology. Dahu dynamically hashes flows onto the available paths in the network, thereby improving network utilization, and reducing infrastructure costs for data centers.

### 1.2.3 SENIC

Server consolidation helps data centers statistically multiplex resources among different applications or services. This necessitates careful resource scheduling, and network bandwidth is no exception. Several recent proposals for bandwidth management and congestion control in data centers rely on programmable rate limiters to enforce bandwidth limits, and achieve network performance isolation between services. These systems require thousands of rate limiters on each server to limit each flow from the server. However, today's NICs support only few tens of traffic classes, while current software based solutions for enforcing rate limits do not work at high link speeds.

SENIC is a NIC design that natively supports tens of thousands of traffic classes or queues in the NIC which can be individually rate limited. The key insight is to store packet queues in host memory for each traffic class, schedule packets from different classes, and pull the packets into the NIC on demand for scheduling. The late binding of packet transfers to the NIC allows SENIC to scale to support thousands of traffic classes, while still enforcing accurate hardware based rate limits at high link speeds.

## 1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of the principles that guide the design and implementation of networking systems. Chapter 3 presents TCP Fast Open, a transport layer solution to improve web service response time. Chapter 4 presents Dahu, a commodity switch design that enables the use of direct-connect topologies in data centers. Chapter 5 presents the design and implementation of SENIC, a NIC that supports tens of thousands of traffic classes natively in the NIC, to meet the requirements of network performance isolation in the data center. Finally, Chapter 6 concludes this dissertation with a summary of contributions.



# Chapter 2

## Design and Implementation Principles

This chapter presents three overarching network design and implementation guidelines or principles that are applicable to several systems and networking challenges. In this chapter, we identify and articulate these principles to consciously exploit them while designing network architectures and protocols. We'll see how these principles are applied to three solutions at different layers of the network stack, in different networks or portions of the network touched by modern web services.

### 2.1 Principles

We present three principles that govern the challenges and solutions proposed in this dissertation.

#### 2.1.1 State Management Principle

Network protocols and algorithms have state that they operate on. With networking problems, state is often distributed and the way it is distributed affects the processing costs for protocols and algorithms. State distribution also affects the kind of synchronization required among different network components and the overhead of achieving this synchronization. Network devices have limited memory, e.g. switches, routers and NICs have few KBs or MBs of memory for protocol state as well as packets. These factors

make state management an important consideration while designing networked systems.

The state management principle is as follows.

*P1. Distribute state to leverage network component strengths. Use hashing and implicit state in messages to achieve persistence.*

### **2.1.2 Short Circuiting Principle**

Latency is an important consideration while designing network protocols, and networking device pipelines. The number of stages in a pipeline or algorithm has an impact on end-to-end latency, reducing which is a goal or requirement for many networking designs. There are significant gains to be had from optimizing latency for the common case. For example, web search users wish to see results as soon as possible, and even small increases in latency have a negative impact on user experience. So search providers usually resort to returning results within a certain deadline even if they don't always have the best results. The short circuiting principle is as follows:

*P2. Discard avoidable stages in network protocols and stacks. Short-circuit for the optimal or common case.*

### **2.1.3 Inheritance Principle**

Network devices and protocols have evolved over time. Modern networks have a lot of legacy infrastructure or investments to deal with. Both hardware and software solutions to networking challenges have been developed and deployed over a long period of time. Networking proposals that require radical changes to existing infrastructure are hard to deploy, and often only see limited deployment in niche domains, or remain as theoretical exercises. Note that while it is important to question traditional designs and ideas, new solutions that aim for large scale adoption have a better chance of succeeding when they have incremental deployment stories. The key to quick adoption is to innovate on functionality and novelty of ideas, but have simple incremental deployment paths.

**Table 2.1.** Summary of implications of the principles

<b>Principle</b>	<b>TCP Fast Open</b>	<b>Dahu</b>	<b>SENIC</b>
State Management	Self certifying tokens, stateless servers	Flow hashing in switches	Store transmit queues in host RAM
Short Circuiting	Data transfer during TCP handshake	Default shortest path forwarding	Kernel or hypervisor bypass
Inheritance	Middlebox and TCP backwards compatibility	Simple changes to commodity switch silicon	Receive path unmodified

The inheritance principle is as follows:

*P3. Design backwards compatible and interoperable systems. Add new functionality with minimal change to existing designs.*

The ideas that see quick adoption are those that provide creative and innovative functionality or improvement in performance, while requiring very simple or incremental changes to prevailing deployments.

## 2.2 Implications of the Principles

We'll now see how these principles guided the design and goals of TCP Fast Open, Dahu, and SENIC. These systems are described in detail in the following chapters, but this section gives a high level overview of how the above principles apply to the designs presented in this dissertation. Table 2.1 presents a summary of the same.

### 2.2.1 State Management Principle

#### TCP Fast Open

Modern web services have multi-tiered architectures, and front-end servers at large scale services like Google or Microsoft can service hundreds of thousands to millions of requests per day. At this scale, any persistent state requirement on the server for each client can add up to have a large memory footprint. Traditional TCP only

maintains state in the server when a client connection is active; when the connection is closed, the server forgets about the connection and the client. If the server needs to maintain any state across multiple connections from a single client, the overall memory requirement of the server dramatically increases. TCP Fast Open is designed to avoid this by pushing state to clients. A typical web user connects much fewer servers in a day, than the number clients served by a popular server. In this way, distributing state to the clients rather than storing it on web servers helps TCP Fast Open to scale, by avoiding per-client state on servers when the respective clients are not connected to the server.

Most large scale services load balance user requests among a pool of servers in a server farm. Multiple connections from the same client to the server farm might be load balanced and serviced by different servers. Any client specific information on servers must be shared by all these servers to be able to service requests. Instead, the way TCP Fast Open addresses this challenge is by having the servers grant hash based authenticating tokens to clients. When a client connects to a server, this token presented by the client authenticates the client's IP address which is implicitly contained in the packet sent by the client. Server farms do not need run-time synchronization among the servers to distribute authentication information about client IP addresses at the transport layer in the network stack.

## **Dahu**

Network switches typically forward packets of any particular host connection (TCP or UDP flow) along persistent paths. This avoids frequent re-ordering of packets belonging to a flow, which can have a negative impact on application performance. Dahu switches use flow level hashing to forward packets along persistent paths. This avoids the need to store per-flow state in the switches corresponding to each flow, but at the same time ensures persistent paths for the flow since all packets of a particular flow have the same hash and end up taking the same path through the network.

## **SENIC**

NICs are highly memory constrained devices. They have on the order of a few MB of memory for packet buffers and any other state requirements [64]. The primary goal of SENIC is to scale the number of supported rate limited traffic classes or queues in the NIC to tens of thousands, compared to the few tens of classes supported today. Efficient state management and distribution is key to the SENIC design. SENIC stores packet queues in the co-located host memory (DRAM), and pulls packets into the NIC on demand for transmission. This avoids the need to store multiple packets and packet descriptors in the NIC for each traffic class, thereby enabling SENIC to scale to a large number of traffic classes. The late binding of packet DMA transfers to the NIC, and computing the final schedule before pulling in packets from host memory are the key design choices that result in a small memory requirement on the NIC to support thousands of traffic classes.

### **2.2.2 Short Circuiting Principle**

#### **TCP Fast Open**

The primary goal of TCP Fast Open is to short circuit the connection establishment phase of TCP connections. TCP Fast Open allows data to be exchanged while the connection handshake is still in progress, thereby reducing the time taken to transfer data to the peer. This is really useful for web services, since a vast majority of HTTP responses involve very short transfers that take only a few round trips [76]. Saving one round trip time (RTT) provides substantial savings in response time for these web services. TCP Fast Open does not completely eliminate the TCP connection handshake, instead it allows the data transfer to begin concurrently with the connection handshake, thereby reducing latency.

There are some scenarios, such as with non-standards compliant middleboxes,

that can prevent TCP Fast Open's short-circuited connection handshake from successfully completing. However, experimental data from the Internet indicates that this occurs only for a small subset of connections [57]. In such cases, TCP Fast Open falls back to a regular TCP 3-way handshake, and the connection just proceeds as in traditional TCP. Thus, TCP Fast Open short-circuits the TCP connection for the common case where most Internet users can successfully connect to web servers using TCP Fast Open's concurrent handshake and data transfer.

### **Dahu**

Dahu is a commodity switch design for data center networks that enables the use of direct-connect topologies. The key feature that Dahu offers is the ability to use non-shortest path routing on demand. Direct connect topologies provide sufficient bandwidth for most common communication patterns at lower cost than tree-based topologies, but this is contingent upon the use of paths of different lengths between servers. As in current multi-rooted tree based data center topologies, direct connect networks have multiple paths between any pair of servers; however, unlike current data center topologies, these paths are not all of the same length. The topology fundamentally requires the use of non-shortest paths to leverage the full capacity offered by the network. Dahu switches short circuit network traffic to take shortest paths by default, and enable longer paths on demand, as the network load on shortest paths increases.

### **SENIC**

SENIC is a NIC design that offers tens of thousands of rate limited traffic classes natively in the NIC. SENIC supports kernel or hypervisor bypass on the transmit datapath. This enables guest VMs in a virtualized environment, or even user applications in a non-virtualized environment, to directly send packets to the NIC without having to go through the hypervisor. Short circuiting the kernel reduces the latency in the host networking stack for transmitting a packet [61]. SENIC offers mechanisms for rate limiting or

weighted sharing of bandwidth between traffic classes natively in the NIC hardware. This allows bypassing the hypervisor, while still ensuring performance isolation among traffic classes.

In a malicious environment, guest VMs may try to take advantage of hypervisor bypass, and falsely classify packets into queues that have larger rate limits. SENIC detects this through packet sampling and disables hypervisor bypass for these guest VMs, forcing these guest VMs to send their traffic through the hypervisor. The common case, where the guest VMs are non-malicious, can still take advantage of hypervisor bypass even when it is disabled for some malicious guest VMs.

### **2.2.3 Inheritance Principle**

#### **TCP Fast Open**

TCP Fast Open is designed to be compatible with NATs and other middleboxes in the Internet. This was a specific design goal of TCP Fast Open so that it could be used by as many Internet users as possible without any disruption to their regular network traffic or any restrictions in terms of the networks where TCP Fast Open can be deployed. TCP Fast Open is completely backwards compatible with traditional TCP endpoints. This ensures TCP Fast Open can be deployed incrementally for different services, and users can start using TCP Fast Open as and when services start supporting TCP Fast Open. Further, the API for using TCP Fast Open is very simple—we use existing system calls with new flags that enable TCP Fast Open functionality. This makes it easier for programmers to take advantage of TCP Fast Open and also be assured that it would work just like traditional TCP if the peer does not support TCP Fast Open.

#### **Dahu**

Dahu proposes a small set of simple hardware enhancements to existing commodity data center Ethernet switches. The changes are confined to switches, which keeps

end-hosts simple and unmodified. The changes to the switch silicon are indirection layers that allow better control over how flows are routed through the network, while providing support for on-demand non-shortest path routing.

## **SENIC**

The SENIC design involves a simple redistribution of host and NIC responsibilities. The receive path is unmodified from current NICs. SENIC exposes the same interface to the operating system as exposed by today's NICs, except that it supports a lot more traffic classes or queues. The interface for enqueueing packets to the NIC on the transmit path, and dequeueing packets on the receive path are the same as they are with traditional NICs. Thus SENIC can be used as a plug-in replacement for any traditional NIC, but the operating system can easily take advantage of many more traffic classes available natively in the NIC.

## **2.3 Summary**

In this chapter, we looked at three overarching network design and implementation principles that apply to several challenges in systems and networking. The State Management Principle argues for distributing state between network components to take advantage of their strengths. The Short Circuiting Principle proposes eliminating or bypassing avoidable stages in network protocols and stacks to improve performance for the common case. The Inheritance Principle advocates the importance of innovating on functionality and improving performance, while maintaining backwards compatibility and having an incremental deployment plan. The following chapters present an in-depth description of the design and implementation of TCP Fast Open, Dahu, and SENIC, further elucidating how these principles are applied.



# Chapter 3

## TCP Fast Open

Today's web services are dominated by TCP flows so short that they terminate a few round trips after handshaking; this handshake is a significant source of latency for such flows. In this chapter, we describe the design, implementation, and deployment of the TCP Fast Open protocol, a new mechanism that enables data exchange during TCP's initial handshake. In doing so, TCP Fast Open decreases application network latency by one full round-trip time, decreasing the delay experienced by such short TCP transfers.

We address the security issues inherent in allowing data exchange during the three-way handshake, which we mitigate using a security token that verifies IP address ownership. We detail other fall-back defense mechanisms and address issues we faced with middleboxes, backwards compatibility for existing network stacks, and incremental deployment. Based on traffic analysis and network emulation, we show that TCP Fast Open would decrease HTTP transaction network latency by 15% and whole-page load time over 10% on average, and in some cases up to 40%.

### 3.1 Introduction

While web pages have grown significantly in recent years, network protocols have not scaled with them. Today's pages are on average over 300KB each, but most web objects are relatively small, with mean and median sizes of 7.3KB and 2.4KB

respectively [76]. As a result of the preponderance of small objects in large pages, web transfer latency has come to be dominated by both the round-trip time (RTT) between the client and server and the number of round trips required to transfer application data. The RTT of a web flow largely comprises two components: transmission delay and propagation delay. Though network bandwidth has grown substantially over the past two decades thereby significantly reducing transmission delays, propagation delay is largely constrained by the speed of light and therefore has remained unchanged. Thus reducing the number of round trips required for the transfer of a web object is the most effective way to improve the latency of web applications [12, 30, 91, 95].

Today's TCP standard permits data exchange only after the client and server perform a handshake to establish a connection. This introduces one RTT of delay for each connection. For short transfers such as those common today on the web, this additional RTT is a significant portion of the flows' network latency [93]. One solution to this problem is to reuse connections for later requests (e.g. HTTP persistent connections [59]). This approach, while widely used, has limited utility. For example, the Chrome browser keeps idle HTTP 1.1 TCP connections open for several minutes to take advantage of persistent connections; despite this over one third of the HTTP requests it makes use new TCP connections. A recent study on a large CDN showed that on average only 2.4 HTTP requests were made per TCP connection [2]. This is due to several reasons as we describe in Section 3.2.

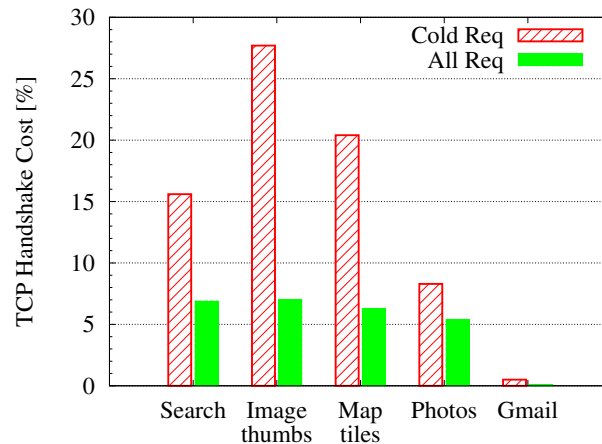
We find that the performance penalty incurred by a web flow due to its TCP handshake is between 10% and 30% of the latency to serve the HTTP request, as we show in detail in Section 3.2. To reduce or eliminate this cost, a simple solution is to exchange data during TCP's initial handshake (e.g. an HTTP GET request / response in SYN packets). However, a straightforward implementation of this idea is vulnerable to denial-of-service (DoS) attacks and may face difficulties with duplicate or stale SYNs.

To avoid these issues, several TCP mechanisms have been proposed to allow data to be included in the initial handshake; however, these mechanisms were designed with different goals in mind, and none enjoy wide deployment due to a variety of compatibility and/or security issues [9, 10, 20].

In this chapter we propose a new TCP mechanism called TCP Fast Open (TFO) that enables data to be exchanged safely during TCP's initial handshake. At the core of TFO is a security cookie that is used by the server to authenticate a client that is initiating a TFO connection. We describe the details of TFO, including how it exchanges data during the handshake, the protocol used for TFO cookies, and socket API extensions to enable TFO. In addition, we analyze the security of TFO and examine both the potential for new security vulnerabilities and their mitigation. We also describe our implementation of TFO in the Linux kernel and in the Chrome web browser and present the performance gains we see in our test-bed experiments. Finally we examine deployment issues and related approaches.

## **3.2 Motivation**

Latency and page load time are important factors that influence user satisfaction with a website. Even small improvements in latency lead to noticeable increases in site visits and user satisfaction, and result in higher revenues [66, 98, 99]. While it is well known that small objects dominate web flows today, we sought to better understand the actual performance characteristics of today's flows and the performance bottlenecks they experience. To do so, we analyzed both Google web server logs and Chrome browser statistics to demonstrate that TCP's handshake is a key performance bottleneck for modern web transfers. Our intent is to highlight this practical problem through the analysis of large scale data and to estimate the potential benefits of TFO.



**Figure 3.1.** TCP handshake time as a percentage of total HTTP request latency for Google.com. For the “All Req” category, handshake time is amortized over all HTTP requests for the Google service in question.

### 3.2.1 Google Server Logs Analysis

We begin by analyzing latency data from Google web server logs to study the impact of TCP’s handshake on user-perceived HTTP request latency. We sampled a few billion HTTP requests (on port 80) to Google servers world-wide over 7 consecutive days in June 2011. These included requests to multiple Google services such as search, email, and photos. For each sampled request, we measured the latency from when the first byte of the request is received by the server to when the entire response is acknowledged. If the request is the first one of the TCP connection, this latency also includes the TCP handshake time since the browser needs to wait for the handshake to complete before sending the request. Note that our latency calculation includes both server processing time and network transfer time.

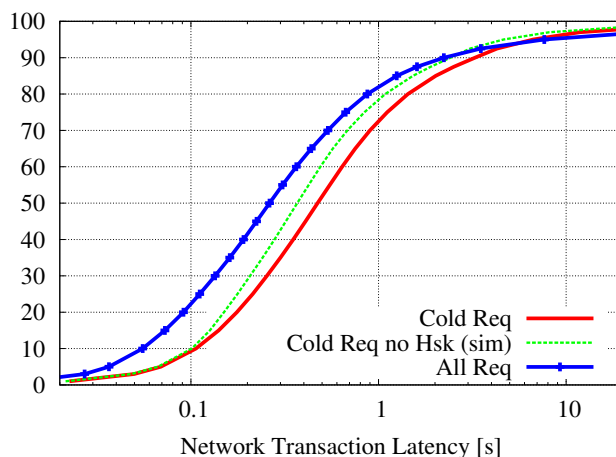
We define requests sent on new TCP connections as *cold requests* and those that reuse TCP connections as *warm requests*. We segregate requests by service and compute the fraction of time spent on TCP handshakes for cold requests. Similarly, we compute the amortized cost of TCP handshakes over both cold and warm requests for

each service. The results shown in Figure 3.1 indicate that TCP handshakes account for 8% to 28% of the latency of cold requests for most services. Even the amortized cost for handshakes accounts for 5-7% of latency across both cold and warm requests, including photo services where the average response size is hundreds of kilobytes. (The only exception is Gmail because it downloads javascript upon a cold request and reuses the same connection for many subsequent warm requests.)

The cost of TCP handshakes is surprisingly high given that 92% of the requests that we see use HTTP/1.1 which supports persistent HTTP connections. Moreover, Google web servers keep idle connections open for several minutes. In theory, most requests should reuse existing TCP connections to minimize the penalty of a TCP handshake, but our analysis indicates this may not be happening. In order to understand if this problem persists for other web sites and what its cause(s) might be, next we analyze statistics from the Chrome web browser.

### **3.2.2 Chrome Browser Statistics**

We processed Chrome browser statistics for 28 consecutive days in 2011; these only cover Chrome users who have opted into statistics collection and only contain anonymized data such as latency statistics. The statistics do however cover requests to all websites and not just Google services. Across billions of sampled HTTP latency records, we found that over 33% of requests made by Chrome are sent on newly created TCP connections even though it uses HTTP 1.1 persistent connections. The restricted effectiveness of persistent connections is due to several factors. Browsers today often open tens of parallel connections to accelerate page downloads, which limits connection reuse. Domain sharding or the placement of resources on different domains by content providers to increase parallelism in loading web pages also exacerbates this issue. In general, hosts and middle-boxes (NATs) also terminate idle TCP connections to minimize



**Figure 3.2.** CDF of the HTTP transaction network latency for Chrome Windows users. The Y-axis is the cumulative distribution of HTTP requests in percentiles. “Cold Req” and “Cold Req no Hsk (sim)” refer to requests that need to open new TCP connections, but the latter excludes TCP connect time. “All Req” refers to all requests, including both HTTP and HTTPS.

resource usage. The middle-box issue may be partly mitigated by using TCP keepalive probes, but this could be prohibitively power hungry on mobile devices [94]. Major mobile browsers close idle connections after mere seconds to conserve power.

To understand the latency impact of waiting for TCP’s handshake to complete before transferring data, we plot the distribution of HTTP transaction network latency for cold requests and all requests in Figure 3.2. We measured network transaction latency from the time the browser schedules a request to the time it receives the entire response. If the browser does not have an idle TCP connection available to serve the request, it attempts to open a TCP connection. Thus TCP’s handshake time is included in the network latency. Chrome also has a limit of 6 parallel connections per domain.

Figure 3.2 reveals that cold requests are often over 50% slower when compared to all requests in the same percentile. For example, the median latencies of cold requests and all requests are 549ms and 308ms, respectively. Many factors including DNS lookup, TCP slow-start, SSL handshake, and TCP handshake, may contribute to this slowdown.

To isolate the cost of the TCP handshake, we plot network transaction latencies of cold requests excluding TCP handshake time.<sup>1</sup> This simulated distribution, labeled as “Cold Req no Hsk” in the figure, suggests that TCP handshake accounts for up to 25% of the latency between the 10th and 90th percentiles.

Thus the results of our analysis of both Google server logs and Chrome browser statistics suggest that sending an HTTP request and response during a TCP handshake can significantly improve HTTP transaction performance.<sup>2</sup>

### 3.3 Design

Our measurement results support the notion that eliminating one round trip from a web flow can provide immediate, measurable performance gains. However, it may be instructive to first consider the constraints we designed within and the assumptions we made while working on TCP Fast Open.

#### 3.3.1 Context and Assumptions

The current TCP specification actually allows a client to include data in its SYN packet when initiating connections to servers, but forbids the servers from delivering the data to applications until the 3-way handshake (3WHS) completes [80]. Suppose for the moment that we were to remove this restriction, and simply enable ordinary TCP-based client applications to send HTTP GET requests in TCP SYN packets and servers to respond with data in their TCP SYN-ACK packets. While this would trivially meet the needs of TCP Fast Open, it would open the protocol up to a straightforward denial-of-service attack of both the server and arbitrary hosts: an attacker or set of attackers

---

<sup>1</sup>We measure TCP handshake time by the time it takes to finish the `connect()` system call in Chrome.

<sup>2</sup>We note that our estimates from Google server logs (which concern only requests for `google.com`) and Chrome browser statistics (which are across the web) differ likely because Google has a lower RTT and processing time than many other websites.

could send HTTP GET requests to a server while spoofing the source address of a victim host, thereby causing the server both to perform potentially expensive request processing and to send a potentially large response to a victim host. Thus we must build security mechanisms into TFO to protect both the server and other hosts from such attacks.

Our goal in designing TCP Fast Open was to enable each end of a TCP connection to safely transmit and process any received data while the 3WHS is still in progress. However, there are several other constraints that we kept in mind and assumptions that we were forced to make. For example, the TCP initial handshake is designed to deal with delayed or duplicate SYN packets received by a server and to prevent such packets from creating unnecessary new connections on the server; server applications are notified of new connections only when the first ACK is received from the client. We found that to manage stale or duplicate SYN packets would add significant complexity to our design, and thus we decided to accept old SYN packets with data in some rare cases; this decision restricts the use of TFO to applications that are tolerant to duplicate connection / data requests. Since a wide variety of applications can tolerate duplicate SYN packets with data (e.g. those that are idempotent or perform query-style transactions), we believe this constitutes an appropriate trade-off.

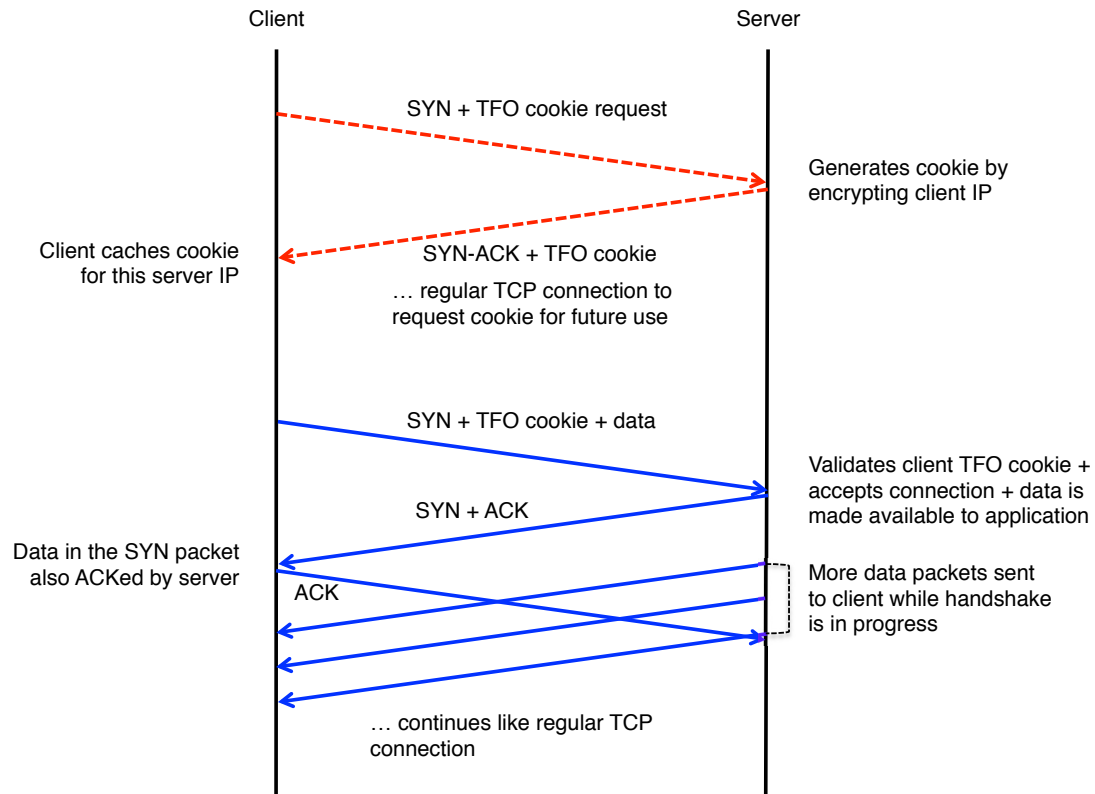
Similarly, we make several assumptions about the setting in which TFO is deployed. We assume that servers cannot maintain permanent or semi-permanent per-client state since this may require too much server memory, and that servers may be behind load balancers or other such network devices. A stateless-server design is more desirable in this setting as it keeps state-management complexity to a minimum.

We also assume that servers cannot perform any operations to support TFO that are not reasonable to implement on the kernel's critical path (e.g. symmetric cryptography is possible, but asymmetric is not). We assume that clients are willing to install new software to support TFO and that small changes to applications are acceptable. Finally,



we assume that it is acceptable to leverage other security mechanisms within a server’s domain (if needed) in concert with TFO to provide the required security guarantees.

### 3.3.2 Design Overview



**Figure 3.3.** TCP Fast Open connection overview

Our primary goal in the design of TCP Fast Open is to prevent the source-address spoofing attack mentioned above. To prevent this attack, we use a security “cookie”. A client that wishes to use TFO requests a cookie—an opaque bytestring—from the server in a regular TCP connection with the TCP Fast Open TCP option included, and uses that cookie to perform fast open in subsequent connections to the same server. Figure 3.3 shows the usage of TFO. We begin by listing the steps a client performs to request a TFO cookie from a server:

1. The client sends a SYN packet to the server with a Fast Open Cookie Request TCP option.
2. The server generates a cookie by encrypting the client's IP address under a secret key. The server responds to the client with a SYN-ACK that includes the generated Fast Open Cookie in a TCP option field.
3. The client caches the cookie for future TFO connections to the same server IP.

To use the fast open cookie that it received from a server, the client performs the following steps:

1. The client sends a SYN with the cached Fast Open cookie (as a TCP option) along with application data.
2. The server validates the cookie by decrypting it and comparing the IP address or by re-encrypting the IP address and comparing against the received cookie.
  - (a) If the cookie is valid, the server sends a SYN-ACK that acknowledges the SYN and the data. The data is delivered to the server application.
  - (b) Otherwise, the server drops the data, and sends a SYN-ACK that only acknowledges the SYN sequence number. The connection proceeds through a regular 3WHS.
3. If the data in the SYN packet was accepted, the server may transmit additional response data segments to the client before receiving the first ACK from the client.
4. The client sends an ACK acknowledging the server SYN. If the client's data was not acknowledged, it is retransmitted with the ACK.
5. The connection then proceeds like a normal TCP connection.

### 3.3.3 Cookie Design

The TFO cookie is an encrypted data string that is used to validate the IP ownership of the client. The server is responsible for generation and validation of TFO cookies. The client or the active-open end of a connection simply caches TFO cookies and returns these cookies to the server on subsequent connection initiations. The server encrypts the source IP address of the SYN packet sent by the client and generates a cookie of length up to 16 bytes. The encryption and decryption / validation operations are fast, comparable to the regular processing time of SYN or SYN-ACK packets.

Without the secret key used by the server upon cookie generation to encrypt the client's IP address, the client cannot generate a valid cookie. If the client were able to generate a valid cookie this would constitute a break of the underlying block cipher used for encryption. The server periodically revokes cookies it granted earlier by rotating the secret key used to generate them. This key rotation prevents malicious parties from harvesting many cookies over time for use in a coordinated attack on the server. Also, since client IP addresses may change periodically (e.g. if the client uses DHCP), revoking cookies granted earlier prevents a client from mounting an attack in which it changes its IP address but continues to spoof its old IP address in order to flood the new host that has that old address.

### 3.3.4 Security Considerations

TFO's goal is to allow data exchange during TCP's initial handshake while avoiding any new security vulnerabilities. Next we describe the main security issues that arise with TFO and how we mitigate them.

## **SYN Flood / Server Resource Exhaustion**

If the server were to always allow data in the SYN packet without any form of authentication or other defense mechanisms, an attacker could flood the server with spurious requests and force the server to spend CPU cycles processing these packets. Such an attack is typically aimed at forcing service failure due to server overload.

As noted earlier, TFO cookie validation is a simple operation that adds very little overhead on modern processors. If the cookies presented by the attacker are invalid, the data in the SYN packets is not accepted. Such connections fall back on regular TCP 3WHS and thus the server can be defended by existing techniques such as SYN cookies [31].

If the cookies that the attacker presents are valid—and note that any client can get a cookie from the server—then the server is vulnerable to resource exhaustion since the connections are accepted, and could consume significant CPU and memory resources on the server once the application is notified by the network stack. Thus it is crucial to restrict such damage.

To this end, we leverage a second mechanism: the server maintains a counter of total pending TFO connection requests either on a per service port basis or for the server as a whole. This counter represents TFO connections that have been accepted by the server but that have not been migrated to the fully-established TCP state, which occurs only after receiving the first ACK from the peer (completion of 3WHS). When the number of pending TFO connections exceeds a certain threshold (that is administratively set), the server temporarily disables TFO and any incoming TFO requests fall back on regular 3WHS. This allows the usual SYN flood defense techniques [31] to prevent further damage until the pending TFO requests falls below the threshold. This limit makes it possible for an attacker to overflow the limit and temporarily disable TFO on

the server, but we believe that this is unlikely to be of interest to an attacker since this would only disable the TFO “fast path” while leaving the service intact.

There is another subtle but important difference between TFO and a regular TCP handshake. When SYN flood attacks originally broke out in the late 1990s, they were aimed at overflowing the short SYN backlog queues on servers that were used to store information about incoming connection requests until the completion of 3WHS. In such an attack, the attacker sends a stream of SYN packets with spoofed source IP addresses until this SYN queue fills up. This causes new SYN packets to be dropped, resulting in service disruption. The attacker typically uses spoofed source IP addresses that are non-responsive; otherwise, the SYN-ACK would trigger a TCP RST from the host whose IP has been spoofed. The TCP RST packet would terminate the connection request on the server and free up the SYN queue, thereby defeating the attack.

With TFO, such RST packets from spoofed hosts would fuel the damage since the RST will terminate the pending TFO request on the server, allowing another spoofed TFO connection request to be accepted. To defend against this, any pending TFO connections that get terminated by a RST should continue to be counted against the threshold for pending TFO connections described previously until a timeout period has passed.

### **Amplified Reflection Attack**

While regular TCP restricts the response from a server to just one SYN-ACK packet, TFO allows the server to send a stream of data packets following the SYN-ACK to the source IP address of the SYN packet. If not for the TFO cookie, this could be used by an attacker to mount an amplified reflection attack against any victim of choice.

As mentioned in the previous section, the number of pending TFO connections on the server has a system limit, so the server is protected from resource exhaustion beyond the limit. This system limit also bounds the damage that an attacker can cause through

reflection from that server. However, the attacker can still create a reflection attack from a large number of servers to a single victim host as follows.

First, the attacker has to steal or otherwise obtain a valid cookie from the target victim. This likely requires the attacker to compromise or collude with the victim host or network. If the victim host is already compromised, the attacker would likely have little value in mounting a reflection attack against the host itself, but the attacker might still be interested in mounting the attack to disrupt the compromised host's network. The stolen cookie is valid for a single server, so the attacker has to steal valid cookies from a large number of servers and use them before they expire to cause a noticeable impact on the compromised host's network.

We argue that if the attacker has already compromised the target network or host, then the attacker could directly start flooding the network from the compromised host without the use of a reflection attack. If servers still want to mitigate such an attack, one possible defense is to wait for the 3WHS to complete before sending data to the client. The server would still accept data in the SYN packet and allow the application to process it, but would make sure it is a valid connection—that is, 3WHS completes before sending the response to the client. For many applications this modification would yield little slowdown versus standard TFO as server processing time is often greater than the RTT of the connection.

### **3.3.5 Handling Duplicate SYN Segments**

The current TCP standard allows SYNs to carry data but forbids delivering the data to the application until the connection handshake is completed in order to handle duplicate SYNs. Although TFO does not retransmit SYNs with data, it's possible that the network duplicates SYN packets with data, causing the data to be replayed at the server. Suppose a TFO client sends a SYN with data and actively closes the connection before

the duplicate SYN arrives at the server. The server, being passively closed, does not retain any state about the closed connection, so accepts the duplicate SYN and processes (replays) the data. If the duplicate is generated within a 2MSL timeout, the server is likely to terminate the connection after receiving an RST since the client would process the server's SYN/ACK in the TIME\_WAIT state. Nevertheless, the request will have been replayed.

One heuristic to address this is to extend the LAST\_ACK state for 2MSL duration at the server (the passive close side) after receiving the final ACK from the client. This prevents some delayed duplicate packets from reaching the server application. Applications that are particularly intolerant to duplicate transactions, such as credit card transactions or banking applications, already have application-level measures to ensure idempotence. Alternatively, they can use a nonce to ensure that a transaction occurs only once. This challenge already exists today in another form: users often click refresh in web browsers if a page does not load quickly, resulting in duplicate transactions.

### 3.3.6 API Changes

One of our design goals was to avoid changes to socket libraries and to reuse existing APIs as much as possible. This minimizes changes to applications that wish to use TFO and poses less of a deployment hurdle.

The server side API to use TFO is extremely simple. A server application enables TFO for incoming connections to a listening socket simply by enabling a new socket option. Figure 3.4 shows sample code to enable TFO on the server. The remaining socket calls (i.e. `listen()`, `accept()`, `send()`, `recv()`, etc.) remain unchanged.

On the client side, using TFO requires the application to provide a destination IP address and port number, as well as the data to send. The `sendto()` and `sendmsg()` system calls already provide such an interface; we extend them for use with TFO. When

```
sd = socket(...);
bind(sd, ...);

int tfo_opt = 1;
setsockopt(sd, SOL_TCP, TCP_LISTEN_TFO,
           (void*)&tfo_opt, 4);

listen(sd, ...);
```

**Figure 3.4.** Server application sample code

these system calls are used on a regular TCP socket, the destination address is ignored and they behave just like a `send()` call. When the new TFO flag is set, these calls are modified to initiate a TFO connection.

If the TFO cookie for the destination IP address is available, a SYN packet with the cookie and data is sent to initiate the TFO connection; the network stack handles this decision without the application's intervention. If the cookie is not available, it falls back on a regular TCP three-way handshake and the data is queued up for transmission when the 3WHS is completed. The SYN packet in this case also includes a TCP option requesting a TFO cookie from the server for later use. In general, the use and handling of TFO cookies is done by the networking stack and is completely transparent to the application. Therefore no API is needed to expose them. After the first `sendmsg()` or `sendto()` call, the rest of the socket calls from the application are unmodified.

The modified `sendto()` and `sendmsg()` calls return the number of bytes of data queued up in the kernel or sent in the SYN packet. They can be used with blocking or non-blocking sockets and their return values upon error are a combination of the error messages returned by `send()` and `connect()`. Figure 3.5 shows sample code that a client application can use to connect to a server using TFO.

Besides these, one could imagine additional less critical APIs that could be provided to expose TFO information related to the connection, such as whether a connection



```
sd = socket(...);  
  
char msg[16] = "Hello TFO World";  
  
send_len = sendto(sd, msg, 15, MSG_TFO,  
                 server_addr, addr_len);
```

**Figure 3.5.** Client application sample code. The `sendto()` call with `MSG_TFO` flag combines `connect()` and `send()` functionalities.

was opened through a regular handshake or TFO, and whether a TFO attempt to a server succeeded; APIs to set TFO secret keys and flush the TFO cache might also prove useful.

The TFO cookie handling is transparent to applications and the cookies received by a client from a server are not directly readable by applications unless they have root privileges to sniff packets on the client. This prevents malicious sites from using simple browser hacks to trick users by making connections to other websites and stealing those TFO cookies for mounting an amplified reflection attack.

## 3.4 Deployability

Given that the main goal of TFO is improving the performance of short transfers such as the retrieval of web objects, being compatible with today's network architecture is key. Thus we designed TFO for incremental deployment. In doing so, we enabled it to gracefully fall back on standard TCP to ensure that current and future TCP connections can proceed in response to unexpected network events. In this section, we discuss the challenges of incremental deployability and our responses to these challenges.

### 3.4.1 New TCP Options / Data in SYN

Our primary deployment concern with TFO is regarding how Internet routers, middle-boxes, end-hosts, and other entities will handle new types of TCP packets such as those with new TCP options, SYN packets with data, and the like, as such packets are

unusual in today's networks. One study found that some middle-boxes and hosts drop packets with unknown TCP options [57]. A more recent study found that 0.015% of the top 10,000 websites do not respond with a SYN-ACK after receiving a SYN with a non-standard or new TCP option [19]. Another study found that 6% of probed paths on the Internet drop SYN packets containing data [52].

If a SYN packet with TCP Fast Open option set does not elicit a response within the timeout period (regardless of where it is dropped), we simply retransmit the SYN without any data or non-standard TCP options. In doing so, TFO falls back on a regular TCP 3WHS and connectivity with the server is not lost. The client also caches the RTT to the server in its cookie cache and sets the SYN retransmit timeout to  $1.5 \times RTT$ , thereby reducing the ordinarily longer SYN timeout. If TFO fails repeatedly to a given server, the client remembers the server's IP address and disables TFO for that server in the future.

### 3.4.2 Server Farms

Given that many large web services place servers in server farms, another point of concern for us is how TFO would be used at such data centers. A common setup for server farms is for many servers to be behind a load balancer, sharing the same server IP. Client TCP connections are load balanced to different physical hosts, often without any stored state about previous connections from the same client IP. Clients cache the TFO cookie based on a server's IP; TFO connections from a particular client might be load balanced to a physical server different from the one that granted the TFO cookie. We use TFO in this setting by sharing a common secret key (used for encrypting and decrypting TFO cookies) among all the servers in the server farm. Secret key updates to the servers are made at about the same time on all the servers.

### 3.4.3 Network Address Translation (NAT)

Network Address Translation (NAT) is another challenge for TFO. Hosts behind a single NAT sharing the same public IP address are granted the same cookie for the same server; nevertheless, the clients can all still use TFO. However some carrier-grade NAT configurations use different public IP addresses for new TCP connections from the same client. In such cases, the TFO cookies cached by the client would not be valid and the server would fall back on a regular 3WHS and reject any data in the SYN packet. Despite this, since the server would reply with an ordinary SYN-ACK, the use of TFO in this scenario would not cause any latency penalty versus an ordinary TCP connection.

### 3.4.4 TCP Option Space

The availability of TCP option space in the SYN and SYN-ACK packets is an issue since many options are negotiated in these packets. We analyzed the connections seen at Google's web servers and found that over 99% of incoming client connections had sufficient option space to accommodate a TFO cookie option with an 8 or 16 byte cookie in the SYN packets. Therefore, this is unlikely to be a concern for web traffic.

If other types of traffic use certain long TCP options (e.g. the TCP MD5 option), and space is insufficient in the TCP options field to accommodate the TFO cookie, the connection can safely fall back on regular 3WHS.

## 3.5 Implementation

We implemented TCP Fast Open in the Linux 2.6.34 kernel and are in the early stages of deploying it across Google. The entire kernel patch is about 2000 lines of code with about 400 lines used for the client-side TFO cookie cache. We also coordinated with the developers of Chrome to implement TFO support within the browser.

### 3.5.1 Kernel Support

While the Linux TCP stack required fairly deep modification, a key aspect to both the design and implementation of TFO is that it *does not affect* TCP congestion control. That is, since congestion control only takes place after TCP's handshake completes, and TFO is only in use during the handshake, the two are entirely separate. Thus we did not have to modify any code relating to congestion control in the Linux kernel. Also note that the maximum number of data segments that a server can send before getting acknowledgments from the client is dictated by the initial congestion window and receiver window, but that neither of these values are affected by TFO. Our modifications included alterations to incoming packet handling in the LISTEN, SYN\_SENT, and SYN\_RCVD states and to the routines that transmit TCP packets (to include appropriate options as required for TFO).

Our implementation uses a fixed size, 8 byte TFO cookie. We use the 128-bit (16 byte) AES block cipher implementation available in the Linux Kernel CryptoAPI to encrypt each client IP value; we truncate the result to 8 bytes to generate the cookie. We pad IPv4 client IP addresses with zeros to create a 16 byte IP value while IPv6 addresses are used in full. To validate the cookie contained within an incoming TFO request, the server recomputes the 8 byte cookie value based upon the incoming source IP address and compares it to the cookie included by the client. The cookie generation and validation operations add cryptographic processing overhead on the server. Many modern processors include AES instructions in hardware and a single CPU core can support tens of thousands of 16-byte AES encryptions per second [43]. This is greater than the connection acceptance rate of many modern servers, and the processing overhead for this cryptographic operation is only a small fraction of typical connection processing time.

For the cookie cache—which is used by client hosts' network stacks—we imple-

mented a simple LRU policy that caches cookies, RTT, and MSS by server IP. While we found that this policy worked well, this cache replacement policy is not in any way tied to the protocol.

### 3.5.2 Application Support

Only small changes are required in user level applications. Server side applications need just a single additional line of code: a call to `setsockopt()` to set the TFO socket option for the listen socket. Client side applications must replace `connect()` and the first `send()` call with a single call to `sendto()` with the appropriate flags. In addition to using TCP Fast Open within our own custom socket programs, the Chrome web browser was also modified to use TFO, as was the web server with which we performed tests.

## 3.6 Evaluation

In this section we evaluate the performance improvement conferred by TCP Fast Open in two contexts. First, we measure the whole-page download gains seen by a TFO-enabled Chrome browser visiting popular websites. Second, we measure the more surprising performance benefits of TFO on the server side.

### 3.6.1 Whole Page Download Performance

The primary goal of TFO is to eliminate one RTT of latency, thereby improving the performance of short flows. This is particularly important for cold HTTP requests. In Section 3.2, we estimated, based on Chrome browser statistics, that TFO could improve HTTP transaction latency by up to 25%. Here we ask a more general question: how much does TFO speed up whole-page downloads? Unfortunately, this question does not have a straightforward answer. On average, major web pages consist of 44 resources distributed

**Table 3.1.** Average page load time (PLT) in seconds for various pages for an emulated residential broadband user with a 4Mbps/256Kbps link. In all tests, the standard deviations of PLT are within 5% of average except for amazon.com with 20ms RTT (7%).

Page	RTT(ms)	PLT : non-TFO (s)	PLT : TFO (s)	Improv.
amazon.com	20	1.54	1.48	4%
	100	2.60	2.34	10%
	200	4.10	3.66	11%
nytimes.com	20	3.70	3.56	4%
	100	4.59	4.30	6%
	200	6.73	5.55	18%
wsj.com	20	5.74	5.48	5%
	100	7.08	6.60	7%
	200	9.46	8.47	11%
TCP wikipedia page	20	2.10	1.95	7%
	100	3.49	2.92	16%
	200	5.15	3.03	41%

across 7 different domains [76], and modern browsers have complex scheduling routines to fetch these resources using multiple parallel TCP connections.

First, we benchmarked several popular websites from the Alexa top 500 websites list [5]. The test-bed for these experiments consists of a single machine (Intel Core 2 Quad CPU 2.4GHz, 8GB RAM) that runs our TFO-enabled Linux kernel and Chrome browser. We used the Google web page replay tool to benchmark the web page download latency for TFO-enabled Chrome and for standard Chrome [97]. The page replay tool has two modes: record and replay. In record mode, the tool passively records all DNS and TCP traffic sent from and received by the browser into a local database. In replay mode, the tool runs a DNS server on the local machine and redirects the browser’s HTTP requests to the local proxy run by the tool. The replay can also leverage dummynet to emulate different network delays, bandwidths, and random packet loss [81]. All connections during replay use the loopback interface with a reduced MTU of 1500 bytes.

In our experiment, we emulated a broadband user with 4Mbps downlink and 256Kbps uplink bandwidth and with a 128KB buffer; this is a popular configuration as

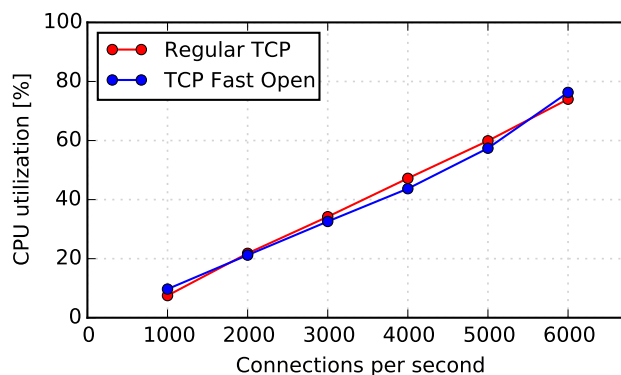
found by Netalyzer [50]. First, we used the tool to record the home pages of Amazon.com, the New York Times, the Wall Street Journal, and the Wikipedia page for TCP<sup>3</sup>. We then replayed each web page 20 times with and without TFO support in Chrome, and did so with three different RTTs: 20ms, 100ms, and 200ms. Therefore, for each page we gathered 120 samples. For each replay we performed a cold start of the browser and used a new user configuration folder (with an empty cache) to avoid caching effects and persistent connections to the replay tool’s proxy. Since all connections use the loopback interface, the TFO-enabled browser always has a valid TFO cookie and thus sends the cold HTTP requests in the SYN packet.

Thus for each replay with TFO-enabled browsing, we emulated a user with an empty browser cache visiting the website with TFO-enabled servers. For each page, the browser reports the page load time (PLT) that is measured from when the browser starts processing the URL until the browser *onload event* begins [96].<sup>4</sup> The PLT includes HTTP redirects, accessing the local cache, DNS lookups, HTTP transactions, and processing the root document. The results of the replays are shown in Table 3.1. As expected, TFO improves the PLT when the RTT is high for all the sites we tested. When the RTT is small and the network delay is only a small fraction of PLT, the resource processing time would exceed network time, so the gains from TFO are expected to be small. But even for pages heavy on content and with short emulated RTT (i.e. 20ms), TFO accelerates PLT by 4–5%. Conversely, for simpler pages such as wikipedia, the browser spends most of its time waiting for network transfers rather than processing the retrieved content, and thus TFO offers significant improvements of 16% and 41% with 100ms and 200ms RTTs respectively. The 200ms RTT figures roughly correspond to the expected performance on mobile devices since mobile RTTs are typically on the order of 100–200ms [84].

---

<sup>3</sup>[http://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](http://en.wikipedia.org/wiki/Transmission_Control_Protocol)

<sup>4</sup>To extract the PLT from Chrome, we opened the browser’s javascript console and entered “performance.timing.loadEventEnd - performance.timing.navigationStart”.



**Figure 3.6.** CPU utilization vs. web server load

### 3.6.2 Server Performance

To measure the impact of TFO on the server, we wrote a client program that generates HTTP 1.0 requests at a constant rate to an Apache server and fetches a 5KB web page. We used client and server machines with configurations similar to that in Section 3.6.1 and connected them through a Gigabit Ethernet switch; the RTT between them was about  $100\mu\text{s}$ . We limited the server machine to only use one CPU core and measured the CPU utilization using OProfile to evaluate the overhead added by cookie generation/validation operations and other TFO related processing. At each request rate, we measured the average CPU utilization across four 5 minute long trials for regular TCP and TFO. Figure 3.6 shows that server CPU utilization is nearly the same with and without TFO—in fact the CPU utilization was marginally lower when using TFO between 2000 and 5000 requests per second. We attribute this to the fewer packets that the server has to process when TFO is enabled as the request is included in the SYN packet. The AES encryption function used for cookie validation accounted for less than 0.3% of CPU utilization even at 6000 requests per second.

To further stress the connection setup process, we created another client load generator that repeatedly makes TFO connections to the server and requests the default



Apache home page. This server response fits within a single response packet thereby making the connection handshake a significant fraction of the entire connection. The client program operates in a closed loop and maintains one outstanding request to the server at any time; it creates a new connection to the server and sends another request as soon as it receives one complete HTTP response. With regular TCP, the server was able to sustain an average rate of 2876.4 transactions per second. Surprisingly, with the use of TFO, the server's sustained rate rose to 3548.7 transactions per second. This result is likely due to several factors: (1) one RTT saved per request, (2) fewer CPU cycles spent by the server to process each request (as the request is received in the SYN packet itself), and (3) one system call saved per request on the client.

## **3.7 Discussion**

Over the time that we spent discussing, designing, and implementing TCP Fast Open we considered several alternative approaches to the design of TFO cookies, to the semantics of TFO, to server-side attack mitigation, and alternative scenarios where TFO is useful. Here we describe those approaches and discuss their benefits and drawbacks.

### **3.7.1 One Time Cookies**

To prevent attacks in which a host reuses a cookie or cookies that it collects either legitimately or illegitimately, next we discuss an alternative design approach that we considered but ultimately did not implement (largely to keep the mechanism as simple as possible). In this approach, each TCP Fast Open cookie is valid for only one Fast Open. A client that wants to do more than one Fast Open must request more cookies to perform those subsequent Fast Opens. All open TCP connections (regardless of how they were opened) would have a limitation that they can only issue one Fast Open cookie for the lifetime of the connection, and that a cookie cannot be issued until the server has received

at least one ACK from the client—this maintains a one-to-one relationship between the number of currently valid cookies issued for a client-server pair and the number of TCP handshakes the pair have completed at some time.

Thus, a client host would a) open a connection with a normal three-way handshake, b) request a one-time Fast Open cookie, c) proceed as usual with the connection and eventually close it, d) open a new connection using its Fast Open cookie, and e) request a new cookie during this new connection. Clients that wish to open parallel TFO connections to a server would acquire multiple cookies to the same server across multiple regular TCP handshakes. In the (client) kernel, this approach would change the abstraction slightly, from a one-to-one mapping from server IP to cookie to a set mapping of server IP to a set of cookies; this change would not affect applications.

To implement one-time cookies, the zero-padding used for IPv4 addresses would be replaced by a 64-bit unsigned integer counter during cookie computation, thus the cookie would be the encryption of the concatenation of the server IP, client IP, and counter. This ensures that each cookie is unique, even for the same client-server pair. In this design, there need only be one counter per server, which is incremented whenever a Fast Open cookie is issued to any host.

There are several methods that could be used to prevent cookie reuse. Standalone servers would keep a lookup table to make sure that a cookie isn't reused in some small time window (e.g. a few minutes), and if it is, the server would fall back on a normal handshake. For servers behind load balancers, the load balancers could either do the same or, alternatively, could always hash the cookie value consistently to a destination back-end server, thereby ensuring that if a cookie is reused then the same server will receive the duplicate requests, so the client(s) will be caught. If no load balancer modification is possible—as may be the case for large production web services—two options are possible: a) the service simply allows for a cookie to be reused  $n$  times where  $n$  is

the number of servers behind a load balancer or b) the servers behind a load balancer periodically exchange information about recently seen cookies.

While this one-time cookie approach is more complex, it may have the benefit of thwarting some amplification and resource exhaustion attacks. Standalone servers or server farms with load balancers modified as described above would also have another benefit of providing TCP's usual semantics and not be exposed to the duplicate SYN issue since the cookie in a duplicate SYN would be rejected.

### **3.7.2 Data After SYN**

Some applications may require the transmission of initial data requests that cannot fit in a single packet. Thus the room provided by TFO in the SYN packet may be insufficient. Our TFO protocol design can easily support transmission of additional data packets following a TFO-enabled SYN packet (before receiving a SYN-ACK from the server). However these data packets would have the ACK flag unset since the initial sequence number of the server is unknown until the receipt of the SYN-ACK. Our experiments revealed that Internet paths originating at several major ISPs drop data packets without the ACK flag. In order to not introduce any additional deployment constraints, we decided to disallow data packets sent by the client following a TFO-enabled SYN packet. This effectively limits the amount of data to be sent by the client during 3WHS to a single MSS; all this data must fit within the initial SYN packet. This is sufficient for many client applications such as HTTP web requests. The server is not limited in this way, and thus will be able to send up to what the advertised receive window in the client's SYN packet and TCP's initial congestion window allow.

### 3.7.3 Server-side TFO Cache

TFO includes a counter of total pending TFO connection requests on a per service port basis or for the whole server. Therefore it is possible for an attacker to force the server to disable TFO for all clients by flooding the server with spurious TFO requests using a cookie it obtained itself or using a stolen cookie from a compromised host.

The server can avoid disabling TFO for all clients by maintaining a small cache of recently received TFO connection requests from different client IP addresses. For each client IP address in the cache, the server stores the number of pending TFO connection requests from the client IP. If the pending TFO requests from a particular client IP exceeds the administratively set threshold, the server can selectively disable TFO for just that client IP address. The cache can store, for example, 10,000 client IP addresses using a modest amount of memory. The cache uses an LRU replacement policy.

The server would still have to maintain the global or listener port-level accounting to serve as the final defense, but smaller thresholds may be used for individual client IP addresses. This is because a large number of compromised hosts can mount a coordinated attack in which they overflow the server-side cache and thus the cache entries replaced are those with the client IP addresses of other compromised hosts which are also flooding the server with spurious requests. Each client IP does not exceed the IP-level pending request threshold before its entry gets evicted from the cache, but it soon sends more spurious requests and is added to the cache again with its pending-requests counter reset. The server-side cache increases the number of valid cookies that the attacker must steal to disable TFO for everyone, but does not completely eliminate the possibility.

### 3.7.4 TCP Fast Open in Low Latency Networks

While TFO was motivated by short TCP transfers in the wide area network with long RTTs, its applicability extends to low latency networks like data centers as well. In

the data center, applications often keep connections alive for extended periods of time to avoid the connection setup overhead each time data has to be transferred to a peer. Keeping many idle connections alive increases the server load and memory requirements. TFO allows such applications to instantly begin data transfers on demand (just like UDP transfers), while getting all the benefits of TCP like reliability and congestion control. Some applications require on-demand fast one-packet ping-pong communication such as sending a periodic short log message to a log server. TFO is again useful for such applications to achieve fast reliable communication without having to maintain long lived connections.

### **3.7.5 Cookie-less TCP Fast Open**

The TFO cookie mechanism mitigates resource exhaustion and amplification attacks. However cookies are not necessary if the server has application-level protection or is immune to these attacks. For example a web server that only replies with a simple HTTP redirect response that fits in the SYN-ACK packet may not care about resource exhaustion. For such an application, the server could decide to disable TFO cookie checks. Similarly, when TFO is used inside a data center between machines in a trusted environment, the TFO cookie checks may be disabled.

Disabling the cookie checks simplifies both the client and the server, as the client no longer needs to cache the cookie and the server no longer needs to check or generate cookies. Disabling cookies also potentially simplifies configuration, as the server no longer needs a secret key. In specific scenarios where it is safe to disable TFO cookies, the server may choose to respond to a client with a null or empty cookie thereby indicating to the client that it is free to send data in the SYN packet.

### 3.8 Related Work

Several instances of prior work aim to improve TCP performance by directly eliminating the three-way handshake, or more generally by designing server-stateless extensions. Here we attempt to place TCP Fast Open in context and compare the design trade-offs that motivated prior work and motivate our work.

TCP Extensions for Transactions (T/TCP), among its other features, bypasses TCP's connection handshake, and thus shares both the goals and the challenges of TFO [20]. T/TCP focuses its effort on combating old or duplicate SYNs, and does not aim to mitigate security vulnerabilities introduced by bypassing 3WHS. Its TAO option and connection count add complexity and require the server to keep state per remote host, while still leaving it open for attack. It is possible for an attacker to fake a congestion control value that will pass the TAO test. Ultimately its scheme is insecure, as discussed by prior analyses [37, 73].

As noted earlier, our focus with TCP Fast Open is on its security and practicality, and thus we made the design decision to allow old, duplicate SYN packets with data. We believe this approach strikes the right balance, and makes TFO simpler and more appealing to TCP implementers and application developers. While TFO's vulnerability to SYN flood attacks is no different from traditional TCP, the damage an attacker can inflict on the server may be worse due to the additional cost of processing application level data, and thus deserves careful consideration. Numerous prior studies discuss approaches to mitigate ordinary SYN flood attacks (i.e., floods of SYN packets without data) [31]. However, none of these approaches, from stateless solutions such as SYN-cookies to stateful solutions such as SYN Caches, can preserve data sent in SYN packets while providing an effective defense. Thus we concluded that the best defense is to simply disable TFO when a host is suspected to be under a SYN flood attack (e.g. when the

SYN backlog is filled). Once TCP Fast Open is disabled, normal SYN flood defenses can be employed.

Like TCP Fast Open, TCPCT also allows SYN and SYN-ACK packets to carry data, though TCPCT is primarily designed to eliminate server state during the initial handshake, and to defend from spoofed denial-of-service attacks [9]. Therefore, TCPCT and TFO are designed to meet different needs and are not directly comparable. A TCPCT-enabled server does not keep any connection state during TCP's initial handshake, and thus the server side application must consume the data in the SYN packet and immediately produce the response data to be included in the SYN-ACK packet. Otherwise, the application's response is forced to wait until the handshake completes. This approach also constrains the application's response size to only one packet. By contrast, TFO allows the server to respond to data during the handshake even after the SYN-ACK is sent. Therefore, we believe TFO is better suited to enabling data exchange during 3WHS at least for web flows.

A recent proposal, Rapid-Restart [10], was proposed after the TFO IETF draft and has similar goals. Rapid-Restart is based on TCPCT; both the server and the client cache TCP control blocks after a connection is terminated, deviating from TCPCT's original design goal of saving server memory. The client sends a SYN with data and the previously stored TCPCT cookie. The server accepts the connection if the cookie and the IP match its cached copies. Rapid-Restart does not scale because it requires per-connection state at the server. Moreover, Rapid-Restart cannot be used in server farms because connection state is retained only by the server that processed the last connection from the client, and a subsequent connection from that client may be directed to a different server in the farm unless the load balancer is modified.

More recently Zhou *et al.* proposed ASAP which provides a solution to reduce DNS and eliminate TCP handshake latency [103]. It employs public-key certificates

issued by a provenance verifier and signed by clients to ensure authenticity of requests to a server. In doing so, it offers more generality at the expense of computational overhead and incremental deployability.

Since none of the proposals we have discussed above are deployed, browser vendors have developed their own feature – “PRECONNECT” to avoid TCP handshake latency. Chrome and Internet Explorer 9 maintain a history of the domains for frequently visited web pages. The browsers then speculatively pre-open TCP connections to these domains before the user initiates any requests for them. Tests show this feature improves overall page load time by 6-10% for the top 35 websites [13,24]. The downside of this approach is that it wastes server and network resources by initiating and maintaining idle connections due to mis-speculation; the hit rate for these mechanisms is fairly low. TFO offers similar performance improvement without the added overhead.

### **3.9 Summary**

To improve the performance of short transfers, we proposed TCP Fast Open (TFO), which enables data to be exchanged safely during TCP’s initial handshake. Our analysis of both Google server logs and Chrome browser statistics shows that handshaking has become a performance bottleneck for web transfers. TFO enables applications to decrease request latency by one round-trip time while avoiding severe security ramifications. At the core of TFO is a security cookie issued by the server to authenticate clients that initiate TFO connections. We believe that this cookie mechanism provides an acceptable defense against potential denial-of-service attacks. TFO is also designed to fall back gracefully on regular TCP handshaking as needed.

Our goal—of including data in TCP SYN and SYN-ACK packets—is not novel. The TCP standard already allows it, but forbids the receiver from processing the data until the handshake completes. Several recent proposals achieve similar goals to TFO but



have not seen wide deployment. The main contribution of TFO is the simplicity of its design, allowing rapid and incremental deployment while maintaining reasonable defense against denial-of-service attacks. We believe TFO interoperates well with existing TCP implementations, middle-boxes, server farms, and legacy server and client applications.

We have implemented TCP Fast Open in the Linux kernel and shown that it imposes minimal performance overhead for clients and servers, with significant latency improvement for short transfers. Our analysis and test-bed results show that TFO can improve single HTTP request latency by over 10% and the overall page load time from 4% to 40%. At the time of writing this dissertation, TFO is under discussion at the IETF for publishing as an Experimental Internet Standard [23]. TFO has been deployed on Google's web servers. TFO is available as a part of the mainline Linux kernel since version 3.6 (client side support), and 3.7 (server side support). The Chrome browser has application level support for users to leverage TFO. ChromeOS which is based on Linux and Chrome is also TFO capable.

### **3.10 Acknowledgments**

We thank the anonymous CoNEXT reviewers and our shepherd Kyoungsoo Park for their comments. We are thankful to Mike Belshe for motivating this work and making the Chrome browser an early adopter of TFO. We thank Adam Langley, Tom Herbert, Roberto Peon, and Mathew Mathis for their insightful comments on early designs of TFO. We would like to also thank the IETF tcpm working group for their comments and feedback on the design. In particular, we wish to acknowledge the efforts of Bob Briscoe, Michael Scharf, Gorry Fairhurst, Rick Jones, William Chan, Neal Cardwell, and Eric Dumazet for offering their feedback and help in improving the design.

Chapter 3, in part, contains material as it appears in the Proceedings of the 7th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT

'11), Tokyo, Japan, December 2011. "TCP Fast Open". Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, contains material as it appears in the IETF Internet Drafts, March 2011 – February 2014. "TCP Fast Open". Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain.

## Chapter 4

# Dahu: Commodity Switches for Direct Connect Data Center Networks

Solving “Big Data” problems requires bridging massive quantities of compute, memory, and storage, which requires a very high bandwidth network. Recently proposed direct connect networks like HyperX [1] and Flattened Butterfly [47] offer large capacity through paths of varying lengths between servers, and are highly cost effective for common data center workloads. However data center deployments are constrained to multi-rooted tree topologies like Fat-tree [3] and VL2 [35] due to shortest path routing and the limitations of commodity data center switch silicon.

In this chapter we present Dahu<sup>1</sup>, simple enhancements to commodity Ethernet switches to support direct connect networks in data centers. Dahu avoids congestion hot-spots by dynamically spreading traffic uniformly across links, and forwarding traffic over non-minimal paths where possible. By performing load balancing primarily using local information, Dahu can act more quickly than centralized approaches, and responds to failure gracefully. Our evaluation shows that Dahu delivers up to 500% improvement in throughput over ECMP in large scale HyperX networks with over 130,000 servers, and up to 50% higher throughput in an 8,192 server Fat-tree network.

---

<sup>1</sup>Dahu is a legendary creature well known in France with legs of differing lengths.

## 4.1 Introduction

Historically, high-speed networks have fallen into two main design spaces. High performance computing (HPC) and supercomputing networks have typically adopted *direct network topologies*, configured so that every switch has some servers connected to it. The remaining ports in each switch are used to connect to other switches in the topology (e.g. mesh, torus, hypercube). This type of network is highly resource efficient, and offers high capacity through the presence of many variable-length paths between a source and destination. However, the choice of which path to forward traffic over is ultimately controlled by proprietary protocols in switches, NICs, and by the end-host application logic. This increases the burden on the developer, and creates a tight coupling between applications and the network.

On the other hand, scale-out data centers have adopted *indirect network topologies*, such as folded Clos and Fat-trees, in which servers are restricted to the edges of the network fabric. There are dedicated switches that are not connected to any servers, but simply route traffic within the network fabric. Data centers have a much looser coupling between applications and network topology, placing the burden of path selection on network switches themselves. Given the limited resources and memory available in commodity switches, data center networks have historically relied on relatively simple mechanisms for choosing paths, e.g., Equal-Cost Multi-Path Routing (ECMP).

ECMP relies on static hashing of flows across a fixed set of shortest paths to a destination. For hierarchical topologies like Fat-trees [3], shortest path routing has been largely sufficient when there are no failures. However, recently proposed direct network topologies like HyperX, BCube, and Flattened Butterfly [1, 36, 47], which employ paths of different lengths, have not seen adoption in data centers due to the limitations imposed by commodity data center switches and shortest path routing. ECMP

leaves lot of network capacity untapped when there is localized congestion or hot-spots as it ignores uncongested longer paths while forwarding. Further, even in hierarchical networks, ECMP makes it hard to route efficiently under failures, when the network is no longer completely symmetric, and some non-shortest paths can be utilized to improve network utilization.

Commodity switches and shortest path routing have led to hierarchical networks in data centers. These restrictions on topology and routing also mean that higher level adaptive protocols like MPTCP [75] are unable to take advantage of the full capacity of direct networks because all paths are not exposed to them through routing/forwarding tables.

The goal of Dahu is to bridge the benefits of direct connect networks—higher capacity with fewer switches (lower cost) for common communication patterns—with the lower complexity, commoditization, and decoupled application logic of data center networks. To that aim, we present Dahu, a lightweight switch mechanism that enables us to leverage non-shortest paths with loop-free forwarding, while operating locally, with small switch state requirements and minimal additional latency. Dahu seeks to obtain the benefits of non-shortest path routing without coupling the application to the underlying topology. Dahu supports dynamic flow-level hashing across links, resulting in higher network utilization. Dahu addresses the local hash imbalance that occurs with ECMP using only local information in the switches.

Dahu makes the following contributions: (1) Novel hardware primitives to efficiently utilize non-minimal paths in different topologies with a modest increase in switch state, while preventing persistent forwarding loops, (2) A *virtual port* abstraction that enables dynamic multipath traffic engineering, (3) A decentralized load balancing algorithm and heuristic, (4) Minimal hardware modifications for easy deployability, and (5) Large scale simulations on networks with over 130K servers to evaluate Dahu’s per-

formance. Our evaluation shows that Dahu delivers up to 50% higher throughput relative to ECMP in an 8,192 server Fat-tree network and up to 500% throughput improvement in large HyperX networks with over 130,000 servers. We are encouraged by these results, and believe that they are a concrete step toward our goal of combining the benefits of HPC and data center network topologies.

## 4.2 Motivation and Requirements

Fully-provisioned multi-rooted tree topologies are ideal for targeting worst case communication patterns—where all hosts in the network simultaneously try to communicate at access link speeds. However, common communication patterns have only few network hot-spots and over-provisioning the topology for worst-case traffic results in high CAPEX. Oversubscribing the multi-rooted tree topology would reduce CAPEX, but network performance would also suffer in the common case since the oversubscribed layers of the tree have even lower capacity to tolerate hot-spots.

Direct networks provide an interesting point in the design space of network topologies, since they provide good performance for most realistic traffic patterns, at much lower cost than fully-provisioned Clos networks [1, pg.8-9] [47, pg.6-8]. There are two defining characteristics of direct networks which distinguish them from tree based topologies. (1) Hosts are embedded throughout the structure of the network. Each switch has some hosts connected to it. (2) There are many network paths between any pair of servers—but they are of varying length. These properties of direct networks allow more flexible use of overall network capacity, with slack bandwidth in one portion of the network available to be leveraged by other congested parts of the network by forwarding traffic along longer less congested paths. In a sense the oversubscription is “spread throughout the network” rather than at specific stages or points in the topology.

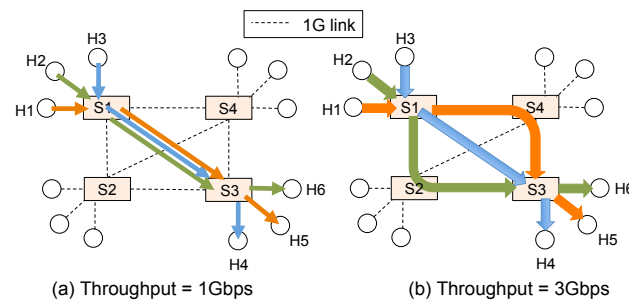
Direct networks are very popular in HPC—Titan, the world’s second fastest

supercomputer, uses a 3D torus, a direct connect topology [92]. However, data centers have been largely constrained to multi-rooted trees due to commodity switch silicon and shortest path based routing protocols. Direct networks have significant potential in meeting the bandwidth requirements for data centers, but have yet to see wide-scale deployments. Dahu presents simple enhancements to commodity Ethernet switches (both hardware and software) to support direct connect topologies in data centers.

### 4.2.1 Challenges

In order to deploy direct connect networks in data centers, we need to address the following challenges:

(1) **Non-shortest path routing:** Current data center switches and routing protocols only support routing over shortest paths. Direct networks, offer large path diversity between servers, but the paths are of varying lengths, typically with only a small number of shortest paths. Shortest path routing artificially constrains bandwidth utilization during periods of localized congestion, and traffic to a destination could potentially achieve higher throughput, if alternate longer paths are also used. Consider a simple mesh network of four switches, as shown in Figure 4.1. In (a), the shortest path connecting the sources and destinations is congested, by a factor of 3 with a resulting total bandwidth of 1Gbps. However, by sending some traffic flows over non-shortest path links, as shown in (b), the total throughput can be increased to 3Gbps.



**Figure 4.1.** (a) Shortest, and (b) Non-shortest path routing

(2) **Cost-effective commodity switch hardware:** Direct networks in supercomputers rely on custom designed silicon that is expensive and topology dependent. Routing is typically integrated with proprietary protocols in NICs and switches. Data center networks on the other hand are built using low cost commodity off-the-shelf switches [3,35]. So commodity Ethernet switch silicon must be enhanced to provide the necessary features to support direct connect topologies in the data center while keeping costs low.

(3) **Dynamic traffic management:** Although shortest path routing is the main roadblock to deploying direct networks, the static nature of ECMP style forwarding in switches presents a challenge—even for indirect networks. Hashing flows on to paths, oblivious to current link demands or flow rates can result in imbalance, significantly reducing achieved throughput compared to innate network capacity [4].

There have been several recent proposals for dynamic traffic engineering in data centers. Centralized approaches like Hedera [4] and MicroTE [17] advocate a central fabric scheduler that periodically measures traffic demands, and computes good paths for bandwidth intensive flows to maximize bandwidth utilization. However they have long control loops that only react to changes in traffic at timescales on the order of hundreds of milliseconds at best, i.e. they are only effective for long lived flows. Further, scaling such centralized approaches to very large numbers of flows in large scale data centers presents challenges in terms of switch state requirements. MPTCP [75] is a transport layer solution that splits flows into subflows that take different paths through the network, and modulates the rates of subflows based on congestion. However, in direct networks, MPTCP requires many subflows to probe the different paths, making it impractical for short flows. We illustrate this in Section 4.6.4.

(4) **Decouple applications and routing:** Direct connect networks in supercomputers tightly couple application development and routing, which requires application developers to be concerned with workloads, routing for certain expected application



behaviors, etc. Data center workloads are much more dynamic, and developers often cannot predict the composed behavior of many co-located applications. Handling routing functionality entirely in the network significantly simplifies application development as is done in data centers today.

#### 4.2.2 Dahu Requirements and Design Decisions

(1) **On-demand non-shortest path routing:** Dahu should only enable non-minimal paths on demand when shortest paths do not have sufficient capacity. Using shorter paths by default results in fewer switch hops and likely lower end-to-end latency for traffic. Dahu must achieve this while ensuring there are no persistent forwarding loops.

(2) **Dynamic traffic engineering:** Dahu chooses to load balance traffic primarily using local decisions in each switch which helps react quickly to changing traffic demands and temporary hot-spots. In addition, it inter-operates with other routing and traffic engineering schemes.

(3) **Readily deployable:** Any proposed changes to switch hardware should be simple enough to be realizable with current technology, with minimal required switch state. Switches should still make flow-level forwarding decisions—i.e., packets of a particular flow should follow the same path through the network to the extent possible. This avoids excessive packet reordering, which can have undesirable consequences for TCP. Moving flows to alternate paths periodically at coarse time scales (e.g., of several RTTs) is acceptable.

(4) **Generic/Topology Independent:** The switch hardware should be topology independent and deployable in a variety of networks including indirect networks. Non-shortest path routing is also beneficial in the case of Clos topologies which are left asymmetric and imbalanced under failures.

(5) **Fault tolerant:** Failures must be handled gracefully, and re-routing of flows upon failure should only affect a small subset of flows, so that the effect of failures is proportional to the region of the network that has failed. To prevent traffic herds, Dahu should not move many flows in the network around when a single path fails or is congested. Rather, it should be possible to make finer-grained decisions and migrate a smaller subset of flows to alternate paths.

Dahu achieves these targets through a combination of switch hardware and software enhancements, which we describe in Sections 4.3 and 4.4 respectively.

### 4.3 Switch Hardware Primitives

Dahu proposes new hardware primitives which enable better ways of utilizing the path diversity in direct connect topologies, and addresses some of the limitations of ECMP.

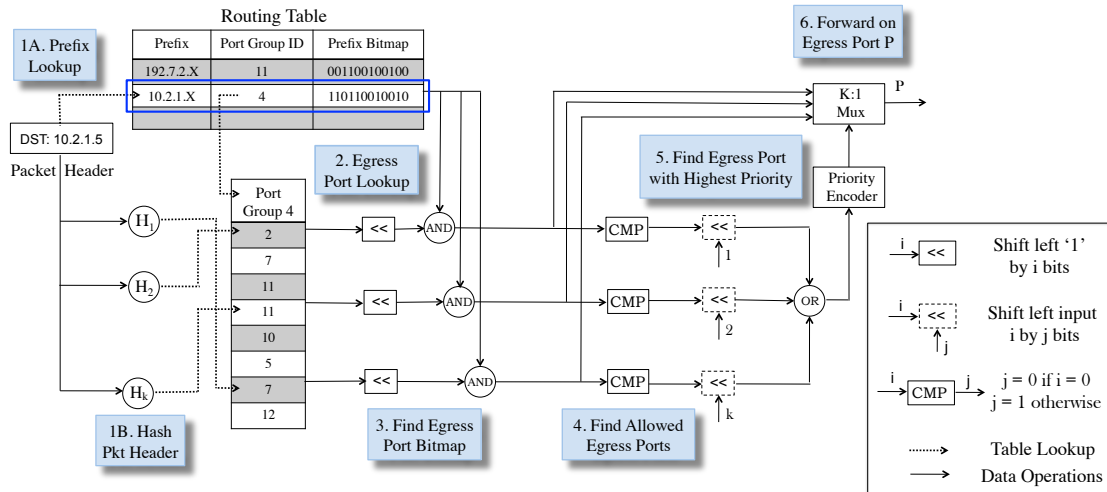


Figure 4.2. Datapath pipeline for packet forwarding in Dahu

### 4.3.1 Port Groups With Virtual Ports

ECMP spreads traffic over multiple equal-cost paths to a destination. Internally, the switch must store state to track which set of ports can be used to reach the destination prefix. A common mechanism is storing a list of egress ports in the routing table, represented as a bitmap. Dahu augments this with a layer of indirection: each router prefix points to a set of *virtual ports*, and each virtual port is mapped to a physical port. In fact, the number of virtual ports can be much larger than the number of physical ports. We define a *port group* as a collection of virtual ports mapped to their corresponding physical ports (a many-to-one mapping). The routing table is modified to allow a pointer to a port group for any destination prefix instead of a physical egress port. When multiple egress choices are available for a particular destination prefix, the routing table entry points to a port group.

When the switch receives a packet, it looks up the port group for the destination prefix from the routing table. It computes a hash based on the packet headers, similar to ECMP, and uses this to index into the port group to choose a virtual port. Finally, it forwards the packet on the egress port to which the virtual port is mapped. This port group mechanism adds one level of indirection in the switch output port lookup pipeline, which we use to achieve better load balancing, and support for non-shortest network paths.

In hardware, a port group is simply an array of integers. The integer at index  $i$  is the egress port number to which virtual port  $i$  in that port group is mapped. Each port group has a fixed number of virtual ports, larger than the number of egress ports in the routing table. Multiple destination prefixes in the routing table may point to the same port group. For the rest of this chapter, the term *member port* of a port group is used to refer to a physical port which has some virtual port in the port group mapped to it.

The virtual port to egress port mapping provides an abstraction for dynamically migrating traffic from one egress port to another within any port group. Each virtual port is mapped to exactly one egress port at any time, but this mapping can be changed dynamically. A key advantage of the indirection layer is that when a virtual port is remapped, only the flows which hash to that virtual port are migrated to other egress ports. Other flows remain on their existing paths and their packets don't get re-ordered. All flows that map to a virtual port can only be remapped as a group to another egress port. Thus, virtual ports dictate the granularity of traffic engineering, with more virtual ports providing finer grained control over traffic. We propose that each port group have a relatively large number of virtual ports—on the order of 1,000 for high-radix switches with 64-128 physical ports. That means each virtual port is responsible for an average of 0.1% or less of the total traffic to the port group. If required, the routing table can be augmented with more fine grained forwarding entries.

Each port group keeps a set of counters corresponding to each member port indicating how much traffic the port group has forwarded to that port. While having a traffic counter for each virtual port provides fine grained state, it comes at a higher cost for two reasons: (1) The memory required to store the counters is fairly large. For example, a switch with 64 port groups, 1,024 virtual ports per port group, and 64 bit traffic counters needs 512KB of on-chip memory just for these port group counters. (2) Reading all 65,536 counters from hardware to switch software would take a long time, increasing the reaction time of traffic engineering schemes that use all these counters for computations. Dahu uses the port group mechanism and associated counters to implement a novel load balancing scheme described in Section 4.4.

### 4.3.2 Allowed Port Bitmaps

Port groups enable the use of multiple egress port choices for any destination prefix. However, it is sometimes useful to have many egress ports in a port group, but use only a subset of them for forwarding traffic to a particular prefix. One reason to do this is to avoid forwarding on failed egress ports. Consider two prefixes in the routing table  $F_1$  and  $F_2$  both of which point to port group  $G_1$  which has member egress ports  $P_1, P_2, P_3, P_4$ . Suppose a link fails and egress port  $P_4$  cannot be used to reach prefix  $F_1$ , whereas all member ports can still be used to reach  $F_2$ . Now, one option is to create another port group  $G_2$  with member ports  $P_1, P_2$  and  $P_3$  only, for prefix  $F_1$ . This can quickly result in the creation of many port groups for a large network which might experience many failures at once. We propose a more scalable approach where we continue to use the existing port group, but restrict the subset of member ports which are used for forwarding traffic to a particular destination.

For each destination prefix in the routing table, we store an *allowed port bitmap*, which indicates the set of egress ports that are allowed to be used to reach the prefix. This bitmap is only used when the routing table entry points to a port group. The bitmap is as wide as the number of egress ports, and only the bits corresponding to the allowed egress ports for the prefix are set. One way to restrict forwarding to the allowed egress ports is to compute a hash for the packet and check if the corresponding port group virtual port maps to an allowed egress port. If not, we compute another hash for the packet and repeat until we find an allowed egress port.

To pick an allowed port efficiently, we propose a parallel scheme where the switch computes 16 different hash functions for the packet in parallel. The first valid allowed egress port among the hashed choices is used for forwarding. In case none of the 16 hash functions picked an allowed egress port, we generate another set of 16 hash values for the

packet and retry. This is repeated some fixed number of times (say 2) to bound output port lookup latency. If an allowed egress port is still not found, we just fall back to randomly picking one of the allowed egress ports, i.e. we ignore the port group mechanism for this packet and just hash it on to one of the allowed egress ports directly. We explore other uses of the allowed port bitmap in Section 4.4.2.

Figure 4.2 illustrates the egress port lookup pipeline incorporating both port groups and allowed egress port mechanisms. The switch supports a fixed number of allowed port bitmaps for each prefix and has a selector field to indicate which bitmap should be used. Dahu uses an *allow all* bitmap which is a hardwired default bitmap  $B_{all}$  where all bits are set, i.e. all member ports of the port group are allowed. The *shortest path* bitmap  $B_{short}$  is an always available bitmap that corresponds to the set of shortest path egress ports to reach the particular destination prefix. Unlike the  $B_{all}$  bitmap,  $B_{short}$  is not in-built and has to be updated by the switch control logic if port group forwarding is used for the prefix. Its use is described in Sections 4.3.3 and 4.4.2. There can be other bitmaps as well for further restricting the set of egress ports for a destination prefix based on other constraints.

### 4.3.3 Eliminating Forwarding Loops

Dahu uses non-shortest path forwarding to avoid congestion hot-spots when possible. The term *derouting* is used to refer to a non-minimal forwarding choice by a switch. The number of times a particular packet has been derouted (routed on an egress port not along shortest paths) is referred to as the *derouting count*. An immediate concern with derouting is that it can result in forwarding loops. To prevent persistent forwarding loops, Dahu augments network packets with a 4-bit field in the IP header to store the derouting count. Switches increment this field only if they choose a non-minimal route for the packet. Servers set this field to zero when they transmit traffic. In practice, the

derouting count need not be a new header field, e.g., part of the TTL field or an IP option may be used instead.

When a switch receives a packet, if the derouting count in the packet header has reached a maximum threshold, then the switch forwards the packet along shortest paths only. This is enforced using the  $B_{short}$  allowed port bitmap for the destination prefix described earlier. The derouting count is also used while computing the packet hash. If a packet loops through the network and revisits a switch, its derouting count will have changed. The resulting change to the hash value will likely forward the packet along a different path to the destination. Each switch also ensures that a packet is not forwarded back on the ingress port that it arrived on. Further, in practice, only a few deroutings are required to achieve benefits from non-minimal routing and the derouting count threshold for the network can be configured by the administrator as appropriate. These factors ensure that any loops that occur due to non-minimal routing are infrequent and don't hinder performance.

As with current distributed routing protocols, transient loops may occur in certain failure scenarios. Dahu uses standard IP TTL defense mechanisms to ensure that packets eventually get dropped if there are loops during routing convergence.

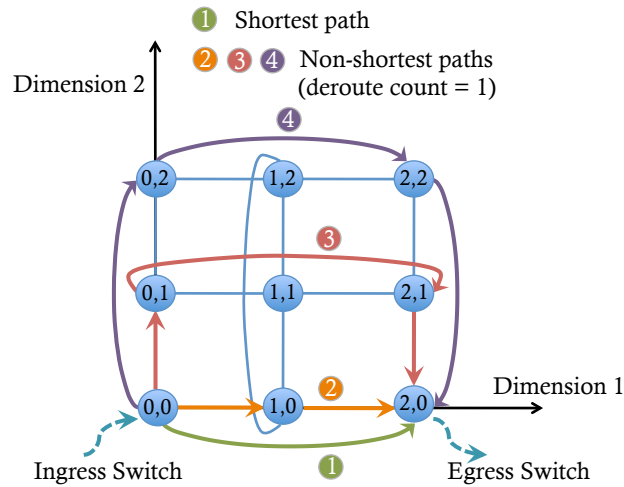
## 4.4 Switch software

Now we look at how Dahu's hardware primitives can more efficiently utilize the network's available capacity. We describe how to leverage non-minimal paths, and then look at dynamic traffic engineering to address local hash imbalances in switches. These techniques rely on Dahu's hardware primitives, but are independent and may be deployed separately. We begin with some background on HyperX topology.

### 4.4.1 Background on HyperX Topology

We use the HyperX topology, a direct connect network for detailing how Dahu’s hardware primitives are used, and for evaluating the techniques. This section summarizes the HyperX topology and related terminology [1].

HyperX is an  $L$ -dimensional direct network with many paths of varying length between any pair of servers. It can be viewed as a generalization of the HyperCube topology. In an  $L$ -dimensional HyperCube, each switch’s position can be denoted by a vector of  $L$  coordinates, each coordinate being 0 or 1. Any two switches that differ in their coordinate vectors in exactly one dimension are connected by a bidirectional link. Each switch has some fixed number  $T$  of servers attached to it. E.g., a regular cube is a 3-dimensional HyperCube with 8 switches and 12 edges.



**Figure 4.3.** HyperX topology ( $L=2$ ,  $S=3$ ). Only switches and the links between them are shown for clarity. The  $T$  servers connected to each switch are not shown in the figure. The position of the switches is shown on a 2-dimensional lattice. The paths between switches  $(0, 0)$  and  $(2, 0)$  with at most 1 derouting are shown. Ingress switch  $(0, 0)$  and the egress switch  $(2, 0)$  are offset along dimension 1 and aligned along dimension 2.

A regular  $(L, S, T)$  HyperX, is a generalization of the HyperCube where the switch coordinates in each dimension are integers in the range  $[0, S-1]$  rather than just 0



or 1. Again, any two switches whose coordinate vectors differ in only one dimension are directly connected. Figure 4.3 shows an example of a 2-dimensional HyperX network with the switches overlaid on a 2-D lattice. An *offset* dimension for a pair of switches is one in which their coordinates differ. Similarly, an *aligned* dimension for a pair of switches is one in which their coordinates are the same. Some examples of HyperX topologies are: (1) A HyperCube is a regular HyperX with  $S=2$ , and (2) An  $L=1$  HyperX is just a fully connected graph.

#### 4.4.2 Non-Minimal Routing

As described earlier, ECMP constrains traffic routes to the set of shortest paths between any pair of switches. While this keeps path lengths low, it can also impose artificial constraints on available bandwidth. Direct connect networks like HyperX have many paths of differing length between any pair of nodes. In a HyperX switch  $S_s$ , there are three classes of egress port choices to reach any destination switch  $S_d$ .

1. Set of shortest path egress ports to reach  $S_d$ . The size of the set is equal to the number of offset dimensions, between the  $S_s$  and  $S_d$ , i.e. dimensions in which the switch coordinates of  $S_s$  and  $S_d$  differ. In Figure 4.3, the egress port on switch (0, 0) along path 1 is a shortest path egress port.
2. Set of egress ports connected to neighbors along offset dimensions excluding the shortest path egress ports. Each of these neighbors is at the same distance from  $S_d$ , equal to the shortest path distance from  $S_s$  to  $S_d$ . In Figure 4.3, the egress port on switch (0, 0) along path 2 is in this set.
3. All the remaining ports that are connected to other switches. The egress ports which connect to neighbors along dimensions already aligned with  $S_d$  are members of this class. Each of these neighbors is one additional hop away from  $S_d$  as

compared to the shortest path distance from  $S_s$  to  $S_d$ . In Figure 4.3, the egress ports on switch (0,0) along paths 3 and 4 are in this set.

Dahu's port group mechanism and allowed port bitmaps enable switches to efficiently route along non-minimal paths. The number of shortest and non-minimal path egress ports for a single destination prefix is not limited artificially, unlike n-way ECMP. The virtual port to physical port mapping, and allowed port bitmaps control how traffic gets forwarded onto shortest paths and non-minimal paths. We now look in more detail at how to enable non-minimal routing in direct connect networks.

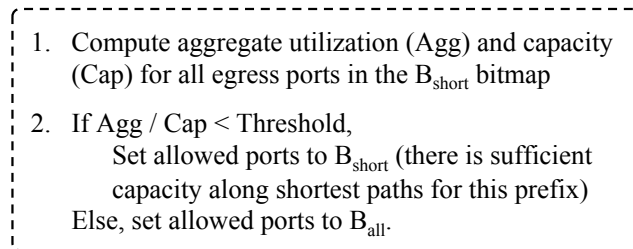
### **Space saving techniques**

A strawman solution for non-minimal routing is to create one port group for each destination prefix. For each prefix's port group, we make all appropriate physical ports (both along shortest paths and non-minimal paths to the destination) members of the port group. When a switch receives a packet, it looks up the port group for the destination prefix, and hashes the packet onto one of the virtual ports in the port group. Use of some of the corresponding egress ports results in non-minimal forwarding.

One characteristic of the HyperX topology is that in any source switch, the set of shortest path egress ports is different for each destination switch. These ports must be stored separately for each destination switch, thereby requiring a separate destination prefix and a separate port group in case of the strawman solution. For a large HyperX network, the corresponding switch memory overhead would be impractical. For example, a network with 2,048 128-port switches, and 1,024 virtual ports per port group would need 2 MB of on-chip SRAM just to store the port group mapping (excluding counters). Thus, we seek to aggregate more prefixes to share port groups. We now describe some techniques to use a small number of port groups to enable the use of non-minimal paths, while using only shortest paths whenever possible.

In a HyperX switch, any egress port that is connected to another switch can be used to reach any destination. However not all egress ports would result in paths of equal length. Let us assume that we only use a single port group  $PG_{all}$ , say with 1,024 virtual ports. All physical ports in the switch connected to other switches are members of the port group  $PG_{all}$ . If we simply used this port group for all prefixes in the routing table, that would enable non-minimal forwarding for all destinations.

Dahu uses the allowed port bitmap hardware primitive to restrict forwarding to shorter paths when possible. If Dahu determines that only shortest paths need to be used for a particular destination prefix, the  $B_{short}$  allowed port bitmap for the prefix is used for forwarding, even when the derouting count has not reached the maximum threshold. Otherwise, the allowed port bitmap is expanded to also include egress ports that would result in one extra hop being used and so on for longer paths. For HyperX, there are only three classes of egress port choices by distance to destination as described earlier; in our basic non-minimal routing scheme, we either restrict forwarding to the shortest path ports or allow all paths to the destination (all member ports of  $PG_{all}$ ).

- 
1. Compute aggregate utilization (Agg) and capacity (Cap) for all egress ports in the  $B_{short}$  bitmap
  2. If  $Agg / Cap < Threshold$ ,  
     Set allowed ports to  $B_{short}$  (there is sufficient capacity along shortest paths for this prefix)  
     Else, set allowed ports to  $B_{all}$ .

**Figure 4.4.** Restricting non-minimal forwarding

We now look at the question of how Dahu determines when additional longer paths have to be enabled for a destination prefix to meet traffic demands. Switches already have port counters for the total traffic transmitted by each physical port. Periodically (e.g., every 10ms), the switch software reads all egress port counters, iterates over each destination prefix, and performs the steps shown in Figure 4.4 to enable non-minimal

paths based on current utilization. It is straightforward to extend this technique to progressively enable paths of increasing lengths instead of all non-minimal paths at once. In summary, we have a complete mechanism for forwarding traffic along shorter paths whenever possible, using just a single port group and enabling non-minimal routing whenever required for capacity reasons.

### **Constrained non-minimal routing**

As described earlier, in a HyperX network, each switch has three classes of egress port choices to reach any destination. Based on this, Dahu defines a constrained routing scheme as follows—a switch can forward a packet only to neighbors along offset dimensions. If a packet is allowed to use non-minimal routing at a switch, it can only be derouted along already offset dimensions. Once a dimension is aligned, we do not further deroute the packet along that dimension. After each forwarding choice along the path taken by a packet, it either moves closer to the destination or stays at the same distance from the destination. We call this scheme *Dahu constrained routing*. For this technique, we create one port group for each possible set of dimensions in which the switch is offset from the destination switch. This uses  $2^L$  port groups where  $L$ , the number of dimensions is usually small, e.g., 3–5. This allows migrating groups of flows between physical ports at an even smaller granularity than with a single port group.

This technique is largely inspired by Dimensionally Adaptive, Load balanced (DAL) routing [1]. However, there are some key differences. DAL uses per-packet load balancing, whereas Dahu uses flow level hashing to reduce TCP reordering. DAL allows at most one derouting in each offset dimension, but Dahu allows any number of deroutings along offset dimensions until the derouting threshold is reached.

### 4.4.3 Traffic Load Balancing

Per-packet uniform distribution of traffic across available paths from a source to destination can theoretically lead to very good network utilization in some symmetric topologies such as Fat-trees. But this is not used in practice due to the effects of packet reordering and faults on the transport protocol. ECMP tries to spread traffic uniformly across shortest length paths at the flow level instead. But due to its static nature, there can be local hash imbalances. Dahu presents a simple load balancing scheme using local information at each switch to spread traffic more uniformly.

Each Dahu switch performs load balancing with the objective of *balancing* or *equalizing* the aggregate load on each egress port. This also balances bandwidth headroom on each egress port, so TCP flow rates can grow. This simplifies our design, and enables us to avoid more complex demand estimation approaches. When multiple egress port choices are available, we can remap virtual ports between physical ports, thus getting fine grained control over traffic. Intuitively, in any port group, the number of virtual ports that map to any member port is a measure of the fraction of traffic from the port group that gets forwarded through that member port. We now describe the constraints and assumptions under which we load balance traffic at each switch in the network.

#### Design Considerations

Periodically, each switch uses local information to rebalance traffic. This allows the switch to react quickly to changes in traffic demand and rebalance port groups more frequently than a centralized approach or one that requires information from peers. Note that this design decision is not fundamental—certainly virtual port mappings can be updated through other approaches. For different topologies, more advanced schemes may be required to achieve global optimality such as through centralized schemes.

We assume that each physical port might also have some traffic that is not *re-routable*. So Dahu’s local load balancing scheme is limited to moving the remainder of traffic within port groups. Dahu’s techniques can inter-operate with other traffic engineering approaches. For example, a centralized controller can make globally optimal decisions for placing elephant flows on efficient paths in the network [4], or higher layer adaptive schemes like MPTCP can direct more traffic onto uncongested paths. Dahu’s heuristic corrects local hashing inefficiencies and can make quick local decisions within a few milliseconds to avoid temporary congestion. This can be complemented by a centralized or alternate approach that achieves global optimality over longer time scales of few hundreds of milliseconds.

### **Control Loop Overview**

Every Dahu switch periodically rebalances the aggregate traffic on its port groups once each epoch (e.g., every 10ms). At the end of each rebalancing epoch, the switch performs the following 3 step process:

*Step 1: Measure current load:* The switch collects the following local information from hardware counters: (1a) for each port group, the amount of traffic that the port group sends to each of the member ports, and (1b) for each egress port, the aggregate bandwidth used on the port.

*Step 2: Compute balanced allocation:* The switch computes a balanced traffic allocation for port groups, i.e. the amount of traffic each port group should send in a balanced setup to each of its member ports. We describe two ways of computing this in Sections 4.4.4 and 4.4.5.

*Step 3: Remap port groups:* The switch then determines which virtual ports in each port group must be remapped to other member ports in order to achieve a balanced traffic allocation, and changes the mapping accordingly. We have the current port group

traffic matrix (measured) and the computed balanced traffic allocation matrix for each port group to its member egress ports.

As mentioned in Section 4.3.1, a switch only maintains counters for the total traffic from a port group to each of its member ports. We treat all virtual ports that map to a particular member port as equals and use port group counters to compute the average traffic that each of the virtual ports is responsible for. Then, we remap an appropriate number of virtual ports to other member ports depending on the intended traffic allocation matrix using a first-fit heuristic. In general, this remapping problem is similar to bin packing.

PG \ Port	0	1	2	3
0	4	1	2	--
1	--	1	--	2
BG	2	2	2	0
Agg	6	4	4	2

Initial Port Group (PG)  
Utilizations

PG \ Port	0	1	2	3
0	$2 \frac{2}{3}$	$1 \frac{2}{3}$	$2 \frac{2}{3}$	--
1	--	1	--	2
BG	2	2	2	0
Agg	$4 \frac{2}{3}$	$4 \frac{2}{3}$	$4 \frac{2}{3}$	2

Step 1:  
Balancing Port Group 0

PG \ Port	0	1	2	3
0	$2 \frac{2}{3}$	$1 \frac{2}{3}$	$2 \frac{2}{3}$	--
1	--	0	--	3
BG	2	2	2	0
Agg	$4 \frac{2}{3}$	$3 \frac{2}{3}$	$4 \frac{2}{3}$	3

Step 2:  
Balancing Port Group 1

PG \ Port	0	1	2	3
0	$2 \frac{2}{3}$	$2 \frac{2}{3}$	$2 \frac{2}{3}$	--
1	--	0	--	3
BG	2	2	2	0
Agg	$4 \frac{1}{3}$	$4 \frac{1}{3}$	$4 \frac{1}{3}$	3

Step 3:  
Balancing Port Group 0 (again)

**Figure 4.5.** Port group rebalancing algorithm. Egress ports that are not a member of the port group are indicated by ‘--’. The last row of the matrix represents the aggregate traffic (Agg) on the member ports (from port counters). The row indicating background traffic (BG) is added for clarity and is not directly measured by Dahu.

#### 4.4.4 Load Balancing Algorithm

We now describe an algorithm for computing a balanced traffic allocation on egress ports. Based on the measured traffic, the switch generates a *port group traffic matrix* where the rows represent port groups and columns represent egress ports in the switch (see Figure 4.5). The elements in a row represent egress ports and the amount of traffic (bandwidth) that the port group is currently forwarding to those egress ports. If an egress port is not a member of the port group corresponding to the matrix row, then the respective matrix element is zeroed. Additionally, the *Aggregate utilization* row of elements stores the total bandwidth utilization on each egress port. This is the bandwidth based on the egress port counter, and accounts for traffic forwarded by any of the port groups, as well as background traffic on the port that is not re-routable using port groups, such as elephant flows pinned to the path by Hedera.

We first pick a port group in the matrix and try to balance the aggregate traffic for each of the member ports by moving traffic from this port group to different member ports where possible. To do this, Dahu computes the average aggregate utilization of all member ports of the port group. Then, it reassigns the traffic for that port group to equalize the aggregate traffic for the member ports to the extent possible. If a member port's aggregate traffic exceeds the average across all members, and the member port receives some traffic from this port group then we reassign the traffic to other member ports as appropriate. Dahu performs this operation for all port groups and repeats until convergence. To ensure convergence, we terminate the algorithm when subsequent iterations offload less than a small constant threshold  $\delta$ . Figure 4.5 shows the steps in the algorithm. Host facing ports in the switch can be ignored when executing this algorithm.



#### 4.4.5 Load Balancing Heuristic

The load balancing algorithm considers all physical ports and port groups in the switch and aims to balance the aggregate load on all of them to the extent possible. However, the algorithm may take many steps to converge for a large switch with many ports and port groups. We now describe a quick and practical heuristic to compute the balanced traffic allocation. The key idea behind the heuristic is to offload traffic from the highest loaded port to the least loaded port with which it shares membership in any of the port group, instead of trying to balance the aggregate load on all ports. By running the heuristic quickly, the switch can balance the port groups at time scales on the order of a few milliseconds.

1. Sort the physical ports by their aggregate utilization
2. Offload traffic from the highest loaded port H1 to the least loaded port with which it shares membership in any port group
3. Continue offloading traffic from H1 to the least loaded ports in order until they are completely balanced or H1 runs out of lesser loaded ports to offload to.

**Figure 4.6.** Load balancing heuristic

The heuristic, as described in Figure 4.6, is repeated for some fixed number  $R$  (say 16) of highest loaded switch ports, and has a low runtime of around 1ms. The runtime depends on the number of physical ports and port groups in the switch and is independent of the number of flows in the system. Our research grade implementation of the heuristic for our simulator running on a general purpose x86 CPU showed average runtimes of few 10's of microseconds to 0.5 milliseconds, even for large networks with over 130,000 servers. We believe an optimized version targeted at a switch ARM or PPC processor

can run within 1ms with a small DRAM requirement of under 10 MB. In the rest of this chapter, we employ this heuristic for load balancing.

#### 4.4.6 Fault Tolerance

Dahu relies on link-level techniques for fault detection, and uses existing protocols to propagate fault updates through the network. If a particular egress link or physical port  $P_f$  on the switch is down, the virtual ports in each port group which map to  $P_f$  are remapped to other member ports of the respective port groups. The remapping is performed by switch software and the actual policy could be as simple as redistributing the virtual ports uniformly to other egress ports or something more complicated.

On the other hand, when the switch receives fault notifications from the rest of the network, a specific egress port  $P_f$  may have to be disabled for only some destination prefixes because of downstream faults. We use the allowed port bitmaps technique described in Section 4.3.2 to just disable  $P_f$  for specific prefixes. The virtual port to physical port mappings in the port groups are left unchanged. In both scenarios, the only flows migrated to other egress ports are ones that were earlier mapped to the failed egress port  $P_f$ . When a physical port comes up, some virtual ports automatically get mapped to it the next time port groups are balanced.

### 4.5 Deployability

Deployability has been an important goal during the design of Dahu. In this section, we look at two primary requirements for adding Dahu support to switches: the logic to implement the functionality, and the memory requirements of the data structures.

To our knowledge, existing switch chips do not provide Dahu-like explicit hardware support for non-minimal routing in conjunction with dynamic traffic engineering. However, there are some similar efforts including Broadcom's resilient hashing fea-

ture [21] in their modern switch chips which is targeted at handling link failure and live topology updates, and the Group Table feature in the recent OpenFlow 1.1 Specification [69] which uses a layer of indirection in the switch datapath for multipath support. The increasing popularity of OpenFlow, software defined networks [49], and custom computing in the control plane (via embedded ARM style processors in modern switch silicon) indicates a new trend that we can leverage where large data centers operators are adopting the idea of a programmable control plane for the switches. The need for switch hardware modification to support customizable control plane for switches is no longer a barrier to innovation, as indicated by the deployment of switches with custom hardware by companies like Google [38].

To implement the hardware logic, we also need sufficient memory in the chip to support the state requirements for Dahu functionality. We now briefly estimate this overhead. Consider a large Dahu switch with 128 physical ports, 64 port groups with 1,024 virtual ports each, 16,384 prefixes in the routing table, and support for up to two different allowed port bitmaps for each prefix. The extra state required for all of Dahu's features is a modest 640 KB. Of this, 64 KB each are required for storing the virtual to physical port mappings for all the port groups, and the port group counters per egress port. 512 KB is required for storing two bitmaps for each destination prefix. A smaller 64 port switch would only need a total of 352 KB for a similar number of port groups and virtual ports. This memory may come at the expense of additional packet buffers (typically around 10 MB); however, recent trends in data center congestion management [7, 8] indicate that trading a small amount of buffer memory for more adaptive routing may be worthwhile.

## 4.6 Evaluation

We evaluated Dahu through flow-level simulations on both HyperX and Fat-tree topologies. Overall, our results show:

1. 10-50% throughput improvement in Fat-tree networks, and 250-500% improvement in HyperX networks compared to ECMP.
2. With an increase of only a single network hop, Dahu achieves significant improvements in throughput.
3. Dahu scales to large networks of over 130,000 nodes.
4. Dahu enables MPTCP to leverage non-shortest paths and achieve higher throughput with fewer subflows.

The evaluation seeks to provide an understanding of Dahu’s effect on throughput and hop count in different network topologies (HyperX and Fat-tree) under different traffic patterns. We first present a description of the simulator that we used for our experiments and the methodology for validating its accuracy. We simulate HyperX networks, large and small, and measure throughput as well as expected hop count for different workloads. We then move on to evaluate Dahu on an 8,192 host Fat-tree network using two communication patterns. We conclude this section by evaluating how MPTCP benefits from Dahu through the use of non-shortest paths.

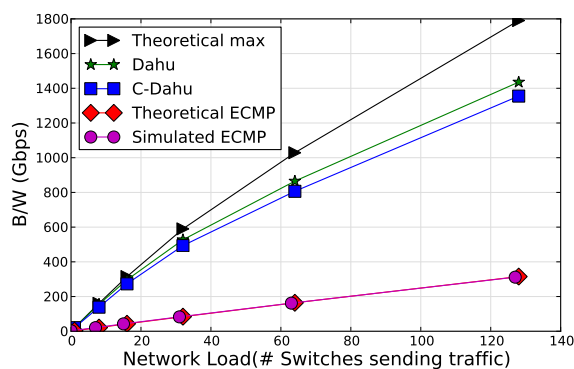
### 4.6.1 Simulator

We evaluated Dahu using a flow level network simulator that models the performance of TCP flows. We used the flow level simulator from Hedera [4], and added support for decentralized routing in each switch, port groups, allowed port bitmaps, and the load balancing heuristic. The Dahu-augmented Hedera simulator evaluates the AIMD

behavior of TCP flow bandwidths to calculate the total throughput achieved by flows in the network.

We built a workload generator that generates open-loop input traffic profiles for the simulator. It creates traffic profiles with different distributions of flow inter-arrival times and flow sizes. This allows us to evaluate Dahu’s performance over a wide range of traffic patterns, including those based on existing literature [16, 35]. Modeling the AIMD behavior of TCP flow bandwidth instead of per-packet behavior means that the simulator does not model TCP timeouts, retransmits, switch buffer occupancies and queuing delay in the network. The simulator only models unidirectional TCP data flows but not the reverse flow for ACKs. We believe this is justified, since the bandwidth consumed by ACKs is quite negligible compared to data. We chose to make these trade-offs in the simulator to evaluate at a large scale—over 130K servers, which would not have been possible otherwise.

We simulated five seconds of traffic in each experiment, and each switch rebalanced port groups (16 highest loaded ports) and recomputed prefix bitmaps every 10ms. For non-shortest path forwarding, switches used 80% of available capacity along shortest paths to the destination as the threshold utilization to dynamically enable non-shortest paths. These values were chosen based on empirical measurements.



**Figure 4.7.** Simulator throughput vs. theoretical maximum

**Simulator Validation:** To validate the throughput numbers reported by the simulator, we generated a range of traffic profiles with a large number of long-lived flows between random hosts in a ( $L=3, S=8, T=48$ ) HyperX network with 1Gbps links; ( $L, S, T$  defined in Section 4.4.1). We computed the theoretical maximum bandwidth achievable for the traffic patterns by formulating maximum multi-commodity network flow problems and solving them using the CPLEX [25] linear program solver—both for shortest path routing and non-minimal routing. We also ran our simulator on the same traffic profile.

As shown in Figure 4.7 the aggregate throughput reported by the simulator was within the theoretical maximum for all the traffic patterns that we validated. In case of shortest path forwarding, the theoretical and simulator numbers matched almost perfectly indicating that the ECMP implementation was valid. With non-minimal forwarding, the simulator’s performance is reasonably close to the theoretical limit. Note that the multi-commodity flow problem simply optimizes for the total network utilization whereas the simulator and TCP in general, also take fairness into account.

In addition, we also explicitly computed the max-min fair flow bandwidths for these traffic profiles using the water-filling algorithm [18]. We compared the resulting aggregate throughput to those reported by the simulator. For all evaluated traffic patterns, the simulator throughput was within 10% of those reported by the max-min validator. This small difference is because the TCP’s AIMD congestion control mechanism only yields approximate max-min fairness in flow bandwidths whereas the validator computes a perfectly max-min fair distribution.

## 4.6.2 HyperX Networks

We first evaluate Dahu with HyperX networks which have many paths of differing lengths between any source and destination. Dahu’s non-shortest path We simulate a

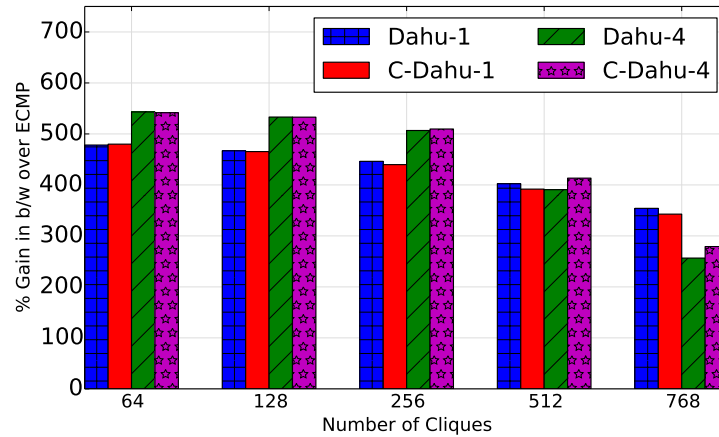
( $L=3, S=14, T=48$ ) HyperX network with 1Gbps links, as described in [1]. This models a large data center with 131,712 servers, interconnected by 2,744 switches, and an oversubscription ratio of 1:8.

We seek to measure how Dahu's non-minimal routing and load balancing affect performance as we vary traffic patterns, the maximum derouting threshold, and non-minimal routing scheme (constrained or not). We run simulations with Clique and Mixed traffic patterns (described next), and compare the throughput, average hop count and link utilizations for Dahu and ECMP. In the graphs, Dahu- $n$  refers to Dahu routing with at most  $n$  deroutings. C-Dahu- $n$  refers to the similar Constrained Dahu routing variant.

### **Clique Traffic Pattern**

A *Clique* is a subset of switches and associated hosts that communicate among themselves; each host communicates with every other host in its clique over time. This represents distributed jobs in a data center which are usually run on a subset of the server pool. A typical job runs on a few racks of servers. There could be multiple cliques (or jobs) running in different parts of the network. We parameterize this traffic pattern by i) clique size, the number of switches in the clique, and ii) total number of cliques in the network. In this experiment, we vary the total number of cliques from 64 to 768, keeping the clique size fixed at 2 switches (96 servers). Each source switch in a clique generates 18Gbps of traffic with 1.5 MB average flow size.

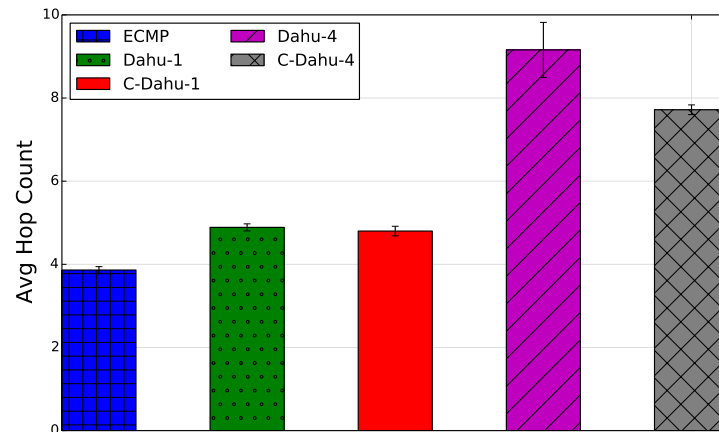
**Bandwidth:** Figure 4.8 shows the bandwidth gains with Dahu relative to ECMP as we vary the number of communicating cliques. Dahu offers substantial gains of 400-500% over ECMP. The performance gain is highest with a smaller number of cliques, showing that indeed derouting and non-shortest path forwarding can effectively take advantage of excess bandwidth in HyperX networks. This validates a major goal of this work, which is improving the statistical multiplexing of bandwidth in direct network



**Figure 4.8.** Throughput gain with Clique traffic pattern

topologies. As the number of cliques increases, the bandwidth slack in the network decreases, and the relative benefit of non-minimal routing comes down to around 250%. Dahu and constrained Dahu have similar performance for the same derouting threshold.

We further find that a large derouting threshold provides larger benefit with less load, since there are many utilized on average, bandwidth slack reduces, and a derouting threshold of one starts performing better.

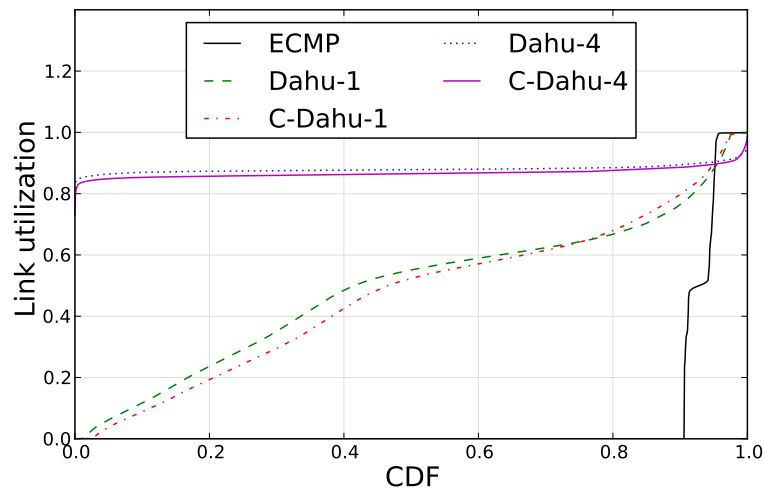


**Figure 4.9.** Average hop count with Clique traffic pattern

**Hop count:** Beyond raw throughput, latency in an important performance metric that is related to network hop count. Figure 4.9 shows the average hop count for each

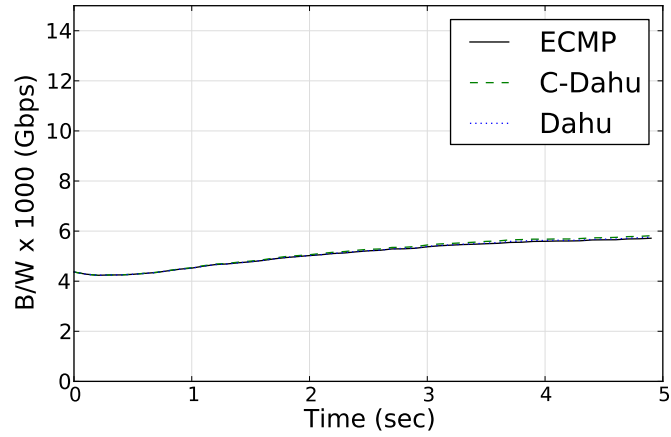


routing scheme. Dahu delivers significantly higher bandwidth with a small increase in average hop count. Average hop count increases with increase in derouting threshold. For smaller derouting threshold, the hop count is similar to that of ECMP while still achieving most of the bandwidth improvements of non-minimal routing. Note that the small error bars indicate that the average hop count is similar while varying the number of cliques.

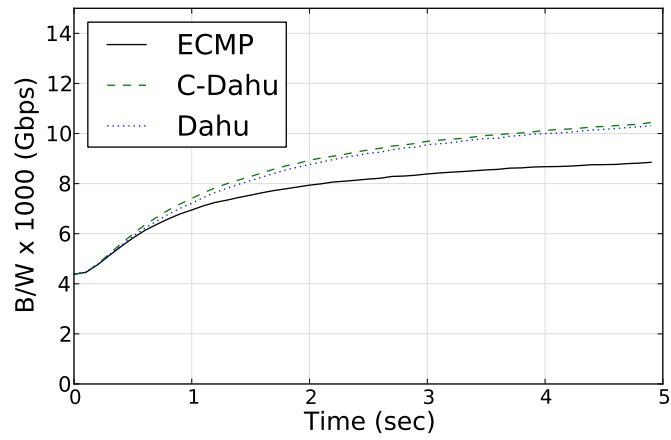


**Figure 4.10.** Link utilization with Clique traffic pattern (number of cliques = 512)

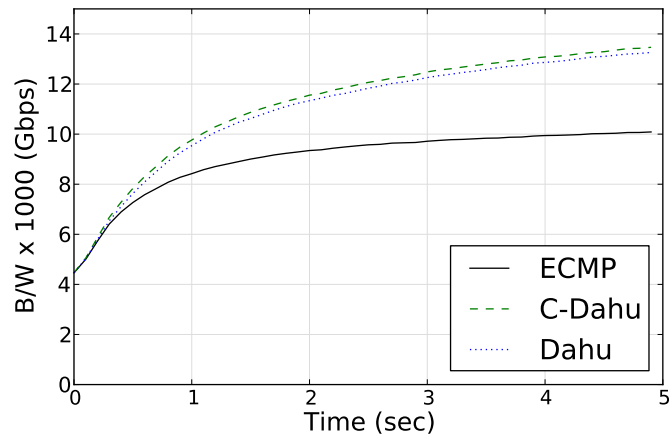
**Link utilization:** Figure 4.10 shows the CDF of inter-switch link utilizations for ECMP and Dahu for the experiment with 512 cliques. With shortest path routing, 90% of the links have zero utilization, whereas Dahu achieves its bandwidth gains by utilizing available capacity on additional links in the network. Also, we see that a single derouting can achieve most of the overall bandwidth gains while consuming bandwidth on significantly fewer links in the network thereby sparing network capacity for more traffic.



(a) Load = 4.5Gbps/switch



(b) Load = 17.5Gbps/switch



(c) Load = 33.5Gbps/switch

**Figure 4.11.** Dahu performance with Mixed Traffic Pattern

### Mixed Traffic Pattern

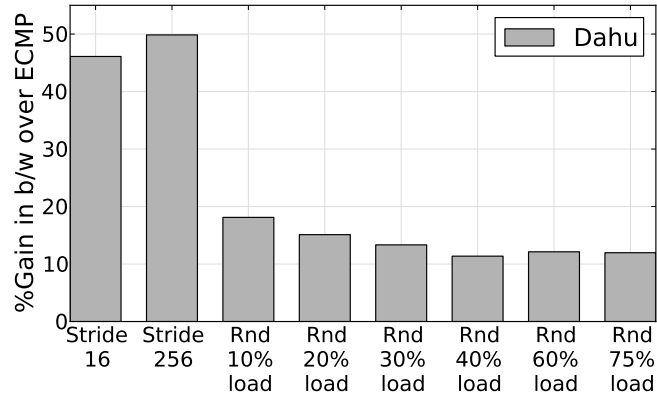
The “mixed traffic pattern” represents an environment with a few *hot racks* that send lot of traffic, representing jobs like data backup. For this traffic pattern, we simulate 50 cliques with 10 switches in each. Every switch acts as a network hot-spot and has flows to other members of is varied from 3Gbps to 32Gbps. We also generate random all-to-all traffic between all the hosts in the network. This background traffic creates an additional load of 1.5Gbps per source switch with average flow size of 200 KB.

Figure 4.11 shows that for low load levels (4.5Gbps total load per switch) ECMP paths are sufficient to fulfill demand. As expected, total bandwidth achieved is same for both ECMP and Dahu. However, at high load (17.5Gbps and 33.5Gbps per switch) Dahu performs significantly better than ECMP by utilizing available slack bandwidth.

### 4.6.3 Fat-Tree Networks

To illustrate Dahu’s generality, we evaluate it in the context of a Fat-tree topology. Fat-trees, unlike HyperX, have a large number of shortest paths between a source and destination, so this evaluation focuses on Dahu’s load balancing behavior, rather than its use of non-shortest paths. We compare Dahu with ECMP, with hosts communicating over long lived flows in a  $k = 32$  Fat-tree (8,192 hosts). We consider these traffic patterns: (1) *Stride*: With  $n$  total hosts and stride length  $x$ , each host  $i$  sends traffic to host  $(i + x) \bmod n$ . (2) *Random*: Each host communicates with another randomly chosen destination host in the network. To study the effect of varying overall network load, we pick a subset of edge switches that send traffic to others and vary the number of hosts on each of these edge switches that originate traffic.

Figure 4.12 shows that Dahu achieves close to 50% improvement with stride traffic patterns. The load balancing heuristic local hash imbalances and improves total throughput. For random traffic patterns, Dahu outperforms ECMP by 10-20%. Overall,



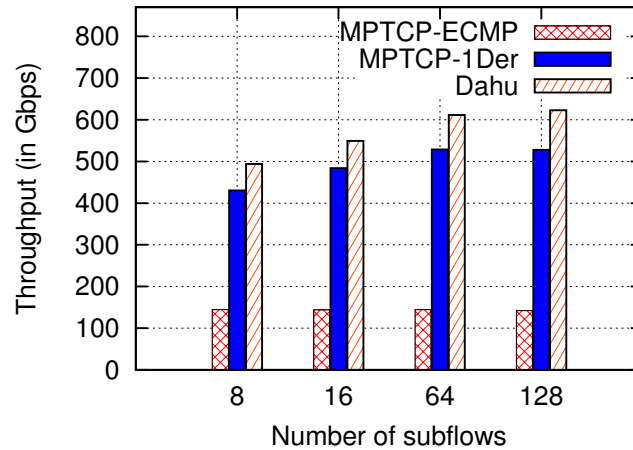
**Figure 4.12.** Throughput gain for  $k = 32$  Fat-tree with Stride and Random (Rnd) traffic patterns

Dahu is better able to utilize network links in Fat-tree networks than ECMP, even when only shortest-path links are used.

#### 4.6.4 MPTCP in HyperX Networks

MPTCP is a recent host-based transport layer solution for traffic engineering [75]. MPTCP relies on splitting each flow into multiple subflows that take different paths through the network, and modulates the amount of data transmitted on each subflow based on congestion, thus improving the network utilization. In this section, we evaluate how MPTCP benefits from Dahu non-shortest path routing, and the additional improvements achieved using Dahu dynamic load balancing.

We extended the *htsim* packet level simulator [39] (used to evaluate MPTCP in [75]), to simulate a ( $L=3, S=10, T=20$ ) HyperX network with 100 Mbps links. This network has 1000 switches, 20,000 hosts and an oversubscription ratio of 1:4. We chose a smaller topology and lower link speed due to the higher computational overhead of packet level simulations. We generated a random permutation matrix (without replacement), and selected a subset of source-destination pairs to create long lived flows, with 50% of total hosts sending traffic. To evaluate the impact of non-shortest path routing, we ran MPTCP



**Figure 4.13.** MPTCP-ECMP, MPTCP-1Der, and Dahu performance for  $L=3$ ,  $S=10$ ,  $T=20$  HyperX topology. Results obtained from packet level simulations for MPTCP and flow level simulations for Dahu.

under two scenarios: (1) ECMP style shortest-path routing (MPTCP-ECMP), and (2) Dahu-style non-shortest path routing with one allowed deroute but no load balancing (MPTCP-1Der). To understand the additional impact of Dahu’s load balancing, we also ran the Dahu simulator on the same topology and traffic pattern by treating each subflow as an independent TCP flow.

Since we used a packet level simulator for MPTCP and a flow level simulator for Dahu, we also validated that the two simulators reported comparable throughput results under identical scenarios [74, pg.12].

Figure 4.13 shows that Dahu’s non-shortest path routing unlocks 300% more bandwidth compared to ECMP. With MPTCP-1Der, throughput increases with the number of subflows, indicating that in order to effectively leverage the large path diversity in direct connect networks, MPTCP needs to generate a large number of subflows, making it unsuitable for short flows. Dahu, on the other hand, is able to achieve a similar throughput with 8 subflows that MPTCP-1Der achieves with 64 or 128 subflows, and can also handle short flows with efficient hash rebalancing. At the transport layer, MPTCP has no way of

distinguishing between shortest and non-shortest paths and can leverage Dahu for better route selection. These results indicate that Dahu effectively enables MPTCP to leverage non-shortest paths, and achieve much better network utilization in direct networks with fewer subflows.

## 4.7 Discussion

As seen in Section 4.6, Dahu exploits non-minimal routing to derive large benefits over ECMP for different topologies and varying communication patterns. Yet, there is a scenario where non-minimal routing can be detrimental. This occurs when the network as a whole is highly saturated; shortest path forwarding itself does well as most links have sufficient traffic and there is no “unused” capacity or slack in the network. With Dahu, a derouted flow consumes bandwidth on more links than if it had used just shortest paths, thereby contributing to congestion on more links. In large data centers, this network saturation scenario is uncommon. Networks have lower average utilization although there may be hot-spots or small cliques of racks with lot of communication between them. Usually, there is network slack or unused capacity that Dahu can leverage. The network saturation case can be dealt with in many ways. For example, a centralized monitoring infrastructure can periodically check if a large fraction of the network is in its saturation regime and notify switches to stop using non-minimal paths.

Alternatively, a simple refinement to the localized load balancing scheme can be used which relies on congestion feedback from neighboring switches to fall back to shortest path forwarding in such high load scenarios. Network packets are modified to store 1 bit in the IP header which is updated by each switch along the path of a packet to indicate whether the switch used a shortest path egress port or derouted the packet. A switch receiving a packet checks if two conditions are satisfied: (1) It doesn't have enough capacity to the destination along shortest paths alone, and (2) The previous hop

derouted the packet. If both conditions are satisfied, it sends congestion feedback to the previous hop notifying it to stop sending derouted traffic through this path for the particular destination prefix, for a certain duration of time (say 5ms). This solution is discussed further in [74].

## 4.8 Related Work

There have been many recent proposals for scale-out multipath data center topologies such as Clos networks [3, 35, 54], direct networks like HyperX [1], Flattened Butterfly [47], DragonFly [48], and even randomly connected topologies proposed in Jellyfish [89]. Many current proposals use ECMP-based techniques which are inadequate to utilize all paths, or to dynamically load balance traffic. Routing proposals for these networks are limited to shortest path routing (or K-shortest path routing with Jellyfish) and end up under-utilizing the network, more so in the presence of failures. While DAL routing [1] allows deroutes, it is limited to HyperX topologies. In contrast, Dahu proposes a topology-independent, deployable solution for non-minimal routing that eliminates routing loops, routes around failures, and achieves high network utilization.

Hedera [4] and MicroTE [17] propose a centralized controller to schedule long lived flows on globally optimal paths. However they operate on longer time scales and scaling them to large networks with many flows is challenging. While DevoFlow [27] improves the scalability through switch hardware changes, it does not support non-minimal routing or dynamic hashing. Dahu can co-exist with such techniques to better handle congestion at finer time scales.

MPTCP [75] proposes a host based approach for multipath load balancing, by splitting a flow into multiple subflows and modulating how much data is sent over different subflows based on congestion. However, as a transport protocol, it does not have control over the network paths taken by subflows. Dahu exposes the path diversity

to MPTCP and enables MPTCP to efficiently utilize the non-shortest paths in a direct connect network. There have also been proposals that employ variants of switch-local per-packet traffic splitting [29,102]. With Dahu, instead of per-packet splitting, we locally rebalance flow aggregates across different paths thereby largely reducing in-network packet reordering.

Traffic engineering has been well studied in the context of wide area networks. TeXCP [45], MATE [32], and REPLEX [33] split flows on different paths based on load, however their long control loops make them inapplicable in the data center context which requires faster response times to deal with short flows and dynamic traffic changes. FLARE [90] exploits the inherent burstiness in TCP flows to schedule “flowlets” (bursts of packets) on different paths to reduce extensive packet reordering.

Finally, a key distinction between Dahu and the related traffic engineering approaches is that Dahu actively routes over non-shortest paths in order to satisfy traffic demand. Dahu decouples non-minimal routing and its mechanism for more balanced hashing and offers a more flexible architecture for better network utilization in direct connect networks.

## 4.9 Summary

Existing solutions for leveraging multipath in the data center rely on ECMP which is insufficient due to its static nature and inability to extend beyond shortest path routing. We present a new switch mechanism, Dahu, that enables dynamic hashing of traffic onto different network paths. Dahu exploits non-shortest path forwarding to reduce congestion while preventing persistent forwarding loops using novel switch hardware primitives and control software. We present a decentralized load balancing heuristic that makes quick, local decisions to mitigate congestion, and show the feasibility of proposed switch hardware modifications. We evaluate Dahu using a simulator for different topologies and



different traffic patterns and show that it significantly outperforms shortest path routing and complements MPTCP performance by selecting good paths for hashing subflows.

## **4.10 Acknowledgments**

Chapter 4, in part, contains material as it appears in the Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13), San Jose, CA, October 2013. “Dahu: Commodity Switches for Direct Connect Data Center Networks”. Sivasankar Radhakrishnan, Malveeka Tewari, Rishi Kapoor, George Porter, and Amin Vahdat. The dissertation author was the primary investigator and author of this paper.

## Chapter 5

# SENIC: Scalable NIC for End-Host Rate Limiting

Rate limiting is an important primitive for managing server network resources. Unfortunately, software-based rate limiting suffers from limited accuracy and high CPU overhead, and modern NICs only support a handful of rate limiters. In this chapter, we present SENIC, a NIC design that can natively support 10s of thousands of rate limiters—100x to 1000x the number available in NICs today. The key idea is that the host CPU only classifies packets, enqueues them in per-class queues in host memory, and specifies rate limits for each traffic class. On the NIC, SENIC maintains class metadata, computes the transmit schedule, and only pulls packets from host memory when they are ready to be transmitted (on a real time basis). We implemented SENIC on NetFPGA, with 1000 rate limiters requiring just 30KB SRAM, and it was able to accurately pace packets. Further, in a memcached benchmark against software rate limiters, SENIC is able to sustain up to 250% higher load, while simultaneously keeping tail latency under 4ms at 90% network utilization.

## 5.1 Introduction

Today's trend towards consolidating servers in dense data centers necessitates careful resource management. It is hence unsurprising that there have been several recent proposals to manage and allocate data center network bandwidth to different services, tenants, and traffic flows. This can be a challenge given the bursty and unpredictable nature of data center traffic, which has necessitated new designs for congestion control [72].

Many of these recent proposals can be realized on top of a simple substrate of *programmable rate limiters*. For example, Seawall [87], Oktopus [11], EyeQ [44] and Gatekeeper [83] use rate limiters between pairs of communicating virtual machines to provide tenant rate guarantees. QCN [6] and D<sup>3</sup> [100] use explicit network feedback to rate limit traffic sources. Such systems need to support thousands of rate limited flows or traffic classes, especially in virtual machine deployments.

Unfortunately these new ideas have been hamstrung by the inability of current NIC hardware to support more than a handful of rate limiters (e.g., 8–128) [40, 58]. This has resulted in delegating packet scheduling functionality to software, which is unable to keep up with line rates, while diverting CPU resources away from application processing. As networks get faster, this problem will only get worse since the capabilities of individual cores will likely not increase. We are left with a compromise between precise hardware rate limiters that are few in number [51, 87] and software rate limiters that support more flows but suffer from high CPU overhead and burstiness (see Table 5.1). Software rate limiters also preclude VMs from bypassing the hypervisor for better performance [61, 65].

The NIC is an ideal place to offload common case or repetitive network functions. Features such as segmentation offload (TSO), and checksum offload are widely used to improve CPU performance as we scale communication rates. However, a key missing functionality is scalable rate limiting.

**Table 5.1.** Pros and cons of current hardware and software approaches to rate limiting

Property	Hardware	Software
Scales to many classes	×	✓
Works at high link speeds	✓	×
Low CPU overhead	✓	×
Precise rate enforcement	✓	×
Supports hypervisor bypass	✓	×

In this work, we present SENIC, a NIC architecture that combines the scalability of software rate limiters with the precision and low overhead of hardware rate limiters. Specifically, in hardware, SENIC supports 10s of thousands of rate limiters, 100–1000x the number available in today’s NICs. The key insight in SENIC is to invert the current duties of the host and the NIC: the OS stores packet queues in host memory, and classifies packets into them. The NIC handles packet scheduling and proactively pulls packets via host memory DMA for transmission. This late-binding enables SENIC to maintain transmit queues for many classes in host memory, while the NIC enforces precise rate limits in real-time.

Our contributions are: (1) identifying the limitations of current operating system and NIC capabilities, (2) the SENIC design that provides scalable rate limiting with low CPU overhead, and supports hypervisor bypass, (3) a unified scheduling algorithm that enforces strict rate limits and gracefully falls back to weighted sharing if the link is oversubscribed, and (4) evaluating SENIC through implementation of a software prototype and a hardware 10G-NetFPGA prototype. Our evaluation shows that SENIC can pull packets on-demand and achieve (nearly) perfect packet pacing. SENIC sustains 43–250% higher memcached load than current software rate limiters, and achieves low tail latency under 4ms even at high loads. SENIC isolates memcached from bandwidth intensive tenants, and sustains the configured rate limits for all tenants even at high loads (9Gb/s), unlike current approaches.

## 5.2 Motivation

We motivate SENIC by describing two capabilities which rely on scalable rate limiting, then describe the limitations of current NICs which prevent these capabilities from being realized.

### 5.2.1 The Need For Scalable Rate Limiting

Scalable rate limiting is required for network virtualization as well as new approaches for data center congestion control, as we now describe.

**Network Virtualization:** Sharing network bandwidth often relies on hierarchical rate limiting and weighted bandwidth sharing. For example, Gatekeeper [83], and EyeQ [44] both rate limit traffic between every communicating source-destination VM pair, as well as use weighted sharing across source VMs on a single machine. With greater server consolidation and increasing number of cores per server, the number of rate limiters needed is only expected to increase.

To quantify the number of rate limiters required for network virtualization, we observe that Moshref et al. [63] cite the need for 10s of thousands of flow rules per server to support VM-to-VM rules in a cluster with 10s of thousands of servers. Extending these to support rate limits would thus necessitate an equal number of rate limiters. For example, if there are 50 VMs/server, each communicating with a modest 50 other VMs, we need 2500 rate limiters to provide bandwidth isolation. Furthermore, supporting native hardware rate limiting is necessary, since VMs with latency sensitive applications may want to bypass the hypervisor entirely [61, 65].

**Data Center Congestion Control:** Congestion control has typically been an end-host responsibility, as exemplified by TCP. Bursty correlated traffic at high link speeds, coupled with small buffers in commodity switches can result in poor application

performance [72]. This has led to the development of QCN [6], DCTCP [7], HULL [8], and D<sup>3</sup> [100] to demonstrate how explicit network feedback can be used to pace or rate limit traffic sources and reduce congestion. In the limit, each flow (potentially thousands [16]) needs its own rate limiter.

## 5.2.2 Limitations of Current Systems

Today, rate limiting is performed either (1) in hardware in the NIC, or (2) in software in the OS or VM hypervisor. We consider these alternatives in detail.

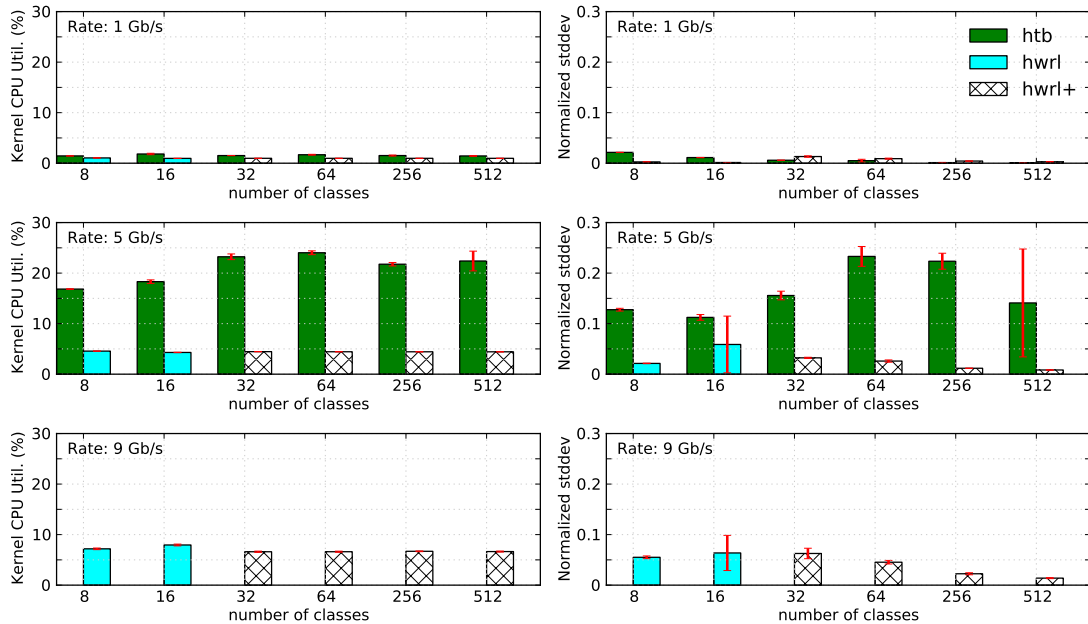
### Hardware Rate Limiting

Modern NICs support a few hardware transmit queues (8-128) that can be rate limited. When the OS transmits a packet, it sends a doorbell request<sup>1</sup> to the NIC notifying it of the packet and the NIC Tx ring buffer to use. The NIC DMA's the packet descriptor from host RAM to its internal SRAM memory. The NIC uses an arbiter to compute the order in which to fetch packets from different Tx ring buffers. It looks up the physical address of the packet in the descriptor, and initiates a DMA transfer of the packet contents to its internal packet buffer. Eventually a scheduler decides when different packets are transmitted.

A straightforward approach of storing per-class packet queues on the NIC does not scale well. For instance, even storing 15KB packet data per queue for 10,000 queues requires around 150MB of SRAM, which is too expensive for commodity NICs. Likewise, storing large packet descriptor ring buffers for each queue is also expensive.

---

<sup>1</sup>A doorbell request is a mechanism whereby the network driver notifies the NIC that packet(s) are ready to be transmitted.



**Figure 5.1.** Comparison of CPU overhead and accuracy of software (Linux htb) and hardware (hwrl, hwrl+) rate limiting. At high rates (5Gb/s and 9Gb/s), hwrl ensures low CPU overhead and high accuracy, while htb is unable to drive more than 6.5Gb/s of aggregate throughput. Accuracy is measured as the ratio between the standard deviation of successive packet departure time differences, to the ideal. For instance, at 0.5Gb/s, 1500B packets should depart at times roughly 24us apart, but a “normalized stddev” of 0.2 means the observed deviation from 24us was as much as  $\sim 4.8$ us.

### Software Rate Limiting

Operating systems and VM hypervisors support rate limiting and per-class prioritization; for example, Linux offers a configurable queueing discipline (QDisc) layer for enforcing packet transmission policies. The QDisc can be configured with traffic classes from which packets are transmitted by the operating system.

In general, handling individual packets in software imposes high CPU overhead due to lock contention and frequent interrupts for computing and enforcing the schedule. To reduce CPU load, the OS transfers packets to the NIC in batches, leveraging features like TSO. Once these batches of packets are in the NIC, the operating system loses control over packet schedules; packets may end up being transmitted at unpredictable times

on the wire, frequently in large bursts (e.g., 64KB with 10Gb/s NICs) of back-to-back MTU-sized packets transmitted at the full line rate.

**Quantifying Software Overheads:** Accurate rate limiting is challenging at 10Gb/s and higher. For instance, at 40Gb/s, accurately pacing 1500B packets means sending a packet approximately every 300ns. Such accuracy is difficult to achieve even with Linux’s high resolution timers, as servicing an interrupt can easily cost thousands of nanoseconds. To quantify the overhead of software rate limiting, we benchmarked Linux’s Hierarchical Token Bucket (htb), and compared it to the hardware rate limiter (hwrl) on an Intel 82599 NIC. The tests were conducted on a dual 4-core, 2-way hyperthreaded Intel Xeon E5520 2.27GHz server running Linux 3.6.6.

We use userspace UDP traffic generators to send 1500B packets, and compare htb and hwrl on two metrics—OS overhead and accuracy—for varying number of classes. Each class is allocated an equal rate (total rate is 1Gb/s, 5Gb/s, or 9Gb/s). When the number of classes exceeds the available hardware rate limiters (16 in our setup), we assign classes to them in a round robin fashion (shown as hwrl+). OS overhead is the total fraction of CPU time spent in the kernel across all cores, and includes overheads in the network stack, packet scheduling, and servicing interrupts. To measure how well traffic is paced, we use a hardware packet sniffer at the receiver, which records timestamps with a 500ns precision. These metrics are plotted in Figure 5.1; the shaded bars indicate that many classes are mapped to one hardware rate limiter (hwrl+).

These experiments show that implementations of rate limiting in hardware are promising and deliver accurate rate limiting at low CPU overheads. However, they only offer few rate limiters, in part due to limited buffering on the NIC. Figure 5.1 shows that htb, while scalable in terms of the number of queues supported, is unable to pace packets at 9Gb/s, resulting in inaccurate rates.



## 5.3 Design

In the previous section, we described limitations of today’s software and hardware approaches to rate limiting. The primary limitation in hardware today is scalability on the transmit path; we do not modify the receive path. In light of this, we now describe the design of the basic features in SENIC, and defer more advanced NIC features to Section 5.5. We begin with the service model abstraction.

### 5.3.1 Service Model

SENIC has a simple service model. The NIC exposes multiple transmit queues (classes), each with an associated rate limit. When the sum of rate limits of active classes does not exceed link capacity, each class is restricted to its rate limit. When it exceeds link capacity (i.e., the link is oversubscribed), SENIC gracefully shares the capacity in the ratio of class rate limits.

### 5.3.2 CPU and NIC Responsibilities

To enforce a service model, we need a packet scheduler, and must store state for all classes. The state and functionality are spread across the CPU/host and the NIC.

**State:** Memory on the NIC (typically SRAM) is expensive, and we therefore use it to only store metadata about the classes. To store packet queues, SENIC leverages the large amount of host memory. Table 5.2 shows an example class metadata structure; the total size for storing 10,000 classes is about 300kB of SRAM. Note that the Myricom 10Gb/s NIC has 2MB SRAM [64].

**Functionality:** At a high level, the CPU classifies and enqueues packets in transmit queues, while the NIC computes a schedule that obeys the rate limits, pulls packets from queues in host memory using DMA, and transmits them on to the wire. The NIC handles all real time per-packet operations and transmit scheduling of packets

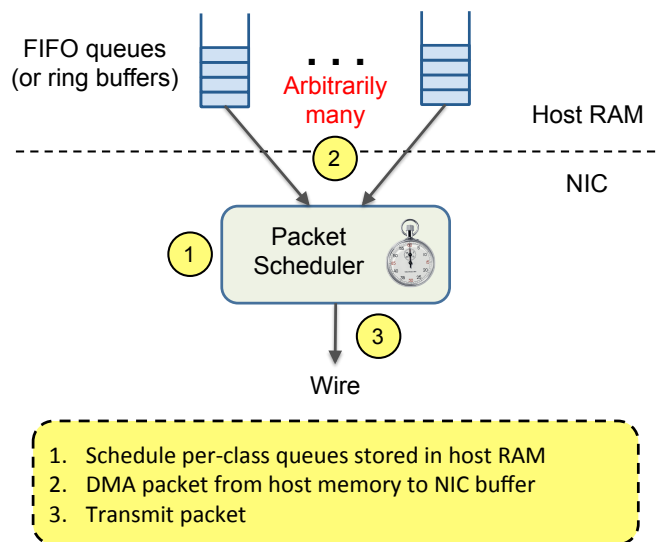
**Table 5.2.** Per-class metadata in NIC SRAM. Total size = 30B

Entry	Bytes	Description
<i>Queue management</i>		
ring_buffer	4	Aligned address of the head of the ring buffer
buffer_size	2	Size of ring buffer (entries)
head_index	2	Index of first packet
tail_index	2	Index of last packet
<i>Head packet descriptor</i>		
head_paddr	8	Address of the first packet
head_plen	2	Length of the first packet (B)
pkt_offset	2	Next segment offset into the packet (for TSO)
<i>Scheduler state (say for token bucket scheduler)</i>		
rate_mbps	2	Rate limit for the queue
tokens_bytes	2	Number of bytes that can be sent from the queue without violating rate limit
timestamp	4	Last timestamp at which tokens were refreshed

from different classes based on their rate limits. This frees up the CPU to batch network processing, which reduces overall CPU utilization. This architecture is illustrated in Figure 5.2, which we now describe in detail.

### CPU Functionality

As in current systems, the OS manages the NIC and initializes the device, creates/deletes classes, and configures rate limits. The OS is also in charge of classifying and enqueueing packets in appropriate queues. In both cases, the OS communicates with the NIC through memory-mapped IO. For instance, when the OS enqueues a packet (or a burst of packets) into a queue, it notifies the NIC through a special doorbell request that it writes to a device-specific memory address.



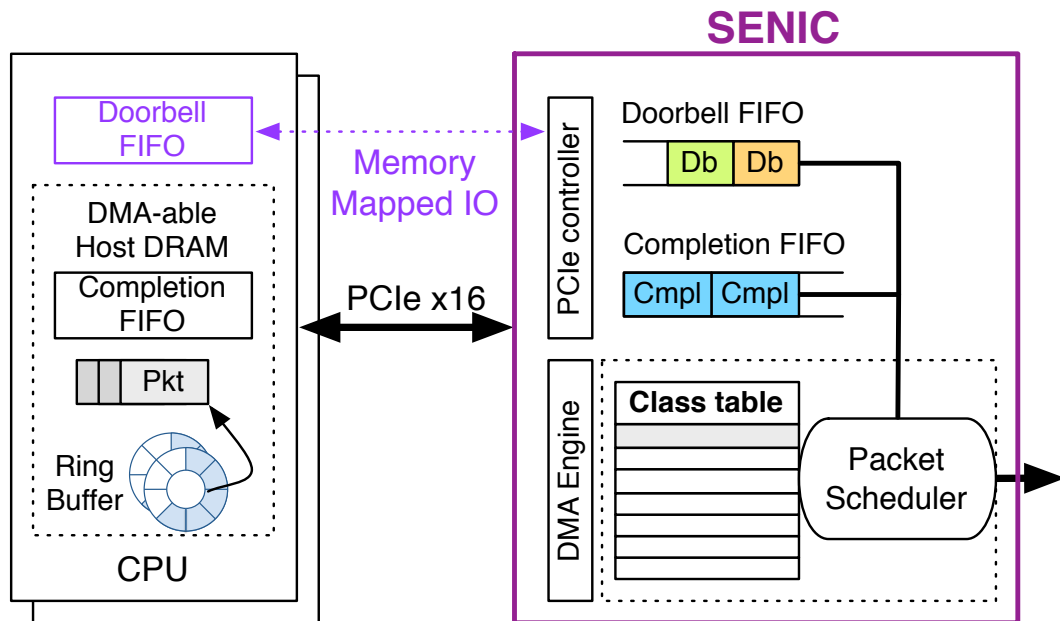
**Figure 5.2.** SENIC— “Schedule and Pull” model

### NIC Functionality

The NIC is responsible for all per-packet real time operations on transmit queues. Since it has limited hardware buffer resources, the NIC first computes the transmit schedule based on the rate limits. It then chooses the next packet that should be transmitted, and DMA's the packet from the per-class queue in host memory to a small internal NIC buffer for transmitting on to the wire. Figure 5.3 shows a schematic of the SENIC hardware and related interfaces from software.

**Metadata:** The NIC maintains state about traffic classes to enforce rate limits. In the case of a token bucket scheduler, each class maintains metadata on the number of tokens, and the global state is a list of active classes with enough tokens to transmit the next packet. The memory footprint is small, easily supporting 10,000 or more traffic classes with a few 100kB of metadata.

**Scheduling:** The NIC schedules and pulls packets from host memory on demand at link speed. Packets are not pulled faster, even though PCIe bandwidth between the NIC and CPU is much higher. This late binding reduces the size of NIC hardware buffers



**Figure 5.3.** SENIC hardware design. Once the NIC DMA's a packet from host memory, there is further processing (e.g. checksum offloads) before the packet is transmitted on the wire. This chapter focuses on the scheduler and the NIC interaction with the operating system and software stack.

required for storing packets. It also avoids head-of-line blocking, and allows the NIC to quickly schedule newly active classes or use updated rate limits. This offloading of scheduling and real time work to the NIC is what enables SENIC to accurately enforce rate limits even at high link speeds.

**Other Functionality:** The NIC does more tasks than just state management and rate limiting. After the packet is DMA'd onto NIC memory, there is a standard pipeline of operations that we leave unmodified. For instance, NICs support TCP and IP checksum offloading, VLAN encapsulation, and send completions to notify the CPU when it can reclaim packet memory.

## 5.4 Packet Scheduling in SENIC

SENIC employs an internal scheduler to rate limit traffic classes. The task of packet scheduling can be realized using a number of algorithms such as Deficit Round Robin (DRR) [88], Weighted Fair Queueing (WFQ) [28], Worst-case Fair weighted Fair Queueing (WF<sup>2</sup>Q) [15], or simple token buckets. The choice of algorithm impacts the sharing model, and packet delay bounds. For instance, token buckets support rate limits, but DRR is work-conserving; simply arbitrating across token buckets in a DRR-like fashion can result in bursty transmissions [14].

In this section, we start with our main requirements to pick the appropriate scheduling algorithm. We desire hierarchical rate limits, so the above work-conserving algorithms (DRR, WFQ, etc.) do not directly suit our needs. We now describe a unified scheduling algorithm that supports hierarchies and rate limits.

### 5.4.1 SENIC Packet Scheduling Algorithm

Recall that the service model exposed by SENIC is rate limits on classes, with fallback to weighted sharing proportional to the class rates. We begin by describing a scheduling algorithm which can enforce this service model. We leverage a virtual time based weighted sharing algorithm, WF<sup>2</sup>Q+ [14], and modify its *system virtual time* ( $V$ ) computation to support strict rate limiting with a fallback to weighted sharing. The algorithm computes a *start* ( $S$ ) and *finish* ( $F$ ) time for every packet based on the class rate  $w_i$ . Packets with  $S \leq V$  are considered *eligible*, and the algorithm transmits eligible packets in increasing order of their finish times.

**Computing Start and Finish Time:** Since each class is a FIFO, the start and finish times are maintained only for the packets at the head of each transmit queue. The start time  $S_i$  of a class  $C_i$  is only updated when a packet is dequeued from that class or a

packet is enqueued into a previously empty class. The finish time  $F_i$  is updated whenever  $S_i$  is updated.  $S_i$  and  $F_i$  for each flow  $C_i$  are computed in the same way as in WF<sup>2</sup>Q+, as follows:

$$S_i = \begin{cases} \max(F_i, V_{enq}) & \text{on enqueue into empty queue} \\ F_i & \text{on dequeue} \end{cases}$$

$$F_i = S_i + \frac{L}{w_i}$$

where  $V_{enq}$  is the *system time*  $V$  (described below) when the packet is enqueued, and  $L$  is the head packet's length.

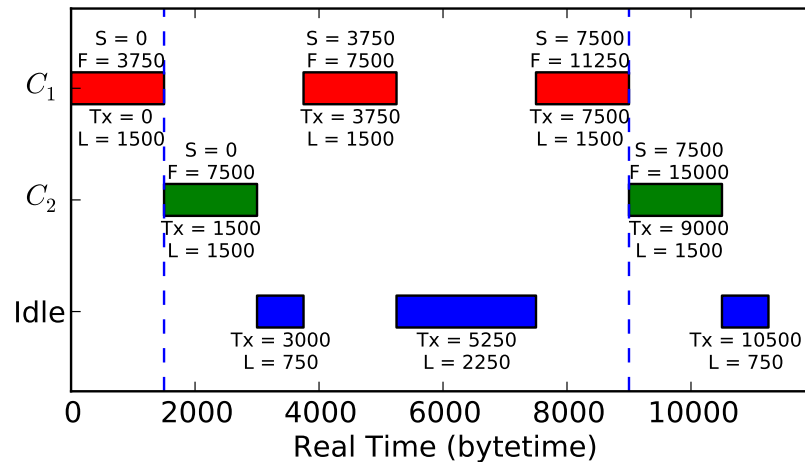
**System Time Computation:** WF<sup>2</sup>Q+ computes a work-conserving schedule where at least one class is always eligible to transmit data. To enforce strict rate limits, SENIC incorporates the notions of real time and the link drain rate ( $R$ ) to compute the transmit schedule. The system time is increased by 1 unit (*bytetime*), in the time it takes to transmit 1B of data at link speed, and thus incorporates the link's known drain rate  $R$  (e.g. 10Gb/s).

SENIC supports graceful fallback to weighted sharing when the link is oversubscribed. When the link is oversubscribed, we slow the system time  $V$  down to reflect the marginal rate at which the active flows are serviced. Without loss of generality, let the rate limits of flows  $C_i$  be represented as fractions  $w_i$  of the link speed  $R$ . We define the *rate oversubscription factor*  $\phi$  to be the sum of rate limits (weights) of currently backlogged classes or flows in the system;  $\phi = 0$  when no flows are active. The scheduler modifies the system time  $V$  to slow down by the rate oversubscription factor and proceed at most as fast as the link speed.  $V$  is computed as:

$$V(0) = 0$$

$$V(t + \tau) = V(t) + R\tau \times \max(1, \phi)$$

where  $\tau$  is a single packet transmission period, or contiguous link idle period, or the period between successive updates to  $\phi$ . Given the system time, the start and finish times of all classes, we schedule packets in the same order as  $WF^2Q+$ , i.e. in order of increasing finish times among all eligible classes at the time of dequeuing.



**Figure 5.4.** Transmit schedule example. Link is not oversubscribed. The interval between the vertical dashed lines indicates repeating sequence in the transmit schedule (only 1 repetition shown for clarity).  $S$ ,  $F$ ,  $L$  as defined in the text.  $T_x$  is the time when a particular packet transmission or idle period starts.

**Example:** We now look at an example transmit schedule computed using these time functions. Assume a 10Gb/s link with two continuously backlogged classes  $C_1$  and  $C_2$  (with rate limits 4Gb/s and 2Gb/s respectively). The transmit schedule is shown in Figure 5.4. The values of  $S_i$  and  $F_i$  are computed using rate limits as a fraction of link speed (so  $w_1 = 0.4$  and  $w_2 = 0.2$ ). All packets are 1500B in length.

If we consider a single iteration (7500 bytetimes),  $C_1$  transmits 3000B,  $C_2$  transmits 1500B, and the link is idle for (750 + 2250 = 3000 bytetimes). Thus  $C_1$  achieves  $3000 / 7500 = 0.4$  of link capacity and  $C_2$  achieves  $1500 / 7500 = 0.2$  of link capacity. The link remains idle for 40% of the time in each iteration, thereby enforcing strict rate limits. Notice also that the packets are appropriately interleaved and accurately paced.

**Delay Guarantees:** The advantage of using virtual time based scheduling algorithms is that they offer strong per-packet delay guarantees. Specifically,  $WF^2Q+$  guarantees that the finish time of a packet in the discretized system is no more than a bounded delay from an ideal fluid model system. SENIC's unified scheduling algorithm offers similar strong guarantees. Algorithms such as DRR do not have such strong guarantees [14].

## 5.4.2 Hierarchical Bandwidth Sharing

So far we discussed a flat rate limiting scheme. In practice, it may be desirable to group classes and enforce another rate limit on the group. For example, an approach useful in multi-tenant environments is a two level hierarchy where the first level implements strict rate limits for each VM on the server, and the second level provides weighted sharing between the flows originating from each VM.

It is possible to enforce any hierarchical allocation by modulating the rate limits of hardware traffic classes. Control logic in the hypervisor can measure demands and hardware counters, and adjust the rates based on preconfigured limits. We instead now describe an extension to the virtual time based scheduler described above to support a simple two level hierarchy.

**Sharing Model:** We define an  $L_1$  (level 1) class as one which is directly attached to the root of the hierarchy. An  $L_2$  (level 2) class is attached to an  $L_1$  class. Each class is configured with a rate limit. The  $L_1$  classes only support strict rate limits, i.e. sum of rate limits of active  $L_1$  classes should not exceed link capacity.  $L_2$  classes support strict rate limiting, but fallback to weighted sharing in the ratio of their rate limits when the active  $L_2$  classes within an  $L_1$  class oversubscribe the rate limit of that  $L_1$  class. An  $L_1$  class might be a leaf or an internal class while  $L_2$  classes can only be leaves.

**Start and Finish Time Computation:** SENIC only computes time variables for



leaf classes as packets are “enqueued” and “dequeued” only at the leaves. For  $L_1$  leaf classes, the scheduler computes start and finish times as usual, using the rate limits of the respective classes. For each  $L_1$  class, it maintains a rate oversubscription factor  $\phi_{L_1}$ , of active  $L_2$  classes within the  $L_1$  class. For  $L_2$  classes, to compute finish time, the scheduler scales the rate limits and uses the minimum of (1) the configured rate limit  $w_i$  of the  $L_2$  class, and (2) the scaled rate limit of the parent  $L_1$  class based on  $L_2$ 's share, given as:

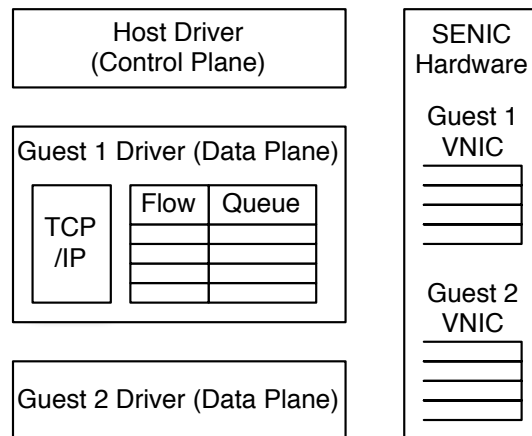
$$w_{i_{scaled}} = \min \left( w_i, w_{L_1} \times \frac{w_i}{\phi_{L_1}} \right)$$

**System Time Computation:** System time is purely based on real time and link drain rate  $R$ , as the  $L_1$  classes are configured such that they never oversubscribe the link. This condition can be easily met even if weighted sharing is required at level 1 of the hierarchy, by simply having the host driver periodically measure demand and adjust the rate limits of the  $L_1$  classes.

**Summary:** Driven by requirements to support rate limits, we described a scheduling algorithm incorporating both weighted sharing and rate limiting into one coherent algorithm. We also extended the algorithm to support two-level rate limits across classes and groups of classes. We realized the unified scheduling algorithm on top of QFQ [22], which in turn implements WF<sup>2</sup>Q+ efficiently. The metadata structure for this QFQ based scheduler is around 40B per class, and it needs only 10kB of global state, thereby scaling easily to 10,000 classes.

## 5.5 Advanced NIC features

This section touches upon advanced features in today’s NICs that are impacted by SENIC’s design, and how we achieve similar functionality with SENIC.



**Figure 5.5.** SENIC architecture, with each guest given a virtualized slice of the NIC (VNIC) using SR-IOV

### 5.5.1 OS and Hypervisor Bypass

Many applications benefit from bypassing the OS network stack to meet their stringent latency and performance requirements [46, 67]. Further, high-performance virtualized workloads benefit from bypassing the hypervisor entirely, and directly access the NIC [61, 65]. To support such requirements, modern NICs expose queues directly to user-space, and include features that virtualize the device state (ring buffers, etc.) through technologies like Single-Root IO Virtualization (SR-IOV [71]). We now describe how SENIC provides these features.

**Configurable SR-IOV Slices or VNICs:** SENIC leverages SR-IOV to expose multiple VNICs. Each VNIC is allocated a configurable number of queues, and guest VMs directly transmit and receive packets through the VNICs, as shown in Figure 5.5. Guest VMs are only aware of queues for their respective VNICs (which is standard SR-IOV functionality), thereby ensuring isolation between transmit queues of different guest VMs. A simple lookup table on the NIC translates VNIC queue IDs to actual queue IDs. A host SENIC driver provides the interface for the hypervisor to configure VNICs,

allocate queues, and configure rate limits. A guest driver running in the VM provides a standard interface to enqueue packets into different queues on the VNIC.

**Classifying Packets:** SENIC relies on the operating system to classify and enqueue packets in the right traffic classes or queues. The host driver residing in the hypervisor maintains the packet classification table. It exports an OpenFlow [68] like API to configure traffic classes and rate limits. When SR-IOV is enabled, the hypervisor is bypassed in the datapath. SENIC therefore relies on the guest VM to perform packet classification.

The guest driver maintains a cached copy of the packet classification table. When the guest driver receives a packet from the network stack for transmission, it looks up its guest packet classification table for a match. If no match is found, it makes a hypercall to the hypervisor for a lookup and caches the matching rule. The actual mapping to the appropriate queue is also cached in the socket data structure to avoid repeated lookups for each packet of a flow. The hypervisor can also proactively setup rules in guest classification tables. Once the rules are cached in the guest, the hypervisor is completely bypassed during packet transmission.

**Untrusted Guests:** It may be unwise to trust guests to classify packets correctly. However, we argue this is not an issue. Even though SR-IOV ensures that a VM can only place packets in queues for its own VNIC, the guest may ignore the hypervisor-specified classification among its queues. We adopt a *trust-but-verify* approach to ensure that guest VMs do not cheat by directing packets to queues with higher rate limits. The key idea is that the hypervisor need not look at every packet to ensure rate limits are not violated, but instead only look at a sampled subset of packets. Since classification is used to provide QoS, sampling packet headers and verifying their classification is sufficient to identify violations. The administrator can be alerted to misbehaving guests, or they can be halted, or forced to give up SR-IOV, and rely on the hypervisor for future packet transmissions.

## 5.5.2 Other Features

Below we describe few other features that are affected by SENIC's design.

**Segmentation Offload:** TCP Segmentation Offload (TSO) is a widely available NIC feature to reduce CPU load by transferring large (upto 64KB) TCP segments to the NIC, which are then divided into MTU sized segments and transmitted with appropriately updated checksums and sequence numbers. SENIC only pulls MTU sized portions of the packet on demand from host memory queues before transmission. This avoids long bursts from a single class, and enables better interleaving and pacing. SENIC augments per-queue metadata with a *TSO-offset* field that indicates which portion of the packet at the head of the queue remains to be transmitted. When interleaving packets, SENIC does not cache packet headers for each class on the NIC, thereby keeping NIC SRAM requirements low. When transmitting TSO packets, SENIC issues two DMA requests: one for the packet header, and another for the MTU sized payload based on TSO-offset.

**Scatter-Gather:** A related optimization is scatter-gather, where the NIC can fetch packet data spread across multiple memory regions, e.g., the header separately from the payload. In such cases, SENIC stores the location of the next segment to be transmitted for each queue and fetches descriptors and data on demand.

**Handling Concurrency:** The design assumed each transmit queue corresponds to one traffic class. To allow multiple CPU cores to concurrently enqueue packets to a class, the SENIC design is extended to support some number of queues (say 8) for each class. Round robin ordering is used among queues within a class, whenever the class gets its turn to transmit. This is easily accomplished by separately storing head and tail indices for each queue in the class metadata table, an active queue bitmap and round robin counter for each class.

**Priority Scheduling:** SENIC can easily also support strict priority scheduling between transmit queues of a class instead of round-robin scheduling. In this case, a priority encoder picks the highest priority active class. One use case is for applications to prioritize their traffic within a given rate limit.

## 5.6 Implementation

We have implemented two SENIC prototypes:

1. A software prototype using a dedicated CPU core to perform custom NIC processing. This implements the unified QFQ-based rate limiting and weighted sharing scheduler described in Section 5.4.1.
2. A NetFPGA-based hardware prototype designed to run microbenchmarks and evaluate the feasibility of pulling packets on demand from host memory for transmission. For engineering expediency, this prototype relies on a simpler, token bucket scheduler (without hierarchies).

We now describe both prototypes in detail. Both prototypes are available for download at <http://sivasankar.me/senic/>.

### 5.6.1 Software Prototype

The software prototype is implemented as a Linux kernel module with modest changes to the kernel. The network packet scheduler is implemented in a new Linux queueing discipline (QDisc) kernel module. We also modified the Linux `tc` utility to enable us to configure the new QDisc module. As described in Section 5.4, SENIC's packet scheduling algorithm is implemented on top of the Quick Fair Queueing (QFQ) scheduler available in the Linux kernel.

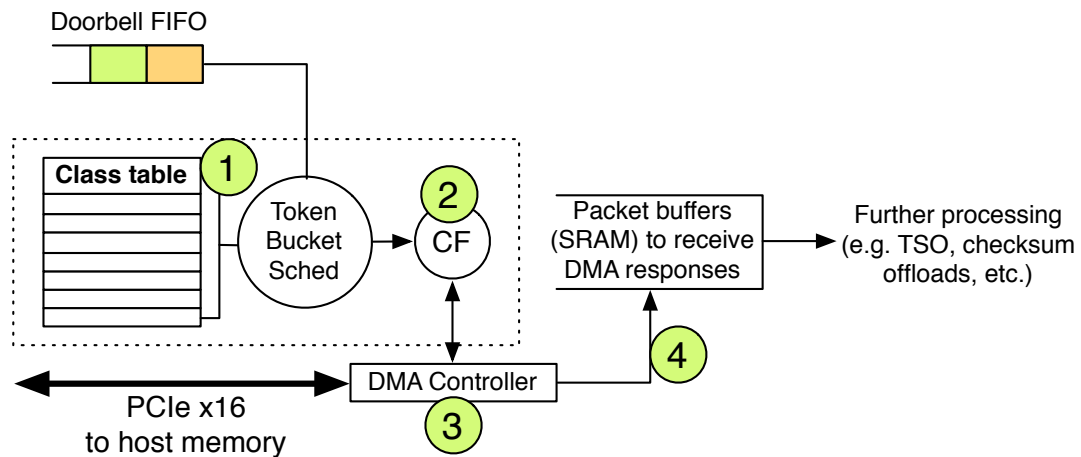
**Transmit Queues and Rate Limits:** The SENIC QDisc maintains per-class FIFO transmit queues in host memory as linked lists. We configure classification rules via `tc`, and also set a rate limit for each class.

**Enqueueing Packets:** In Linux, when the transport layer wants to transmit a packet, it hands it down to the IP layer, which in turn hands it to the QDisc layer. When the QDisc receives a packet from IP, it first classifies the packet, then enqueues it in the corresponding queue, marking the class as active.

**Dedicated CPU Core for Packet Scheduling:** In today's kernel, the dequeue operation starts right after enqueue. However, to mimic NIC functionality, we modified the kernel so the enqueue call immediately returns to the caller, and dedicate a CPU core to perform all NIC scheduling (i.e. dequeuing). The dedicated CPU core runs a kernel thread that computes the schedule based on configured rate limits, and pulls packets from the active transmit queues when they should be transmitted. Packets are transferred to the physical NIC using the standard NIC driver. We disabled TSO to control the transmit schedule at a fine granularity and avoid traffic bursts.

## 5.6.2 NetFPGA Prototype

We now describe our SENIC hardware implementation on a NetFPGA [55]. The primary hardware components of SENIC are (a) the packet scheduler with the class table, (b) doorbell FIFOs to process notifications from the host, and (c) completion FIFOs to send notifications to the host. Each component maintains its own independent state machine and executes in parallel. Figure 5.6 below zooms into the operation of the packet scheduler. We now describe each component in detail.



**Figure 5.6.** The 4 stages of scheduling a packet: (1) pick a class for dequeuing, (2) submit work-request to the class-fetch (CF) module, (3) DMA descriptors and packet payload from the class, (4) handoff packet payload for further processing.

### Packet Scheduler

The scheduler operates on the class metadata table (SRAM block), and performs the following operations:

- It cycles through all active classes (i.e., classes with at least one enqueued packet), and determines if a class has enough tokens to transmit a packet (i.e., whether it is *eligible*). If not, the scheduler refreshes the class's tokens and continues with other classes.
- If the class is eligible, the scheduler submits a work-request to a 'class-fetch' (CF) module and disables the class. Each CF module has a small FIFO to accept requests from the scheduler.
- If the CF module's FIFO is full, the scheduler stalls and waits for feedback from the CF module.
- In parallel, the scheduler processes any pending doorbell requests that modify the

class metadata table. For instance, if the doorbell request is an enqueue operation, the scheduler parses the class ID in the request and updates the class table.

### **Class-Fetch Module**

The class-fetch (CF) module is given a class entry, and its task is to dequeue as many packets as possible until limited by (a) the tokens available for the class, or (b) the burst size of the class. The class entry only stores the descriptor for the first packet. Therefore the CF dispatches DMA requests to (a) fetch the descriptor of the next packet in the ring buffer, and (b) fetch the packet payload of the first descriptor stored in the class entry. The module then synchronously waits for the first DMA to complete, and repeats the process until it exhausts the class tokens, or burst size. Finally, it issues (a) feedback to the scheduler with the new class entry state (updated tokens, tail pointer, and the first packet descriptor), and (b) a completion notification for the class.

The latency to make a scheduling decision, and the DMA fetch latency determine the maximum achievable throughput. We evaluate this in detail in Section 5.7.1.

### **Host Notifications**

SENIC uses standard notification mechanisms to synchronize state between the NIC and the host: doorbell requests and completions. Doorbells update class state on the NIC (e.g., new packets and new rates), and completions notify the host about transmitted packets and processed doorbells. Doorbells and completions are stored in FIFO ring buffers, on the NIC and host respectively.

**Doorbells:** The doorbell is a 16B message written by the host to the memory mapped doorbell FIFO on the NIC. The FIFO is a circular buffer—the host enqueues at the tail while the NIC dequeues at the head. The host synchronizes the head index when it receives completions from the NIC, thereby freeing FIFO entries.



**Completions:** The NIC issues completions by DMA'ing an entry into the completion FIFO in host memory and interrupting the CPU. Each entry indicates (1) the class and number of packets transmitted from the class, or (2) the number of doorbell requests processed. This information is used by the host to reclaim packet memory, and doorbell FIFO entries. These event notifications are similar to BSD's kqueue mechanism [53].

**Avoiding Write Conflicts:** Note that the CF module's feedback, and host notifications both modify the class entry state. However, the feedback only modifies tokens, the first packet's length and address; the host notification only modifies the tail index. If the class's rate changes while it is being serviced, the new rate takes effect only the next iteration when the scheduler refreshes tokens.

## 5.7 Evaluation

This section dissects SENIC to answer the following aspects of the system:

- How scalable and accurate are the hardware rate limiters? We synthesized our hardware prototype with 1000 rate limiters. At 1Gb/s, we found the mean inter-packet timing was within 10ns of ideal, and the standard deviation was 191ns (less than 1.6% of the mean).
- How many packets should be pipelined for achieving line rate at various link speeds? This value depends on the scheduling and DMA latency, and the dominant factor is the DMA latency across the PCIe bus.
- How effective is SENIC at supporting high loads and delivering low latency compared to state of the art software rate limiters? We compare SENIC against Linux HTB and a Parallel Token Bucket (PTB) implementation in software (used in EyeQ [44]). We found that at very low load, all approaches have comparable

latencies. But SENIC sustains 55% higher load compared to PTB, and 250% higher than HTB while keeping memcached 99.9th percentile latency under 3ms.

- How effectively can SENIC isolate different tenants—memcached latency sensitive tenants and a background bandwidth intensive UDP tenant? We found that SENIC could comfortably sustain the configured 3Gb/s of UDP traffic and nearly 6Gb/s of memcached traffic with tail latency under 4ms. However, HTB and PTB had trouble sustaining more than 1.4Gb/s of UDP traffic. SENIC sustains 233% higher memcached load compared to HTB and 43% higher than PTB.

### 5.7.1 Hardware Microbenchmarks

#### Scalability and Accuracy

Due to limitations on the number of outstanding DMA requests<sup>2</sup>, and pipeline datawidth, we were unable to sustain more than 3Gb/s packet transmission rate, and we restrict our tests to rates less than 3Gb/s.

**Table 5.3.** Rate limit accuracy as we vary the number of rate limiters  $N$ , and the rate per class. We see that SENIC is within  $10^{-2}\%$  of ideal even as we approach the maximum throughput we could push through the NetFPGA (3Gb/s).

$N$	Rate	$\mu \pm \sigma$	Rel. error in $\mu$
500	1Mb/s	12ms $\pm$ 7.1us	$3.1 \times 10^{-6}$
1	10Mb/s	1.2ms $\pm$ 233ns	$1.5 \times 10^{-6}$
10		1.2ms $\pm$ 240ns	$1.5 \times 10^{-6}$
100		1.2ms $\pm$ 1.3 $\mu$ s	$2.3 \times 10^{-5}$
1	100Mb/s	120 $\mu$ s $\pm$ 87ns	$1.7 \times 10^{-7}$
10		120 $\mu$ s $\pm$ 173ns	$1.6 \times 10^{-6}$
1	1Gb/s	11.25 $\mu$ s <sup>†</sup> $\pm$ 161ns	$3.5 \times 10^{-4}$
3		11.25 $\mu$ s <sup>†</sup> $\pm$ 191ns	$3.8 \times 10^{-4}$

Table 5.3 shows the rate limiting accuracy of one of the classes, as we vary the number of eligible classes on the NIC. We measure accuracy by timestamping

<sup>2</sup>Our NetFPGA stalls the processor if it has more than 2 outstanding DMA requests. Others have reported a similar issue with the Virtex5 FPGA [101].

every packet with a clock resolution of 10ns, and retrieving the inter-packet timestamp difference for packets of that one class. We compute the mean ( $\mu$ ) and standard deviation ( $\sigma$ ), and also the relative error in  $\mu$  as  $|\mu_{\text{empirical}} - \mu_{\text{ideal}}|/\mu_{\text{ideal}}$ . We see that SENIC very accurately enforces the configured rate even with 500 classes each operating at 1Mb/s.

†**Note:** NetFPGA supports rates that are of form 12.8Gb/s/K, where K is an integer. Therefore, though we set the rate limit to 1Gb/s, the output will 12.8/12 Gb/s (1.067Gb/s), for which the inter-packet time is 11.25 $\mu$ s.

### Scheduler Latency

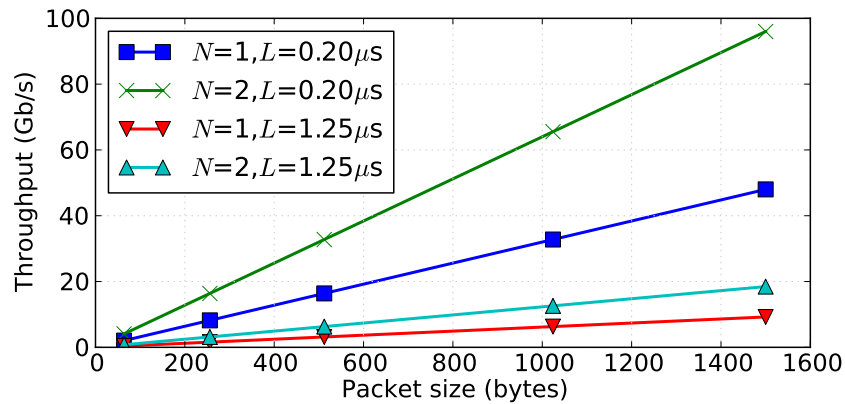
We dig deeper into how long it takes for a scheduling operation in hardware. On the NetFPGA, the SRAM has a datawidth of 512 bits (64B), an access latency of 1 cycle, and enough bandwidth to support one operation (either a read or a write) every cycle. In the *worst case*, each scheduler iteration takes at most 5 cycles:

- 1 for reading the class metadata from SRAM.
- 1 for refreshing the tokens and CF-enqueue.
- 1 SRAM write for processing CF-feedback.
- 2 for processing a doorbell: 1 for reading the class metadata from SRAM, and 1 for updating class metadata and writing it back.

We synthesized our NetFPGA prototype at 100MHz (10ns per clock cycle), and therefore, it takes no more than 50ns to make a scheduling decision. We expect a production-quality NIC to have a higher clock rate, and thus a faster scheduler. For instance, the ASIC in Myricom 10Gb/s Ethernet NIC runs at a clock rate of 364.6MHz [64]. The QFQ based scheduler takes about twice as many cycles as simple token buckets [22], so with a higher clock rate, it can still complete in 50ns.

## Maximum Per-Class Throughput

In this experiment, we first analyze the DMA latency which affects the achievable throughput per-class. We measure the time interval between sending a DMA request from the CF-module to fetch 16B from host memory, and receiving the response. We find that the average latency is  $L = 1.25\mu\text{s}$  ( $\sigma = 40\text{ns}$ ) with the NetFPGA platform (using a second generation PCIe x8 bus). However, the number is often better with a production-quality NIC. For instance, the DMA latency on an Intel NIC was found to be close to 200ns [77].



**Figure 5.7.** Maximum throughput per class as a function of the packet size, and the number  $N$  of CF modules operating in parallel, and the DMA latency  $L$ . We see that the achievable throughput on the NetFPGA ( $L = 1.25\mu\text{s}$ ) with  $N = 1$  is 9.23Gb/s with 1500B packets (if not for the DMA request constraints described earlier).

Recall that the CF-module processes each class by issuing a DMA request for the class's second packet descriptor, followed by the request for the class's first packet payload. With a burst size of 1 packet per class, the maximum achievable throughput per class depends on the sum of DMA latency and scheduler latency. For instance, if the scheduler takes 50ns to dispatch a class to the CF module, the DMA latency to fetch a packet descriptor is 1250ns, and burst size is 1 packet, the maximum achievable throughput per-class is about 1500B (MTU) every 1300ns. Therefore, to achieve line rate

we can instantiate multiple CF modules, and the scheduler dispatches classes to them in parallel. Further, using TSO, or multiple queues per class enables higher throughput per class. Figure 5.7 shows the trend.

## 5.7.2 Software Macrobenchmarks

We ran experiments with our software based SENIC prototype to evaluate the application level performance when SENIC is used for rate limiting traffic.

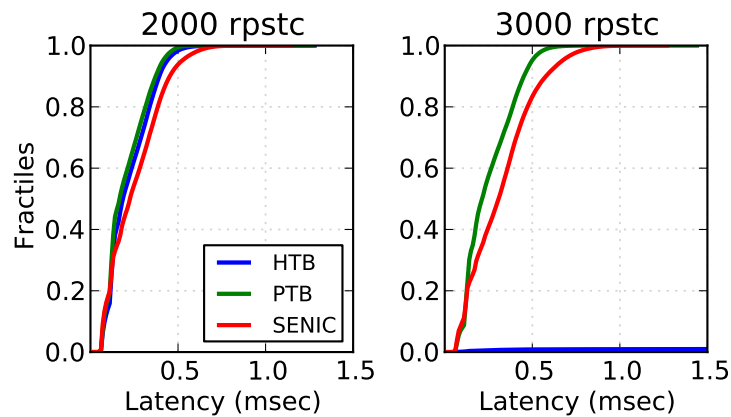
### Memcached

We conducted an experiment with several memcached tenants sharing a cluster—10 tenants on each machine in an 8 node cluster. Each node is a dual 4-core, 2-way hyperthreaded Intel Xeon E5520 2.27GHz server, with 24GB of RAM, a 10Gb/s NIC (Intel or Myricom), and running Linux 3.9.0. Each tenant was allocated 1 CPU hyperthread on each machine, and 2GB of RAM. One machine ( $M_{srv}$ ) had 10 memcached server instances—1 for each tenant. We pre-populated them with 12B-key, 2KB-value pairs. Each of the other 7 machines ( $M_{cli}$ ) ran 10 memcached client processes that sent GET requests to the respective tenant’s memcached server instance.

Rate limits were configured for each memcached client-server pair. The total rate limit was 9.5Gb/s on  $M_{srv}$ , and 6Gb/s on  $M_{cli}$  machines. Each tenant got an equal share of the total rate, divided equally among its own destinations. These limits were chosen to be large enough that memcached would not be bandwidth limited. We ran experiments using HTB, PTB, and the SENIC software prototype.

We define the unit *rpstc*, requests per second per tenant per client, to denote the load on the system. For instance, 2,000 rpsct means each of the 7 client instances of each tenant generates a load of 2,000 req/s, resulting in a total load on  $M_{srv}$  of 140,000 req/s.

**Latency:** We varied the client load (2000, 3000 rpsct) and observed the latency

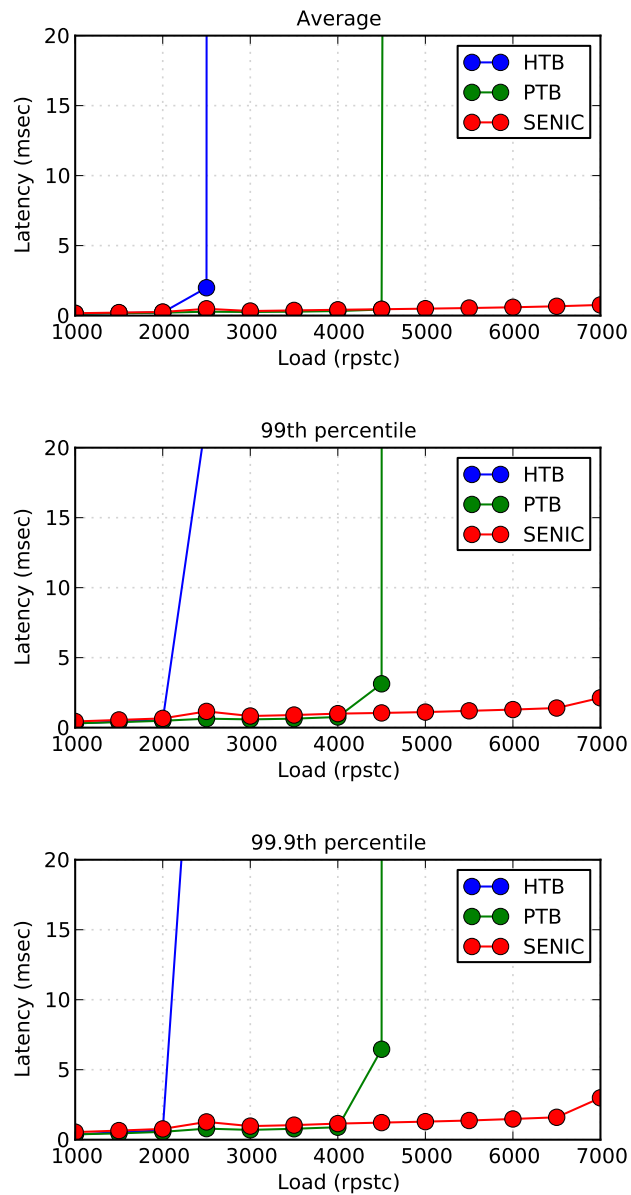


**Figure 5.8.** CDF of memcached response latency at different loads. SENIC, HTB and PTB have similar latency at 2,000 rpstc, but HTB latency shoots up at 3,000 rpstc.

distribution of memcached responses (Figure 5.8). The total egress bandwidth utilization on  $M_{srv}$  is quite low at 2.3Gb/s and 2.9Gb/s respectively at the two loads. At 2,000 rpstc, we observed that HTB, PTB and SENIC perform similarly. But at 3,000 rpstc, HTB's latency suffers a drastic hit, whereas PTB and SENIC are able to keep up. With HTB, requests keep getting backlogged as the scheduler is the bottleneck and is unable to push packets out of the server fast enough. At the fairly low load of 3,000 rpstc, PTB has marginally lower latency than the SENIC software prototype due to the cache misses incurred for pulling and transmitting all packets from a single CPU core. A hardware SENIC implementation would not have this penalty.

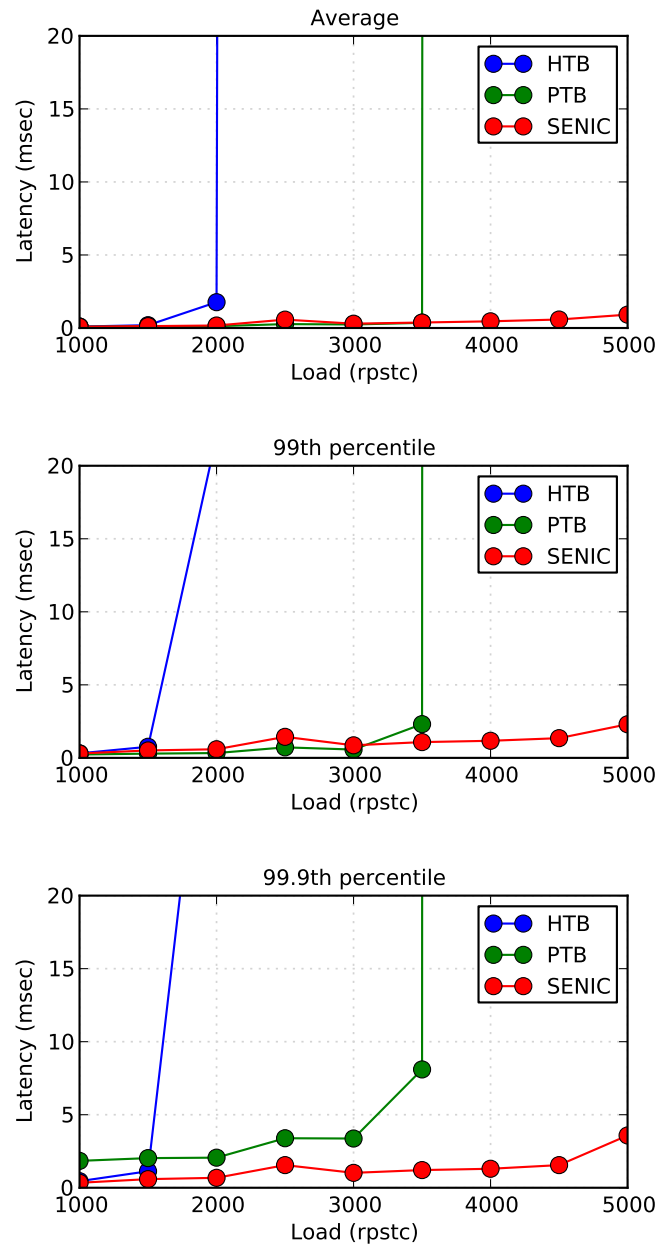
**Throughput:** We varied the memcached load and measured the average, 99th, and 99.9th percentile latency in each case. Figure 5.9 shows that SENIC could comfortably handle 7,000 rpstc, sustaining 55% higher load compared to PTB, and 250% higher than HTB. We stopped at 7,000 rpstc as that was the maximum load the cluster could sustain even without any rate limiters (with the default Linux multi queue QDisc).

While the SENIC software prototype is much better than HTB and PTB, a hardware SENIC implementation would perform even better as there would not be cache



**Figure 5.9.** Memcached response latency at different loads (average, 99th perc. and 99.9th perc. latency). We see that SENIC easily sustains 7,000 rpstc (which was also the maximum load the cluster sustained without any rate limiting). However HTB and PTB latencies spike up at much lower loads.

misses for each transmit operation. Further, if hypervisor bypass is used by VMs to communicate directly with SENIC hardware, the relative latency and throughput benefits of the hardware solution would be even more.



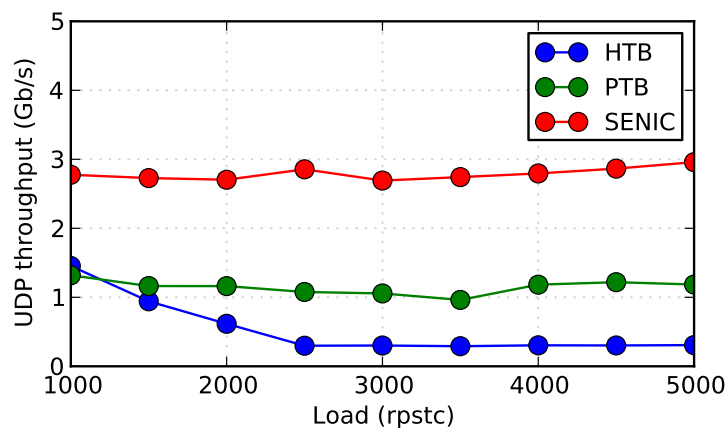
**Figure 5.10.** Memcached latency at different loads, with configured background all-to-all UDP traffic of 3Gb/s from each server. We see that SENIC could sustain 5,000 rpstc (network throughput was roughly equal to the configured limit of 6Gb/s). HTB and PTB on the other hand, fell over at lower loads.



## Memcached and UDP Tenant Isolation

To evaluate how effectively SENIC can isolate different tenants, we repeated the above experiments with 1 co-located UDP tenant on each machine, that generates all-to-all UDP traffic as fast as it can. The total rate limit was set at 3Gb/s for UDP traffic, and 6Gb/s for memcached on each machine—divided equally among respective tenants and destinations. The maximum memcached bandwidth utilization we tested was around 5.75Gb/s on  $M_{srv}$ , so memcached was again not bandwidth limited.

**Memcached Latency and Throughput:** As shown in Figure 5.10, SENIC was able to sustain 5,000 rpstc memcached throughput (5.75Gb/s) with 99.9th percentile latency around 4ms while simultaneously delivering very close to the configured 3Gb/s of total UDP tenant traffic on the memcached server machine. On the other hand, HTB was only able to sustain 1,500 rpstc, while PTB sustained 3,500 rpstc.



**Figure 5.11.** Throughput achieved by UDP background traffic. The configured rate limit was 3Gb/s. We found that SENIC could sustain very close to the configured 3Gb/s throughput, but HTB and PTB had trouble delivering more than 1.3Gb/s.

**UDP Tenant Throughput:** We measured the total throughput the UDP tenant achieved on  $M_{srv}$  as it was the primary machine under heavy overall load. Figure 5.11 shows that while SENIC sustained the configured 3Gb/s of throughput for the bandwidth

intensive UDP tenant, HTB and PTB had difficulty keeping up. Even at lower mem-cached loads, HTB and PTB had trouble delivering more than 1.3Gb/s UDP throughput. Measurements showed that the CPU cores allocated to the UDP tenant were highly loaded, indicating that current software approaches suffer when CPU load increases and the tenants with high CPU load might notice degraded performance as the rate limiter is unable to keep up.

## 5.8 Practical Considerations

SENIC's design goals expose a tension in its implementation. Its on-board packet scheduler must be able to transfer sequences of individual packets from a potentially large number of traffic classes for fine-grained rate control. Yet, to drive high line rates, it must support a high overall DMA transfer rate to transfer packets from host memory to the wire. Thus, the performance of SENIC is upper-bounded by the performance of the host's underlying DMA subsystem.

Today's NICs rely on a number of optimizations to drive high link rates, while lowering their impact on the DMA subsystem. For example, when TSO is enabled, they can transfer the packet header just once from memory and cache it on the NIC. The NIC can then pull in the rest of the payload (issuing the appropriate DMA operations), combine it with the cached header and transmit MTU-sized segments. SENIC's design supports interleaving MTU-sized segments from different traffic classes, depending on their configured rates and burst sizes. Because the number of such classes can be quite large, SENIC does not cache packet headers on the NIC for each class. Thus, SENIC's impact on the underlying DMA subsystem is going to be greater than a traditional NIC with TSO. We now briefly examine this impact.

In the absence of TSO, SENIC requires the same number of DMA transfers from host memory as current NICs—one for each packet, in addition to the packet descriptors.

However when TSO is active, SENIC issues a DMA operation for the header in addition to one for the payload, for each MTU-sized segment. Note that NICs today are capable of processing many more DMA transfers per second than required for handling MTU-sized frames at line rate. This headroom allows SENIC to drive high line rates even when TSO is enabled, despite the larger number of DMA transfers it requires.

To ground this claim experimentally, we examined the DMA subsystem performance of both 10Gb/s and 40Gb/s commercial NICs. Using a Myricom 10Gb/s NIC, we were able to sustain a throughput of 13–14 million 64 byte packets per second (pps). Since packets were randomly spread across host memory, each packet required at least one DMA transfer, and thus the NIC can sustain roughly the same number of DMA transfers per second.

For 40Gb/s, we used a Mellanox Connect-X3 NIC [58] to transmit 64 byte packets. We observed that it could only support about 13.1 Mpps, which is less than the rate required to sustain 40Gb/s with 64 byte packets. However, using MTU-sized frames, and TSO disabled, it was able to drive 3.25 Mpps, which was sufficient to sustain line rate at 40Gb/s.

These reference points allow us to gauge the performance of SENIC at both 10Gb/s and 40Gb/s. For instance, at 40Gb/s, SENIC would require  $3.25 \times 2 = 6.5$  million DMA transfers per second (to DMA both payloads and headers) to achieve line rate. This is well under the 13.1 million transfers per second we were able to sustain on the same NIC. Hence, we believe that SENIC can support line rate performance with TSO enabled for MTU-sized segments. Since SENIC does not introduce additional DMA requests for non-TSO packets, it should perform comparably to current commercial NICs.

## 5.9 Related Work

We classify related work into two parts: (1) hardware improvements, and (2) software improvements, some of which try to work around limited hardware capabilities. The NIC hardware datapath has only recently received attention from the research community in light of the requirements listed in Section 5.2.

**Hardware Efforts:** Commercial NICs support transport offloading to support millions of connection endpoints, such as ‘queue pairs’ in InfiniBand [78], or TCP sockets in case of TCP offload engines [60]. The SENIC design is simpler as we only offload rate limiting, and leave the task of reliable delivery to software.

Recent work [61,85] calls for changes in the NIC architecture in light of low-latency applications (e.g. RAMCloud [34]), and virtualized environments (e.g. public clouds). Such efforts are complementary to SENIC, which focuses only on scaling transmit scheduling. ServerSwitch [56] presented a programmable NIC to support packet classification and configurable congestion management. ServerSwitch can directly benefit from the large number of rate limiters in SENIC.

A number of efforts have focused on scalable packet schedulers in switches [62, 79]. A NIC is conceptually no different from a switch; however, switch schedulers have to deal with additional complexity due to limited on-chip SRAM, and the fact that they cannot control the exogenous traffic arrival rate. Thus, commercial switches often resort to simpler approaches like AFD [70] which can scale to 1000s of policers, but can only *drop* packets (instead of accurate pacing). On the other hand, the NIC being the first hop is in a unique position—its design can be made considerably simpler by leveraging host DRAM to store all packets. This approach enables SENIC to simultaneously scale to, and accurately pace, a large number of traffic classes.

**Software Efforts:** An alternate approach to deal with limited NIC rate limiters is

to share them in some fashion, which has been explored by approaches like vShaper [51] and FasTrak [65]. SENIC eases the burden on such approaches, as we believe the NIC is particularly amenable to large-scale rate limiting by taking advantage of host DRAM. However, if unforeseen applications require more rate limiters than SENIC can offer, such techniques come in handy.

IsoStack [86] proposed offloading the entire TCP/IP network stack to dedicated cores. Our SENIC software prototype mimics this approach (offloading only the scheduler to a dedicated core), which explains the performance benefits in our evaluation. Architectures for fast packet IO such as Netmap [82] are orthogonal to SENIC, and they only stand to benefit from scalable rate limiting in the NIC.

## 5.10 Summary

Historically, the NIC has been an ideal place to offload common network tasks such as packet segmentation, VLAN encapsulation, checksumming, and rate limiting is no exception. Today's NICs offer only a handful of rate limiters, however new requirements such as performance isolation and OS-bypass for low-latency transport demand more rate limiters. We argued why it makes sense to pursue a hardware offload approach to rate limiting: at data center scale, a custom ASIC is cheaper than dedicating CPU resources for a task that requires real time packet processing. We implemented a proof-of-concept NIC on the NetFPGA to demonstrate the feasibility of scaling hardware rate limiters to thousands of queues. We believe the NIC hardware is *the* cost-effective place to implement rate limiting, especially as we scale the bandwidth per-server to 40Gb/s and beyond.

## 5.11 Acknowledgments

This research was supported in part by the NSF through grants CNS-1314921 and CNS-1040190. Additional funding was provided by a Google Focused Research Award. We would like to thank our shepherd Saikat Guha and the anonymous NSDI reviewers.

Chapter 5, in part, contains material as it appears in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14), Seattle, WA, April 2014. "SENIC: Scalable NIC for End-Host Rate Limiting". Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, contains material as it appears in the Proceedings of the 5th USENIX Conference on Hot Topics in Cloud Computing (HotCloud '13), San Jose, CA, June 2013. "NicPic: Scalable and Accurate End-Host Rate Limiting". Sivasankar Radhakrishnan, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. The dissertation author was the primary investigator and author of this paper.

# Chapter 6

## Conclusion

Modern web services are complex systems that have sophisticated network subsystems. In this dissertation, we took a holistic look at how web services are designed and used, identified three key networking challenges they face, and presented solutions.

TCP Fast Open is an improvement to TCP, that enables data transfer to begin instantly without any delay for setting up a connection. This has a significant impact on the latency for short TCP transfers such as HTTP web requests, which was one of the driving use cases for the design. TCP Fast Open enables the safe exchange of data during TCP's 3-way handshake, without compromising on server security or making the server susceptible to denial-of-service attacks. TCP Fast Open relies on a simple security cookie granted by the server to authenticate clients connecting to it using TCP Fast Open, providing defense against denial-of-service attacks. Authenticating client IP addresses through the cookie also helps protect against amplified reflection attacks. TCP Fast Open has a simple design, enabling rapid and incremental deployment. It interoperates well with middle-boxes, traditional TCP implementations, and server farms. The simple API for applications enables web services to easily use TCP Fast Open, and we believe this can have a significant impact on reducing web service latency. TCP Fast Open is widely deployed and is available as part of the Linux kernel since version 3.6. It has been deployed across all of Google's web servers. At the time of writing this dissertation, TCP

Fast Open is under discussion at the IETF for publishing as an Experimental Internet Standard [23]. The Chrome web browser has application level support for users to take advantage of TCP Fast Open.

Dahu is a switch mechanism for commodity data center switches, that enables the adoption of direct connect networks in data centers. Existing data center multipath designs are constrained by the static nature of ECMP, and the inability of commodity switches to go beyond shortest path routing. Dahu enables dynamic hashing of traffic along different network paths. Further, Dahu actively exploits non-shortest path forwarding to reduce congestion, while preventing persistent forwarding loops. We presented a decentralized load balancing heuristic that can quickly respond to congestion by making local decisions in switches, and be more reactive than centralized approaches. Dahu enables data centers to adopt direct connect topologies, and build networks that provision the required capacity with fewer switches for common communication patterns. Dahu offers the same ease of use, decoupled application logic, and lower complexity and commoditization of current data center network designs that use commodity switches. We believe this can maximize network utilization in the data center.

Network performance isolation in the data center is a problem that has had significant interest recently. With the rise of virtualization and the increasing level of statistical multiplexing of resources in the data center, performance isolation has become an important consideration. Server consolidation, and the increasing density of compute, storage, and bandwidth resources in servers makes this problem more challenging as many more services share physical resources. SENIC is a scalable NIC design that supports tens of thousands of traffic classes natively in the NIC to meet the needs of performance isolation in the data center. SENIC can be used as a plug in replacement for today's NICs — it exposes the same familiar interface to the operating system to enqueue packets to the NIC for transmission, and to configure rate limits or weights for



different traffic classes on the NIC. SENIC also supports hypervisor and kernel bypass to reduce software packet processing overheads, and to reduce the latency in the datapath. We built a hardware and a software prototype to demonstrate the feasibility and benefits of the SENIC design. We showed that as link speeds to servers increase to 40Gb/s and beyond, the NIC hardware is the most cost-effective place to implement rate limiting and performance isolation.

The design of modern web services, and the relentless focus on improving performance has made network design an important consideration. Today's planetary scale services have stringent performance needs from the network. The complexity of the networking subsystem requires a holistic approach to consider the challenges, identify the key components that have a large impact on network performance, and design solutions that are innovative, but that can be incrementally or easily deployed at a large scale.

# Bibliography

- [1] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.
- [2] Mohammad Al-Fares, Khaled Elmeleegy, Benjamin Reed, and Igor Gashinsky. Overclocking the Yahoo! CDN for Faster Web Page Loads. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2011.
- [3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2008.
- [4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.
- [5] Alexa top 500 global sites. <http://www.alexa.com/topsites>. Retrieved 4 October 2011.
- [6] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, Rong Pan, B. Prabhakar, and M. Seaman. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *46th Annual Allerton Conference on Communication, Control, and Computing*, 2008.
- [7] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.
- [8] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

- [9] William Allen Simpson. TCP Cookie Transactions (TCPCT), 2011. RFC 6013.
- [10] William Allen Simpson. TCP Cookie Transactions (TCPCT) Rapid Restart, July 2011. IETF Internet draft (work in progress).
- [11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.
- [12] Mike Belshe. More Bandwidth Doesn't Matter (Much). <http://www.belshe.com/2010/05/24/more-bandwidth-doesnt-matter-much/>. Retrieved 9 March 2014.
- [13] Mike Belshe. The Era of Browser Preconnect. <http://www.belshe.com/2011/02/10/the-era-of-browser-preconnect/>. Retrieved 9 March 2014.
- [14] Jon C. R. Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 1996.
- [15] Jon C.R. Bennett and Hui Zhang. WF<sup>2</sup>Q: Worst-case Fair Weighted Fair Queueing. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 1996.
- [16] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.
- [17] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2011.
- [18] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1987.
- [19] Andrea Bittau, Michael Hamburg, Mark Handley, David Maziers, and Dan Boneh. The Case for Ubiquitous Transport-Level Encryption. In *Proceedings of the USENIX Security Symposium*, 2010.
- [20] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification, 1994. RFC 1644.
- [21] Broadcom Smart-Hash Technology. [http://www.broadcom.com/collateral/wp/StrataXGS\\_SmartSwitch-WP200-R.pdf](http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP200-R.pdf). Retrieved 9 March 2014.
- [22] Fabio Checconi, Luigi Rizzo, and Paolo Valente. QFQ: Efficient Packet Scheduling With Tight Guarantees. *IEEE/ACM Transactions on Networking (TON)*, June 2013.

- [23] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. TCP Fast Open. <http://tools.ietf.org/html/draft-ietf-tcpm-fastopen-07>, February 2014. IETF Internet Draft.
- [24] Chromium by “pre-connect” to accelerate web browsing. <http://goo.gl/KPoNW>. Retrieved 4 October 2011.
- [25] CPLEX Linear Program Solver. <http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/>. Retrieved 9 March 2014.
- [26] Cray XT3 Supercomputer. [http://www.craysupercomputers.com/downloads/CrayXT3/CrayXT3\\_Datasheet.pdf](http://www.craysupercomputers.com/downloads/CrayXT3/CrayXT3_Datasheet.pdf), Retrieved 4 March 2014.
- [27] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.
- [28] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 1989.
- [29] Advait Abhay Dixit, Pawan Prakash, and Ramana Rao Kompella. On the Efficacy of Fine-Grained Traffic Splitting Protocols in Data Center Networks. Technical Report Purdue/CSD-TR 11-011, Department of Computer Science, Purdue University, 2011.
- [30] Nandita Dukkupati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An Argument for Increasing TCP’s Initial Congestion Window. *ACM SIGCOMM Computer Communication Review (CCR)*, July 2010.
- [31] Wesley Eddy. TCP SYN Flooding Attacks and Common Mitigations, 2007. RFC 4987.
- [32] Anwar Elwalid, Cheng Jin, Steven Low, and Indra Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2001.
- [33] Simon Fischer, Nils Kammenhuber, and Anja Feldmann. REPLEX: Dynamic Traffic Engineering Based on Wardrop Routing Policies. In *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2006.
- [34] Mario Flajslik and Mendel Rosenblum. Network Interface Design for Low Latency Request-response Protocols. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.

- [35] Albert 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 *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2009.
- [36] 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 *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.
- [37] Charles M. Hannum. Security problems associated with T/TCP. <http://www.midway.org/doc/ttcp-sec.txt>, 1996. Retrieved 4 October 2011.
- [38] Urs Hölzle. OpenFlow @ Google. Talk at Open Networking Summit, 2012.
- [39] MPTCP htsim simulator. <http://nrg.cs.ucl.ac.uk/mptcp/implementation.html>. Retrieved 9 March 2014.
- [40] Intel 82599 10GbE Controller. <http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf>. Retrieved 9 March 2014.
- [41] Internet World Stats: Usage and Population Statistics. <http://www.internetworldstats.com/stats.htm>, Retrieved 14 January 2014.
- [42] Cisco Global Cloud Index: Forecast and Methodology. [http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud\\_Index\\_White\\_Paper.pdf](http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf), Retrieved 4 March 2014.
- [43] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and Kyoungsoo Park. SSLShader: Cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.
- [44] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
- [45] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2005.
- [46] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2012.

- [47] John Kim, William J. Dally, and Dennis Abts. Flattened butterfly: A Cost-efficient Topology for High-radix networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.
- [48] John Kim, William J. Dally, Steve Scott, and Dennis Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.
- [49] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform For Large-scale Production Networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [50] Christian Kreibich, Nicholas Weaver, Boris Nechaev, and Vern Paxson. Netalyzr: Illuminating the Edge Network. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.
- [51] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. Virtualizing Traffic Shapers for Practical Resource Allocation. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.
- [52] Adam Langley. Probing the viability of TCP extensions. <http://www.imperialviolet.org/binary/ecntest.pdf>.
- [53] Jonathan Lemon. Kqueue - A Generic and Scalable Event Notification Facility. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2001.
- [54] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
- [55] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *Proceedings of the IEEE International Conference on Microelectronic Systems Education*, 2007.
- [56] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.
- [57] Alberto Medina, Mark Allman, and Sally Floyd. Measuring Interactions Between Transport Protocols and Middleboxes. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2004.

- [58] Mellanox Connect-X3. [http://www.mellanox.com/related-docs/prod\\_adapter\\_cards/ConnectX3\\_EN\\_Card.pdf](http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf). Retrieved 9 March 2014.
- [59] Jeffrey C. Mogul. The Case for Persistent-Connection HTTP. *ACM SIGCOMM Computer Communication Review (CCR)*, October 1995.
- [60] Jeffrey C. Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.
- [61] Jeffrey C. Mogul, Jayaram Mudigonda, Jose Renato Santos, and Yoshio Turner. The NIC Is the Hypervisor: Bare-Metal Guests in IaaS Clouds. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.
- [62] Sung-Whan Moon, Jennifer Rexford, and Kang G Shin. Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches. *ACM Transactions on Computer Systems (TOCS)*, November 2000.
- [63] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable Rule Management for Data Centers. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
- [64] Myri-10G PCI Express Network Adapter. <https://www.myricom.com/products/network-adapters/10g-pcie-8b-2s.html>, Retrieved 25 September 2013.
- [65] Radhika Niranjana Mysore, George Porter, and Amin Vahdat. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2013.
- [66] The need for speed. [http://www.technologyreview.com/files/54902/GoogleSpeed\\_charts.pdf](http://www.technologyreview.com/files/54902/GoogleSpeed_charts.pdf). Retrieved 4 October 2011.
- [67] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
- [68] OpenFlow Consortium. <http://www.openflow.org>. Retrieved 9 March 2014.
- [69] OpenFlow Switch Specification - Version 1.1. <http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf>. Retrieved 9 March 2014.
- [70] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate Fairness Through Differential Dropping. *ACM SIGCOMM Computer Communication Review (CCR)*, April 2003.

- [71] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. <http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html>, Retrieved 25 September 2013.
- [72] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008.
- [73] T/TCP Vulnerabilities. *Phrack Magazine*, 8(53), July 1998.
- [74] Sivasankar Radhakrishnan, Rishi Kapoor, Malveeka Tewari, George Porter, and Amin Vahdat. Dahu: Improved Data Center Multipath Forwarding. Technical Report UCSD/CS2013-0992, Department of Computer Science, University of California, San Diego, February 2013.
- [75] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.
- [76] Sreeram Ramachandran. Web metrics: Size and number of resources. <http://code.google.com/speed/articles/web-metrics.html>. Retrieved 9 March 2014.
- [77] RAMCloud RPC Performance Numbers. <https://ramcloud.stanford.edu/wiki/display/ramcloud/RPC+Performance+Numbers>, Retrieved 25 September 2013.
- [78] RDMA Aware Networks Programming User Manual. [http://www.mellanox.com/related-docs/prod\\_software/RDMA\\_Aware\\_Programming\\_user\\_manual.pdf](http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf), Retrieved 25 September 2013.
- [79] Jennifer Rexford, Flavio Bonomi, Albert Greenberg, and Albert Wong. A Scalable Architecture for Fair Leaky-Bucket Shaping. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 1997.
- [80] Transmission Control Protocol, 1981. RFC 793.
- [81] Luigi Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM SIGCOMM Computer Communication Review (CCR)*, January 1997.
- [82] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.
- [83] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant



- Datacenter Networks. In *Proceedings of the USENIX Workshop on I/O Virtualization*, 2011.
- [84] Peter Romirer-Maierhofer, Fabio Ricciato, Alessandro D'Alconzo, Robert Franzan, and Wolfgang Karner. Network-Wide Measurements of TCP RTT in 3G. In *Proceedings of the Workshop on Traffic Monitoring and Analysis*. Springer-Verlag, 2009.
- [85] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. It's Time for Low Latency. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.
- [86] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.
- [87] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.
- [88] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 1995.
- [89] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.
- [90] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2004.
- [91] Peng Sun, Minlan Yu, Michael J. Freedman, and Jennifer Rexford. Identifying Performance Bottlenecks in CDNs through TCP-Level Monitoring. In *Proceedings of the ACM SIGCOMM Workshop on Measurements Up the Stack*, 2011.
- [92] Titan Supercomputer. <http://www.olcf.ornl.gov/support/system-user-guides/titan-user-guide/>. Retrieved 9 March 2014.
- [93] J. Touch, J. Heidemann, and K. Obraczka. Analysis of HTTP performance. *USC/ISI Research Report 98-463*, December 1998.
- [94] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An Untold Story of Middleboxes in Cellular Networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.

- [95] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. Why are Web Browsers Slow on Smartphones. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011.
- [96] Zhiheng Wang. Navigation Timing. <http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html>. Retrieved 9 March 2014.
- [97] Web Page Replay Tool. <http://code.google.com/p/web-page-replay/>. Retrieved 9 March 2014.
- [98] Web performance and ecommerce. <http://www.strangeloopnetworks.com/resources/#Infographics>. Retrieved 4 October 2011.
- [99] When seconds count. <ftp://ftp.software.ibm.com/software/au/downloads/GomezWebSpeedSurvey.pdf>, Retrieved 4 March 2014.
- [100] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2011.
- [101] Xilinx User Community Forums: ML506 board: Why my DMA IP hangs OS? <http://forums.xilinx.com/t5/PCI-Express/ML506-board-Why-my-DMA-IP-hangs-OS/td-p/94298>, Retrieved 25 September 2013.
- [102] David Zats, Tathagata Das, Prashanth Mohan, Dhruva Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2012.
- [103] Wenxuan Zhou, Qingxi Li, Matthew Caesar, and Brighten Godfrey. ASAP: A Low-Latency Transport Layer. In *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2011.