Technical Report

UCAM-CL-TR-943
ISSN 1476-2986

Number 943

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Latency-First datacenter network scheduling

Mathew P. Grosvenor

January 2020

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 2017 by the author for the degree of Doctor of Philosophy to the University of Cambridge.

ISSN 1476-2986

Latency-First Datacenter Network Scheduling

Matthew Philip Grosvenor

Summary

Every day we take for granted that, with the click of a mouse or a tap on a touchscreen, we can instantly access the Internet to globally exchange information, finances and physical goods. The computational machinery behind the Internet is found in *datacenters* scattered all over the globe. Each datacenter contains many tens of thousands of computers connected together through a *datacenter network*. Like the Internet, datacenter networks suffer from *network interference*. Network interference occurs when congestion caused by some applications, delays or *interferes* with traffic from other applications. Network interference makes it difficult to predict *network latency*: the time that any given packet will take to traverse the network. The lack of predictability makes it difficult to build fast, efficient, and responsive datacenter applications.

In this dissertation I address the problem of network interference in datacenter networks. I do so primarily by exploiting *network scheduling* techniques. Network scheduling techniques were previously developed to provide predictability in the Internet. However, they were complex to deploy and administer because few assumptions could be made about the network. Unlike the Internet, datacenter networks are administered by a single entity, and have well known physical properties, that rarely change.

The thesis of this dissertation is that it is both possible and practical to resolve network interference in datacenter networks by using network scheduling techniques. I show that it is possible to resolve network interference by deriving a simple, and general, network scheduler and traffic regulator. I take the novel step of basing my scheduler design on a simple, but realistic, model of a datacenter switch. By regulating the flow of traffic into each switch, I show that it is possible to bound latency across the network. I develop the *leaky token bucket* regulator to perform this function. I show that my network scheduler design is practical by implementing a simple, and immediately deployable, system called QJUMP. QJUMP is thoroughly evaluated and demonstrated to resolve network interference between datacenter applications in both simulation and on a physical test-bed. I further show that QJUMP can be extended and improved upon in a variety of ways. I therefore conclude that it is both possible and practical to control network interference in datacenter networks using network scheduling techniques.

Acknowledgements

First and foremost, I would like to acknowledge the support and care of my supervisor, Andrew W. Moore. Andrew is a true gentleman and a scholar. Beyond his excellent academic credentials and impressive grant writing skills, he is a deeply caring human being. He has always placed my physical and emotional wellbeing ahead of my academic progress and, for that, I am truly grateful. I could not have asked for more from my supervisor.

My deepest thanks also go to my second supervisor Robert N. M. Watson. Robert too has deeply cared for my wellbeing as well as my academic and career progression. Despite his significant commitments, whenever I have needed it, Robert has always made the time for me. I will always be thankful for his marathon effort of reading, and providing deeply insightful comments on this dissertation, during a 12 hour flight to the US. The document has been much improved as a result.

I will forever be indebted to Andrew and Robert for their time, love and support over these years. It has been an honour to work with both of you and I am privileged to call you friends. I sincerely hope to continue our collaborations into the future. It is thanks to the efforts of both Robert and Andrew that I have been generously supported by the EPSRC INTERNET, DARPA MRC2 and EU Horizon 2020 SSICLOPS research programs. These programs have kept me warm, dry, and fed, as well as opening up the world to me through collaboration and conference travel to exotic locations (like Oakland, California).

The Cambridge Computer Laboratory has a proud history reaching back as far as the very beginning of computing as we know it. My thanks go to all the members, past and present, of the Computer Laboratory, especially the Systems Research Group (SRG) and the Networks and Operating Systems (NetOS) group. In particular I would like to thank Steven Hand for his keen insights and Jon Crowcroft for his impressive (random?) idea generation abilities. Jon read and provided feedback to this dissertation multiple times (sometimes even unsolicited) which has immensely improved the quality of the final document.

To my first, and best, post-doc Yury Audzevich. You taught me everything I know about networks and switches. Your quiet humility is an inspiration and I've truly missed you since you moved on to better things. This dissertation would not have been possible if you hadn't taken me under your wing during my first weeks in Cambridge.

There will always be a special place in my heart for James Snee. You've taught me so much about how the Ph.D. marathon is run. Your race to completion, with all the highs and lows showed me that it is possible to finish. Your gentle (and sometimes not so gentle) hand rescued me from the abyss more than once. When I had nothing left and wanted to quit, you were there to push me back on track. Thank you. Like many others James volunteered part of his brain, and much of his sanity, to read and comment on the early versions of this document.

My sincerest gratitude also goes to George Neville-Neil, Tim Harris, Peter von Konigsmark, Christopher Bailey, Nick McKeown and Charles P. “Chuck” Thacker for reading and providing detailed feedback on earlier versions of this document. Each of your efforts made meaningful and lasting improvements to the quality of this dissertation and the work contained within it.

Thanks also go to Ionel Gog and Malte Schwarzkopf AKA “CamSaS”. Through many late nights we’ve struggled, debated, argued, fought, Cherry Box’d, joked and celebrated. You two have been the force behind our best publications, my worst ideas, and inspiration to keep going all these years. Thank you both.

It is easy to acknowledge those that have been close and present over the last 5 years, but there are some who have shaped this dissertation since well before it began. To Matthew “Doc” Chapman, Geoff Erickson, and Greg “Robo” Robinson, you are the role models that encouraged me to leave the trappings of a well paid job and become an impoverished student once again. Thank you? Thanks also go all of my friends at Zomojo / Exablaze for your support. I have never met a more genuine, honest or caring group of people and I am immensely looking forward to joining the team again.

Thanks also to Kevin Elphinstone and Gernot Heiser, for teaching (torturing?) me about operating systems and reminding me that I would never start a Ph.D after I had started working, and that I would never return to Australia after living in Europe. I always love a challenge...

Going all the way back to the start, my family have been behind and in front of me the whole way. To my brave parents, Philip and Kathy Grosvenor, who left the comfort of South Africa for a better education for your children, I will never be able to truly thank you for the impact that your have had. The support and encouragement of my parents and my brother, David Grosvenor, have kept me going in the darkest hours. Without your love and sacrifices no part of this work would have been possible. Dad, you taught me everything I know about computers, Mom, you taught me everything I know about writing and David, you taught me everything I know about perseverance. From the bottom of my heart, thank you all.

It is one thing to supervise, assist or befriend a Ph.D. student. It is another thing to live with one. It is something else entirely to marry one. To my precious Evianne. My lieve-anne’tje. My honing konijntje. How can I ever thank you? You have been my rock and my strength all of these years. You’ve celebrated my success and tended to my wounds. You’ve pushed me to keep writing and you’ve read this dissertation cover to cover. You’ve struggled through every single day of this dissertation with me; every tear and every smile. No moment has gone by without your love and support. This dissertation is as much yours as it is mine. All my thanks, all my love, and everything I am and have, go to you, Mrs Dr Evianne. L. (van Gijn) Grosvenor.

Contents

Chapter 1

Introduction

EVERY day we take it for granted that with the click of a mouse or a tap on a touchscreen we can instantly access information [1], keep in touch with friends [2], and globally transact finances [3, 4], music [5, 6], movies [5, 6], and physical goods [7–9] over the Internet. The computational machinery behind the Internet services provided by Amazon [7], Apple [10], Facebook [2], Google [11], Microsoft [12], and others is found in warehouses scattered all over the globe. These warehouses, or *datacenters*, contain many tens of thousands of computers, all working together to produce, manage, and maintain our online world [13]. We now take for granted that from a desktop, laptop, phone, or watch we can perform hundreds of Internet searches every day. Yet it takes 1,000s of computers working and communicating with each other to produce a single Google search result [14]. Such a result will be produced in less than 0.2 seconds. For so many computers to work together, they must be connected by a communications network.

The communications networks used inside datacenters inherit many of their features from the Internet. Like the Internet, information is transmitted through the network in discrete units called *packets*. Datacenter networks use *Internet Protocol (IP)* [15] encapsulated inside of *Ethernet frames* [16] to transfer packets. Computers (also known as *hosts*, *end-hosts*, or *machines*) are attached to the network and send packets to other computers via *Network Interface Controllers (NICs)*. Copper or optical fibre *links* connect NICs to network *switches*. Switches *forward* (or *route*) packets between their *ports*, some of which may be be connected to host NICs while other ports are connected to additional switches. Like Internet routers, datacenter switches use *statistical multiplexing* to forward packets between their ports. In a statistically multiplexed network, packets share the network in a first come, first served manner [17]. A packet arriving at a switch (or router) is either, forwarded immediately if the output link is free, or, forced to wait in a *queue* until the link becomes free. This means that different packet sources can *interfere* with each other in the network.

Network interference makes it difficult to predict the time that packets will take to traverse the network from their source to their destination. This time is known the *network delay* or *network latency*. Some packets may experience very high network latencies in extreme cases. These extreme cases are called *tail latencies* because they occur due to the combination of low-probability events, and, as a result, fall at the far end (or *tail*) of the latency distribution. Latency distributions in datacenter networks are typically non-normal and may have tails that are many orders of magnitude greater than the minimum and median values [13].

The close coupling of applications in datacenters magnifies the effects of tail latencies [13]. When 1,000s of computers work together, the latency of the slowest message dominates the overall performance. For example, in a realistic datacenter model, if as few as one machine in 10,000 is delayed, up to 18% of responses can experience long tail latencies [13]. This can have a tangible impact on user engagement and thus potential revenue [18, 19].

In this dissertation I address the problem of network interference causing tail latencies in datacenter networks. I do so by taking a *latency-first* approach. That is, I arrange my investigation to concentrate primarily on the problem of network latency caused by network interference. All other concerns (scalability, utilisation, throughput, cost, deployability etc.) are treated as secondary. I address these secondary concerns only after I have proposed solutions to network latency tails caused by network interference.

I begin my investigation by conducting a thorough historical, analytical and experimental analysis of the causes of, and potential mitigations to, network interference. Ultimately I conclude that *network scheduling* techniques are the best tool to address this problem. Network scheduling techniques provide a principled and mathematically sound method of providing *isolation* between different applications sharing a network. Previous work [20–23] on network scheduling typically sought to provide isolation in the form of throughput *guarantees*. Doing so often also resulted in guaranteed (*bounded*) network latency. In my work, I take the opposite approach. I first consider network scheduling techniques that offer bounded latency, and only subsequently concern myself with throughput availability and throughput guarantees.

Network scheduling techniques have been extensively studied in the context of the Internet network. When applied in this context, the techniques were found to be difficult to configure, required universal deployment of specialised (expensive) network hardware, and often required cooperation between multiple competing entities [24]. All of these problems can be resolved in the new context of datacenter networks.

The thesis of this dissertation is that it is both possible and practical to resolve network interference in datacenter networks by using network scheduling techniques. I show that it is possible to resolve network interference by taking a similar approach to Parekh and Gallager [20, 21]. Like Parekh and Gallager, I use a switch scheduling model and an input traffic regulator to provide isolation in the network. Unlike Parekh and Gallager, I take a latency-first, datacenter specific approach, rooted in the design of datacenter networks as they exist today. To do so, I create a simple, but realistic, switch scheduler model based on known and measured properties of existing datacenter switches. Using this model, I derive the necessary traffic regulation conditions to bound delay across the scheduler and consequently through the network. I express the traffic regulation conditions using a new type of traffic regulator called a *Leaky Token Bucket (LTB)* regulator. Finally, I express the resulting delay bound using a simple equation, parameterised over the maximum packet size, the number of senders, network link-rates, and switch performance. In doing so, I show that it is possible to precisely control latency in the network and therefore mitigate the effects of network interference.

To show that my approach is practical, I apply the Leaky Token Bucket regulator, and delay-bound calculation, in a system called QJUMP. QJUMP [25, 26] combines low-latency and high-bandwidth flows in datacenter networks while resolving the problem of network interference. The system is simple and coordination free, requires no changes to hardware or application software, and is designed to be immediately deployable in datacenter environments.

QJUMP operates by regulating the injection rate of low-latency traffic into the network using a fast implementation of a Leaky Token Bucket regulator. This bounds the time that the low-latency traffic will take to cross the network and thus mitigates the effects of network interference. Bounded-latency traffic is prioritised over high-bandwidth traffic so that both types of traffic can co-exist on the same network. This results in a coordination-free system offering both low-rate, bounded-latency traffic and high-rate, unbounded-latency traffic service levels. I thus show that it is practical to resolve network interference using a network scheduling approach.

Unfortunately, the QJUMP implementation does not offer a high-rate *and* bounded-latency service level. There is nothing intrinsic in my scheduling model, or regulator design to prohibit this. To resolve this problem, I propose three extensions to QJUMP: (*i*) EYEQJUMP, (*ii*) FASTJUMP, and (*iii*) R3CJUMP. The main difference between QJUMP and the following proposals is the introduction of dynamic, runtime coordination. Each system implements this coordination in different ways: EYEQJUMP uses end-host based coordination, FASTJUMP uses a centralised coordinator and R3CJUMP uses distributed coordination. Regardless of the implementation mechanism, the introduction of dynamic, runtime coordination allows hosts to negotiate *channels* in the network that have bounded latency and (up to) line-rate throughput. As part of this negotiation, limits can also be placed on the lowest acceptable throughput and/or highest acceptable latency of a channel. The result is an emulation of circuit switching behaviour over a packet switched network. The network can therefore be made to offer both statistical and bounded service simultaneously over the same infrastructure. I conclude that network scheduling is a viable and practical approach to controlling network interference in datacenter networks.

The remainder of this chapter contains a brief overview of my contributions, related publications, and highlights of the dissertation to follow.

1.1 Contributions

In this dissertation I make the following primary contributions:

1. I propose a novel, discrete scheduling model based on known and measured properties of commodity datacenter switches. Using this model, I determine the packet transmission

conditions required to provide isolation and bounded latency in datacenter networks using commodity hardware. I describe these transmission conditions using a new, fluid-flow regulator called a Leaky Token Bucket (LTB). I express the resulting delay bound using a simple equation, parameterised over only four parameters. Finally I show that the combination of my discrete scheduling model and fluid-flow traffic regulator can achieve similar bounds to previous work, which used a fluid-flow scheduler and a discrete traffic regulator (Chapter 3).

2. I show that the model derived in contribution (1) can be implemented in a simple, coordination free, and immediately deployable manner in datacenters. This implementation requires minimal modifications to host kernels and requires no modifications to applications, host hardware and the datacenter network. It offers bounded latency communication for low-rate, latency-sensitive applications in datacenter networks. The implementation results in measurable improvements to network interference between datacenter applications (see Chapter 4).

3. I discuss in detail three proposals to extend the implementation from contribution (2) to accommodate guaranteed latency at line-rate throughput. Each of these proposals requires dynamic coordination. I discuss end-host, centralised and distributed coordination schemes. The proposals come at the cost of increased deployment complexity and/or network requirements (see Chapter 5).

All of the models, algorithms, physical implementations, experimental tools, and analysis used to produce the above contributions are my own work. However, as with any research project, parts of the work leading to this dissertation were completed in collaboration with others. In particular, Ionel Gog implemented and configured the Musketeer [27] system which was used to generate and run Hadoop Map-Reduce workloads used in some experiments, especially those found in Sections 2.4 and 4.5. Ionel also implemented an initial version of the NS2 simulation of QJUMP used to produce the results in Section 4.5.3. Additionally, Malte Schwarzkopf and I implemented parallel versions of a tool called `dag-join`, which was used to analyse packet traces and measure latency across network devices. This tool was important for producing the in-network latency measurements used in many places throughout the dissertation particularly in Sections 3.4.6 and 4.5.1. Malte and I cross-checked the results of our implementations to find and remove bugs resulting in a tool that I am confident in. Furthermore, many of the plots throughout the document were generated using the `matplotlib` library with the aid of scripts initially written by Malte. In addition to specific assistance, both Malte and Ionel, along with my supervisors, co-authors, colleagues, fellow researchers and students of my own have contributed to discussions and the progression of the ideas found within. This to-and-fro, although indefinable, was vital to the development of the ideas, implementations and publications leading to this final dissertation.

1.2 Publications

This dissertation is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other any qualification. Some of the contents also appears in the following publications:

- MATTHEW P. GROSVENOR, MALTE SCHWARZKOPF, ROBERT N. M. WATSON, ANDREW W. MOORE. R2D2: Bufferless, Switchless Data Center Networks Using Commodity Ethernet Hardware (poster). In *Proceedings of the Special Interest Group on Communications 2013* (SIGCOMM 2013 - Hong Kong).
- MATTHEW P. GROSVENOR, MALTE SCHWARZKOPF, IONEL GOG, ANDREW W. MOORE Jump the Queue to Lower Latency. In *;login: The Usenix Magazine* April 2015, Vol. 40, No. 2.
- MATTHEW P. GROSVENOR, MALTE SCHWARZKOPF, IONEL GOG, ROBERT N. M. WATSON, ANDREW W. MOORE, STEVEN HAND AND JON CROWCROFT. Queues don't matter if you can JUMP them! In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation* (NSDI 2015) (*best paper award*).
- HITESH BALLANI, PAOLO COSTA, CHRISTOS GKANTSIDIS, MATTHEW P. GROSVENOR, THOMAS KARAGIANNIS, LAZAROS KOROMILAS, GREG O'SHEA. Enabling End-Host Network Functions. In *Proceedings of the Special Interest Group on Communications 2015* (SIGCOMM 2015 - London) (*NB: authors listed alphabetically by surname*).
- NOA ZILBERMAN, MATTHEW P. GROSVENOR, DIANA POPESCU, NEELAKANDAN MANIHATTY BOJAN, GIANNI ANTICHI, MARCIN WOJCIK, AND ANDREW W. MOORE Where Has My Time Gone? In *Proceedings of Passive and Active Measurements 2017* (PAM 2017 - Sydney)

In addition to the above, the following publications that I have authored, or contributed to, have influenced my work, although they are not directly related to the contents of this dissertation:

- MATTHEW P. GROSVENOR. uvNIC: Rapid Prototyping Network Interface Controller Device Drivers (poster). In *Proceedings of the Special Interest Group on Communications 2012* (SIGCOMM 2012 - Helsinki).
- MALTE SCHWARZKOPF, MATTHEW P. GROSVENOR AND STEVEN HAND. New wine in old skins: the case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (APSys 2013 - Singapore).
- IONEL GOG, MALTE SCHWARZKOPF, NATACHA CROOKS, MATTHEW P. GROSVENOR, ALLEN CLEMENT, STEVEN HAND. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the 10th European Conference on Computer Systems* (EuroSys 2015).

1.3 Dissertation Overview

The rest of this dissertation is organised as follows:

- Chapter 2 — *Background* — Introduces the design and construction of datacenter networks. It contains a thorough discussion of the historical context of network interference, related literature and contemporary approaches to resolving network interference. It also contains a brief measurement based study demonstrating the effects of network interference under controlled conditions and when using typical datacenter applications. Additionally it contains (where possible) measurements of the effectiveness of contemporary approaches to resolving network inference.

- Chapter 3 — *Bounding Delay in Datacenter Networks* — Includes a description of a simple, but realistic, scheduling model derived from datacenter switches. This model is used to derive a new type of traffic regulator called a *Leaky Token Bucket* regulator. The chapter details how delay bounds can be calculated and enforced by combining these models and how they relate to previous work. It also includes a description of a fast software-based implementation of the regulator and discusses (in detail) many practical considerations for implementing network scheduling in datacenter networks.

- Chapter 4 — *Implementing Predictable Datacenter Networks* — Describes and evaluates QJUMP. QJUMP is a simple but useful implementation of a co-ordination free, distributed system implementing the equations and regulator derived in Chapter 3. This system was first published in my paper "Queues don't matter when you can JUMP them" [25, 26] (Chapter 4).

- Chapter 5 — *Improvements and Future Work* — Describes future work and potential solutions to some of the problems found in Chapter 4. This includes a detailed description of three proposed systems: (*i*) EYEQJUMP, (*ii*) FASTJUMP and; (*iii*) R3CJUMP. The EYEQJUMP proposal extends on QJUMP to allow for centralised coordination using end-host cooperation in full-bisection bandwidth networks. This improves on the throughput limitations imposed by QJUMP. The EYEQJUMP proposal is generalised to operate on arbitrary network topologies by including a centralised coordinator in the FASTJUMP proposal. This proposal introduces planing and routing traffic in the network, but has a central point of failure (the coordinator). The R3CJUMP proposal improves upon the FASTJUMP proposal by introducing fully distributed control but introduces broadcast overheads which may be intolerable at certain scales or with certain workloads.

- Chapter 6 — *Conclusions* — Contains some concluding statements and final remarks.

To facilitate the reader all substantial chapters (2-5) conclude with a short chapter summary. The chapter summaries should provide the reader with enough background material to start reading the following chapter independently of the rest of the dissertation.

Chapter 2

Background

2.1 The Rise of Hyper-scale Datacenters

MODERN web-services companies such as Amazon, Google, Facebook and Microsoft run at “hyper-scale” [28, 29]. Their systems support billions of users[1], index over a billon websites[2], and serve billions of requests per day[3]. Furthermore, these services are growing rapidly; Facebook now adds two new datacenters nearly every year [30] and, every day, Amazon adds enough new server capacity to support all of its global infrastructure needs from 2004 (when it was a $7B annual revenue company) [31]. The network infrastructure required by hyper-scale datacenter operators must scale to support tens of millions of endpoints (i.e. bare metal or virtualised servers) and middleboxes (i.e. physical and virtualized network functions) [32] and supply petabits per second of bandwidth across the network[4] [34].

In general there are few concrete details about the construction of large-scale datacenters and their networks. Internet giants such as Amazon, Apple, Google and Microsoft continue to view the technical details of their datacenter infrastructure as trade secrets, which yield competitive advantages over one another. This has made them historically reluctant to share many details. Recently, Facebook has defied the convention by launching the “Open Compute Project (OCP)”[35]. The OCP aims to “democratize access to the best server, storage and data center technologies available” [35]. Over time, Apple and Microsoft[5] have joined this effort (to varying degrees) and both Amazon [31] and Google [34] have recently also revealed some details of the scale of their operations. Although details are still limited (and often dated) it is possible

[1] As of June 30, 2016, Facebook reports that it has 1.13 billion daily active users. The company further claims to have had an average of 1.71 billion active users over the same month. See Appendix A.11.1.

[2] As of September 2016, Netcraft estimates that there are over 1.29 billion websites. See Appendix A.22.1.

[3] The Statistic Brain Research Institute estimates that in 2015, an average of 7.8 billon Google searches were performed each day. See Appendix A.25.1.

[4] Alexey Andreyev, Facebook Engineering blog (14 November 2014) [33]. See Appendix A.11.3.

[5] Cade Metz, wired.com (March, 2015) and, Frank Frankovsky, opencompute.org blog (16 January 2013). See Appendix A.27.1 and A.23.3.

to combine these sources to arrive at reasonable estimate of how contemporary datacenters are constructed and operate.

2.1.1 Datacenter Applications

The need for the hyper-scale datacenters operated by Facebook, Google, Microsoft and others arose out of the scale of problems being tackled. These included indexing and interactively searching the entire Internet and providing social networking and business productivity services to billions of people. Unlike the provision of shared infrastructure *cloud services*[6], these problems require massively parallel, highly distributed software systems [36]. And, unlike super-computers, the software systems require flexibility, generality and 99.999% or better uptime over several years of service [36]. As a result datacenters typically run a small number of very large applications using shared management and deployment infrastructure [36].

The applications running in datacenter environments can broadly be classified into three distinct layers: bottom, middle and top:

1. **Bottom layer** – Distributed Coordination – In this layer are management, scheduling and coordination services. These include distributed timing and consensus services such as the Precision Time Protocol daemon (PTPd) [37], Chubby [38], and Zookeeper [39], and task placement, management and scheduling systems like Borg [40], and Omega [41].

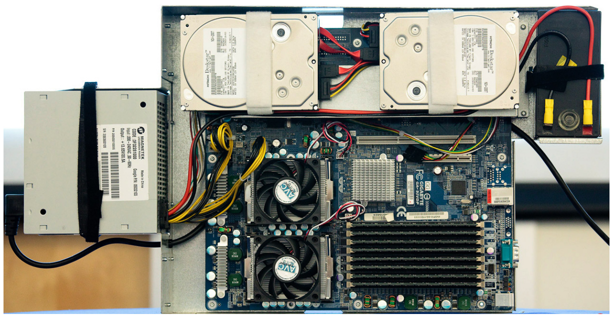
2. **Middle layer** – Distributed Data Access and Storage – This middle layer runs services that provide distributed data storage and access systems. It includes block storage systems such as the Google File System (GFS) [42] or the Hadoop Distributed File System (HDFS) [43] to provide unstructured distributed data storage. Above these, structured data services such as BigTable [44], MegaStore [45], Spanner [46], and MySQL [47] provide structure/database abstractions with various consistency levels. To keep these systems fast, some operators (e.g. Facebook) use in-memory caching services like memcached [48].

3. **Top layer** – Distributed Programming Frameworks and Distributed Applications – On the top layer are the distributed programming frameworks such as Google Map-Reduce [49], Hadoop Map-Reduce [50], Spark [51], and Naiad [52]. Actual datacenter applications such as web-search engines, social networking systems, email and office productivity systems and video and image delivery systems run within these higher level frameworks. These systems often have "front-end" language extensions such as FlumeJava [53], Hive [54], and Pig [55]. Recent work has extended these abstractions to a single unified interface which can automatically select the correct abstraction based on job properties [27].

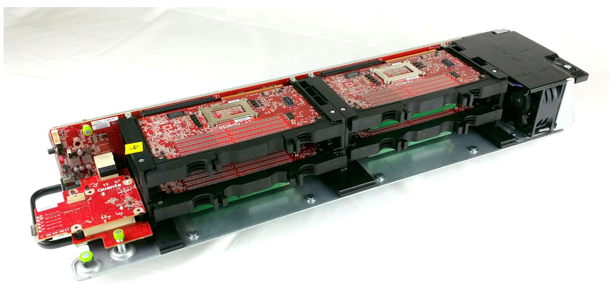
2.1.2 Datacenter Host Configurations

The software systems running inside of datacenters operate on commodity x86 [56] based Central Processing Units (CPUs). Depending on the source, it is generally agreed that warehouse scale

[6]From providers such as Amazon Web Services , Microsoft Azure and Google Compute Engine



(a) The Google custom server chassis (circa 2005)



(b) The Facebook Yosemite modular sled (circa 2015)

Figure 2.1: *Server designs from hyperscale datacenter operators (Facebook and Google).*

datacenters operate using between 40,000[7] and 80,000 [30, 31] hosts per facility with 100,000 hosts being the upper limit [8] [31]. Given the scale of operation, it is not surprising that the hosts themselves are custom built. In 2009 Google demonstrated their (2005 vintage) custom built server and chassis (See Fig. 2.1a). The machines were a 2 socket x86 CPU design with, 2 hard disk drives, 8 memory slots and a lead-acid backup battery.

Since 2009, Amazon, Facebook[9] and Microsoft[10], have also revealed that they build custom severs. Amazon, Facebook and others have also had Intel build custom CPUs for them. For example, Amazon's CPUs contain higher than usual core counts (10 CPUs), running faster than usual and are said to include an FPGA fabric for extra flexibility[11].

The most recently published host design is the "Yosemite modular micro-sever" published by Facebook in early 2015[12] (See Fig. 2.1b). According to Facebook, Yosemite was designed in response to power and scaling issues experienced with "mainstream" two socket server designs. This suggests that the two socket design used by Google in 2005 has likely been prevalent throughout the industry for at least the last decade.

The Yosemite modular micro-sever comprises four single socket System on Chip (SoC) Xeon-D[13] based server cards, each connected to a carrier board and housed in a "sled" enclosure[14]. The carrier sled bears a strong resemblance to Facebook's earlier "Group Hug" design[15]. In each

[7]Rich Miller, datacenterknowledge.com (1 April 2009) and Stephen Shankland, cnet.com (1 April 2009) who both attended the 2009 Efficient Datacenter Summit held by Google. See Appendix A.5.1,A.6.1 and A.13.1.

[8]Rich Miller, datacenterfrontier.com (23 September 2015) who attended the 2014 Amazon Re:Invent conference. See Appendix A.7.1.

[9]Hu Li, Facebook Engineering Blog (10 March 2015). See Appendix A.11.2.

[10]The Open CloudServer design on the OCP is attributed to Microsoft. Corroborated by marketing material from Microsoft. See Appendix A.21.1 and A.23.1.

[11]Yevgeniy Sverdlik, datacenterknowledge.com (13 November 2014) and corroborated in part by a press release from intel.com (June 12, 2015). See Appendix A.6.2 and A.16.1.

[12]Hu Li, Facebook Engineering Blog (10 March 2015). See Appendix A.11.2.

[13]Marketing materials, Intel. See Appendix A.16.2.

[14]As detailed in the Open Compute project Yosemite project specification. See Appendix A.23.2 for more details

[15]Joel Hruska, extremetech.com (28 January 2013), corroborated by Frank Frankovsky, opencompute.org blog (16 January 2013). See Appendix A.10.1 and A.23.3.

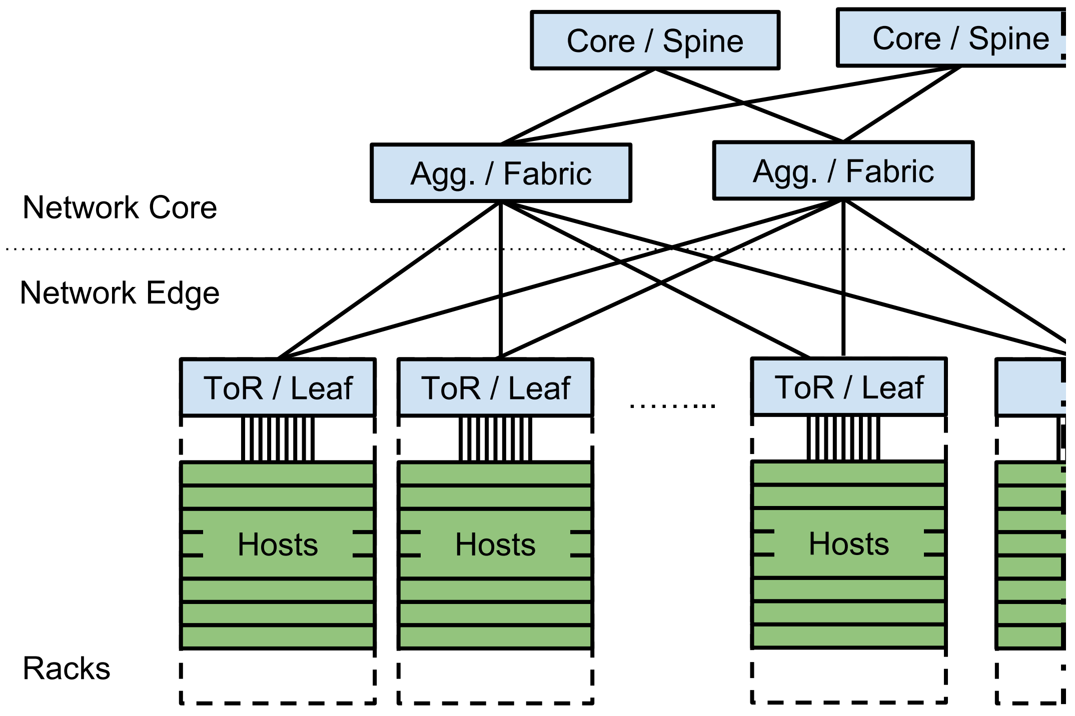


Figure 2.2: *Datacenter network architecture.*

sled a 4-port "multi-host" NIC connects the host to 40Gb/s or 50Gb/s networking.

2.1.3 Datacenter Network Architectures

Figure 2.2 shows a high level depiction of a datacenter network. At the bottom of the figure are hosts, contained within *racks*. The term "rack" refers to a 19 inch "relay rack" (or more recently "telecommunications rack") which was standardised as early 1934 [57]. A standard rack comprises 42 Rack Units (RUs) (or Us), each of which are 1.752 inches high and 19 inches wide. Facebook's "Open Rack" has the same external dimensions as a standard rack, but accommodates modules that are 21 inches wide [35]. A rack is the standard unit of deployment for datacenter hosts. The Facebook Yosemite sled design is intended to house up to 192 hosts (SoCs) in each rack, giving it a density of one host per 0.25RU.

Various terminologies are used to describe the components of a datacenter network[16] [28, 29, 34, 36]. Hosts within a rack are connected to a *Top of Rack (ToR)* [17] (or *leaf*) switch. These switches are connected to *aggregation* (or *fabric*) switches, which are connected to *core* (or *spine*) switches (see Fig. 2.2). ToR switches and end-hosts lie at the *edge* of the network while aggregation and core switches fall in the *core* of the network.

The arrangement of connections between components in the network (the *network*

[16]Alexey Andreyev, Facebook Engineering blog (14 November 2014). See Appendix A.11.3.

[17]The term "Top of Rack" switch is a misnomer. In the datacenter context, Top of Rack (ToR) switches are usually mounted in the middle of the rack to shorten cable lengths [36].

topology) varies by deployment. Facebook[18], Google [34] and Microsoft [28, 29] use variants of the Clos architecture [58] in the core of their networks. Clos networks provide *non-blocking* switching capacity that is greater than the capacity of any single switch. In a non-blocking network, any input to the network can be connected to any unused output regardless of any existing connections [59]. Non-blocking networks offer *full bisection bandwidth*. The bisection bandwidth of a network is the capacity of the network when partitioned in half [60]. If a network offers full bisection bandwidth, then any input to the network can communicate with any unused output at full speed (*line-rate*).

The edge of the network is typically *oversubscribed*. The Facebook "Fabric"[19] ToR switches provide 12× 40Gb/s connections to hosts the rack and 4× 40Gb/s connections to the fabric switches. The Google "Jupiter" [34] network is much denser. In one configuration, 48× 40Gb/s connections are provided to hosts in the rack and 16× 40Gb/s to the aggregation switches. In both cases, 16-port, 40Gb/s merchant Ethernet [16] switching chips are used, and, in both cases, there is a 3:1 *oversubscription ratio* across each ToR switch. Both Facebook and Google use their own switch designs [20] [34]. Microsoft also appears to be using 40Gb/s networking to the hosts[21] [28, 29], although the networking infrastructure appears to be commodity, provided by Arista Networks[22].

There is evidence that Facebook (and by inference, Amazon, Google, Microsoft *et al.*) are moving towards 50Gb/s and 100Gb/s networks. The Open Compute Project specifications require the Yosemite shared host NIC (see §2.1.2) to be capable of running four, 10Gb/s connections or two, 25Gb/s connections over a Quad 25Gb/s Small Form-factor Pluggable (QSFP28) connector[23]. The QSFP28 connector[24] is used by vendors of 50 Gb/s and 100Gb/s Ethernet equipment. In late 2015[25] and early 2016[26], Facebook announced work on an expanded ToR switch called the "Wedge 100", which is based on the Broadcom BCM56960 "Tomahawk" Ethernet switching silicon[27]. The Wedge 100 is a 32 port QSFP28, 100Gb/s switching platform. As of yet, no concrete details have emerged about how, or if, this change will affect Facebook's datacenter network topology designs.

Both Facebook's and Google's networks are comprised of sub-networks which are used

[18]Alexey Andreyev, Facebook Engineering blog (14 November 2014). See Appendix A.11.3.

[19]Alexey Andreyev, Facebook Engineering blog (14 November 2014). See Appendix A.11.3.

[20]Yuval Bachar and Adam Simpkins, Facebook Engineering Blog (18 June 2014). See Appendix A.11.4.

[21]The Open CloudServer design on the OpenCompute project (`www.opencompute.org`) is attributed to Microsoft. Corroborated by marketing material from Microsoft. See Appendix A.21.1 and A.23.1.

[22]Press release, Arista. See Appendix A.2.1. Corroborated by personal discussions while working at Microsoft Research

[23]Standardised by the Small Form Factor committee of the Storage Networking Industry Association. See Appendix A.26.1.

[24]Also known as QSFP100 by some vendors e.g Arista. See Appendix A.2.2 for more details

[25]Jasmeet Bagga and Zhiping Yao, Facebook Engineering Blog (19 November 2015). See Appendix A.11.4.

[26]Jasmeet Bagga, Hany Morsy and Zhiping Yao, Facebook Engineering Blog (9 March 2016). See Appendix A.11.5.

[27]Marketing material, Broadcom. See Appendix A.3.1.

to facilitate management and resource allocation. These sub-networks are called "cells" [61], "clusters" [34], "containers" [62] or "pods"[28]. For the remainder of this dissertation I will adopt the nomenclature of *pod-scale network* to refer to one of these sub-networks. Facebook defines a pod as a standard "unit of network". Each Facebook pod includes 48 server racks. Assuming each rack contains between 48 and 192 hosts, this means that each pod contains between 2000 and 9000 hosts. For the remainder of this work, I will assume that a pod-scale network contains approximately 5000 hosts unless otherwise stated.

Recently there has been increased discussion of "rack-scale computing" and "rack-scale networks"[63–65]. The key observation of rack-scale computing is that, since datacenter operators provision hosts at a granularity of whole racks, it logical to construct larger *rack-scale computers* at the same granularity. A rack-scale computer would do away with the boundaries between hosts within a rack and *disaggregate* their resources to increase performance, density, power efficiency, and/or reduce costs. For example, disk or memory resources could be shared among many SoCs in a rack, obviating the need to have a disk per host. However, doing so would place more pressure on the internal network in such system. It is predicted that[29] rack-scale computers could reach the density of small pods, containing the equivalent of thousands hosts. This suggests that much of the work presented in this dissertation will be equally relevant to rack-scale networks.

2.1.4 Datacenter Network Protocols

Datacenter networks owe their existence and much of their history to the development of the Internet. Key to the development of the Internet was the co-development of the Internet Protocol (IP) and Transmission Control Protocol (TCP) [66]. On the Internet today there is a mixture of IP version 4 (IPv4) [15] and IP version 6 (IPv6) [67] traffic. The most substantial difference between the two protocol versions is the address size. While IPv4 is limited to approximately four billion addresses (32 address bits), IPv6 supports around 3.4×10^{38} addresses (128 address bits). The majority of datacenter operators appear to still be using IPv4 in their internal networks[30]. One notable exception is Facebook. In 2014 Facebook announced a long running project to convert their datacenter infrastructure to IPv6. According to Facebook, in 2014 "100% of our hosts we care about respond on IPv6" and "75% of our internal traffic is now IPv6"[31] The transition to IPv6 was expected to be fully complete within the following 2 years (i.e. at the time of writing this dissertation).

The scale of datacenter networks means that traditional layer 2 services such as Address Resolution Protocol (ARP) [68] perform poorly [69]. To combat these issues, datacenter operators

[28]Alexey Andreyev, Facebook Engineering blog (14 November 2014). See Appendix A.11.3.

[29]Marketing material, Hewlett Packard Research and Intel. See Appendix A.14.1 and A.16.3.

[30]Microsoft Azure Frequently Asked Questions, Amazon Web Services Documentation – Load Balancing, and Google Cloud Platform Documentation – Networking and Firewalls. See Appendix A.21.2, A.1.1 and A.13.2.

[31]Presented at the IPv6 World Congress 2014, (see Appendix A.18.1) by Paul Saab. Slides (see Appendix A.11.6) posted by to the IPv6 public group on Facebook. See A.11.7. Corroborated by Dan York (22 March 2014) on the Internet Society blog (see Appendix A.17.1) who attended the conference.

employ IP layer routing protocols like Border Gateway Protocol (BGP) [70]. Facebook uses "standard BGP4 as the only routing protocol" with a "centralized BGP controller that is able to override any routing paths", while Google has resorted to building its own protocol dubbed "FirePath". Like Facebook's solution, FirePath introduces distributed routing decisions in merchant silicon switch ICs but yields control to a logically centralised controller [34]. Microsoft is thought to be using a system much like VL2 [28, 29, 69].

Regardless of the particular variant of IP and the routing protocol used, connections between hosts are most often made using Transmission Control Protocol (TCP). A 2010 study found that up to 99.91% of datacenter traffic uses TCP [71].

There are two defining characteristics for TCP: (*i*) the 'on-the-wire' packet format, and; (*ii*) the congestion control algorithm. The TCP packet format has remained relatively unchanged since version 4 was first specified in 1978 [72]. On the other hand, numerous TCP congestion control algorithms have been proposed.

2.2 TCP Congestion Control

In 1987 the early Internet was in trouble. A series of 'congestion collapses' saw throughput between physically close sites drop from 32kb/s to only 40b/s. The culprit was determined to be the 4.3BSD implementation of TCP [73]. At the time, TCP had no congestion control. In the seminal paper *Congestion Avoidance and Control*, Van Jacobson (with credit to Mike Karels) introduced the first TCP congestion control algorithm, retrospectively named "TCP Tahoe". The key insight motivating the development of the algorithm was the principle of "conservation of packets". Specifically that:

> A new packet isn't put into the network until an old packet leaves.

Jacobson argued that if this principle could be obeyed, then congestion collapse would only occur in exceptional circumstances. He therefore approached the TCP congestion control problem as a task of "...finding places that violate conservation and fixing them." Although seemingly trivial, these insights will become a key part of my work described Chapter 3. On the basis of these insights, Jacobson developed the first TCP congestion control algorithm. It had three essential components:

1. Slow-start: While a new connection was starting, the algorithm would start at a low (slow) transmission rate and exponentially increase its sending window to probe for available bandwidth to be consumed.

2. Round-trip time estimation: Once the connection had reached equilibrium, it would use a fast to method estimate the variance in Round-trip Time (RTT) measurements which would be used to make good estimates for retransmit timeouts.

3. Congestion avoidance: When a stable connection with good RTT estimates experienced packet loss, it could assume (with high probability) that congestion was occurring. This would be mitigated by exponentially reducing the sending rate and then linearly increasing it again to probe for unused bandwidth.

Using these simple principles, Jacobson brought the Internet's congestion collapse problems under control.

There are now many TCP congestion control algorithms in common use. These algorithms are mostly small variations on the original ideas and are optimised for different types of connections. For example, high bandwidth, low delay networks (e.g. datacenters) have different properties to low bandwidth, high loss rate networks (e.g microwave/wireless links) which are again different to high bandwidth, high loss, high delay networks (e.g. satellite links). Some of the major variants of TCP in production today are:

- TCP CUBIC [74]: the default TCP congestion control algorithm used by the Linux kernel since version 2.6.19 (still used today in version 4.6) as well as in Mac OS X Yosemite. TCP CUBIC is optimised for high-bandwidth, high-latency connections which are found in homes, businesses and Internet facing devices.
- Compound TCP [75]: the default TCP congestion control algorithm used by the Windows kernel. It is optimised for "high-speed, long distance" networks which in practice is similar to the constraints above.
- TCP New-Reno [76]: the default TCP congestion control algorithm used by the FreeBSD kernel. TCP New-Reno is a modified version of TCP Reno which includes selective acknowledgements to improve bandwidth utilisation.
- Datacenter TCP (DCTCP) [71]: First published in 2010, DCTCP is optimised for high-bandwidth, low-latency datacenter networks with switches that support Explicit Congestion Notification (ECN). DCTCP uses the rate at which ECN markings are received to estimate queue occupancy and thereby control queue depths. DCTCP has been integrated into the Windows Server 2012 operating system[32] as well as the Linux 3.18 kernel[33]. Google use an algorithm similar to DCTCP in their datacenters [34] .

Despite their superficial differences, all variants of TCP attempt to answer the same fundamental question as the original: "What is the appropriate sending rate for the current network path?" [77]. This question is not answered explicitly. Rather, each TCP connection determines its sending rate by probing the network path. This probing works by increasing the sending rate until a congestion signal is detected, then backing off and trying again. The congestion signal will trigger when multiple connections compete for a bottleneck resource

[32]Discussed on Microsoft Technet (9 May 2012). See Appendix A.21.3.
[33]Linux 3.18 release discussed on kernelnewbies.org (7 December 2014). See Appendix A.19.1.

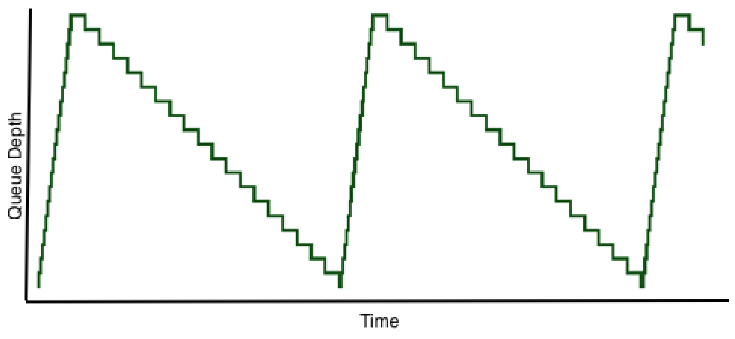


Figure 2.3: *TCP's characteristic sawtooth queueing pattern (Nichols and Jacobson [81]).*

causing queuing to exceed some bound. Typically, the bound is the size of memory supporting the queues, and the trigger is packet loss when the queue is exhausted. Some variants of TCP employ more sophisticated congestion signalling instead of/in addition to packet loss. For example, modified switches/routers may insert Explicit Congestion Notification (ECN) markings into packets [71, 78] and/or TCP may use fine-grained timing [75, 79, 80] provided by the NIC and/or hosts.

Regardless of the signalling method, probing for congestion and then backing off again causes queues in the network to build and then decrease in a "saw-tooth" wave shape [71, 81] (e.g. see Fig. 2.3). In the early Internet this was a feature. It ensured that the slowest (yet typically most expensive[34]), links were kept fully utilised. However, as queues build up so too does the delay across them. Packets sharing the same queue but belonging to different connections/protocols will also experience this delay. Depending on the amount of memory/depth of the queue, these delays might be many milliseconds in the worst case. In the early Internet, memory was limited by expense and delays were high. As a result, the time spent in queues was a relatively small faction of the end-to-end delay. But, as speeds at the edge of the Internet have increased, and memory has become inexpensive, buffers have deepened and long delays have now become problematic. This phenomenon has become known as "bufferbloat" [82]. As I will describe in the next section, effects similar to bufferbloat are also experienced in datacenter environments.

2.3 Network Interference in Datacenters

Networks in datacenters operate at very high speeds (10Gb/s to 40Gb/s) and transfer packets over paths only tens of meters long (nanoseconds of delay). This means the queueing delays can account for a sizeable fraction of end-to-end delays. Additionally, datacenter applications are distributed and tightly coupled, leading to correlated packet arrival events. For example, in a *distribute-aggregate* workload, work may be distributed to many hosts (known as *fan-out*), which then report their results back to a single host (known as *fan-in*). A high degree of fan-in can cause periods of intense congestion resulting in *congestion collapse* (e.g. "TCP incast" [83])

[34] For example long distance terrestrial links, under-sea cables and satellite links.

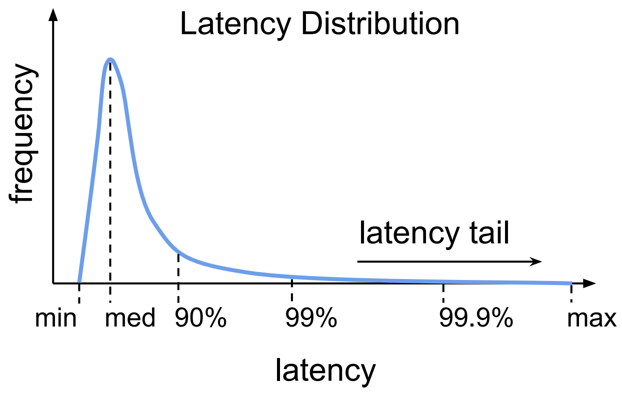


Figure 2.4: *A depiction of network latency distributions and latency tails.*

and further exacerbate latency and queueing issues.

Datacenter applications (see §2.1.1) are mixed and have varying network needs. For example, timing and coordination applications are often dependent on consistent, low latency network performance. By contrast, storage and data processing applications can be limited by network throughput[35]. However, switch queues are shared between these different types of applications. This means that congestion from throughput-intensive applications (like Hadoop) may cause queueing that delays packets from latency-sensitive applications (like PTPd). In effect, throughput intensive applications can produce a form of *interference*[36] or *crosstalk*[37] which affects latency sensitive applications. For the remainder of this dissertation I will refer to this effect as *network interference*.

When latency sensitive applications are subject to network interference, skewed latency distributions develop. An example of such a distribution is given in Figure 2.4. In the figure, the most frequently occurring latency values are at the left of the curve, and there is small region between which the minimum, medium and 90th percentile of values lie. At the right-hand side of the curve are infrequently occurring values, which have very high latencies. This region is known as the *long tail* or simply the *tail* [13, 84, 85]. In practice, latency tails may be many orders of magnitude greater than the minimum and median values. Much of the work in this dissertation is concerned with minimising and controlling latency tails.

The close coupling of applications in datacenters magnifies the effects of tail latencies [13]. When 1,000s of computers work together, the latency of the slowest message can dominate the overall performance. For example, Barroso *et al.* considered a hypothetical system,

[35]Research conducted at the University of Cambridge by Frank McSherry (an author of Naiad [52]), in collaboration with my colleague and QJUMP collaborator Malte Schwarzkopf. Published on Frank's personal blog (8 July 2015). See Appendix A.12.1.

[36] In general, extraneous energy, from natural or man-made sources, that impedes the reception of desired signals. `http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm`

[37] Any phenomenon by which a signal transmitted on one circuit or channel of a transmission system creates an undesired effect in another circuit or channel. `http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm`

Setup / Percentile	*10th (μs)*	50th (μs)	*90th (μs)*	*99th (μs)*
otherwise idle network (A)	36	85	118	126
shared switches (B)	37	110	120	130
shared host egress (C)	168	228	259	268
shared host ingress (D)	55	125	249	278
shared host ingress and egress (E)	171	221	224	229
shared switch queue (F)	1,790	1,920	2,010	2,100

Table 2.1: *Latency of `ping` vs. `iperf` with various degrees of network resource sharing. All results taken over 5,000 samples.*

with a typical response time of 1ms, but with a 99.99th percentile response time of 1s. In such a system, if each request required 2,000 servers to handle it, up to 18% of requests would take over 1 second to complete [13, 36]. Such delays can have a tangible impact on user engagement and thus potential revenue [18, 19]. Controlling tail latencies and isolating latency sensitive applications from interference caused by throughput intensive applications is therefore an important problem to be solved in datacenter networking.

2.3.1 Quantifying Network Interference

Network interference may occur at any place where packets share network resources, particularly in shared queues or buffers. Applications may share ingress or egress queues in the host, share the same network switch scheduler, or share the same queue in a switch. In the following section I present the results of an empirical study, designed to assess the potential impact of network interference from different types of sharing, and to determine the appropriate location to begin resolving it.

To perform the experiment, I emulate two major classes of datacenter applications: (*i*) a latency-sensitive Remote Procedure Call (RPC) application using `ping`, and; (*ii*) a throughput-intensive bulk transfer application using one or more instances of `iperf`. The `ping` application sends small Internet Control Message Protocol (ICMP) echo-request messages to a destination host, which then responds with ICMP echo-reply messages [86]. The time between the request and the reply is measured, resulting in an estimate of the network *Round-trip Time (RTT)*. The `iperf` application establishes a TCP connection between an `iperf-client` and an `iperf-server`. The client then attempts to saturate the connection by sending many large sized messages. The rate at which these messages are sent/received is measured, resulting in an estimate of available network bandwidth.

Many of the distributed systems described in Section 2.1.1 implement RPC and/or bulk-transfer messaging. Bulk transfers are likely to cause congestion, while RPCs are likely to suffer from network interference. By measuring increases or decreases in `ping` RTTs, it is possible to measure directly an application's observed impact from network interference.

Table 2.1 shows the results of arranging `ping` and `iperf` with various types of network queue sharing as detailed in Figure 2.5. The arrangements are as follows:

A **Otherwise idle network** (Fig. 2.5a) : This is the base case. Here only one instance of `ping` is probing the network, it is otherwise idle. The median (50th percentile) `ping` round trip time is 85μs and the 99th percentile is 126μs.

B **Shared switches** (Fig. 2.5b): In this case, there is an `iperf` client sending traffic to an `iperf` server and a `ping` source probing a destination. Four independent hosts are used (one for each source/destination pair). Nothing is shared between the hosts except the switches in-between them. The switches fairly share traffic between the two independent sources. Sharing switches produces only a small degree of network interference. The median (50th percentile) RTT increases by 25μs but the 10th, 90th and 99thpercentiles move by less than 4μs.

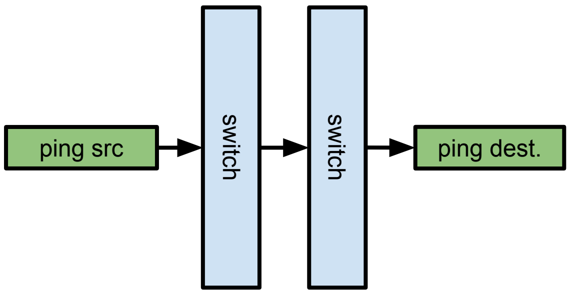
C **Shared host egress** (Fig. 2.5c): This is the same as case (B), however, both the `ping` and `iperf` sources are located on the same host. This means that the entire host egress path (i.e. kernel, driver & NIC) is shared between the sources. In this case the impact of sharing is significant. Across all percentiles an impact of at least 110μs is seen. These results are roughly double the base case values.

D **Shared host ingress** (Fig. 2.5d): This is the opposite of case (C). The `ping` and `iperf` sources are located on separate hosts, but the destinations are located on the same destination host. Again, the entire host ingress path (i.e. NIC, driver & kernel) is shared. Interestingly, in this case, the 10th and 50th percentile (median) values are much lower than in case (C). High speed network cards perform an optimisation called "interrupt moderation"[38]. This delays packets inside the NIC until either a sufficient number of packets have been received or a timeout has expired. When `ping` shares a NIC queue with `iperf`, many more packets are received leading to shorter interrupt moderation periods.

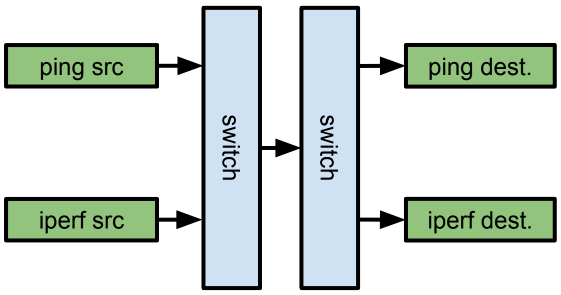
E **Shared ingress and egress** (Fig. 2.5e): This is the combination of case (D) and (D). In this case, only two hosts are used. Both sources reside on one and both destinations reside on the other. It would appear that the results here are dominated by the interference in the source egress path because the results in this test are approximately the same as the shared source host egress case (see case (C)).

F **Shared switch queue** (Fig. 2.5f): This final case is the same as case (E) with the addition of second pair of hosts acting as an `iperf` client/server pair. The addition of the extra pair creates contention across the shared link between the switches. Both `iperf` clients will attempt to transmit at 10Gb/s, but neither will be able to achieve this. Instead, TCP's congestion control algorithm will infer a fair share of 5Gb/s for each source. As described in Section 2.2, the congestion control algorithm will cause queues to build until full, then reduce, and build again in a characteristic sawtooth pattern. Since `iperf` and `ping` share a source host, `ping` packets will also be affected by the queues building. This will cause `ping`'s traffic to be delayed. The result is that, in every percentile, `ping` packets are delayed by at least an order of magnitude greater than the any pervious case. The 99th

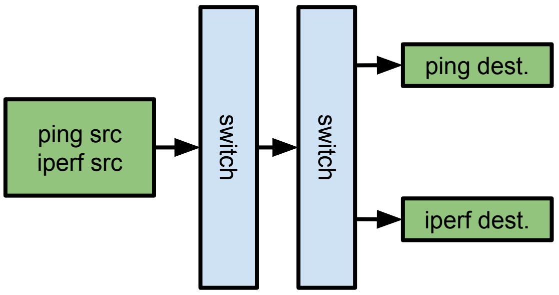
[38]Intel XL710 Ethernet Controller datasheet. See A.16.4.



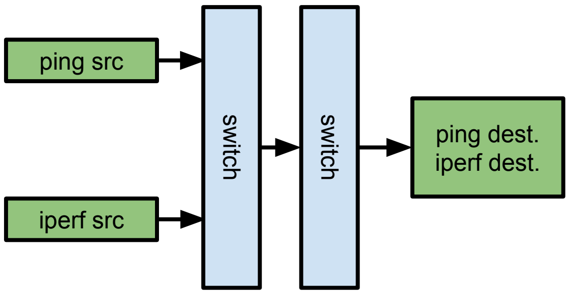
(a) *Otherwise idle network.*



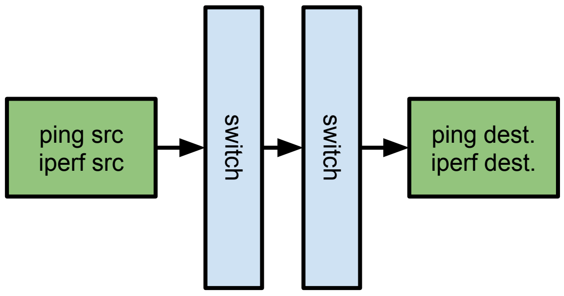
(b) *Two hosts, shared switches.*



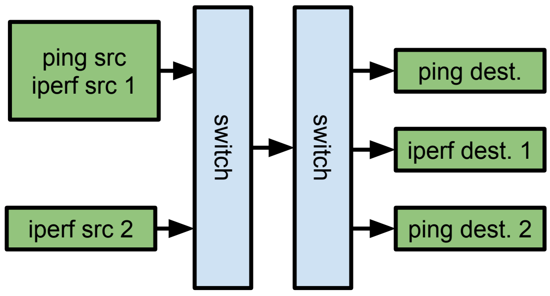
(c) *Shared host egress.*



(d) *Shared host ingress.*



(e) *Shared host ingress and egress.*



(f) *Shared switch queue.*

Figure 2.5: *Arrangements for* `ping` */* `iperf` *network sharing tests.*

percentile `ping` latency is degraded by over 16× compared to the base case.

Comparing case (F) (the shared switch queue case) with all other cases reveals that, although any degree of sharing results in interference, the effect is worst when applications share a congested switch queue.

2.3.2 Testing with Datacenter Applications

The experiments conducted above demonstrated that network interference is both an application measurable effect, and, that it occurs most severely in shared switch queue situations. However, these experiments used `ping` and `iperf` which lie at extreme ends of the latency and bandwidth requirements spectrum. The degree to which realistic datacenter applications can cause and/or be affected by network interference remains untested. To do this requires the use of realistic datacenter applications on a realistic network topology.

In Section 2.1.1, I described three layers of datacenter applications: (*i*) the distributed coordination layer, (*ii*) the distributed data access layer, and, (*iii*) the distributed programming frameworks/ applications layer. To test if realistic datacenter applications are capable of producing and/or being effected by network interference, I have selected one application from each layer: PTPd from the distributed coordination layer (see §2.3.2.1), memcached from the distributed data access layer (see §2.3.2.2) and Hadoop Map-Reduce from the distributed data-processing layer (see §2.3.2.3). I arranged these applications to run on 12 machine, 10Gb/s test network (see Fig. 2.6). Each application has a unique performance metric that will be measured on this network. I call this the *application specific metric of interest*. Specific details of the applications and their performance metrics are as follows:

2.3.2.1 PTPd - Clock Synchronisation

Precise clock synchronisation is important to many distributed systems (e.g. Spanner [46], Fastpass [87] etc.). The Precision Time Protocol daemon (PTPd)[39] offers nanosecond-granularity time synchronisation to machines on a local network. It operates by exchanging timing messages between a "master" (server) and a "slave" (client). The messages are issued at a low rate, usually once per second. The master issues `sync` messages which indicate the current time. The slave issues `delay_req` messages which are used to calculate the network delay. Using these two values, the slave can estimate its clock offset from the master clock and modify its clock to match. The Precision Time Protocol (PTP) requires the network to provide near constant latency (otherwise known as low *jitter*). It also assumes both slowly changing network delays and symmetrical network paths. These assumptions make the protocol highly susceptible to network interference. In the following experiments, the application specific metric of interest for Precision Time Protocol (PTP) is network jitter. This will be expressed at the application level as the perceived time offset between the slave and master. PTPd reports this value as part of its built-in performance statistics.

2.3.2.2 Memcached - Key/Value Store Cache

Memcached[40] is a popular in-memory key-value store used by Facebook and others to accelerate frequent item access [48]. The application stores *values* of a few kilobytes which are accessed by providing smaller *keys*. As the name implies, these keys and values are stored in memory so that they can be accessed quickly. Memcached clients may issue `get` or `set` messages to a memcached server. A `get` message causes the server to return a value associated with a key. A `set` message causes the server to assign a new value to a key. In either case, the application does very minimal processing work. Its performance is nearly entirely limited by network latency and throughput. A well configured memcached installation can achieve a request rate 200-300 thousand requests per second, at a throughput of 3-5Gb/s [48]. Users of memcached require the application to respond quickly to individual requests so that individual systems can achieve fast response times. Users also require that memcached can sustain high request rates so that many

[39]Available from `https://github.com/ptpd/ptpd`.

[40]Available from `https://github.com/memcached/memcached`.

parallel systems can operate with a single memcached instance. In the following experiments, the application specific metric of interest for memcached is request latency. This is the time measured between a request being issued and a response being received. A secondary application specific metric is throughput, which is the number of requests that can be served per second. Measurements of these metrics are made by an external application which issues requests and receives responses.

In my measurements I use a slightly modified version of the `memaslap` load generator. The default `memaslap` load generator only reports summary statistics at the end of each experiment. My modified version stores the latency of every request in memory and returns all values once when the test is complete or memory runs out. I then process the raw values myself.

2.3.2.3 Hadoop Map-Reduce - Distributed Programming Framework

Hadoop Map-Reduce [50] (commonly known as Hadoop MR or simply Hadoop) is an open source implementation of the Google Map-Reduce [49] system. The system provides a high level framework for programming distributed systems. In a Map-Reduce system there are two phases of computation. The first *map* phase takes a set of key/value inputs and produces a (possibly empty) set of intermediate values for every key. The second *reduce* phase takes the set of values for each key and merges them together to form a possibly smaller set. Usually the output from the reduce phase is a single value or no value at all. Between the map and reduce phases, a *shuffle* phase communicates all values with a given key to at least one reducer. In practice mappers and reducers use the same physical machines, so the shuffle phase becomes a synchronised all-to-all communication step. Synchronised all-to-all communication puts immense stress on the network and can result in TCP incast [83] throughput collapse. The application specific metric for Map-Reduce is therefore Flow Completion Time (FCT). This will be expressed at the application level as the time it takes for a Map-Reduce job to complete.

The Hadoop Map-Reduce system stores its data in a distributed data-store called the Hadoop Distributed File System (HDFS) [43]. Loosely speaking, HDFS is an implementation of Google File System (GFS) [42]. HDFS segregates data into blocks of a fixed size. Typically these blocks are 128MB. Block data is stored on multiple *datanodes* where this data is replicated. The replication is usually at least three ways and attempts are made in larger clusters to ensure the replicas reside in different failure domains. HDFS uses a centralised *namenode* on which meta-data about the location of each block is stored and the relationship that blocks have to any given file. This name node may have a backup for failover purposes.

Without a user supplied program, Hadoop is nothing more than a framework. To exercise Hadoop, I supply two 512MB files, each containing uniformly randomly generated numbers in ASCII text format (39M rows). The files are loaded into HDFS. Using the Musketeer system [27] Hadoop code is generated to apply a natural join to these datasets. The output of which is 29GB in size (1.5B rows). Although the join operation is simple and the data size is relatively small, this operation is a common component of many higher level services

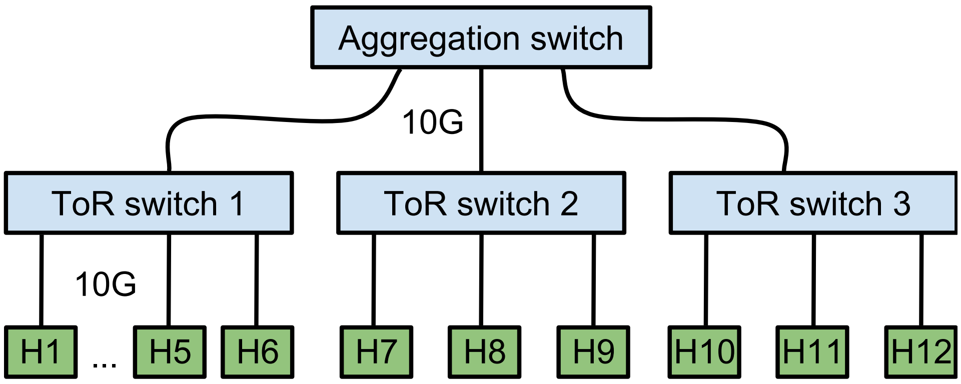


Figure 2.6: *Network topology of my test-bed.*

and algorithms used by datacenter operators [27]. It is thus indicative of a realistic datacenter workload.

2.3.3 Experimental Configuration

The above applications need somewhere to run in order to test the effects of interference between them. I use the following experimental configuration to run and test them. The physical test-bed comprises an otherwise idle, 12 node cluster of recent AMD Opteron and Intel Xeon-based machines running Linux kernel 3.4.55 with an Ubuntu 14.04 software distribution. Each machine has a two-port 10Gb/s NIC installed and is connected to a 10Gb/s Ethernet/IP network. The network is comprised of four Arista 7050 switches arranged as per Figure 2.6. This architecture is broadly similar to those described in Section 2.1.3 and approximates[41] the 3:1 oversubscription ratio found in production datacenters (see §2.1.3). I use `ptpd` v2.1.0[42], memcached v1.4.14[43], and I generate load for memcached using `memaslap` from `libmemcached` v1.0.15[44]. `memaslap` is modified as described in Section 2.3.2.2. It is configured using binary protocol with a mixed get/set workload of 1kB requests in TCP mode with 128 concurrent requests[45]. Finally, Hadoop 2.0.0-mr1-cdh4.5.1 is deployed on eight of our twelve nodes, with the HDFS data in `tmpfs` and the replication factor set to six[46].

2.4 Congestion Control in Datacenters

Datacenter network congestion control continues to be an active area of research, development, and implementation efforts. Some of these efforts are readily deployable while others are still at a prototype or simulation only stage. The following section contains a description of the relevant contemporary approaches. Key properties of the approaches are summarised in Table 2.2.

[41] Within the constraints of the equipment available to me
[42] `https://github.com/ptpd/ptpd`.
[43] `https://github.com/memcached/memcached`.
[44] `http://libmemcached.org/`
[45] Flags: `-X 1024 -B -T2 -c 128 -t 120s -S 1s`
[46] This simulates the traffic a larger Hadoop cluster would generate.

		Unmodified				Coord.	Flow	Bounded	Imple-
	System	hardware	proto.	OS kernel	apps.	free	deadlines	latency	mented
Deployable	TCP [74]	✓	✓	✓	✓	✓	✗	✗	✓‡
	EFC [88]	✓	✓	✓	✓	✓	✗	✗	✓‡
	ECN [78]	✓*, ECN	✓	✓	✓	✓	✗	✗	✓‡
	DCTCP [71]	✓*, ECN	✓*	✗	✓	✓	✗	✗	✓‡
	Fastpass[87]	✓	✓	✓, module	✓	✗	✗	✗	✓‡
	EyeQ [89]	✓*, ECN	✓	✗	✓	✗	✗	✗	✓‡
	QJump	✓	✓	✓, module	✓	✓	✓*	✓	✓‡
	EyeQJump	✓	✓	✓, module	✓	✗	✓	✓	✗
	FastJump	✓	✓	✓, module	✓	✗	✓	✓	✗
	R3CJump	✓*, ECN	✓	✓, module	✓	✗	✓	✓	✗
Not deployable	D^{2}TCP [90]	✓*, ECN	✓*	✗	✗	✗*	✓	✗	✓
	HULL [91]	✗	✓*	✗	✓	✓	✗	✗	✓*
	D^{3} [92]	✗	✗	✗	✗	✓	✓	✗	✗*, softw.
	PDQ [93]	✗	✗	✗	✗	✗	✓	✗	✗
	pFabric [94]	✗	✗	✗	✓	✓	✓*	✗	✗
	DeTail [85]	✗	✓	✓	✗	✗*	✗	✗	✗*, softw.
	TDMA [95]	✓*	✓*	✗	✓*	✗	✗	✓	✓

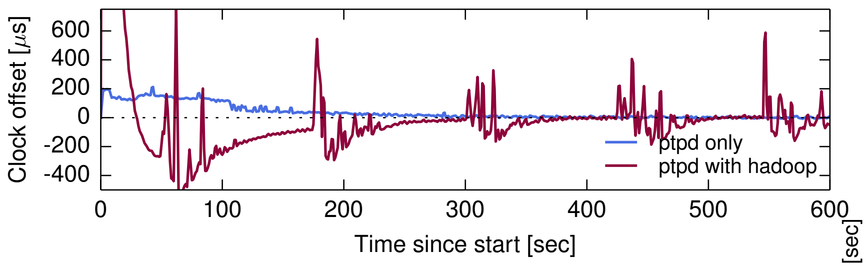
Table 2.2: *Comparison of related systems.* **with caveats, see text;* ‡*implementation publicly available.*

Systems are categorised as *deployable* if they function on commodity hardware using unmodified transport protocols and unmodified application source code. They are classified as *implemented* if tests have been run on physical test-beds (i.e. not in simulation alone) and if any special hardware required has at least been prototyped. For some approaches the source code is publicly available. Where possible, publicly available systems are tested to measure their effectiveness at resolving network interference on my test-bed. For comparison, the table also contains rows for QJump, EyeQJump, FastJump and R3CJump which will be described more fully in Chapters 4 and 5.

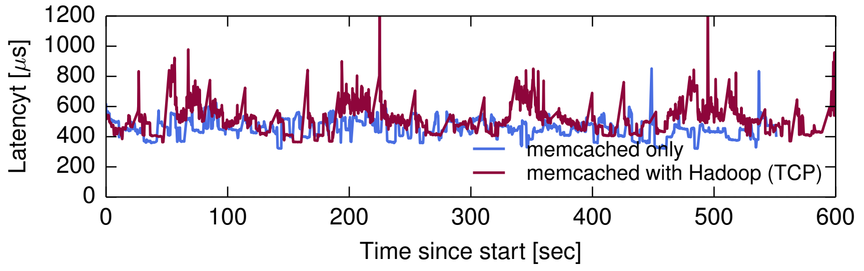
2.4.1 Transmission Control Protocol (TCP CUBIC)

The default TCP congestion control algorithm used by the Linux kernel 2.6.19 to 4.6 and Mac OS X Yosemite is TCP CUBIC [74]. The protocol modifies the linear window growth function of previous TCP standards to be a cubic function. This is in order to improve the scalability of TCP over fast and long distance networks (e.g. the Internet). CUBIC is not specifically designed for use in datacenters, but since it is the default algorithm included with Linux, it makes a useful comparison point.

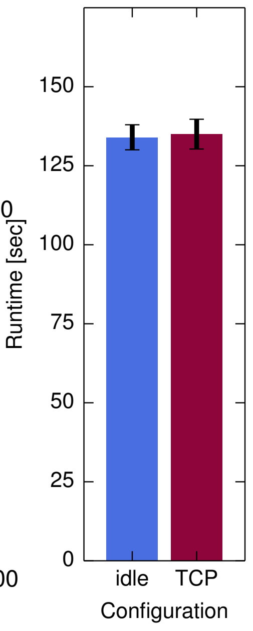
It is worth noting that PTP does not use TCP to transfer messages. The protocol uses either User Datagram Protocol (UDP) [96] or Ethernet frames directly. It is however very sensitive to delays in the network which can be caused by TCP filling switch queues. PTP is in



***(a)** PTPd performance (synchronisation offset)*



***(b)** memcached performance (request latency)*



***(c)** Hadoop (runtime)*

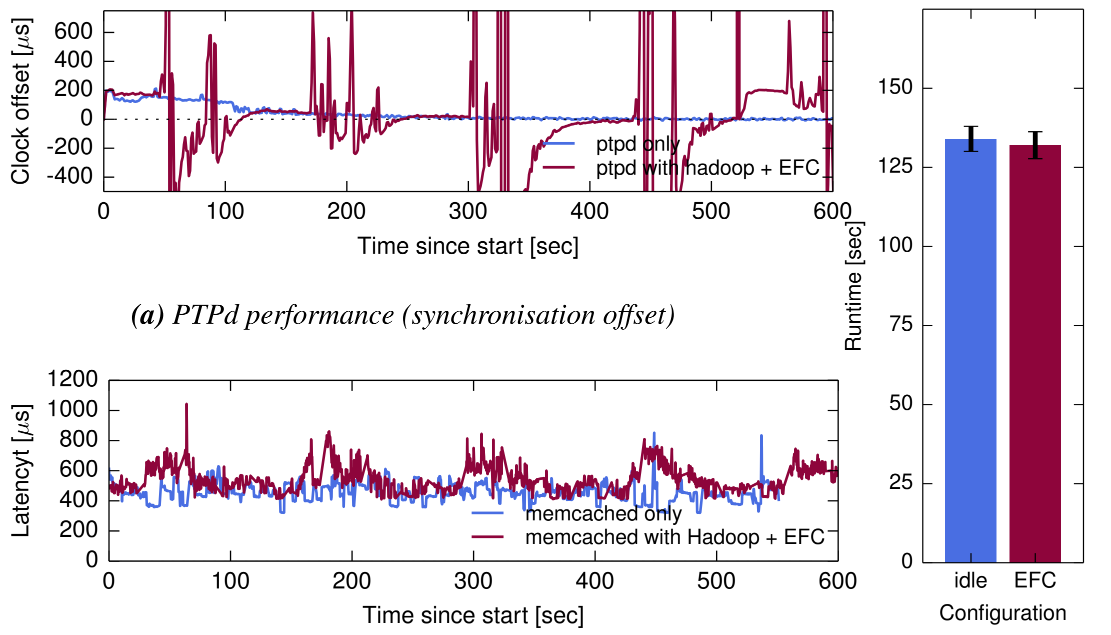
***Figure 2.7:** Comparison of PTPd, memcached and Hadoop performance when run in isolation, or on a shared network using only TCP congestion control to mitigate interference.*

effect an external measure of the delays caused by TCP queueing in the network.

In Figure 2.7a, I show a timeline of PTPd synchronising a host clock on both an idle network and when sharing the network with Hadoop. PTP takes approximately 5 minutes to stabilise clock synchronisation to within $\leq 10\mu s$ while there is no background traffic (see Figure 2.7a). When PTP shares the network with Hadoop, Hadoop's shuffle phases causes queueing. This delays PTPd's synchronisation packets and causes PTPd to temporarily fall 200–500μs out of synchronisation. The effect is is $50\times$ worse than on an idle network. The result is clear: PTP is both affected by network interference and network interference can be caused by Hadoop. As expected, TCP CUBIC does little to reduce this interference.

Figure 2.7b shows that a similar situation occurs with memcached. The figure reports the same timeline, this time showing request latencies for memcached, also on an idle network and on a network shared with Hadoop. With Hadoop running, the 99$^{\text{th}}$ percentile request latency degrades by $1.5\times$ from 779μs to 1196μs. Furthermore, around 1 in 6,000 requests takes over 200ms to complete. This is likely because severe packet loss triggers a TCP Retransmit Timeout (RTO).

A Retransmit Timeout (RTO) fires when TCP fails to receive several consecutive acknowledgements. The protocol cannot know if the acknowledgements have been lost, or if they are being delayed by severe congestion. Injecting more packets into the network may exacerbate severe congestion, but missing acknowledgements may also indicate that data has not been received and needs to be retransmitted. The protocol must therefore wait for a 'safe' period



(a) PTPd performance (synchronisation offset)

(b) memcached performance (request latency)

(c) Hadoop (runtime)

Figure 2.8: *Comparison of PTPd, memcached and Hadoop performance when run in isolation, or on a shared network while using Ethernet flow control (IEEE 802.3x) to mitigate interference.*

so that any outstanding (delayed) acknowledgements can be received, but not too long so that bandwidth is wasted because required retransmissions have not been sent. The value of 'safe' is determined dynamically [66] with a minimum value which is called the *minimum RTO (minRTO)*. The default minRTO value is 200ms in Linux, (see §3.4.1) for more details. Messages delayed by a minRTO are over 85× worse than the maximum latency on an idle network.

In Figure 2.7c I show the runtime of Hadoop with and without memcached and PTPd sharing the network. The impact of PTPd and memcached on the Hadoop job runtime is minimal.

2.4.2 Ethernet Flow Control (IEEE 802.3x)

Ethernet Flow Control (EFC) is a data link layer congestion control mechanism. It is a form of explicit congestion notification and one of the central components of the Datacenter Bridging (DCB) extensions to Ethernet [95, 97]. When using Ethernet Flow Control (EFC), hosts and switches issue special link-local *pause frames* when their queues are full (or nearly full). These messages alert the sender to stop sending for a specified pause period [95] which is contained in the message. Pause frames allow the receiver to clear local buffers before they overflow. One problem with this approach is that several higher layer (e.g. TCP, UDP) flows may share a single Ethernet layer connection. If one of these flows exceeds the output capacity of the switch, a pause frame may be issued and all flows will be delayed. This is otherwise known as *Head of Line (HOL)* blocking. To mitigate HOL blocking effects, there is an extension to IEEE 802.3x

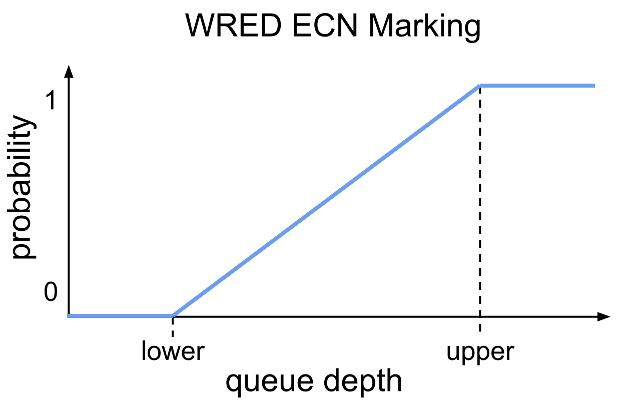


Figure 2.9: *WRED marking probably function*

called Priority Flow Control (PFC) (IEEE 802.1Qbb [88]). PFC extends the flow control format to support 8 different traffic classes. By sending an 802.1Qbb PFC pause frame, different traffic classes can be paused [95] for different amounts of time (or not at all).

Figure 2.8 shows the effect of using Ethernet Flow Control in the same experimental configuration as above. It shows that EFC has a limited positive influence on memcached (Fig 2.8b). This is because EFC limits loss in the network. Although limiting loss in the network would appear to be beneficial, it also has secondary effects. Ordinarily, TCP would use loss as a signal to reduce its sending rate. Since EFC limits loss, TCP keeps all queues occupied and pause messages add further delay to the network. This has a strongly negative influence on PTPd (Fig 2.8a) performance. Hadoop's performance is throughput bound and remains unaffected (Fig 2.8c).

2.4.3 Explicit Congestion Notification (ECN)

In comparison to EFC/PFC, Explicit Congestion Notification (ECN) [78] provides a more fine-grained congestion signal. ECN is a network layer mechanism in which switches indicate queueing to end-hosts by marking IP packets with a special flag. Our Arista 7050 switch implements ECN with Weighted Random Early Detection (WRED)[47]. WRED is a type of Active Queue Management (AQM) scheme and is a variant of Random Early Detection (RED) [98]. In a WRED system, an administrator must configure two *marking thresholds*: an upper and lower threshold (See Fig. 2.9). These thresholds denote queue lengths within the switch. If the queue length is below the lower threshold, no action is taken. When queuing exceeds the lower threshold, packets are randomly marked by setting explicit congestion notification bit. The markings are added with linearly increasing probability until the upper threshold is reached. Once the queue length exceeds the upper threshold, all packets are marked with the ECN bit set. The ECN bit expresses to senders and receivers that congestion is occurring or will occur soon. It allows senders to reduce their rates well before packet loss becomes a problem.

[47] Arista switch configuration manual. See Appendix A.2.4.

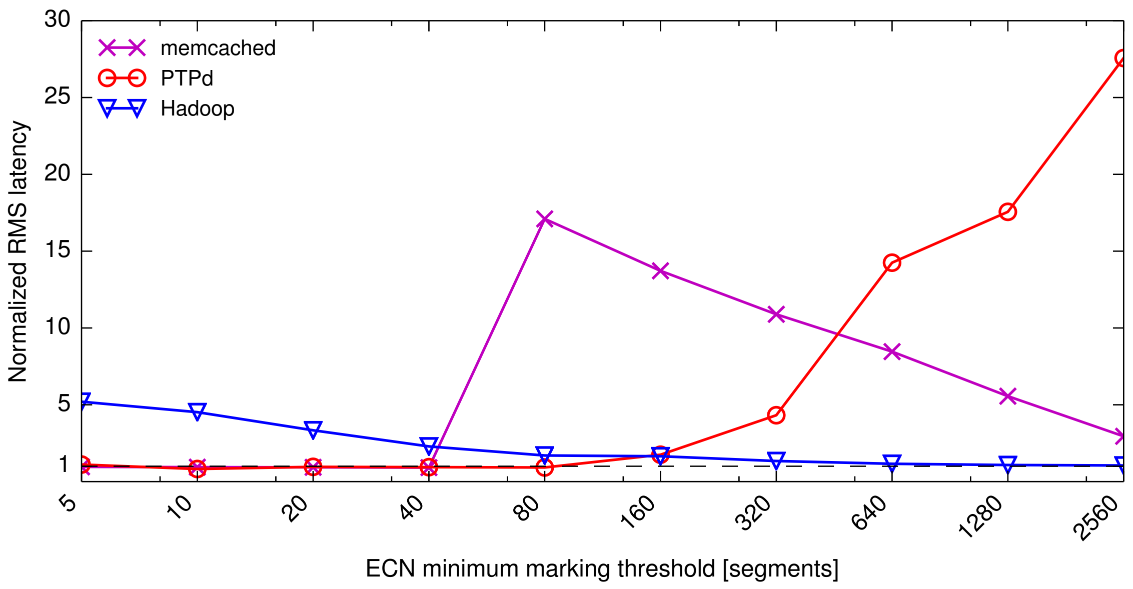
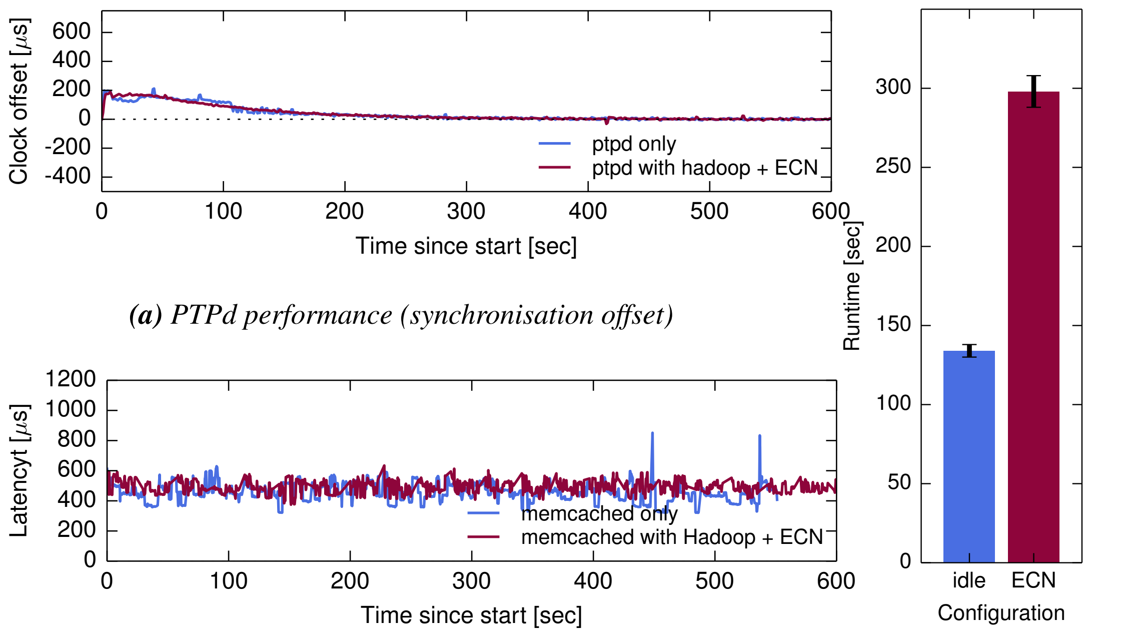


Figure 2.10: *Normalised Root Mean Squared (RMS) performance for PTPd, memcached and Hadoop when run in on a shared network while using explicit congestion notifications (ECN). Results are normalised to applications running on an idle network. Applications are reported using their application specific metric; PTPd: synchronisation offset, memcached: request latency, and Hadoop: job completion time.*

One of the difficulties with deploying RED/WRED systems is choosing the correct marking thresholds. The choice of marking threshold can have a dramatic impact on on application's performance. Figure 2.10 shows the results of using ten different marking thresholds pairs, ranging between [5, 10] and [2560, 5120] ([lower,upper], in packets). The figure shows the RMS value of each application specific metric, normalised to the idle case. The application specific metrics are; PTPd: synchronisation offset, memcached: request latency, and Hadoop: job completion time (for more details see §4.5.2). None of these settings achieve ideal performance for all three applications, but the best compromise is at [40, 80]. This point lies directly before an inversion point where memcached starts to be heavily affected by Hadoop's traffic, but at the minimum reduction in Hadoop's traffic. With this configuration, ECN effectively resolves the interference experienced by PTPd and memcached. However, this comes at the expense of substantially increased Hadoop runtimes. Figure 2.11 is included to facilitate comparisons with previous and forthcoming experiments. This shows the performance of the ECN using the same style of plots previously used (e.g Fig. 2.8), when configured at thresholds of [40,80]. The figure shows the improved performance of PTPd and memcached, but also the decreased Hadoop performance.

It is interesting to note that the recommended ECN thresholds for DCTCP [71] (see §2.4.4) falls nearly exactly in the middle of this range ([65,65]). Finally, contrary to claims by previous work (e.g. [71, 91]), at the time of writing I found it difficult to acquire switches that support ECN. Only one (now superseded) model of Arista switches supported ECN and it has



***(a)** PTPd performance (synchronisation offset)*

***(b)** memcached performance (request latency)*

***(c)** Hadoop (runtime)*

***Figure 2.11:** Comparison of ECN, memcached and Hadoop performance when run in isolation, or while on a shared network using Explicit Congestion Notification (ECN) configured to [min,max] marking thresholds of [40,80] to mitigate interference.*

proven difficult to find new switches that do. Nevertheless ECN appears to be widely used by Google, Facebook and others[34].

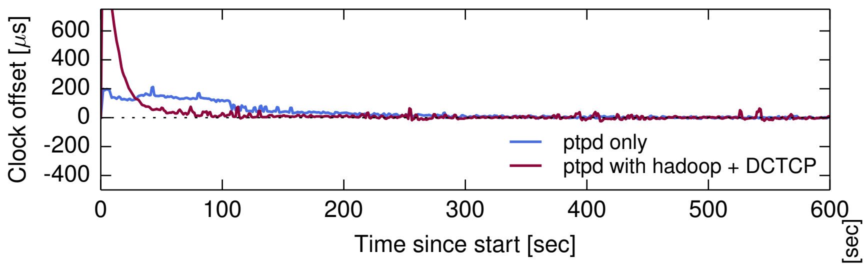
2.4.4 Datacenter TCP (DCTCP)

Datacenter TCP (DCTCP) has recently been proposed as a solution to TCP problems such as TCP incast found in datacenter networks [71]. DCTCP is optimised for high-bandwidth, low-latency datacenter networks with switches that support ECN. It improves upon ECN (see §2.4.3) by estimating the fraction of bytes that encounter congestion, rather than simply detecting that some congestion has occurred. It then scales the TCP congestion window based on this estimate. This method is claimed to achieve high burst tolerance, low latency, and high throughput with shallow-buffered switches. DCTCP has seen impressive uptake: It has been integrated into the Windows Server 2012 operating system[48] as well as the Linux 3.18 kernel[49] and Google use an algorithm similar to DCTCP in their datacenters [34].

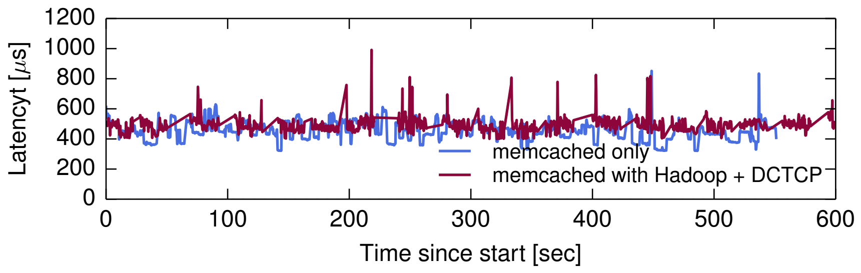
As with previous approaches, I configured an ran DCTCP with PTPd, memcached and Hadoop sharing the test network. Figure 2.12 shows the results. The figure largely supports the DCTCP authors' claims. DCTCP reduces the variance in PTPd synchronisation and memcached latency compared to the contended case. However, this comes with a small, but marked, increase

[48]Discussed on Microsoft Technet (9 May 2012). See Appendix A.21.3.

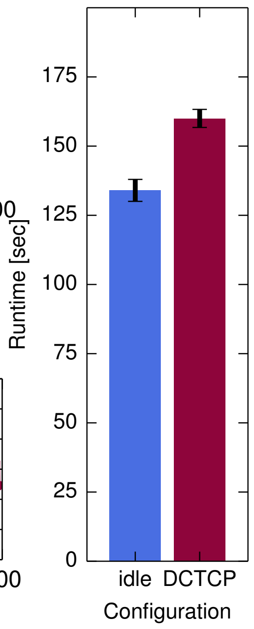
[49]Linux 3.18 release discussed on kernelnewbies.org (7 December 2014). See Appendix A.19.1.



(a) PTPd performance (synchronisation offset)



(b) memcached performance (request latency)



(c) Hadoop (runtime)

Figure 2.12: *Comparison of PTPd, memcached and Hadoop performance when run in isolation, or on a shared network using datacenter TCP (DCTCP) to mitigate interference.*

in Hadoop job runtimes. Nevertheless, DCTCP performs best out of all previously tested approaches.

2.4.5 High-bandwidth Ultra-low Latency (HULL)

Since DCTCP was proposed, several variants have been proposed that extend it in various ways. High bandwidth Ultra Low Latency (HULL) [91] is one such example. With HULL, the authors are motivated by a desire to provide predictable, "ultra-low latency" in datacenter networks without sacrificing (too much) bandwidth. The authors of HULL make an important observation which will become a central element in my work:

> Achieving predictable and low fabric latency essentially requires congestion signalling before any queueing occurs. That is, achieving the lowest level of queueing latency *imposes a fundamental trade-off with bandwidth* (emphasis added).

To resolve this trade-off, the authors propose a Phantom Queue (PQ) approach associated with each switch port. Rather than marking packets when queueing occurs, the PQ simulates queue buildup for a virtual egress port running at a speed slightly slower than the physical link speed. The PQ is then used to mark packets (using ECN flags) when queuing would occur on the slower link (but has not yet actually occurred). This produces a sensitive imminent congestion signal based on utilisation rather than actual queuing. However, the problem with this signal is that it is susceptible to bursty behaviour in the network. For a variety of reasons including TCP slow-start (see §2.2 and [99]), acknoweldgement compression [100] and various offload features in NICs like Large Segment Offload (LSO) [91] transmissions are bursty [101]. To overcome this packet-pacing is required in the NIC. The packet-pacing feature ensures a consistent rate of packet transmissions into the network. The authors observe that:

> It is paradoxical that the pacer must queue packets at the edge (end-hosts) so that queueing inside the network is reduced. Such edge queueing can actually increase end-to-end latency, offsetting any benefits of reduced in-network queueing.

To combat this problem, pacing is selectively applied only to long flows using an online classification system.

Despite impressive performance gains over DCTCP, HULL suffers from several drawbacks. Like DCTCP, its rate-limiting is applied in reaction to ECN-marked packets which requires the support of ECN enabled switches[50]. Furthermore, it requires phantom queue modifications to those switches, which would require custom switch chips to be produced as well as pacing features to be integrated into NICs. These are invasive hardware changes to both the edge and the core of the network hardware. It is difficult to see how HULL might be deployed in current datacenters where commodity hardware dominates purchasing decisions.

2.4.6 D^{2}TCP and D^3 and PDQ

Another alternative to DCTCP is Deadline-aware Datacenter TCP (D^{2}TCP) [90]. D^{2}TCP extends DCTCP's window adjustment algorithm with the notion of flow deadlines. Flows with later deadlines aggressively reduce their sending rate in the face of congestion whereas flows with near deadlines reduce their sending window less aggressively or not at all. Like DCTCP, D^{2}TCP requires switches that support ECN.

Another deadline aware datacenter network transport is Deadline Driven Delivery (D^3) control protocol. D^3 is a radical departure from the TCP/IP based protocols described so far. It introduces an entirely new protocol, header format and network switch/router architecture. D^3 introduces the notion that unfairness can increase performance. Under D^3, hosts calculate a desired rate based on the remaining flow size and deadline. This calculation is used to generate a request which is sent to each switch/router along the path. Rate allocations are made by the switches which greedily allocate rate to request packets on each round trip and store them in the new header format.

A substantial enhancement over D^3 is Preemptive Distributed Quick (PDQ). PDQ also champions the usefulness of unfairness in datacenter networks but takes it one step further. PDQ implements a distributed preemptive scheduling mechanism. Switches along the network path not only assign a rate to a flow, but can also "pause" (similar to §2.4.2) or even terminate a flow. PDQ can be used to implement Earliest Deadline First (EDF) and/or Shortest Job Remaining

[50]Only one in five 10Gb/s switches I looked at supports ECN.

(SJR) scheduling algorithms. However, like D^3, PDQ also requires substantial modifications to the switches, kernels and applications.

D^2TCP, D^3 and PDQ all introduce new congestion control algorithms to the datacenter. Furthermore, all champion the application of unfairness in datacenters. Yet no system fundamentally changes the network interference properties of the network. Under all systems queue occupancy is similar to that of DCTCP, but is instead unfairly allocated proportionally to flows dependent on their respective deadlines. An interesting consequence is that all systems show that differentiating between different types of traffic ("unfairness") in the datacenter network can be beneficial overall. I apply this principle in my work, especially in QJUMP (see Chapter 4).

2.4.7 TDMA for Datacenter Ethernet

One observation that motivated the distributed design of PDQ is that it is "unrealistic" to implement a "centralised algorithm" for scheduling flows in a datacenter network. The authors of *TDMA for Datacenter Ethernet* [95] do exactly this. They argue that datacenter communication patterns look less like the traditional wide-area workloads that TCP was designed for and more like the tightly coupled communication network of a supercomputer backplane. However, despite supercomputer-like workloads, datacenter networks eschew "boutique" link layers designed for supercomputers (e.g. Infiniband [102]) because of the cost advantages of Ethernet. As a response, the paper introduces the design of a Time Division Multiple Access (TDMA) Media Access Controller layer using commodity Ethernet hardware.

Like PDQ, the TDMA Ethernet design issues special messages to selectively pause or preempt flows on each host. However unlike PDQ, these messages are sent to all hosts simultaneously by a centralised scheduler. The messages serve as a way to gate flows into the network. This is how the time division, multiple access schedule is implemented.

TDMA Ethernet uses a centralised scheduler. The aim of the scheduler is to decide when and for how long each flow is allowed to use a given network path. This is to ensure that no two flows attempt to use the same network path at the same time (which would cause congestion). The scheduler is given network resource demands and capacities via an out-of-band network and uses these to calculate the schedule for a short interval.

TDMA Ethernet is evaluated using an Hadoop Map-Reduce shuffle (all-to-all) like workload. The results show a significant improvement in flow completion times. Furthermore, the system is explicitly tested for its ability to handle network interference. A similar test to my own from Section 2.3.1, case (6), is used. In the test, a UDP based `ping` application measures Round-trip Times (RTTs) in the presence of congestion caused by two TCP senders. With the TDMA system enabled, the RTTs are reduced by a factor of around $3\times$. However, this reduction comes at the expense of throughput for the high rate senders.

Although the results of the TDMA Media Access Controller (MAC) system evaluation show promise, the design, implementation and evaluation fall short in several ways:

1. Scheduling messages are implemented using 802.1x/802.1Qbb Ethernet Flow Control (EFC) / Priority Flow Control (PFC) (see also 2.4.2). The design uses the traffic class identifier in the PFC message to selectively pause specific routes through the network. However, standard PFC messages are limited to only 8 different traffic classes (using a 3-bit class identifier). To overcome this limitation, the authors modify the PFC format, extending it to 11 bits and use a kernel bypass network stack to implement this. This significant modification deviates from the stated goal of using commodity Ethernet hardware and would require substantial hardware and software changes in a real deployment.
2. Overheads in the system limit the minimum time interval for each transmission slot. In the prototype implementation a minimum time slot of $300\mu s$ with an additional "guard time" of $15\mu s$ is used. The guard time imposes a minimum 5% bandwidth penalty on any flow, while the minimum slot time requires that each flow send at least 375kB per interval. If a flow cannot use its entire slot, significantly more bandwidth will be wasted.
3. No attempt is made to handle path heterogeneity, a common feature in datacenter networks (see §2.1.3).
4. Few details are given about the scheduling algorithm employed. The prototype implementation uses a "round-robin scheduler that leverages some simplifying assumptions about the network topology", but no further details are given. The optimal scheduling problem is known to be NP-hard [87, 94, 103] and thus scales poorly with both increasing network size and utilisation. Furthermore, the authors explicitly state that the scheduler is unable to estimate demand for short, latency sensitive flows in (paper §§4.1). However, the network interference test (paper §6.2 and described above) would appear to rely on this property. No details are given on how the interference test is made possible given this scheduler limitation.

2.4.8 Fastpass

The development of efficient centralised network flow scheduler is left as an open problem by the authors of TDMA Ethernet (see §2.4.7). Fastpass tackles this problem directly, but takes it one step further. Rather than implement a centralised *flow scheduler*, Fastpass implements a centralised *packet scheduler*. Like TDMA Ethernet, the goal of Fastpass is to implement a "zero-queue" datacenter network. To achieve this, Fastpass treats the network as if it were a single (complex) switch. Like TDMA Ethernet, Fastpass hosts issue requests to a centralised arbiter for permission to send packets into the network. In Fastpass, the arbiter performs a similar job to classical switch schedulers like iSLIP [103] and PIM [104].

In essence Fastpass implements a packet-by-packet Time Division Multiple Access (TDMA) scheme over a datacenter network. In many respects this is a refined version of the approaches taken in TDMA Ethernet (see §2.4.7). The approach also echoes the the work in High bandwidth Ultra Low Latency (HULL) (§2.4.5) where the scheduler "queues packets at the edge (end-hosts) so that queueing inside the network is reduced [or eliminated]". While Fastpass eliminates in-network queueing, requests for an allocation must queue at the centralised arbiter and packets must wait at the end-hosts until a schedule can be issued. Furthermore, unlike TDMA Ethernet, Fastpass does not take into consideration of capacity at the destination hosts. Delays at the end hosts would lead to dropped packets, despite the significant resources expended in getting them there.

The Fastpass software arbiter is capable of scheduling 2.21Tb/s of traffic on a single 8 core machine. The arbiter is tested in a live rack in a Facebook cluster. Only a single rack is tested, so no path planning is required, which does reduce the complexity of the test. As its main result, the test demonstrates a $2.5\times$ reduction in TCP retransmissions from an average 4 per second down to 1.6 per second. The authors do not elaborate on any benefit gained by this reduction. Although retransmission timeouts (RTOs) can lead high latencies (see 2.4.1), 4 timeouts per second already appears to be a very low value. A reduction from 4 per second down to 1.6 per second would appear to be inconsequential, especially given that a 10Gb/s Ethernet link is capable of transmitting between 800K and 14.4M frames per second [105].

Fastpass lacks a specific evaluation of its impact on overall, end-to-end packet latency. Once again echoing the earlier work in HULL,"edge queueing can actually increase end-to-end latency, offsetting any benefits of reduced in-network queueing." This appears to be the case. The paper reports a similar `ping` and `iperf` experiment to that conducted in Section 2.3.1. In the test, `iperf` is demonstrated to interfere with `ping` performance. Fastpass is then shown to resolve this interference. Unfortunately, no baseline values for `ping` without Fastpass or `iperf` are given. In the reported results, `ping` traffic using Fastpass has a 50th percentile RTT of $230\mu s$ and a 99th percentile of $380\mu s$. These values are approximately $3\times$ the baseline latency values that I obtained in my experiments. They exceed all other types of interference other than shared switch queues (case 6) in my experiments. I cannot say for certain that my experimental arrangement was identical to theirs, but it would appear prima facie that Fastpass is introducing significant delays to the `ping` traffic.

One final issue with the Fastpass approach is that it requires the network to be rearrangeably non-blocking (RNB). An RNB network differs slightly from the non-blocking networks described in Section 2.1.3. In an RNB, any input port can still be connected to any free output port however, existing connections may need to be rearranged to do so [106]. Although this is a weak form of non-blocking network, it still rules out oversubscribed networks such as those commonly found in datacenters (see §2.1.3). This requirement may be the reason that only a single rack was used in the Facebook evaluation test.

2.4.9 pFabric

Unlike Fastpass, pFabric is a completely clean-slate approach to datacenter network congestion control. The system aims to take principled, minimal, approach to the problem of scheduling datacenter networks.

A key insight of the paper is to decouple flow scheduling from rate control. This is achieved by including a single number in the header of every packet which encodes its priority.

Switches in the network simply schedule packets on the basis of this priority value. The paper argues that such switches then require only shallow buffers and simple hardware construct. The authors go to great lengths to substantiate this claim by describing, in detail, how the scheme could realised in simple hardware. The priority value is set independently by each flow. It can be used to encode many different scheduling values. Examples include: the remaining flow size, the flow deadline or the absolute flow size.

For the remainder of the paper, the authors choose "remaining flow size" as the value to be encoded. Encoding the remaining flow size into the priority field allows pFabric to approximate the Shortest Remaining Processing Time (SRPT) scheduler. This policy is known to be optimal for minimising Flow Completion Time (FCT) on a single link. However, even a simplified version of the problem for multiple links is known to be NP-Hard. pFabric implements a greedy approximation of SRPT policy. The greedy approximation has been proven to be at least within a factor of 2 of the optimal, though the authors claim that, in practice, their solution is much closer.

Using the above scheduling mechanisms, the need for rate control in pFabric is minimal. Only one corner case mandates its use: if a packet traverses the entire network but is dropped at the last hop, significant resources will have been wasted. To cope with this, pFabric uses TCP's most basic rate control mechanisms, and removes the initial slow-start phase as well as several congestion heuristics.

pFabric is evaluated in simulation using two synthetic workloads. The workloads replicate the flow size distributions found in previous datacenter network studies [69, 71]. Using these workloads, the evaluation focuses mainly on Flow Completion Time (FCT) performance. pFabric is shown to significantly outperform TCP (see §2.2 and §2.4.1), DCTCP (see §2.4.4) and PDQ (see §2.4.6) as well as closely approximating the optimal scheduling solution. However, pFabric's near-optimality is strictly limited to minimising average Flow Completion Times (FCT). Although the simulations show impressive 99th percentile flow completion time results, there is no guarantee that these results will hold under other traffic patterns. Furthermore, the performance is predicated on the assumption that flows have a fixed size which applications can predeclare ahead of time. Predeclaring flow sizes ahead of time is not trivial to do with practical applications.

While workloads like Hadoop and HDFS transfer data in chunks with fixed sizes, streaming workloads cannot be easily described in a fixed size manner. One approach to doing this might be to break large flows into smaller *flowlets* of application level messages. Techniques for doing this are described in *Enabling End-Host Network Functions* [107]. The problem with using flowlets (especially at high load) is that it will produce many small sized messages with similar priorities. In this case, packets will queue and contend with each other in a manner that resembles existing systems. pFabric requires its minimal TCP-like rate control mechanisms for situations like this. However, when invoked, the performance of the system will degrade to TCP-like performance.

As with other priority based systems, pFabric's priority system can also be gamed. Large users like Hadoop and HDFS might game the system by starting many short flows rather than one large flow. The short flows would consume a higher fraction of high priority time slices in switches potentially leading to unfairness at best, and poor service for all at worst.

Finally, as a clean-slate design, pFabric requires new switches, network adapters and substantial end-host modifications. This would make the system difficult to deploy into a production datacenter environment.

2.4.10 pHost and EyeQ

The Fastpass system (see 2.4.8) required networks that are rearrangeably non-blocking (RNB). Given the requirement for a non-blocking network, alternative system designs are possible. pHost [108] is one such design. pHost assumes a full bisection bandwidth network [109]. It further assumes that network switches are configured to spread packets uniformly across the set of available paths through the network. This is called *packet spraying* [110].

The authors observe that "using packet-spraying in a full-bisection-bandwidth network can eliminate almost all congestion in the core" which means that "sophisticated path-level scheduling (as in Fastpass)" and " detailed packet scheduling (as in pFabric)" are unneeded. In a full bisection bandwidth network using packet spraying, queueing is restricted to contention at the end hosts. To resolve this, pHost employs a Request To Send (RTS)/Clear To Send (CTS) scheme. Under this scheme, sending hosts send RTS messages to an arbitrator at the destination host. The arbitrator applies a scheduling policy to senders and issues CTS tokens to senders along with a timeout for those tokens. Doing so places a bound on both the rate at which the sender may inject new packets into the network and a bound on the time over which that injection may occur.

Like PDQ (§2.4.6) and Fastpass, the pHost CTS token mechanism allows the scheduler to preempt flows by issuing no new tokens for a given flow. This ability makes the pHost scheduler very flexible. It can implement many different scheduling policies including minimising flow completion times, deadline aware scheduling (e.g. §2.4.6) and fairness across multiple tenants.

In a simulation based evaluation, pHost is compared to Fastpass [87] (see §2.4.8) and pFabric [94] (see §2.4.9). The authors show that pHost performs better than both Fastpass and pFabric (which itself is near optimal). Further they argue that the scheme is readily implementable over commodity networks, although, no real-world implementation is demonstrated.

pHost is not evaluated in terms of end-to-end latency nor in terms of network interference. Like Fastpass, pHost implements a packet by packet schedule with requests being arbitrated at a centralised (destination host) arbiter. It is therefore likely that pHost will suffer from similar latency performance artefacts to Fastpass, which introduced a 3$\times$ latency penalty.

The pHost system assumes a full-bisection bandwidth network. As discussed in Section 2.1.3 (pg. 20), these are not common in real datacenter deployments. Typically datacenter deployments have an oversubscription ratio of around 3:1. The authors of EyeQ [89] concur with this observation. However, using measurements from a production Microsoft Azure cluster they go on to qualify that, despite the oversubscription, persistent congestion mostly occurs at edge links and not in the core. This motivates the design of EyeQ which, like pHost, pushes per-tenant state and rate management to the edge of the network.

EyeQ is designed to provide bandwidth isolation to multiple Virtual Machines (VMs) sharing a datacenter network. It operates in a manner conceptually similar to pHost. A receiver module at each destination measures the rate at which traffic is received from each source over short time intervals of 200μs. On the basis of these measurements, the receiver calculates the required rate limits for senders. The calculation assumes that no change will occur until the next measurement interval. Using a measurement interval limits the time over which tenants may interfere with each other. The interference period cannot exceed the measurement interval plus scheduling, communication and rate limiting delays. It does not, however, limit the potential extent of the interference which may still be very large in synchronised burst scenarios.

EyeQ is explicitly tested for its ability to control network interference. The system is stressed using a memcached based TCP sender and a bursty UDP sender. In the base case, the memcached sender achieves a 50th and 99th percentile latency of 98μs and 666μs respectively. When the bursty UDP sender is enabled, these values degrade by three orders of magnitude. With EyeQ enabled, the latencies are controlled to within 30% of the base values. EyeQ does however degrade the overall performance of high throughput tasks. In an all-to-all shuffle test (modelled on Map-Reduce) job completion times increase by 16%. This is for two reasons. First, EyeQ's congestion detecters maintain a 10% bandwidth headroom in order to work at end host without network ECN support. Second, the receivers do not share bandwidth over periods of less than 200μs leading to short periods of under utilisation. Once again, EyeQ exposes the fundamental trade-off between latency and bandwidth discussed by the authors of HULL (see §2.4.5).

2.4.11 DeTail

DeTail [85] takes a cross-layered, engineering centric, approach to minimising flow completion times in datacenter networks. The DeTail authors argue that latency tails are due to "flash congestion", though they do not define this term. Further they argue that "flash congestion" aggravates three issues present in datacenter networks:

1. *Packet-loss and retransmissions* which can lead to TCP timeouts. This can be particularly serious for short flows that are still in the slow-start phase. For these flows, typical RTTs would be on the order of 100s of microseconds, whereas TCP timeouts are likely to be 10s of milliseconds. TCP timeouts would lead to tails many orders of magnitude greater than the median.
2. *Mixed workloads* including both latency sensitive and throughput intensive traffic. When congestion occurs, latency sensitive traffic may be queued behind throughput intensive traffic. This queueing will contribute to the tails experienced by both types of traffic.

3. *Uneven load-balancing* can lead to congestion hotspots, exacerbating issues (1) and (2) above. The authors demonstrate scenarios where both flow hashing (e.g. ECMP [111]) and packet spraying [110] based load balancing are deficient and can lead to high 99th percentile flow completion times.

To resolve these issues, DeTail introduces a new switch design. It uses (*i*) priority flow control (PFC) to create a lossless fabric, (*ii*) packet prioritisation to differentiate between latency sensitive and throughput intensive workload types, (*iii*) Explicit Congestion Notification (ECN) to signal congestion, and (*iv*) adaptive load balancing to alleviate congestion hotspots. Features (*i*), (*ii*) and (*iii*) where all already available in commodity switches at the time the paper was written.

By applying techniques (*i*)-(*iii*), DeTail assumes that any queueing at a switch is an indication of congestion. Using this information, DeTail routes traffic through the path with the shortest egress queue length. The authors assume that the entire datacenter network is IP routed, with multiple paths between senders and receivers and that a routing protocol such as BGP or Open Shortest Path First (OSPF) is in operation to determine these routes. This assumption is not justified by the authors, however it is consistent with descriptions of existing datacenter networks (see §2.1.4, pg. 22)

The main benefit of DeTail is its ability to reduce extreme latency tails caused by TCP timeouts rather than to resolve interference caused by queueing in switches. The system relies on both Ethernet Flow Control (see Section 2.4.2, pg. 35) and Explicit Congestion Notifications (see Section 2.4.3, pg. 36), both of which I observe exacerbate, rather than resolve network interference. Furthemore, like pFabric, DeTail introduces significant changes to into the switching architecture. Unlike pFabric, no detailed description is offered for how these changes might be implemented in hardware. This makes DeTail unlikely to see adoption in “the age of merchant silicon” [112].

2.5 Multiplexer Scheduling – In Theory and in Practice

Datacenter networks are a shared resource of limited capacity. The congestion control solutions discussed in Section 2.4 attempt to divide this resource equitably between network users. Some systems (§2.4.1-§2.4.5) attempt to arrive at fair-share allocation for each user. Others (§2.4.6–§2.4.11) attempt to optimise for a specific criterion, e.g. minimising average Flow Completion Times (FCT), or minimising missed deadlines. However, congestion control systems are not alone in the task of allocating network resources. Throughout the network are multiplexing points where users contend for access to network resources. Multiplexing points can be found in software, in hardware, or at the borders between software and hardware. For example, multiple software processes, each with one or more flows, may contend to send or receive via the (software) kernel. Similarly multiple flows, from multiple hosts may contend for access to switch hardware to send or receive via that switch. Finally, many flows to (or from) the network adapter may contend for software (or hardware) resources in the end-host.

At each point of contention, some system must be used to determine how to apportion network resources across competing demands. Typically this system comprises one or more queues, and a scheduling algorithm. The scheduling algorithm needs to decide how to allocate three nearly independent quantities, each of which has an impact on network interference. These quantities are [22]: (*i*) which packets get transmitted (bandwidth), (*ii*) when those packets get transmitted (delay), and (*iii*) which packets are discarded (buffer space) when capacity is overrun.

The congestion control systems in Section 2.4 primarily attempt to control ingress and egress rates into and out of the network. Queue depths at the multiplexers are treated as side effect. With the exceptions of pFabric 2.4.9 and DeTail 2.4.11, none of the congestion control systems in §2.4 take into account scheduler behaviour at multiplexing points. Furthermore, all of the solutions rely on cooperation between the hosts using the network. For example, if a single TCP sender chose to ignore its congestion window, it could unfairly monopolise the network to the disadvantage of other senders. Likewise, if a host in the Fastpass network chose to ignore its scheduled transmission intervals, it too could unfairly monopolise the network. This problem is not a new one, nor is it restricted to datacenter network contexts.

Several years before Van Jacobson published his work on *TCP Congestion Avoidance and Control* (see §2.2), John Nagle was grappling with a similar problem [113]. He too was observing congestion based throughput collapse in early ARPANET/Internet systems. To resolve these issues, he proposed two solutions: (*i*) the now ubiquitous[51] 'Nagle's Algorithm' for controlling small packet overheads, and; (*ii*) a congestion avoidance scheme for TCP using ICMP source quench messages. The latter is similar in spirit to (although less developed than) Jacobson's efforts. Having proposed these end-host based solutions, Nagle observed that they were not enough:

> Host misbehavior by excessive traffic generation can prevent not only the host's own traffic from getting through, but can interfere with other unrelated traffic.

My arguments in Sections 2.2 and 2.3 echo this observation. In a followup paper [114], Nagle showed that even in switches with infinite memory, an overload situation caused by misbehaving hosts could result in all packets being dropped. This surprising result lead Nagle to argue that:

> If the packet switches queue on a strictly first in, first out basis, the badly behaved host will interfere with the transmission of data by other, better-behaved hosts.

The key to this argument is the observation that *first in, first out (FIFO)* ordered scheduling at packet switches can allow malicious or malfunctioning users to unfairly monopolise the network. In the Internet, where hosts come from a range of disparate administrative domains, this could be a serious problem. Ideally, different users of the network should be *protected* from each other so that interference cannot occur. At the same time, network utilisation should be maximised.

[51]Available in every major operating system distribution: Linux (`http://linux.die.net/man/7/tcp`), FreeBSD/macOS (`https://www.freebsd.org/cgi/man.cgi?query=tcp&sektion=4&manpath=FreeBSD+9.0-RELEASE`), and Windows (`https://msdn.microsoft.com/en-gb/library/windows/desktop/ms740476(v=vs.85).aspx`).

2.5.1 Scheduling – In Theory

By applying a game theoretic approach to the problem, Nagle proposed a protection solution which would correctly incentivise senders to limit their own sending rates and would protect users from senders who did not [114]. In his solution, each source in the network would be given its own queue at the switch. The queues would be serviced in round-robin order. If a queue was empty, it would be skipped and its service time would be equally (fairly) distributed to other users. Nagle argued that this mechanism would be fair, and that outgoing link bandwidth would thus be parcelled out equally amongst all source hosts whilst maximising overall utilisation. This would be true if all users sent equally sized packets. However, many modern packet switched networks such as the Internet and Ethernet (as used in datacenters) have variable packet sizes. If variable packet sizes are used, then users sending larger packets can still obtain an unfair proportion of the available throughput.

2.5.1.1 (Weighted) Fair Queueing - WFQ

The variable sized packet deficiency in Nagle's round-robin scheduling was addressed by Demers *et al.* [22]. To overcome the deficiency, Demers initially proposed an idealised, hypothetical service discipline. Similar to Nagle's solution, in this discipline, each source/destination pair would be placed into its own queue at the switch. However, rather than scheduling whole packets, every non-empty queue would receive only 1 bit of service per scheduling round. Because every packet must be at least 1 bit in size, the discipline would guarantee fair service between all competing sources, regardless of packet sizes. This discipline has its roots in the Generalised Processor Sharing (GPS) model described in Section 2.5.1.2. The problem with this model is that it is not practically realisable.

Practical packet switches must service one whole packet at a time. To compensate for this, Demers *et al.* proposed the Fair Queueing (FQ) approximation. To chose the next packet to be serviced, an FQ scheduler first runs a 'simulation' of the idealised bit-by-bit scheduling scheme, given the current outstanding packets to be serviced. The simulation calculates the time at which all packets would have been serviced, had they been using the idealised scheme. The FQ scheduler then chooses the packet with the minimum finishing time as the next packet to be serviced. This allows an FQ scheduler to approximate the idealised discipline while still dealing in units of whole packets.

Fair queueing can be generalised to allow for arbitrary bandwidth divisions. To do so, every user is given a weight which determines the relative share of bandwidth that they are allocated. This generalisation is known as Weighted Fair Queueing (WFQ). Weighted fair queueing was independently developed by Parekh *et al.* under the name "Packet-by-packet Generalised Processor Sharing (PGPS)". It has been proven (by Parekh *et al.*) that PGPS/WFQ schedulers are never worse than 1 maximum sized packet behind the ideal discipline [20].

2.5.1.2 (Packetised) (Generalised) Processor Sharing - (P)(G)PS

Processor Sharing (PS) and Generalised Processor Sharing (GPS) are the natural, mathematical extensions of the bit-by-bit round-robin scheme discussed in Section 2.5.1.1 (although the roots of these approaches are much older [115]). Generalised Processor Sharing (GPS) schemes were first proposed by Kleinrock in the context of scheduling processor access in time-shared computers [116]. Although practically unimplementable, these schemes achieve a theoretical best case which forms a bound on the performance of realisable algorithms. For this reason, PS and PGPS are the standard against which all other scheduling algorithms are compared.

Like the bit-by-bit system described above in Section 2.5.1.1, a processor sharing scheduler gives each user, or *session*, its own FIFO queue. These queues are serviced in round robin order, with each queue receiving a quantum Q of service time. However, unlike previously described systems, PS schedulers employ an extreme approach, taking the limit as Q approaches 0 ($Q \to 0$). This implies that each queue will receive an infinitesimal amount of service. Thus, if there are N non-empty queues, every queue will receive an instantaneous service which is exactly one N^{th} of the total output rate. Therefore, in contrast to previously described approaches (see §2.5–§2.5.1.1), PS schedulers are precisely fair. Processor sharing schedulers are also known as *fluid-flow* schedulers because of their infinitesimal division of work.

Processor sharing schedulers serve all sessions at the same rate. However, it may be beneficial to allow each session to be served at some arbitrary faction of the overall capacity. This is much like the Weighted Fair Queueing extension to Fair Queueing (see 2.5.1.1). Applying this extension to processor sharing is called Generalised Processor Sharing (GPS).

A GPS scheduler serves each session in proportion to its weight. More formally [20]: a session is defined as *backlogged* if it has a non-empty queue. If, over some time period $(\tau, t]$, there are N sessions being served, and each session has a positive, real weight $\phi_1, \phi_2, \ldots \phi_N$, then, for any backlogged session i, and any other session j, the following relation holds:

$$\frac{S_i(\tau,t)}{S_j(\tau,t)} \leq \frac{\phi_i}{\phi_j} \tag{2.1}$$

Where $S_x(\tau, t)$ is the amount of data served from the x^{th} session. Furthermore, by summing over all sessions j, the i^{th} session is guaranteed a service rate of g_i, which is a function of the link speed r and the session weight ϕ_i:

$$g_i = \frac{\phi_i}{\sum_j \phi_j} \times r \tag{2.2}$$

Parekh *et al.* proposed a realisable packet based approximation to GPS which they called PGPS. This algorithm, although independently developed, is identical to the WFQ algorithm discussed in Section 2.5.1.1. Parekh proved two important properties for PGPS based systems: (*i*) a packet will never depart later than one maximum sized packet time from a GPS system, and; (*ii*) the amount of data served by a PGPS system will never be less than one maximum sized packet deviation from a GPS system. More formally:

$$d^k_{iWFQ} - d^k_{iGPS} \leq \frac{P_{max}}{r} \tag{2.3}$$

$$S_{iGPS}(\tau, t) - S_{iWFQ}(\tau, t) \leq P_{max} \tag{2.4}$$

Where:

- P_{max} is a maximum sized packet and *r* is the rate of the rate of service.
- d^k_{iWFQ} and d^k_{iGPS} are the times at which the *k*th packet on session *i* departs under WFQ and GPS respectively.
- $S_{iWFQ}(\tau, t)$ and $S_{iGPS}(\tau, t)$ are the amount of data serviced over a time period $(\tau, t]$ under WFQ and GPS respectively.

On this basis, Parekh further proved that PGPS servers could implement bounded delay scheduling when combined with a token bucket regulator [20] and that this delay could be bounded end-to-end if every scheduler in the network was also a PGPS server [21]. The bound on delay (D) in this case is given by the relation:

$$D \leq \frac{\sigma}{g} + \sum_{s=1}^{K} \frac{P_{max}}{r_s} + \sum_{s=1}^{K-1} \frac{M_{max}}{g_s} \tag{2.5}$$

Where:

- All sources are governed by a token bucket abstraction with rate ρ and burst size σ.
- The maximum size of packets in the network is P_{max} and maximum size of packets in the session is M_{max}.
- The packets pass through *K* switches (schedulers) in total.
- For each switch *s*, there is a total rate r_s of which each session receives a rate g_s.
- *g* is the minimum of all g_s.
- It is assumed that $\rho \leq g$, i.e. the session is underutilised or at most perfectly utilised.

The terms in this equation can be understood intuitively. The first term σ/g is the pure GPS term. This is the delay that would be experienced without a packetised GPS approximation. The second term, $\sum_{s=1}^{K} \frac{P_{max}}{r_s}$, is the PGPS correction term. This term adjusts for the difference between GPS and the PGPS approximation at each switch. Recall from Equation 2.3 that PGPS WFQ approximations are never worse than one full sized packet behind a GPS scheduler (see also [20]). The final term, $\sum_{s=1}^{K-1} \frac{M_{max}}{g_s}$, is the serialisation delay correction term. This term adjusts for the time it takes each packet to be serialised as is passes between each switch.

These result above is important for two reasons. First, it proves that, with the help of schedulers along the way, isolation is possible end-to-end in a packet switched network. Second, it proves that, with the help of token bucket regulators at the source, an upper bound on delay is possible end-to-end in packet switched network. Combined, these properties show that, at least mathematically, network interference can be solved in packet switched networks.

2.5.1.3 Worst-case Fair, Wighted Fair Queuing - WF^{2}Q

Equations 2.3 and 2.4 can easily be misinterpreted. They show that the amount of data served by a PGPS system will never be *less than* one maximum sized packet deviation from the equivalent

GPS system. However, this does not imply that a PGPS scheduler provides near identical service to a GPS scheduler. It is possible for a PGPS scheduler to serve much more data than the equivalent GPS server. This can occur when backlogged packets are served ahead of the time at which they would have begun service in a GPS system. The authors of Worst Case Fair, Weighted Fair Queueing (WF^{2}Q) [23] demonstrate this issue and resolve it. The WF^{2}Q scheduler resolves this discrepancy by introducing a further constraint into the GPS simulation. Packets are only chosen for service from the set of packets which would have started (or possibly finished) service in the equivalent GPS system. WF^{2}Q is proved to have a tighter bound than WFQ. It deviates by no more than one maximum sized packet, either behind, or ahead of GPS.

2.5.1.4 Latency Rate LR Servers

Beyond WFQ (see §2.5.1.1), PGPS (see §2.5.1.2) and WF^{2}Q (see §2.5.1.3) are a range of other scheduling algorithms that approximate GPS scheduling. These include Self-Clocked Fair Queuing [117], Virtual Clock [118], Deficit Round Robin [119], Weighted Round Robin [120] and many others. Most have similar operational principles and performance bounds. Typically these different schedulers make different trade-offs between the closeness of the approximation and the implementation complexity or speed. Some have only very subtle differences to PGPS (e.g. Virtual Clock [118]). It is therefore not surprising that all of the above schedulers can be classified into a single, more general, scheduling class.

Stiliadis *et al.* describe the class of *latency rate (LR)* schedulers [121] which encapsulates the above scheduling algorithms. latency rate servers are defined by their deviation from GPS during a period of backlogged service. Specifically, an LR scheduler is defined by two properties, latency (Θ) and rate (ρ). Formally, an LR server is one that satisfies the following relation:

$$S_{i,j}(\tau, t) \geq \max(0, \rho_i(t - \tau - \Theta_i)) \tag{2.6}$$

Where:

- The j^{th} busy period of session i begins at time τ and finishes at time t.
- The total service provided to session i during the during the interval $(\tau, t]$ is given by S
- ρ_i is the rate allocated to session i
- Θ_i is the minimum non-negative number that satisfies the inequality.

For a scheduling algorithm to be in the class LR, Equation 2.6 simply requires that there is a lower bound on the rate of service it offers. Intuitively, the parameter Θ quantifies this bound as the maximum deviation in delay from GPS that a given scheduler will achieve over a given busy period. Recall that in Section 2.5.1.2 the bound of a PGPS system (Eq. 2.5) is given by three terms, a pure GPS term (σ/g) and two correction terms ($\sum_{s=1}^{K} \frac{P_{max}}{r_s}$ and $\sum_{s=1}^{K-1} \frac{M_{max}}{g_s}$). As defined above, Θ is essentially a generic term that generalises over the correction terms for different schedulers. Given this understanding, it should intuitively follow (and Stiliads *et al.* prove), that a sequence of LR schedulers have a bound given by:

$$D_i \leq \frac{\sigma_i}{g_i} + \sum_{s=1}^{K} \Theta_{i,s} \tag{2.7}$$

The result above is important because it proves that bounded delay across a network does not specifically require a PGPS scheduler. Instead, all schedulers that can be described within the LR class can be arbitrarily combined, yet bounded delay can still be achieved. This means that even on a complex network comprising many different sub-networks, where each sub-network operates its own subtly different schedulers, bounded delay can still be achieved. The Internet is an example of a complex network of sub-networks, independently operated and administered, where bounded delay would be a desirable property. One notable implementation of bounded delay over heterogeneous networks, founded on the ideas of LR networks, is the Integrated Services framework (described in §2.5.2.1).

2.5.2 Scheduling – In Practice

Early in the history of the Internet, researchers and practitioners became aware that the Internet might be used for many different purposes. For example, it might be used for interactive remote sessions, and, real-time video and audio [20, 21, 118, 122, 123], as well as being used for email and file transfer systems. While file transfer systems work best if given high throughput, realtime audio/video systems were expected[52] to require bounded delay and jitter properties. Designers/researchers reasoned that it would be ideal to "integrate" these different "service" requirements into the same network infrastructure. This idea lead to the development of the Integrated Services standard (IntServ) [123].

2.5.2.1 Integrated Services - IntServ

The Integrated Services (IntServ) (IS) architecture was designed to support realtime as well as non-realtime services over the Internet. The core idea is to abstract the notion of "packet service" as provided by the Internet network. By abstracting packet service, different packet schedules could be added to the Internet, allowing different types and quality of service to be offered. While the original Internet design would only supply best effort delivery, the IS authors argued that there would be "an inescapable requirement for routers to be able to reserve resources in order to provide special Quality of Service (QoS) for specific packet streams" [123]. This would in turn require flow specific state at each router.

[52]It is beyond the scope of this dissertation, although this point raises some interesting questions: To the best of my knowledge, end-to-end realtime/prioritised packet delivery and scheduling systems like Integrated Services (IS) have not been deployed widely over the internet at large. Yet, realtime audio and video systems like FaceTime, Skype, Viber. WhatsApp etc. have proliferated. This begs the questions: (*i*) are advanced scheduling systems like IS more common than it is generally accepted? (and if so, who is paying for the premium service?) (*ii*) if not, what did the IS authors not predict about the future (current) internet, that has made IS unnecessary for supporting these realtime services? Is it simply that bandwidth has increased or is there more to it?

The IntServ design requires two essential components: (*i*) a resource reservation protocol, and (*ii*) a per-router implementation architecture. The standard reservation protocol used by IntServ is the Resource reSerVation Protocol (RSVP) [124]. RSVP is designed as a generic mechanism for making flow specific requests for network resources and state along a route from source to sender. The protocol requests resources for simplex flows, i.e. it requests resources in only one direction. Importantly, these requests are receiver oriented. This means that the requests are made by a receiver and follow the reverse path of the flow from receiver back to the source that they are reserving resources for. The resources reserved by an RSVP request depend on the Quality of Service (QoS) requirements of the application and are described by a FlowSpec. The packets affected by the reservation are selected by a FilterSpec. Together a FlowSpec and a FilterSpec form a FlowDescriptor which is issued in an RSVP reservation request message.

The second component of the IntServ design is the per-router architecture. The architecture comprised three key components:

- **Packet classifiers** – which essentially implement the desires of FilterSpecs. Packet classifiers map each incoming packet into a class that will be acted upon by a scheduler.
- **Packet schedulers** – which apply scheduling policies to queues of classified packets. Packet schedulers implement the QoS desires of applications as given by FlowSpecs using algorithms such as those described in Section 2.5.1
- **Admission control** – which accepts or rejects FlowDescriptors based on the requested QoS level and the available resources.

IntServ makes a collection of different quality of service options available [125, 126]. In the context of network interference mitigations, the most relevant is the service model for real-time applications called the *guaranteed service* [126]. This service class is specifically aimed at schedulers like those described in Sections 2.5.1.1–2.5.1.3. The end-to-end behaviour of the guaranteed service class is based on the fluid-flow model described in Section 2.5.1.2. In the guaranteed service model, queueing delays must not exceed the equivalent fluid-flow delays by more than a specified error bound. According to the standard, the error bound is described by two values C and D in the equation:

$$D \leq \frac{b}{R} + \sum_{s=1}^{K} \frac{C_s}{R} + \sum_{s=1}^{K} D_s \tag{2.8}$$

Where:

- the flow passes through each router s and there are K routers in total[53].
- the flow obeys a token bucket rate limiter with rate r and burst size b.
- the flow rate r is less than or equal to network capacity for that flow R.

[53] While the standard does not use the s, K notation, I have transformed the notation for clarity and consistency of this text.

- C_s describes the rate-based deviation from the fluid-flow model at each router s
- D_s describes the rate independent deviation from the fluid-flow model at each router s

By coalescing terms C_s and D_s into a generic deviation term Θ_s and renaming the variables b and R into σ and g respectively, Equation 2.8 can be rewritten as:

$$D \leq \frac{\sigma}{g} + \sum_{s=1}^{K} \Theta_s \tag{2.9}$$

which corresponds exactly with the Equation 2.7. This means that the Guaranteed Quality of Service class for IntServ simply requires any LR (see §2.5.1.4) class scheduler to be implemented at each router. When a receiver makes a reservation for guaranteed service, it includes a value S or *slack time* in the RSVP request [126, 127]. This value defines the deviation from a fluid-flow model that the application can tolerate. This value is the same as the "latency" component of an LR server.

2.5.2.2 Differentiated Services - DiffServ

IntServ requires that all routers along a path implement per-flow state. This puts pressure on memory I/O and a fixed upper bound on the number of flows that a given router can support. Furthermore, it requires that each Internet service provider cooperate by allowing RSVP messages to traverse their infrastructure and make per flow state/reservation changes. Even from the outset, it was acknowledged that this would cause scalability and deployment issues [123], but these issues were never fully addressed by the standard. In response to these issues, an alternative standard was proposed. The authors of the standard reasoned that it would be sufficient to simply differentiate between differed classes of service using coarse grained mechanics. This lead to the development of the *Differentiated Services (DS) (DiffServ)* standard [128].

The differentiated services architecture is based on a simple model where traffic entering a network is classified and possibly conditioned at the boundaries of the network. This traffic is assigned to a class known as a *behaviour aggregate (BA)*. Each behaviour aggregate is identified by a single *Differentiated Services Code Point (DSCP)*. A code point is a 6-bit field in the IP header which defines how packets within the core of the network are forwarded. At each hop over the network a *Per-hop Behaviour (PHB)* determines the actions that will be applied for a given differentiated services code point value. There are (currently) four well defined PHBs.

- The default PHB [129] specifies that a packet marked with the DSCP value '000000' gets the traditional best effort service. If a packet arrives at a DS compliant node and its DSCP value is not mapped to any of the other PHBs, it will get mapped to the default PHB[54]

- The Class-Selector PHBs [129] implement a backwards compatible layer that can interoperate with the earlier IP Type of Service (ToS) [130] coded packets.

[54]Cisco white-paper. See Appendix A.4.1.

- The Expedited Forwarding PHB [131] provides similar operation to the IntServ Guaranteed Service class for low-loss, low-latency, low-jitter and assured bandwidth requests.

- The Assured Forwarding PHB [131] is similar to the IntServ controlled load class. It defines a method where BAs can divide traffic into different assurance level classes. These can be used to, for example, subdivide bandwidth between different categories of traffic.

The DiffServ (DS) model distinguishes between different DS domains. A domain is a contiguous set of DS nodes which operate with a common service policy and common set of PHB groups implemented on each node. To provide end-to-end Quality of Service (QoS), each domain along the path must support the DiffServ code-points requested and some external mechanism must be used to negotiate the policy or Service Level Agreement (SLA) that will be applied. Unlike IntServ, the DiffServ standard does not explicitly contain a mechanism or guidelines on how this negotiation might be completed. In this sense, DiffServe is better suited to intranet service differentiation (such as within an Internet Service Provider (ISP) or a Local Area Network (LAN)) rather than Internet differentiation.

2.5.2.3 IntServ vs. DiffServ

IntServ (see 2.5.2.1) and DiffServ (see 2.5.2.2) take fundamentally different approaches to the problems of providing quality of service. At one extreme, DiffServ is pragmatic, rooted in easily implementable mechanics with a minimum of hardware support. But, DiffServ is not an end-to-end solution, it works only within a specific DS domain, and even within a domain it provides no guarantees of performance. It is up to a DiffServ operator to determine, implement and deploy the correct policies. Furthermore, the DiffServ standards (e.g. [131]) provide almost no guidance about how to implement robust guaranteed bounded performance within a DiffServ domain.

At the other extreme, IntServ is rooted in the fundamental theorems of infinitesimally divisible fluid-flow models. It provides strong, mathematically provable bounds on bandwidth, delay and hence interference. However, these bounds are based on perfect separation of individual flows into distinct queues, knowledge of the minimum and maximum requirements of every flow and require routers to implement complex scheduling algorithms that apply sophisticated scheduling and prioritisation of packets on a per-flow basis. The implementation of these algorithms is not trivial [132] and it is difficult to find modern systems that support them. It would be ideal to be able to provide the benefits of the rigorously developed mathematical guarantees of IntServ, with the pragmatic implementation focused design of DiffServ, tailored for the resources available in modern datacenter networks. This goal will be the focus of the following chapters of this dissertation.

2.6 Conclusions

In this chapter I discussed the construction and operation of warehouse scale datacenters (§2.1). In particular, the problems associated with congestion, which causes interference between applications (§2.2–2.3). Although there are a variety of approaches to congestion control in datacenters, none provide an effective solution to network interference (§2.4) and most approaches require drastic changes to the hardware and software, making them impractical to deploy. I also discussed the work of Parkeh *et al.* that builds on Generalised Processor Sharing (GPS) (§2.5). GPS is a scheduling model where work is assumed to be infinitesimally divisible. It provides perfect isolation and, by definition, makes interference impossible. Unfortunately GPS is unimplementable. Parekh *et al.* proposed a discretised approximation of GPS in the network context called Packet-by-packet Generalised Processor Sharing (PGPS)[55] which has a small but known deviation from the ideal schedule. Furthermore, Parekh *et al.* showed that when the traffic load is governed by a token bucket regulator, PGPS can deliver bounded latency across the network. Stiliadis *et al.* went on to show that PGPS is just one example in a class of "Latency Rate" servers which can be combined to offer bounded latency across a network. LR servers form the basis of the IntServ standard which was proposed to offer different service types in the Internet. In reaction to difficulties with deploying IntServ, DiffServ was developed which takes a more course-grained, practical approach to providing Quality of Service.

2.7 Chapter Summary

- Warehouse scale datacenters are a necessary component of modern Internet services (§2.1).
  - They are built using commodity x86 hosts and merchant Ethernet switches, and run at a scale of approximately 50,000 hosts per facility (§2.1.2).
  - Datacenter networks are built using Clos-like topologies typically with a top-of-rack oversubscription ratio of about 3:1 (§2.1.3)
  - The networks are organised into subnetworks called pods which have a few thousand hosts in each (§2.1.3).
  - The predominant protocol used in datacenter networks is TCP (§2.1.4).
- The TCP congestion control algorithm (§2.2) tries to sense congestion by filling up queues which can lead to interference between applications (§2.3).
  - Interference occurs when packets from latency sensitive applications share congested queues with throughput intensive applications.
  - Congested switch queues can cause delays of $16\times$ the idle case (§2.3.1).
- Conventional and proposed mechanisms for congestion control in datacenter networks fail to resolve network interference (§2.4).

[55] Also proposed by Demers *et al.* as Weighted Fair Queueing (WFQ) [22].

- There are mathematical models which show that it is possible to achieve near perfect isolation in networks which will bound the amount interference that can result (§2.5.1).
  - These models have been implemented in QoS schemes for the Internet, but they are complicated to deploy and require per-flow state (§2.5.2.1).
  - To combat this, coarse-grained classification has also been proposed, but the standards provide no guidance on how to implement isolation and guarantees using coarse-grained classification (§2.5.2.2).

Chapter 3

Bounding Delay in Datacenter Networks

IN the preceding chapter I discussed the work of Parkeh *et al.* that builds on Generalised Processor Sharing (GPS) (§2.5) and Stiliadis *et al.* who showed that PGPS is just one example in the class of latency rate (LR) servers. All LR servers can be combined to offer bounded latency across a network. The problem with these approaches is the implicit assumption that network switches can be arbitrarily altered to implement a given scheduling policy. This is not true in datacenters because they are constructed using commodity switch silicon components[1] (see §2.1.2–§2.1.3). To derive a delay bound and implement isolation in a datacenter network, we must take into account the properties of datacenter switches as they are, rather than how we would like them to be.

In this chapter, I take a similar approach to Parekh *et al.*. I combine a switch scheduling model and an input traffic regulator to provide bounded delay through the network. The bound provides guarantees that are as tight as circuit switches. However, rather than beginning from an idealised GPS fluid-flow scheduling model, I take the novel step of starting with known and measured properties of existing datacenter switches. Based on these properties, I create simple, but realistic, discrete switch scheduling model. Using the model, I then derive the necessary traffic regulation conditions to bound delay across the scheduler and consequently through the network. I express the traffic regulation conditions using a new type of fluid-flow traffic regulator called a *Leaky Token Bucket (LTB)* and the resulting delay bound using a simple equation, parameterised over the maximum packet size, the number of senders, network link-rates, and switch performance. In later chapters I will relax these bounds and demonstrate how the model, traffic constraints, and leaky token bucket regulator can be implemented and deployed in datacenter practical networks. In doing so, I show that it is possible and practical to precisely control datacenter network latency and therefore mitigate the effects of network interference.

[1]To the best of my knowledge. As noted in §2.1.2, datacenter operators have already requested significant CPU modifications from Intel, so there is no reason, in principle, that similar requests could not also be made to switch silicon vendors. However, at present, there is no evidence of this happening.

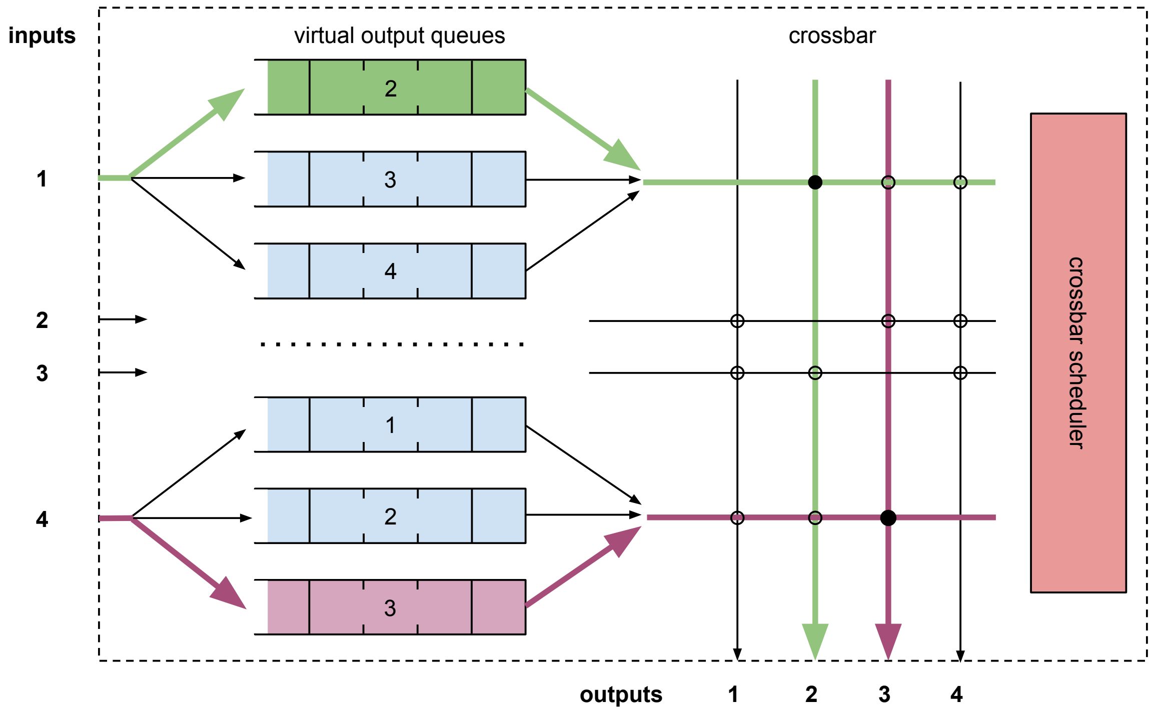


Figure 3.1: *An architectural model of a Virtual Output Queue (VOQ) datacenter switch*

3.1 A Model of a Datacenter Switch

For the remainder of this dissertation I will assume that datacenter switches can be modelled according to the high level architectural diagram given in Figure 3.1. The figure shows a 4 port switch, with input and output *ports* numbered 1 to 4. Inputs 1 to 4 are shown on the left-hand side of the figure while outputs 1 to 4 are shown on the bottom of the figure. In the figure, each input has 3 *output queues* associated with it (the queues for inputs 2 and 3 are omitted for clarity). The output queues are used to store packets before they are sent (*forwarded*) to the correct output. This type of queuing is called *Virtual Output Queue (VOQ)* [133] because the output queues are located on the input side of the switch. It is generally understood that VOQ designs are the basis of modern high speed switching devices[2] [134, 135].

When a packet arrives at the switch, the destination address is decoded from the packet header and the packet is placed into the corresponding virtual output queue. For example, a packet arriving on input 4, destined for output 3 will be placed into the third output queue associated with input 4. Ethernet prohibits circular forwarding so inputs do not have an output queue corresponding to themselves. That is, in this example, input 1 contains queues for outputs 2, 3, and 4, but no queue for output 1. If there are N ports on the switch, each input will have at least $N-1$ output queues dedicated to it. This arrangement means that there will be at least $(N-1)\times(N-1)\approx N^2$ queues in total in the switch.

In practice there are usually many more than N output queues per input. Packets

[2]Cisco 5000 Series white-paper. See Appendix A.4.2.

arriving at the switch will be sorted into different queues based on the destination output port as well as the type of packet (e.g. if the packet is unicast/multicast) and any class/priority of service tags (e.g. DSCP). For example, on each input, the Cisco Nexus 5548, 48 port, 10Gb/s datacenter switch[3] supports 128 multicast queues, and, 8 different priority queues per output. The device has 512 queues in total per port (i.e. $128 + 48 \times 8 = 512$) and a little under 25,000 queues in total (i.e. $48 \times 512 \approx 25,000$). The 8 priority queues per input are used to implement simple Quality of Service (QoS) features such as priority queueing via Ethernet Virtual Local Area Network (VLAN) priorities [136] and/or the DiffServ [123] code point, both of which are widely supported[4]. However, there is no support for more advanced per-flow queueing QoS schemes such WFQ, WF^{2}Q etc. (described in Sections 2.5.1.1–2.5.1.4) which would require vastly more queues and memory I/O bandwidth.

Once packets have been sorted into their appropriate virtual output queues, they need to be transferred to the output ports. To do so, packets need to cross the *crossbar*. In the figure, the crossbar is the grid structure with all input ports on the left, and all output ports on the bottom. At the intersection of each input/output pair, an electrical connection must be made to join the output queue to the actual output. The figure show input 1 connected electrically to output 2, and input 4 connected electrically to output 3. The paths between input 1 and output 2, and, input 4 and output 3 are also highlighted. Only one input port can transmit to one [5] output port at a time. A crucial function of high performance switches is to determine the schedule of crossbar connections. In general crossbar schedulers need to achieve high utilisation and fairness across competing input queues as well as being simple and implementable in hardware. This makes crossbar schedulers an important component of high-speed switches.

Unfortunately specific details of crossbar scheduler designs are viewed as a trade secret by switch silicon manufacturers. This is because the overall performance of the switch is tied to the quality of the scheduler's decisions. One well known scheduling algorithm is iSLIP, proposed by McKeown [103]. It is widely believed that variants of iSLIP are used in production switch chips, though there is little direct evidence to support this. The Cisco 5000 series switches are a notable exception. Marketing materials for the switch state that it is "based on an enhanced iSLIP algorithm" which has been modified "to accommodate cut-through switching of different packet sizes"[6]. No further details of the modifications are available and other manufacturers are reluctant to say even this much. As a result, in my model I do not (and cannot) assume a specific switch scheduling algorithm. I will instead assume that the switch crossbar scheduler implementation has at least the following properties:

1. *port-independent, parallel matching*: the scheduler should independently and in parallel perform matching for each output port. Scheduling one output port should not interfere

[3]Cisco 5548 switch white-paper. See Appendix A.4.3.

[4]Broadcom BCM56580 "Trident II+" marketing material, Intel FM5000/FM6000 datasheet and Cisco 5000/5500 series white-papers. See Appendix A.3.3, A.16.5, A.4.2 and A.4.3.

[5]or more if doing multicast

[6]According to Cisco 5548 white-paper. See Appendix A.4.4.

with scheduling another. For example, scheduling packets to output port 1 should not interfere with or delay scheduling packets for output port 3.

2. *work conserving*: if an output is available and there is a packet destined for it, the packet will be scheduled. Packets should not wait unless another packet is already being served.
3. *throughput conserving*: if there are multiple packets destined for an output, the switch will schedule them back-to-back. The minimum sized *Inter-frame Gap (IFG)* [16] should be maintained so that line-rate throughput is possible at the output.

Note that the above definitions do not include any explicit notion of fairness across competing inputs for a given output. It will become clear in Section 3.2 why this is not a requirement.

The iSLIP algorithm uses a parallel matching process which satisfies these requirements. My switching model is therefore applicable to at least any reasonable switch silicon implementation of an iSLIP-like algorithm (e.g. the Cisco 5548 switch[7]). Furthermore, the experiments conducted in Section 3.4.6 show that the popular Trident II+[8] switching silicon also meets these requirements, although it is not known if it implements iSLIP internally.

Beyond hardware switches, end-hosts also implement software multiplexing points. 1-to-n and n-to-1 multiplexers are a common component of the network path in end hosts where many applications share one network card and vice-versa. Software implementations acting as either n-to-1 multiplexers or 1-to-n demultiplexers are likely to satisfy the above requirements. However, it is unlikely that general purpose n-to-n software switch implementations like OpenVSwitch [137] will be able to satisfy the above requirements. Such a software switch would need to dedicate a CPU core to each output if it was to *perform port-independent, parallel matching* (requirement 1). Furthermore shared memory and data structures are likely to induce some degree of interference. Finally, obtaining *throughput conserving performance* (requirement 3) in a software switch may be a serious challenge. Further work, outside of the scope of this dissertation, is needed to examine effects, applicability and potential modifications needed to apply this model to general n-to-n software switches.

3.2 Bounding Delay in Datacenter Networks

Using the switch model described in Section 3.1 it is possible to derive a delay bound for datacenter networks. To do this, I start by deriving the conditions necessary to obtain a delay bound in a simplified *one-shot* scenario (§3.2.1). In this scenario there is only one switch and the senders are allowed to send at most one packet, once to the switch. In Section 3.2.2, I will relax the one-shot restriction to allow hosts to send an arbitrary number of packets. Finally in Section 3.2.4, I will expand the bound conditions to include multi-hop network topologies. To begin the analysis, I will make the following initial assumptions:

[7]According to Cisco 5548 white-paper. See Appendix A.4.4.

[8]The Broadcom BCM56850 is also known as "Trident II+" according to Broadcom marketing material. See Appendix A.3.2.

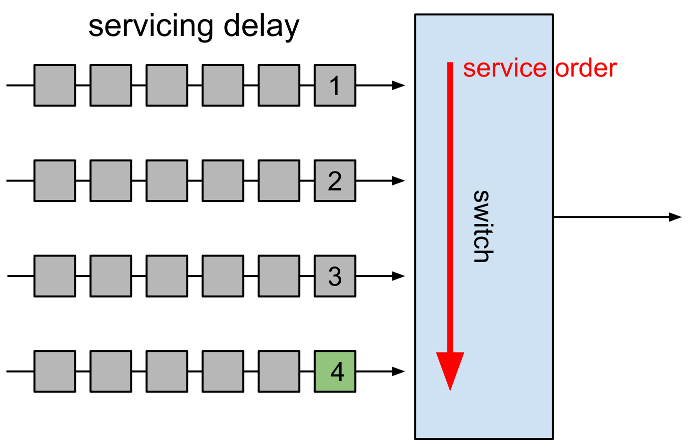


Figure 3.2: *Packet number 4 is unlucky. It waits for 3 previous packets to be serviced before it is serviced.*

1. There is a single switch with a single switch crossbar inside[9].
2. The switch is a store-and-forward type. That is, all packets will be buffered at least once inside the switch[10].
3. The switch implements a scheduler with the properties described in Section 3.1.
4. The switch is initially idle. No packets are in flight or are being processed at the switch and no packets are generated at the switch[11].
5. Each sender is attached to a single port.
6. All senders may send at most one packet to the switch at any time.
7. All packets are a fixed size. Furthermore, the Inter-frame Gap (IFG) period is considered to be part this fixed size. This means that IFGs can be ignored.

I will relax assumptions (4) in Section 3.2.2, assumption (6) in Section 3.2.3 and assumption (1) in Section 3.2.4. Assumption (3) will be discussed in more detail in Section 3.4.6.

3.2.1 The One-shot Case

First I assume a single switch (as described in §3.1) with a single host connected to it. The host may send only a single packet. I define the *delay* (or *latency*) across a switch as the interval between the last bit of the packet arriving at the switch, and the last bit of the packet leaving from the switch[12]. This definition takes into account the serialisation time required to output the packet onto the output link and is thus dependent on both packet size and the switch's output rate. It is not dependent on the switch's input rate.

[9]More complex multi-chip switches are essentially multi-hop networks and can be treated as such.

[10]As opposed to cut-through [138] switches. A store-and-forward switch queuing model is a reasonable approximation of a cut-through switch model and is always a strict upper bound.

[11]For example Link Layer Discovery Protocol [139] or Spanning Tree Protocol [140] packets.

[12]My definition is in contrast to RFC1242 [141] which defines delay across a store-and-forward device as the period between the last bit arriving, and the first bit departing.

When a packet of size P arrives at a store-and-forward switch, it must be fully buffered before it can leave. The last bit to arrive will need to wait until all previous bits have been serialised through the output before it can bit output. Assuming that there are P bits in the packet and the switch has an output speed of r bits/second, it will take P/r seconds to serialise the packet. The switch also adds some amount of intrinsic delay ϵ_i as the packet moves through the switch pipeline and crossbar. Thus, the delay experienced by the a single packet across a single switch is given by:

$$d_{1p} = \frac{P}{r} + \epsilon_i \tag{3.1}$$

Where:

- d_{1p} is the delay in seconds to service one packet
- P is the packet size in bits, and r is the output link rate in bits per second.
- ϵ_i is the switch intrinsic delay constant (in seconds) added by the switch.

If instead of one host, there are m hosts connected to the switch, and each host can send a single packet, then the worst case delay occurs when all packets arrive at the same time. In this situation the last packet to be serviced experiences the longest delay. I call this kind of delay the *switch servicing delay* (see Figure 3.2). The servicing delay period is given by the sum of the per-packet bounds:

$$d \leq \sum_{p=1}^{m} \left(\frac{P}{r_p}\right) + \epsilon_i \tag{3.2}$$

Where:

- d is the worst case delay (in seconds) for the switch to service one packet from each of m hosts.
- the packet size is P bits, and the minimum link rate for a given port p is r_i bits per second.
- ϵ_i is the switch intrinsic delay constant (in seconds) added by the switch.

Note that the intrinsic delay (ϵ_i) is included only once. Since the switch is required to be throughput conserving (see requirement 3, §3.1) only the first packet will experience the intrinsic switch delay. For other packets this delay will be hidden. Furthermore, let us assume that all ports on the switch run at a rate of at least r. This gives us the equation:

$$d \leq m \times \frac{P}{r} + \epsilon_{sw} \tag{3.3}$$

On a modern switch, the quoted switch processing delay ϵ_{sw} is typically between 95ns and 350ns[13]. Since queueing, host, and application latencies are often 10–$1,000\times$ greater (e.g. see Table 2.1), the switch processing delay can usually be ignored.

3.2.2 The Multi-shot Case

To calculate the multi-shot delay bound, we must consider a further source of delay. I will call this type of delay *queueing delay*. Queuing delay happens when a new packet arrives at the tail

[13]Exablaze ExaLink Fusion, Arista 7150 and Cisco Nexus 3548 marketing material. See Appendix A.9.1, A.2.3 and A.4.5.

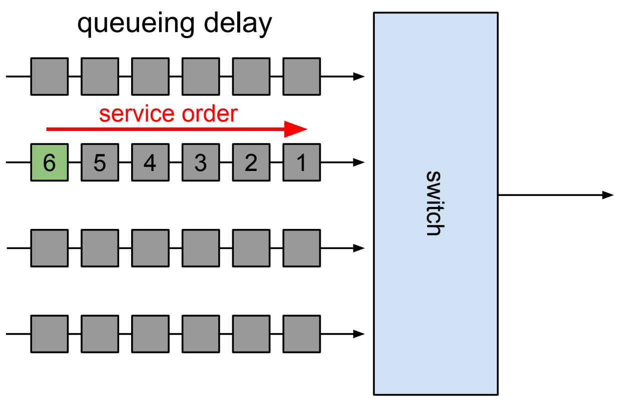


Figure 3.3: *Packet number 6 is at the back of the queue, it must wait for 5 packets to be removed from the head of queue before it will be serviced.*

of a queue and must wait until it reaches the head so that it can be scheduled (see Figure 3.3). Queuing delay is thus a property of the rate at which packets are added to and removed from each queue. If the input rate is higher than the output rate, a queue may build up indefinitely and overflow. This is not a new problem. Recall that in order to maintain a delay bound, Parekh *et al.* required the network to be underutilised or at most perfectly utilised (see §2.5.1.2). Specifically that:

> $\rho \leq g$; where: ρ is the rate of the token bucket regulator, and g is the minimum rate assigned to the session.

The solution to the problem is also not new. Recall the principle of 'conservation of packets' as proposed by Jacobson (see §2.2). Specifically that:

> A new packet isn't put into the network [switch queue] until an old packet leaves.

It follows that, if the switch is initially idle (assumption 4, §3.2), and we ensure that packets are only added at a rate that the switch scheduler can service them, then they will never experience queuing delay. This means that the delay bound can be maintained indefinitely.

In Section 3.2.1 I derived a bound for the *switch servicing delay* in a single shot scenario (Eq. 3.3). This bound is the time the switch will take to service all packets sent to it if each host sends only one packet and the switch is idle. If we would like hosts to be able to send multiple packets, we can regulate each host to send one packet for each switch servicing delay period. If each host can send only one packet per switch servicing delay period then, by definition, the switch will service all packets in the period and every packet will arrive at an empty queue. I will henceforth refer to the switch servicing delay period as the *switch epoch* (e_{sw}) to distinguish it from the end-to-end delay bound (d). In single switch scenarios these values are identical. In more complex arrangements they are not. Remembering that packet sizes are assumed to be fixed (assumption 7,3.2), by applying a switch epoch based rate limit to all senders, senders will receive a guaranteed share of the bandwidth and a guaranteed end-to-end delay bound. The rate limit R is given by dividing the maximum packet size P_{max} by the switch epoch e_{sw}. The epoch in this case is the same as switch servicing delay d from Equation 3.2.1. Formally:

$$R = \frac{P}{e_{sw}} = \frac{P}{mP/r + \epsilon} \approx \frac{r}{m} \tag{3.4}$$

Note: the rate computed by the above Equation 3.4 is subtly misleading. The resulting value R in bits per second appears to be a continuous value over time. It would seem to imply that in any interval $(t, \tau]$, at most $(\tau - t) \times r$ bits may be sent into the network. This is not true. If the value of $\tau - t$ is greater than the servicing interval, then more than P bits might be sent into the network, violating the packet size restriction (P) and thus potentially violating the delay bound. In order to maintain the delay bound, both conditions must hold. That is, a rate of R must not be exceeded and packet size of P must not be exceeded. I therefore refer to the *average rate-limit* as $R_{ave} = P/e_{sw}$ and the *effective rate-limit* R_{eff} as the tuple (P, e_{sw}) that captures this restriction.

3.2.3 Variable-sized Packets and the Leaky Token Bucket

If a host is (P, e_{sw}) limited according to Equation 3.3, and issues a packet that is smaller than P, this will not affect the upper bound on the servicing delay. We can therefore relax the assumption that packets are a fixed size P (assumption 7, §3.2), and instead allow packets to be at most P_{max} bits in size. The switch epoch e_{sw} given by equation 3.3 thus becomes:

$$e_{sw} \leq m \times \frac{P_{max}}{r} + \epsilon_{sw} \tag{3.5}$$

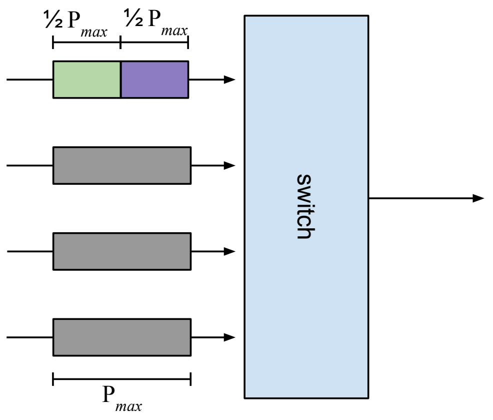
In this case, the effective rate-limit R_{eff} is now described by the tuple (P_{max}, e_{sw}) and the average rate-limit R_{ave} is given by:

$$R_{ave} = \frac{P_{max}}{e_{sw}} = \frac{P_{max}}{mP_{max}/r + \epsilon} \approx \frac{r}{m} \tag{3.6}$$

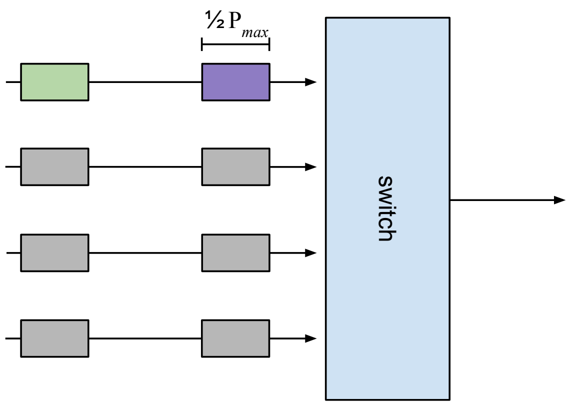
Allowing hosts to send smaller packets causes a secondary problem. Recall that a source must limit its transmissions to one packet of size $\leq P_{max}$ every e_{sw} seconds to maintain the delay bound. If a host issues a packet of size $l < P_{max}$, it will need to wait e_{sw} seconds before sending another packet and thus waste bandwidth proportional to $P_{max} - l$.

To alleviate this bandwidth reduction, consider the case where a host is allowed to issue two, half-sized packets ($P_{max}/2$) back-to-back[14] (see Fig. 3.4a). In this case the delay bound will still be maintained because the amount of work that the switch needs to do does not change, i.e., the switch still needs to service at most $m \times P_{max}$ bits within a period of e_{sw}. However, the advantage of sending half-sized packets is that the sender may choose to space these packets appart. To maintain the delay bound, we need ensure that the switch will always have enough time to service half-sized packets, regardless of the spacing between them, or how many there

[14]As in assumption 7, §3.2, the Inter-frame Gap (IFG) is assumed to be encapsulated into this size so that it can be safely ignored.



***(a)** A back-to-back half-sized packet.*



***(b)** All packets are half-sized.*

***Figure 3.4:** Half-sized packet considerations.*

are. The limiting case is thus when all hosts are allowed to send half-sized packets, and these packets arrive at the switch, at same time (see Fig. 3.4b).

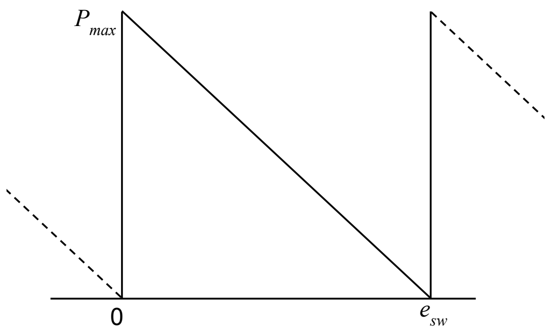
Equation 3.5 can be applied to give the service time required for this case. From the equation, the switch epoch (maximum servicing time) e_{sw} is proportional to the maximum packet size P_{max} (ignoring ϵ_{sw}). If we halve the packet size, the epoch is also halved. That is, if we set $P'_{max} = P_{max}/2$ then $e'_{sw} = e_{sw}/2$. Therefore, the latest moment that any of the half size packets may arrive is $e_{sw}/2$.

We can easily generalise this logic for any packet size. If $P'_{max} = P_{max}/x$ then e'_{sw} is simply e_{sw}/x. Figure 3.5a shows this relationship graphically. As time progresses across the x-axis from 0 to e_{sw}, the maximum number of bits that can be issued into the network decreases proportionally from P_{max} down to 0. Formally at a time v where $0 \leq v \leq e_{sw}$, a host may send at most b bits into the network where b is given by:

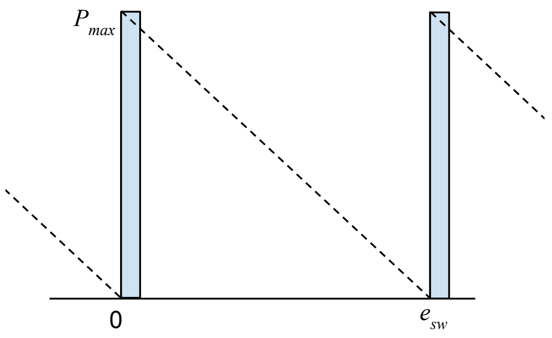
$$b \leq \frac{e_{sw} - v}{e_{sw}} \times P_{max} \quad 0 \leq v \leq e_{sw} \tag{3.7}$$

Note that this relation only remains true provided that the sender continues to be (P, e_{sw}) constrained. That is, at most P_{max} bits can sent in total during any time e_{sw}. Figure 3.5b again shows this relationship graphically. In the figure, a full sized packet is issued at time 0. As a result, P_{max} bits have been sent and no more bits can be transmitted for another e_{sw} seconds. In Figure 3.5c, a half-sized packet is issued at time 0. At any time up to $e_{sw}/2$ later, another half-sized packet may be issued. Finally, in Figure 3.5d, minimum sized packets are issued across the whole period. Again the delay bound will be maintained provided that no more than P_{max} bits are sent in total, and provided that no packet is larger than the result from Equation 3.7.

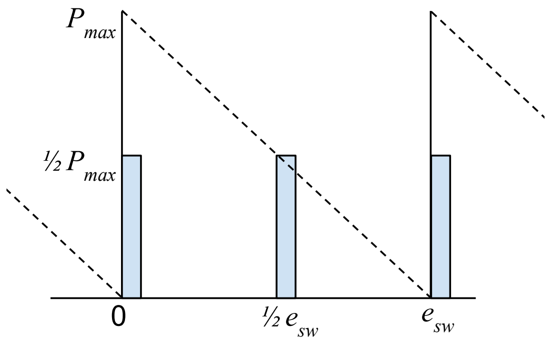
The generalisation above naturally leads to a new kind of traffic regulator abstraction. I call this abstraction a *Leaky Token Bucket (LTB)* regulator. The Leaky Token Bucket is based on the token bucket regulator shown in Figure 3.6). In a standard token bucket regulator, tokens



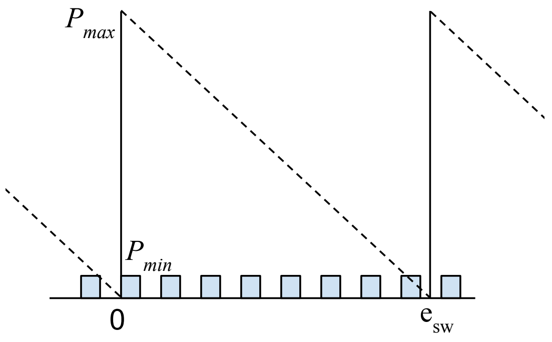
*(**a**) Allowable packet transmission sizes throughout a switch epoch.*



*(**b**) Full sized packets consuming the allowable allocation.*



*(**c**) half-sized packets consuming the allowable allocation.*



*(**d**) Minimum sized packets distributed across the switch epoch.*

Figure 3.5: *The generalised delay model for packet spacing with variable packet sized*

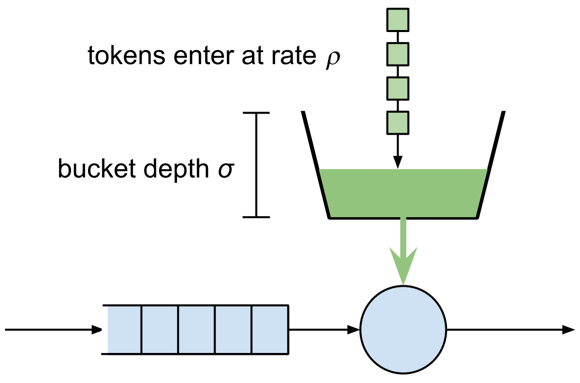


Figure 3.6: *The token bucket regulator. Tokens are added at a rate ρ to a bucket of size σ. Sending a packet of size p will consume p tokens from the bucket. σ sets the maximum burst size and ρ sets the average departure rate.*

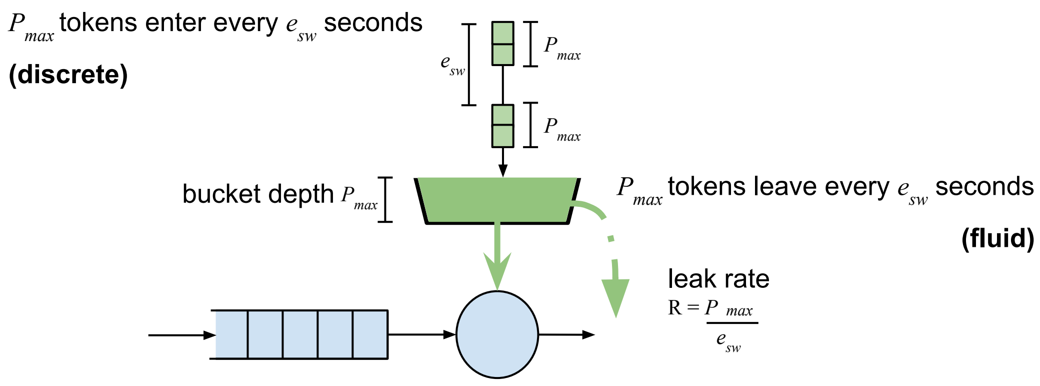

P_{max} tokens enter every e_{sw} seconds
(discrete)
e_{sw}
P_{max}
P_{max}
bucket depth P_{max}
P_{max} tokens leave every e_{sw} seconds
(fluid)
leak rate
$R = \frac{P_{max}}{e_{sw}}$

Figure 3.7: *The leaky token bucket regulator. P_{max} tokens are added once every e_{sw} seconds to a bucket with maximum capacity P_{max}. The tokens leak out of the bucket at a constant rate of $R = P_{max}/e_{sw}$, such that the bucket will be fully empty at most e_{sw} seconds after the tokens were added, or sooner if they used up to send packets.*

(bits) are added to a bucket of a size σ at a rate ρ. If the bucket fills up and overflows, excess tokens are dropped (ignored). When a packet is transmitted, it uses tokens from the bucket corresponding with the size of the packet. If there are not enough tokens in the bucket, the packet is queued until the more tokens arrive, or dropped if the queue is full.

A Leaky Token Bucket adds two more features to this model. First, tokens are added in discrete batches, one batch per time interval; P_{max} tokens are added at the start of every interval of e_{sw} seconds, maintaining an average rate of $\rho = P_{max}/e_{sw}$. Second, the bucket has a leak. Tokens leak out of the bucket at a constant (fluid-flow rate) of L, which is given by $L = P_{max}/e_{sw}$. Importantly, the average arrival rate of tokens is exactly equal to the average leak rate. If no packets arrive to consume tokens, the bucket will be empty at the end of each interval, just before new tokens arrive to replenish it. If packets do arrive, the bucket will empty sooner. The leaky token bucket regulator therefore enforces the packet spacing limits set by Equation 3.7 (shown graphically in Fig. 3.5a) as well the effective rate-limit (P_{max}, e_{sw}).

From the perspective of its output, the a leaky token bucket is a special case of a token bucket; it always emits at most the same amount of traffic as a classical token bucket regulator configured with the same parameters ($\rho = R$ and $\sigma = P_{max}$). This means that the LTB can be used in situations requiring a token bucket regulator abstraction (such as Equation 2.5) without altering the upper bound on delay. Conversely, token bucket regulators cannot be used in place of LTBs. Since a token bucket regulator "fills" rather than "leaks", cases can result where more than P_{max} bytes are issued in a single epoch. For an example of such a case see Figure 3.8. The figure shows the amount of tokens in a token bucket as function of time. As time proceeds, the bucket fills to capacity, then at t_0 a small packet of size p_{tx_1} is transmitted. Some period later, another packet of size p_{tx_2} is sent. In this case, the sum of p_{tx_1} and p_{tx_2} is greater than P_{max}. This violates the constraint that nor more than P_{max} bytes are transmitted in any period of e_{sw}.

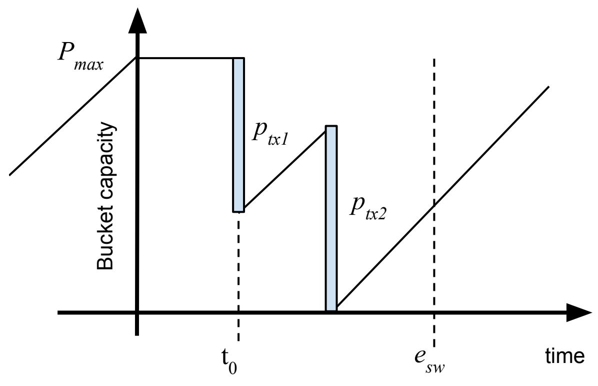


Figure 3.8: *The token bucket regulator admits a packet P_{tx_1} a time t_0. At some time later, it admits a further packet P_{tx_2}. This means that in a period e_{sw}, it admits $P_{tx_1} + P_{tx_2} > P_{max}$ bytes. This violates the constraint that no more than P_{max} bytes are admitted in any period of e_{sw}.*

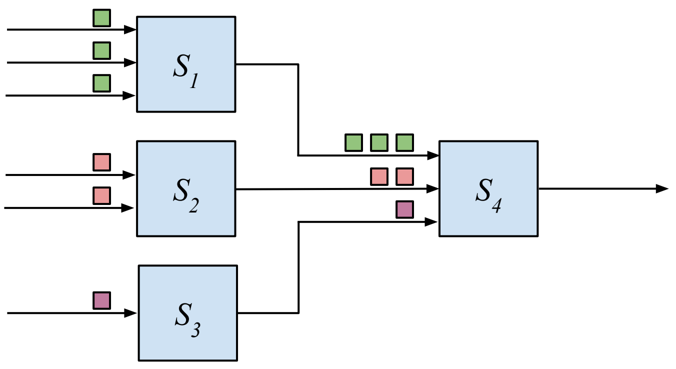


Figure 3.9: *The maximum burst size depends on the number of ports on the preceding switch.*

3.2.4 The Multi-switch Case

In Sections 3.2.1–3.2.3, I showed that a switch with m ports, can deliver bounded latency provided that all sources are Leaky Token Bucket (LTB) constrained. The LTB ensures that a single switch will have exactly enough time to service all packets sent to it, before the next servicing interval begins. This constraint ensures that a delay bound can be maintained indefinitely across that switch. In this section, I consider how to cascade multiple switches together, so that a similar delay bound can be developed for more complex network topologies.

Figure 3.9 shows an example of a simple, cascaded network topology. The figure shows 4 switches, S_1–S_4, connected together. Switches S_1–S_3 have 3, 2, and 1 inputs respectively. The outputs of switches S_1–S_3 all *fan-in* to the inputs on switch S_4.

To determine the behaviour of the network as a whole, we must first consider the output traffic patterns generated by each switch. For example, if 3 packets arrive at the same time on the inputs to S_1, then the switch will serve those packets in the minimum servicing time, given by Equation 3.5. Recall from Section 3.1 that switches are required to be throughput conserving (requirement 3). Serving all three packets in a throughput conserving way at switch S_1 will produce a *packet train* comprising 3 packets back-to-back at the switch's output. A similar result will be produced by switches S_2 and S_3.

In the worst case, the head packet from each of these trains will arrive on each input to switch S_4 at the same time. Strictly speaking, switch S_4 will operate in parallel to switches S_1–S_3. This means that, at the moment that the first packet from each packet train arrives at S_4, the switch S_4 will immediately[15] choose one packet and begin serving it. Assuming that the input and output rates of all switches are the same, this first packet will just have completed service through S_4 at the same moment that the second packets in the packet trains generated S_1 and S_2 will arrive.

Analysis of parallel processing becomes difficult if the network supports multiple input/output rates as datacenters do. To simplify the analysis, I assume that all switches at the same *depth* in the network operate in parallel, but that switches at different depths operate in series (this same assumption is implied by related work [21, 23, 121]). This, simplifying, assumption allows me to treat the queueing behaviour in sequential switches additively. It is thus straightforward to calculate an upper bound on the number of packets that switch S_4 will receive, and therefore a bound on the delay through the switch. However, since the bound does not fully take into account parallelism, this is not a tight upper bound. A lower, worst-case bound is still possible. I leave its development to future work.

In my example, the switches S_1–S_3 are at the same depth in the network, while switch S_4 is one layer deeper. S_1–S_3 are thus assumed to operate in parallel, while S_4 is assumed to only begin processing when the last packets produced by switches S_1–S_3 have arrived. Under this assumption, switch S_4 will need to process n packets from the preceding switches, where n is given by the total number of sources that fan-in to those switches. In this case n will be 6 ($= 3 + 2 + 1$)(see Fig. 3.9). The delay across the switch S_4 is thus simply given by Equation 3.5 where the number of ports on the switch (m) is replaced with n_k, the number of sources fanning-in to switch k. Formally:

$$d_k \leq n_k \times \frac{P_{max}}{r_k} + \epsilon_k \tag{3.8}$$

Where:

- d_k is the worst-case (sequential assumption) servicing delay at a switch k
- n_k is the maximum number of sources from all previous switches fanning-in to switch k
- P_{max} the maximum packet size (in bits)
- r_k is the output rate of the switch (in bits per second)
- ϵ_k is the processing delay introduced by the switch k (ins seconds)

The worst case delay through a network comprising K switches in total is calculated additively.

[15]From switch requirement 2: the switch is *work conserving* meaning that if there work to be done, it will be started immediately. §3.1

That is:

$$d_{net} \leq \sum_{k=1}^{K} \frac{n_k P_{max}}{r_k} + \epsilon_k \tag{3.9}$$

Where d_{net} is the total delay bound through the network and otherwise the same definitions as Equation 3.8 are used.

Equation 3.9 gives the delay bound where each source can send only a single packet. This is the network-wide equivalent of the single switch, one-shot case discussed in Section 3.2.1. To allow hosts to send multiple packets into the network, we need to determine the maximum rate at which packets can be issued, so that no switch queue builds indefinitely. That is, we need to find the minimum period over which each source must wait before sending a new packet. I call this period the *network epoch* (e_{net}). This is the network-wide equivalent to the switch epoch (e_{sw}), discussed in the single switch, multi-shot case in Section 3.2.2.

Recall from Section 2.2 (pg. 23) that a TCP connection will remain congestion free provided that "a new packet isn't put into the network until an old packet leaves." A similar principle applies in the network-wide case. If we ensure that hosts do not issue new packets into the network until their previous packets have left, the network delay will remain bounded indefinitely. Intuitively, the minimum network epoch value will be bounded by the switch with the maximum servicing delay. To maintain the bound, all packets must have left this switch before new packets can be added. The network epoch is thus given by:

$$e_{net} = \max\Big(\frac{n_k P_{max}}{r_k} + \epsilon_k\Big) \quad \forall \quad 0 \leq k \leq K \tag{3.10}$$

Where e_{net} is the network epoch and otherwise the same definitions as Equation 3.8 are used. The average rate-limit for each host is thus given by:

$$R_{ave} = \frac{P_{max}}{e_{net}} = \frac{P_{max}}{n_M P_{max}/r_M + \epsilon} \approx \frac{r_M}{n_M} \tag{3.11}$$

where:

- R_{ave} is the average rate-limit (in bits per second)
- P_{max} is the maximum packet size (in bits)
- e_{net} is the maximum worst-case servicing delay for all switches (in seconds).
- e_{net} occurs at the M^{th} switch along the path.
- n_M is number of hosts preceding switch M.
- r_M is the minimum output rate of switch M (in bits per second).

As in section 3.2.2 it is more correct to use the *effective rate* limit R_{eff} given by (P_{max}, e_{net}). The effective rate-limit is a better description because it ensures that there is a space of at least e_{net} seconds between each transmission of size P_{max}. Furthermore, applying the same logic from Section 3.2.3 it is sufficient for the senders to be Leaky Token Bucket (LTB) constrained.

To understand why the above intuition is true, consider Figure 3.10. The figure shows two switches connected to each other in series. The first switch (S_1) has h inputs and an output

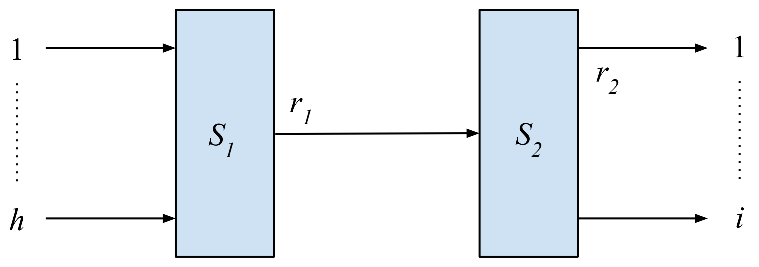


Figure 3.10: *The limiting case for cascaded switches.*

rate of r_1. Switch S_1 is connected directly to switch S_2. Without loss of generality, I assume that the only input to switch S_2 is from S_1. This means that both S_1 and S_2 have the same number of preceding source hosts (h). The second switch (S_2) has i outputs and a minimum output rate of r_2.

The worst case delay for both switches occurs when all n inputs on S_1 are directed to a single output on S_2. Therefore, we may assume that the number of outputs $i = 1$. From Equation 3.8, the worst case delay d_k across any switch S_k is proportional to n_k (the number of preceding hosts) and P_{max} (the maximum packet size), and inversely proportional to r_k (the minimum output rate), (assuming that ϵ_k is insignificant). Recall that P_{max} is a constant and since all h inputs on switch S_1 precede switch S_2, then $n_1 = n_2 = h$ (which is also a constant). Therefore, both the per-packet delay d_k at each switch (from Eq. 3.1), and the servicing delay bound D_k across each switch (from Eq. 3.8) are inversely proportional to r_k. More formally:

$$d_k \propto \frac{1}{r_k} \quad \text{and} \quad D_k \propto \frac{1}{r_k} \quad \text{for any switch } k \tag{3.12}$$

With two switches cascaded, we must consider three cases to determine if the intuitive bound holds.

1. $r_1 < r_2$ (switch S_1 is limiting): Assume that two packets P_a and P_b arrive at S_1 and that P_a is the first packet to be serviced. Assuming no transmission delay and no intrinsic switch delay, the last bit of packet P_a will arrive at switch S_2 at the same instant that the first bit of packet P_b starts to be served. It will take a time d_{1a} to service P_a at switch S_1. From Equation 3.12, if $r_1 < r_2$ then $d_{1a} > d_{2a}$ Therefore, switch S_2 will always have completed serving P_a, before P_b arrives at it. Provided that no more than n packets of size P_{max} are injected into the network in a period of D_1, it follows that no queue in the network will build indefinitely.
2. $r_1 = r_2$ (neither switch is limiting): This is just a special case of the description above. In this case, if $r_1 = r_2$ then $d_{1a} = d_{2a}$. It follows that the packet P_a will leave S_2 at the same instant that the packet P_b arrives at S_2. Therefore the bound continues to hold under the same conditions.

3. $r_1 > r_2$ (switch S_2 is limiting): We know from above that when $r_1 = r_2$ the delay bound is maintained. I therefore assume the opposite extreme: the limit as $r_1 \to \infty$. Since $d_1 \propto 1/r_1$ then, in the limit as $r_1 \to \infty$, $d_1 \to 0$. With $d_1 \to 0$, the switch S_1 will appear invisible to S_2. That is, it will appear as if all n sources are directly connected to S_2. Therefore, using the same logic as in Section 3.2.2, if no more than n packets of size P_{max} are issued into the network during a period of D_2 then no queue will build indefinitely and the bound will be maintained.

We can thus conclude that the intuition from Equation 3.10 is correct and that the injection rate into the network will be limited by the switch with the longest servicing delay.

3.2.5 Datacenter Network Reduction

For any reasonable datacenter network, a number of additional properties will also hold. These properties make it possible to simplify the calculation of the network epoch from Equation 3.10. The properties are:

- The core of the network is at least as fast as the edge of the network. This is true of all datacenter network architectures discussed in Section 2.1.3.
- All hosts are connected to the edge of the network. This is again true of all the datacenter network architectures discussed in Section 2.1.3.
- All hosts in the network participate in the latency bound and are permitted to send traffic to a single destination host. This property causes the maximum potential fan-in to be equal to the total number of hosts in the network.

Assuming that all of the above properties hold, the worst case delay will always occur at the edge switch with the slowest output speed (which is likely to be uniform across the network). The network epoch calculation from Equation 3.10 therefore reduces to:

$$e_{dcnet} = \frac{N \times P_{max}}{r_{edge}} + \epsilon_{edge} \tag{3.13}$$

Where:

- e_{dcnet} is the network epoch for a reasonable datacenter network.
- N is the total number of hosts in the network
- P_{max} the maximum packet size (in bits)
- r_{edge} is the slowest edge speed of the network (in bits per second)
- ϵ_{edge} is the processing delay introduced by the slowest edge switch

The results from Equation 3.9 can therefore be used to calculate the maximum delay that any packet will experience in a datacenter network of any topology, using a reasonable switch model, and a Leaky Token Bucket, configured according to Equation 3.13.

By replacing the network epoch e_{net} with the datacenter network epoch e_{dcnet} in Equation 3.11, we can calculate the effective rate offered to hosts in a datacenter network. That is:

$$R_{ave} = \frac{P_{max}}{e_{dcnet}} \approx \frac{r_{edge}}{N} \tag{3.14}$$

Where the terms have the same definitions as in Equation 3.13.

Practically speaking, there is a problem with Equation 3.14. From the equation, the average rate given to each host scales inversely proportionally to the number of hosts in the network. For example, in the small sample network (d) given in Section 3.4.7, there are 144 hosts and an edge speed of 10Gb/s, but each host receives an average rate of only 69Mb/s. That is, approximately 99.3% of the network bandwidth is 'wasted'. In larger, more realistic, pod-scale networks of 1000-5000 hosts, the effect is more pronounced. For example, on a 10Gb/s network with 5000 hosts, each host will receive only 2Mb/s of average throughput (i.e. 99.98% capacity lost). It should be remembered that, although the average bandwidth is low, all packets transmitted at this low rate will receive bounded latency in the network and therefore interference-free performance, without packet loss due to congestion. This is a much stronger property than low bandwidth alone. The subject of Chapters 4 and 5 are methods to mitigate and/or adapt to the issue of low bandwidth whilst retaining bounded delay.

3.3 Relationship to PGPS Delay Bound

The delay bound equation given in Equation 3.9 has a strong relationship to the PGPS delay bound [21] discussed in Section 2.5.1.2 (pg. 50) and given in Equation 2.5. The PGPS delay bound equation is replicated below for convenience:

$$D \leq \frac{\sigma}{g} + \sum_{s=1}^{K} \frac{P_{max}}{r_s} + \sum_{s=1}^{K-1} \frac{M_{max}}{g_s} \tag{3.15}$$

Where:

- All sources are governed by a token bucket abstraction with rate ρ and burst size σ.
- The maximum size of packets in the network is P_{max} and maximum size of packets in the session is M_{max}.
- The packets pass through K switches in total.
- For each switch s, there is a total rate r_s of which each session receives a rate g_s.
- g is the minimum of all g_s.
- It is assumed that $\rho \leq g$, i.e. the session is underutilised or at most perfectly utilised.

Recall from Section 2.5.1.2 that the terms in this equation can be understood intuitively. The first term σ/g is the pure GPS term. This is the fluid-flow delay that would be experienced without a packetised GPS approximation. The second term, $\sum_{s=1}^{K} \frac{P_{max}}{r_s}$, is the PGPS correction term. This term adjusts for the difference between GPS and the PGPS approximation at each switch. PGPS approximations are never worse than a full sized packet behind a GPS scheduler (see also [20]). The final term, $\sum_{s=1}^{K-1} \frac{M_{max}}{g_s}$, is the serialisation delay correction term. This term adjusts for the time it takes each packet to be serialised as is passes between each switch.

To draw a meaningful comparison between the two approaches, let us rewrite the PGPS delay bound equation with the same parameters used in the delay bound equation (3.9). Specifically I assume that:

- the network is a reasonable datacenter network with the same properties used to produce Equation 3.13.
- Each host is given a fair share of each switch that it is connected to i.e. $g_s = r_s/n_s$
- The network has n hosts in total.
- The minimum rate given to a host is one n^{th} of the rate at the edge , i.e $g = min(g_s) = r_{edge}/n$
- P_{max} is the same for all hosts and all sessions i.e $P_{max} = M_{max}$
- The token bucket regulator is configured with burst size is $\sigma = P_{max}$ and average rate $\rho = g = r_{edge}/n$.

Substituting these parameters into Equation 3.15 results in:

$$D \leq \frac{n \times P_{max}}{r_{edge}} + \sum_{s=1}^{K} \frac{P_{max}}{r_s} + \sum_{s=1}^{K-1} \frac{n_s \times P_{max}}{r_s} \tag{3.16}$$

Note that the first term is essentially the same as Equation 3.13. This is the limiting case for the network rate to be undersubscribed yet, for all hosts in the network to participate in bounded delay. It is also just a special case of the third term in the equation. Since the edge switch is the final switch in a path, and the summation in the third term fails to take this into account (i.e the sum is between switches 1 and $k-1$), it can simply be incorporated into the final summation to become:

$$D \leq \sum_{s=1}^{K} \frac{P_{max}}{r_s} + \sum_{s=1}^{K} \frac{n_s \times P_{max}}{r_s} \tag{3.17}$$

Note now that the second term in Equation 3.17 is essentially the same as Equation 3.9. Parekh *et al.* did not take into account intrinsic switch delays (σ_k), it is otherwise identical.

Comparing Equation 3.9 and Equation 3.17 the only notable difference is the first term. This term compensates for the difference between a GPS fluid-flow model scheduler, and a PGPS discrete approximation to fluid-flow model scheduler. It may seem paradoxical that Equation 3.9, which assumes a very simple scheduling model, has a lower delay lower bound than Equation 3.17, which assumes a more sophisticated PGPS scheduler. This can be accounted for by the assumed traffic models. Equation 3.17 assumes a token bucket regulator. This regulator can issue a burst of at most P_{max} at any time. The PGPS scheduler will incorporate this into the fair share model, applying a fluid-flow model and adding a worst case delay of P_{max}/r. By contrast, the Leaky Token Bucket regulator applies a fluid-flow abstraction at the source. It ensures that bursts of size P_{max} are spaced over a period of least $n \times P_{max}/r$ and that, no more than P_{max} bytes are issued over each period. This is a much more restrictive traffic regulator, which allows the switch schedulers to have a much simpler service model. Finally, note that 3.17 can be rearranged as follows:

$$D \leq \sum_{s=1}^{K} \frac{(n_s + 1) \times P_{max}}{r_s} \tag{3.18}$$

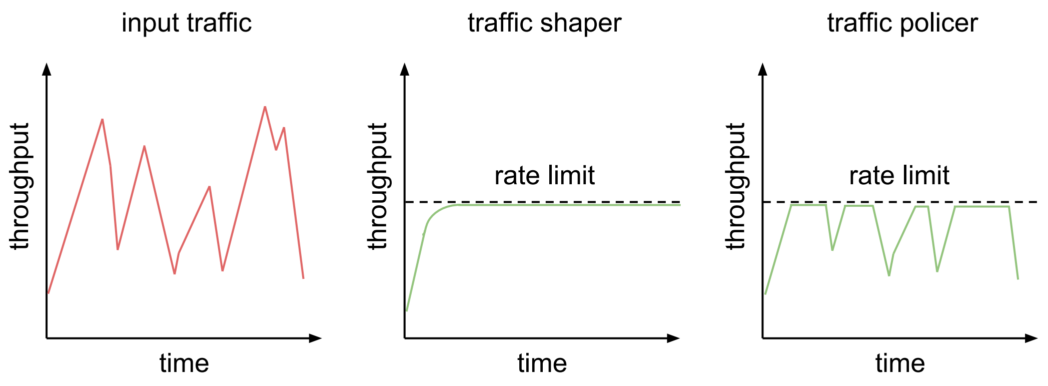


Figure 3.11: *Different approaches to traffic regulation, traffic shaping and traffic policing.*

For large n this reduces to

$$D \leq \sum_{s=1}^{K} \frac{n_s \times P_{max}}{r_s} \tag{3.19}$$

which is essentially identical to Equation 3.9. The PGPS scheduler thus provides approximately the same bounds as Equation 3.9 (sans ϵ_k) . It does so using a complex (fluid-flow) scheduler and a simple (bursty) regulator. By contrast Equation 3.9 provides its bound using a simple (bursty) scheduler and a complex (fluid-flow) regulator.

3.4 Practical Considerations

Section 3.2 describes a mathematical model for bounding latency in datacenter networks. Doing so assumes ideal hosts, networks and software as well as perfect regulators and protocols. In this section I discuss some of the practical considerations for realising the aims of Section 3.2 in realistic datacenter networks. Many of these considerations lead to configuration trade-off's. For these configuration questions there are no easy or correct solutions. Ultimately the right decision will depend on critical operations information and operational trade-offs.

3.4.1 Regulator Queue Depth

A subtle but important component of the Token Bucket / Leaky Token Bucket regulators described in Section 3.2.3 is the input queue (see Fig. 3.6/3.7). The input queue is used to hold packets that arrive while there are not enough tokens in the bucket to allow transmission. If the queue is full, packets will be dropped. An important configuration parameter is the length of this queue.

When configuring the input queue length, the user must choose between two opposing goals: (*i*) to maximise burst tolerance, the regulator should have a long queue, ideally infinitely long; and (*ii*) to minimise latency impact, the regulator should have a short queue, ideally of zero length. This distinction leads to a classification scheme for traffic regulators.

Traffic regulators can be classified as either *policing* or *shaping*[16]. A regulator with a

[16]Cisco Configuration Guide, Release 12.2. See Appendix A.4.7 for more details

queue length greater than zero is a shaping regulator. Shaping regulators alter the "shape" of the throughput curve over time. When the offered load exceeds the capacity of the bucket, the limiter will queue the excess traffic. When the offered load falls short of the output rate, the queue will drain. This buffering leads to a smoother output traffic function overall. An example of traffic shaping is shown in Figure 3.11. The input traffic throughput curve is shown on the left of the figure. The traffic has periods of high rate and periods of low rate. The shaping regulator (middle) alters the shape of the traffic, smoothing it over time to form a uniform throughput curve.

The hope with a shaping regulator is that the average offered load will not exceed the average output rate for too long. As a result, the queue will not build indefinitely. The value of "too long" depends on the length of the queue. Practically speaking, queues will be limited to some fraction of available memory. A shallow queue will use less memory, while a longer queue will be more resilient to bursts and offer a longer shaping period. The choice between these opposing trade-offs is an operator specific decision.

Although shaping regulators lead to a smoother output traffic function, the additional queueing leads to the potential for increased latency in the network. To eliminate this extra latency, the queue length could be configured to be zero. A regulator with a queue length of zero is called a *policing* regulator.

Policing regulators do not buffer packets. They apply an admissibility criterion to each packet and simply drop packets that do not match. This limits the ability of the sender to exceed a given instantaneous transmission rate. As a result, no smoothing is applied and traffic burstiness is forwarded (see Fig. 3.11 right) to the destination. This makes their output more predictable. In addition to providing better latency properties, policing rate limiters are also simpler to implement. They can be implemented in only a few lines of code, require no memory, and have a minimal impact on overall system performance (i.e. memory use, CPU performance etc). The drawback of a zero queue length (policing) regulator is that potential bandwidth is wasted when the offered load falls short of the output rate-limit. Policing regulators can also trigger poor protocol behaviour as discussed later in Section 3.4.5. Ultimately, regulator queue length configuration is an operator specific decision and, one that will depend on how, as well as where the limiter is implemented.

3.4.2 Regulator Placement

The regulators discussed in §3.4.1 can be implemented at several places between an application and the network. The choice of location will affect the amount of CPU, memory and hardware resources used, the maintainability of the system, application impact as well as the fidelity of regulation. A summary of regulator placement trade-offs discussed in this section is shown in Table 3.1.

Application The first option is to insert the traffic regulator code into applications. This can be achieved in two ways: (*i*) the application's code can be modified directly and recompiled;

Placement	Fidelity	Configurability	HW Accel.	Num Limiters	App. Mods	CPU use
Application	Low	High	✗	Per flow	Yes	Medium
LD Preload	Low	High	✗	Per socket	No	Medium
Kernel (e.g. TC)	Medium	Medium	✗	Per socket	No	Medium
NIC (e.g. [142])	High	Limited	✓	HW Limited	No	Low
Switch	High	Limited	✓	HW Limited	No	None

Table 3.1: *Comparison of regulator placement trade-offs.*

or, (*ii*) the application's system-calls can be captured and routed through additional code (for example using the Linux Dynamic Linker and configuring the `LD_PRELOAD`[17] environment variable). The first method is advantageous because it can be aware of application internal state. This means that it can distinguish between individual messages and between flows that are multiplexed over a single TCP connection. However, this method requires the application code to be modified. By contrast, dynamic injection can be used without modifications to the application source code or binary, but cannot be application aware. Both methods are advantageous because they are relatively simple to implement and debug. For applications that use byte stream protocols (e.g. TCP), these regulators can segment application messages which are larger than P_{max} into smaller messages by making multiple calls to `send()`. The segmentation will have some CPU overhead and performing many system-calls can impose significant latency penalties. Furthermore, neither method has direct access to the network hardware, which means that both methods suffer from low fidelity regulation. For example, throughput enhancing features in the kernel or network adapter may attempt to batch messages/packets together (e.g. Nagle's Algorithm[113]) which can disrupt the inter-packet timing applied by these kinds of regulators.

Kernel An alternative arrangement is to place the regulator code inside the kernel, for example using the Linux Traffic Control (TC)[143] interface. This avoids the system-call overheads associated with the methods described above. TC modules are placed between the network stack and the network adapter driver. The module receives fully formed frames from the TCP/IP/Ethernet stack where protocol/kernel layer batching operations have already been performed. This means that TC models have high fidelity control over what gets sent to the network adapter. However, this also causes difficulties if the module receives frames larger than P_{max}. Network acceleration mechanics such as TCP Segment Offload (TSO), Large Segment Offload (LSO) and Generic Segment Offload (GSO) operate by sending large size segments down to the NIC where they are re-segmented into Maximum Transfer Unit (MTU) sized frames. If a large segment is received by the regulator it must either: drop the entire segment, or re-segment it into smaller segments. It can be computationally expensive to re-segment large frames in software. Finally, kernel-based regulators do not have direct control over the network adapter hardware. They provide high fidelity control over what gets sent to the adapter but over not what gets put into the network.

[17]See Appendix A.19.2.

Network Adapter Network adapters often also offer rate-limiting options. For example, the Intel 82599 chipset (as used in x520 and x540 adapters) supports hardware based transmit rate limiters[18]. There are several problems with this implementation. The chipset supports only one rate-limiter per transmit queue and only 128 transmit queues. This limits the availability of hardware based rate-limiting to at most 128 concurrent configurations. Furthermore, the Linux operating system does not expose a mechanism for configuring and using rate-limiter options on network adapters. Finally, these rate-limiters have limited configuration options. The Intel X520 chipset has an output rate configuration value only. There is no direct support for modifying queue lengths. Furthermore, no details are given about rate-limiting scheduler implementation. It is unlikely that Leaky Token Bucket regulator functionality will be directly supported and it is unlikely that it could be approximated without more NIC configurability.

In response to configurability issues, limited queue counts and other shortcomings, Radhakrishnan *et al.* proposed *SENIC* [142], a purpose built network adapter and software stack that could support "10s of thousands of rate limiters". Similarly, my previous work proposed *Eden*, a generic mechanism to enable custom end-host network functions such as rate-limiting/traffic regulation [107] to be offloaded into network adapters. The key feature of both of these systems was the ability for applications to tag flows which would then be handled specially in hardware. Eden specifically provides a Domain Specific Language (DSL) for describing these messages and for describing functions (e.g. regulators) to be implemented in the hardware. Unfortunately, both projects require custom hardware which is not currently found in datacenters.

Software A final alternative is to use rate-limiting functions offered by switches. Commodity switches from Cisco and Arista[19] offer egress shaping and policing configurations. These features are typically configured around a specific port or DSCP field value. This limits the number of effective rate-limiters to between 1 and 8 configurations. Once again, the rate-limiters implemented in commodity switches have limited configuration options and limited details about their internal scheduler operation. It is therefore unknowable if Leaky Token Bucket functionality could be implemented or approximated using these features.

On balance, I concluded that a kernel-based policing regulator offers the highest fidelity and configurability with the lowest runtime and implementation cost. Unless otherwise noted, I will employ a kernel-based policing regulator for the remainder of this dissertation Operators with more significant hardware or software control may balance this choice differently.

3.4.3 Packet Size Considerations

In Equations 3.9 and 3.10, the end-to-end delay and rate limit are (approximately) proportional to the value of P_{max}. To minimise the delay bound and maximise throughput, P_{max} should therefore be as small as possible. The Ethernet standard limits frames to a minimum size of

[18]Intel 82599 datasheet. See Appendix A.16.6.

[19]Cisco and Arista configuration guides. See Appendix A.4.6 and A.2.4.

64B [16]. This imposes a constraint on the allowable values for the minimum delay. For example, on a 32 port, 40Gb/s switch with a service time of $\epsilon = 0.45\mu\text{s}$ (e.g. Arista 7060X[20]), assuming all ports are participating, the minimum delay guarantee given by Eq. 3.9 with 64B frames is:

$$d \leq 32 \times \frac{64\text{B}}{40\text{Gb/s}} + 0.45\mu\text{s} \leq 0.86\mu\text{s} \tag{3.20}$$

This delay cannot be further reduced by decreasing the frame size. The only other option to further reduce the guaranteed minimum delay is to reduce the number of participating hosts/ports/connections (n). The operator is therefore left in the position of deciding between reducing the bound on network delay and reducing the number of hosts that can participate. In the simplest deployments, all hosts in the network participate delay bound (see Chap. 4). This places a lower bound minimum delay bound that can be guaranteed through the network. Chapter 5 discusses some ways in which this problem may be alleviated.

The placement of the regulator (See §3.4.2) also has an effect on the maximum reasonable value of P_{max}. In standard Ethernet networks, the largest frame allowable is 1518B [16]. In some extended Ethernet devices this can be configured using *jumboframes* up to 9000B [144]. As discussed in 3.2.3, sending two packets back-to-back has the same effect on the network as sending one large packet. Therefore, it would seem illogical to set P_{max} larger than the maximum frame size that Ethernet supports. Doing so would require multiple packets to fulfil and would have a negative effect on application messages and the delay bound. However, this is not true if the regulator is placed in software. Network acceleration mechanics such as TCP TCP Segment Offload (TSO), Large Segment Offload (LSO) and Generic Segment Offload (GSO) operate by sending large messages from the host software to the network card where they are re-segmented into maximum sized frames. In-kernel/application-based regulators may receive these large frames (typically 64kB) and have to choose between dropping them (which wastes bandwidth), or re-segmenting them (which is computationally expensive and wastes bandwidth). Therefore, the maximum reasonable size for P_{max} may indeed be larger than maximum Ethernet packet, and is instead bounded by the segmentation offload size.

3.4.4 Regulator Implementation

A regulator is required if we wish to bound latency across the network. The regulator must at least ensure that its output is (P_{max}, e_{net}) constrained. Pseudocode for a minimal implementation of a discrete (P_{max}, e_{net}) enforcing, policing regulator is given in Listing 3.1. The code takes a new packet as an argument. It is assumed that the size of the packet is known. The code first checks to see if a sufficient time has elapsed since the last transmission (line 10). If a time of at least e_{net} seconds has passed, it resets the timer (line 11) and adds a fresh allocation of bytes to the token bucket (line 12). On line 16, the code checks to see if there are enough tokens remaining in the bucket to send a packet. If true, the packet is transmitted (line 18). If not, the packet is dropped (line 15). Line 17 is important: this line implements the (P_{max}, e_{net}) constraint.

[20] Arista 7060X marketing material. See Appendix A.2.5.

```
1
2  const long epoch_cycles = to_cycles(e_net_secs); //Assumes CPU speed
3  long timestart_cycles   = 0;
4  long bucket_tokens      = 0;
5
6  int limiter_packet_event(buffer packet) {
7    long now_cycles = asm("rdtsc"); // read cycle counter
8
9    //Reset timer if sufficient time has passed
10   if (now_cycles > timestart_cycles + epoch_cycles) { //Enough time passed?
11         timestart_cycles = now_cycles; //Reset timer
12         bucket_tokens    = P_max_bytes; //Add fresh tokens
13       }
14   }
15
16   if(packet.size <= bucket_tokens ){ //Enough tokens to send?
17         bucket_tokens = 0; //Remove all tokens
18         return sendToHWQueue(packet); //Send the packet to the hardware
19   }
20
21   return DROP;  //Packet requires too many tokens drop it
22 }
```

Listing 3.1: *Pseudocode for a fast policing regulator.*

When a packet is sent, the bucket is completely drained (`bucket_tokens = 0`), thus enforcing the constraint.

Since the code applies to every packet, it needs to be fast. Note that the current “time” measured on line 6 is a call to read the system timestamp cycle counter (RDTSC) rather than to obtain the wall-clock time. Furthermore, the delay value is pre-computed into cycles on line 2. These operations allow the code to perform relatively cheap cycles based operations rather than expensive wall-clock time keeping operations. According to the Intel 64/IA32 Software Developers Manual [56]:

> On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers). [. . .] TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource. . .

The cycle space optimisation assumes that the host machine has invariant Time Stamp Counter (TSC). An invariant TSC will run at a constant rate in all power states and across all cores.

```

const long epoch_cycles = to_cycles(epoch_secs); //Assumes CPU speed
long timestart_cycles   = 0;
long bucket_tokens      = 0;

int limiter_packet_event(buffer packet) {
  long now_cycles = asm("rdtsc"); // read cycle counter

  //Reset timer if sufficient time has passed
  if (now_cycles > timestart_cycles + epoch_cycles) { //Enough time passed?
      timestart_cycles = now_cycles;   //Reset timer
      bucket_tokens    = P_max_bytes; //Reset token bucket
    }
  }

  const long leaked_tokens = P_max_bytes * (time_now_cycles - timestart_cycles)
    / (epoch_cycles); //How many tokens leaked out?
  if(packet.size <= bucket_tokens - leaked_tokens ){ //Enough tokens?
      bucket_tokens -= packet.size;
      return sendToHWQueue(packet); //Send the packet to the hardware
  }

  return DROP;  //Packet requires too many tokens drop it
}
```

Listing 3.2: *Pseudocode for a fast Leaky Token Bucket (LTB) based policing regulator.*

Recent commodity x86 CPUs that are run in datacenter operations (See 2.1.2) support this feature. However, this optimisation assumes that a calibration routine (such as that described in the Intel Developer's Manual [145]) has been run to calculate or infer the CPU clock speed and to make these conversions possible.

The problem with the above regulator is that it wastes bandwidth. Every time a packet is sent the bucket is reset to zero. This is resolved by using a Leaky Token Bucket (LTB) regulator (§3.2). An LTB ensures that at most P_{max} bits are issued into the network with a minimum spacing of time e_{net} seconds. However, it can also issue traffic into the network in intervals smaller than e_{net} so long as its bucket is "leaky". That is, it may issue b bits into the network where b is given by the following equation (see §3.2.3 for more details):

$$b \leq \frac{e_{net} - v}{e_{net}} \times P_{max} \quad \forall \quad 0 \leq v \leq e_{net} \tag{3.21}$$

Listing 3.2 extends Listing 3.1 to a full Leaky Token Bucket policing regulator. The listing very slightly modifies Listing 3.1 with an extra calculation on line 16. This calculation is an implementation of Equation 3.21. It determines the number of bytes that would have leaked out of the bucket during the epoch interval. The modification makes this code similar to the operations performed by PGPS schedulers (see §2.5.1). In essence, the modified code runs a fluid-flow simulation of finishing time of the leaking bucket "on the side". To remain fast, the calculation is performed using integer mathematics rather than floating point computation. This many introduce small rounding errors of one byte which I consider to be insignificant. Line 18 is modified to compare the packet size to the bucket contents minus the amount that would have leaked out. If there are sufficient tokens remaining, the packet is transmitted and the tokens are removed from the bucket. If not, the packet is dropped.

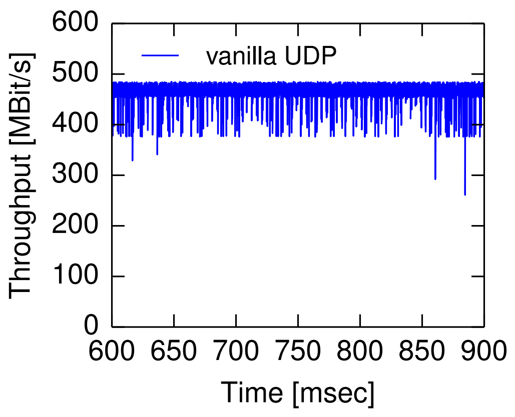
On my test machines, I have found no measurable effect of this regulator on CPU utilisation or throughput. On average it imposes a cost of approximately 35 cycles per packet ($\sigma = 18.6$; 99th% = 69 cycles) on the Linux kernel critical path of ≈8,000 cycles. This amounts to a less than 0.5% overhead. It is important to note that this regulator is a policer without resegmentation. No buffering is performed and all packets that are bigger than P_{max} will be dropped. Despite these shortcomings I found this solution to work adequately with realistic applications (see Chapter 4).

3.4.5 Policing Limiter Protocol Effects

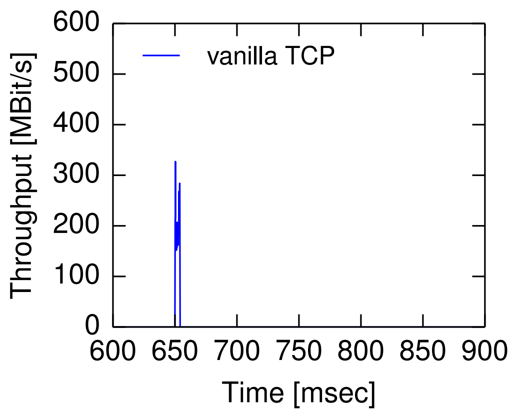
The algorithm presented in Section 3.4.4 is a policing algorithm. Packets are admitted or rejected from the network depending on the time since the last transmission. This leads to the desirable property of predictable network behaviour. However, it can also have a detrimental effect on the end-to-end performance of TCP. Since TCP is the most common protocol in datacenter environments (See §2.2 and §2.4.1–§2.4.4), it is important to understand and mitigate these effects.

Figure 3.12a shows the throughput performance of UDP traffic generated by `iperf` when subjected to a 500Mb/s policing rate limiter. As shown in the figure, the UDP traffic is stable at between 400Mb/s and 490Mb/s, never exceeding the 500Mb/s limit. Figure 3.12b shows the throughput performance of `iperf` generating TCP CUBIC when subjected to the same 500Mb/s policing rate limiter.

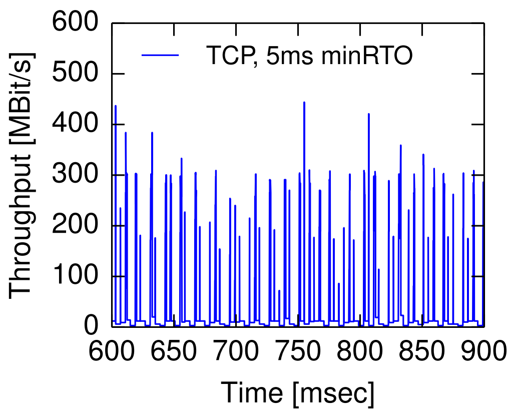
As seen in Figure 3.12b, the TCP throughput results are quite different. Over a 300 millisecond window only 3 packets are sent. This happens because the policer drops 3 out of 4 packets. As a result, TCP fails to receive 3 consecutive acknowledgements. The protocol cannot know if the acknowledgements have been lost, or if they are being delayed by severe congestion. If they are delayed by congestion, injecting more packets into the network may exacerbate the congestion. But, if the packets genuinely have been lost, the lost packets need to be retransmitted. In response, TCP waits for a 'safe' period so that any outstanding (delayed) acknowledgements can be received, but not too long so that bandwidth is wasted because the retransmissions have not been sent. The value of 'safe' is determined dynamically [66] with a minimum value which



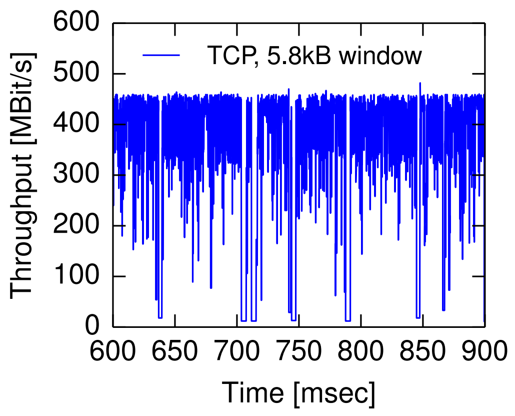
(a) Vanilla UDP behaviour



(b) Vanilla TCP behaviour



(c) TCP with reduce retransmit timout



(d) TCP with socket buffer windowing.

***Figure 3.12:** Protocol effects of applying policing regulaltors*

is called the *minRTO*. On the 3.4.55 Linux Kernel the default minRTO value is 200ms[21]. The result of waiting for 200ms is complete TCP throughput collapse.

Two simple mitigations can be applied to resolve the throughput collapse shown in Figure 3.12b. First, the default minRTO value of 200ms is overly cautious for datacenter networks where round-trip-times are measured in microseconds. Figure 3.12c shows the effect of reducing the minRTO value to 5ms. The figure shows a marked improvement in TCP's behaviour, however, it is apparent that timeouts still dominate the performance.

An alternative mitigation is to limit the TCP transmit socket buffer size given to applications. The transmit socket buffer is used by applications to transfer data from the application into the kernel, but before the TCP stack is invoked. Limiting the buffer size limits the ability of applications to inject traffic into the network stack, effectively applying a simple rate-limiter to the application. The effect of resizing the socket buffer to just under 4 frames is shown in Figure 3.12d. As shown, this mitigation leads TCP to more stable throughput, approaching an

[21]See Linux Kernel 3.4.55 source code, include/net/tcp.h, line 135, e.g. `http://lxr.free-electrons.com/source/include/net/tcp.h#L135`

average transfer rate of 500Mb/s and suffering from far fewer minRTO timeouts.

Unfortunately, limiting the size of the TCP socket buffer can be problematic for some applications because it results in a kind of back-pressure which may not be correctly handled by the application code. When the buffer is full, the kernel will signal the application by returning an `ENOBUFS`[22] error code. If the application is not expecting this return value it may a) crash or b) fail into an inconsistent state. Furthermore, the Linux user manual explicitly states that `ENOBUFS` errors "normally, do not occur in Linux" which means that few programmers will think to handle it explicitly.

The effects of policing limiters are most pronounced for aggressive rate limits of a few hundred megabits per second. At less aggressive limits, TCP successfully transitions into congestion avoidance mode and the limiter operates as expected. Nevertheless, operators wishing to use TCP with aggressive limits are left in a difficult position. They must decide which is preferable:

1. Good TCP performance with unpredictable latency arising from a shaping rate limiter.
2. Poor TCP performance with predictable per packet latencies from a strict policing rate limiter.
3. Improved TCP performance using a shorter RTO timeout, but with increased kernel fragility due reconfiguration (and recompilation) of kernel defaults.
4. Improved TCP performance using socket-buffer windowing, but with increased application fragility due to `ENOBUFS` error handling errors.

The choice between these options will depend on operator specific constraints. In my experiments making these choices has not been necessary. PTPd uses UDP traffic at low rate (1 packet per second), memcached uses TCP at medium rate (about 5Gb/s) and Hadoop uses TCP up to line-rate (about 10Gb/s) (see Chapter 4 for more details).

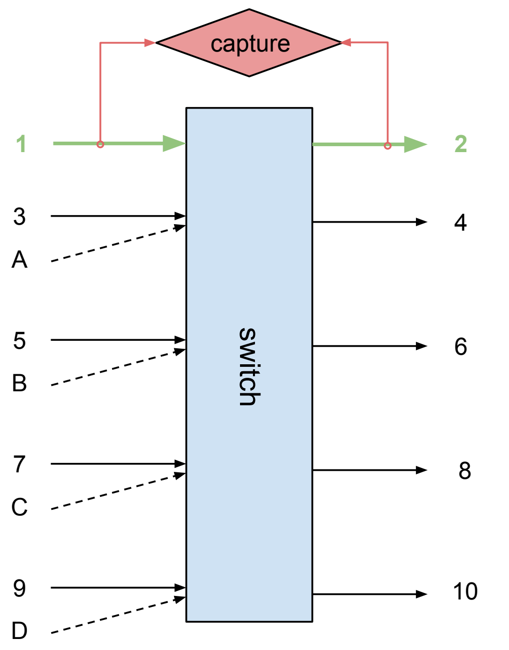
3.4.6 Switch Behaviour

In Section 3.1 I assumed that switch schedulers are: (*i*) parallel and port-independent, (*ii*) work-conserving and, (*iii*) throughput conserving. As a rule, switch silicon manufacturers keep the precise implementation details of their schedulers secret. To understand the behaviour of representative datacenter switches and verify that they operate as required by the model, I ran the following experiments on an Arista 7050 switch, which uses a 10Gb/s Broadcom Trident II+ switching silicon[23] internally.

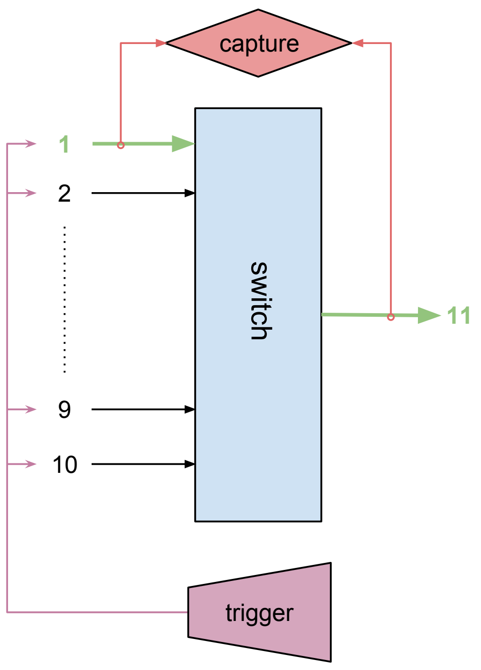
3.4.6.1 Port-independent, parallel matching

I first verified that the switch scheduler is port-independent and parallel. To do so, I arranged 14 hosts connected to each other via the switch as shown in Figure 3.13a. I placed optical taps

[22]Linux man page – sendto(2). See Appendix A.19.3 for more details
[23]Broadcom BCM56580 "Trident II+" marketing material. See Appendix A.3.3



(a) Port independent, parallel matching test arrangement



(b) Work conserving and throughput conserving test arrangements

Figure 3.13: *Switch model validation experimental configurations.*

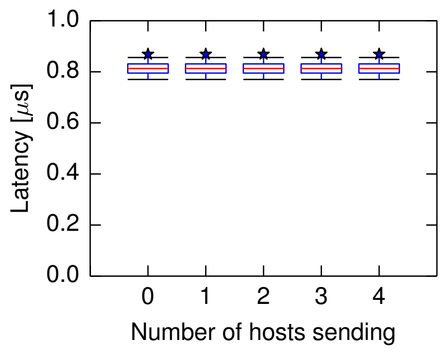
between the output of host 1 and the input to the switch, and, between the output of switch and the input to host 2. The taps forward packets to a hardware time-stamping network capture device (6ns resolution[24]) which is used to calculate the latency experienced across the switch. The first pair of hosts (1 and 2) exchange `ping` messages (ICMP echo requests/replies) in “flood ping” [25] mode (around 13Mb/s). These small, low rate messages provide a way to sample the port-to-port switch latency, including the crossbar scheduler latency. Hosts 3 and 4, 5 and 6, (7 and 8, etc) are arranged as `iperf` clients and servers respectively. As the high rate `iperf` traffic crosses the switch fabric, the switch scheduler will need to schedule this traffic as well as the `iping` traffic. A parallel scheduler should exhibit no change in the latency of `iping` traffic regardless of number of `iperf` pairs. The last group of hosts (A-D) act as additional clients for each `iperf` server. These hosts add further strain to the scheduler by adding contention for output ports. In this case, multiple input ports will have outstanding packets in output queues destined for the same output port. This is done to ensure that the switch scheduler is not exercising a fast path in the one-to-one case.

The results of this experiment are shown in Figures 3.14a and 3.14b[26]. Each figure contains 5 box and whisker plots. The box and whisker plots show the observed latency distribution across the switch as a function of the number of `iperf` clients. The boxes are at

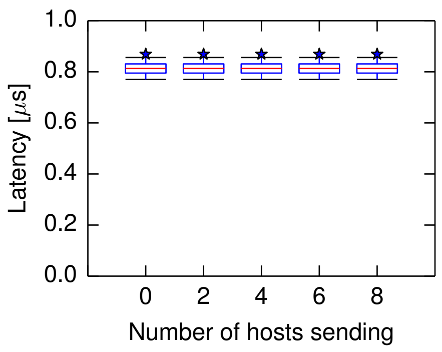
[24]Exablaze X10 datasheet. See Appendix A.9.2.

[25]In flood ping mode, the `iping` sender emits ICMP echo requests as quickly as it can.

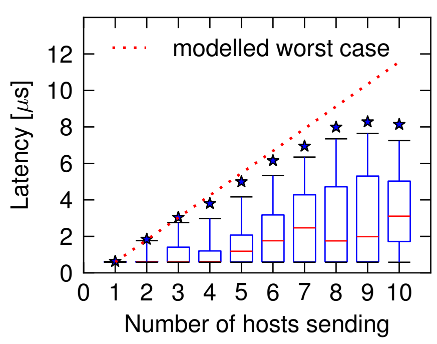
[26]Each box and whisker plot is sourced from approx. 500,000 measurement data points. Although the results appear identical, I have verified that each source is uniquely derived from each experiment.



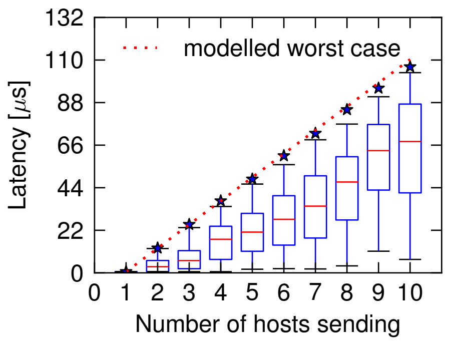
***(a)** Scheduler is parallel and port-independent (single source, single destination)*



***(b)** Scheduler is parallel and port-independent (double source, single destination)*



***(c)** Scheduler is work conserving.*



***(d)** Scheduler is throughput conserving.*

***Figure 3.14:** Experimental results of switch model verification tests.*

the 25th and 75th percentiles, whiskers are 1st and 99th percentiles, and the stars indicate the maximum latency observed over approximately 500,000 samples. Figure 3.14a shows the results for the uncontended cases while Figure 3.14b shows the results for contended cases. Both figures include the idle case where there are no `iperf` clients/servers in operation.

The figures show that the `ping` measurements are unaffected by the traffic on other ports. There is no measurable difference between the latency of `ping` requests with or without the `iperf` traffic. I therefore conclude experimentally that at least one popular datacenter switch silicon implements the required parallel scheduling functionality. As discussed in Section 3.1, the Cisco Nexus 55xx switches implement an iSLIP like algorithm that will also meet these requirements.

3.4.6.2 Work Conservation

The next experiment tests if the switch scheduler is work conserving. This is required by Section 3.2.1 and is the key to Equation 3.4 and all that follow. To do so, I arranged ten hosts as per Figure 3.13b. Between one source host and the switch, and between the switch and the destination host I paced optical taps (similar to that in the experiment above). Once again the taps feed into a high resolution network capture device to measure and calculate the latency across the switch. In this experiment, the measurement device offers a measurement granularity

7.5ns[27].

Each host generates packets using the Netmap [105] kernel bypass system. Using the Netmap system allows me to accurately control the number of packets that are sent to the network card for transmission. Each host generates full-sized (1518B) packets in response to a broadcast trigger message being received over a separate control network. The hosts send a configurable number of packets in a train to the destination. This creates a synchronised burst of traffic arriving at the switch which is the same as the modelled case in Section 3.2.1.

For the first experiment, each host sends a single packet to the destination in response to the trigger. The triggers are spaced at 1ms to ensure a quiescent system. Figure 3.14c shows the measured latency results across the switch. The figure comprises 10 box and whisker plots similar to those described in Section 3.4.6.1 above. The plots show the observed latency distribution as a function of the number of hosts (n) and the modelled worst case using Equation 3.1. The modelled worst case uses the parameters $P = $ 1518B, $R = $ 10Gb/s, $\epsilon_k = $ 588ns[28] and $n = [1, 2, 3, ..., 10]$. In the figure, the observed worst-case packet switching latency increases nearly linearly with the number of sending hosts. It tracks the predicted worst case closely up to six senders and then begins to fall short. This is because the probability of all senders being exactly synchronised (i.e. all 10 packets arriving within a $\approx 1\mu$s window) decays exponentially with the number of hosts. Interestingly this observation suggests that in a larger, asynchronous systems, the observed latencies will generally be lower than those predicted by the worst case in Equation 3.1, but that the worst case bound does hold as expected.

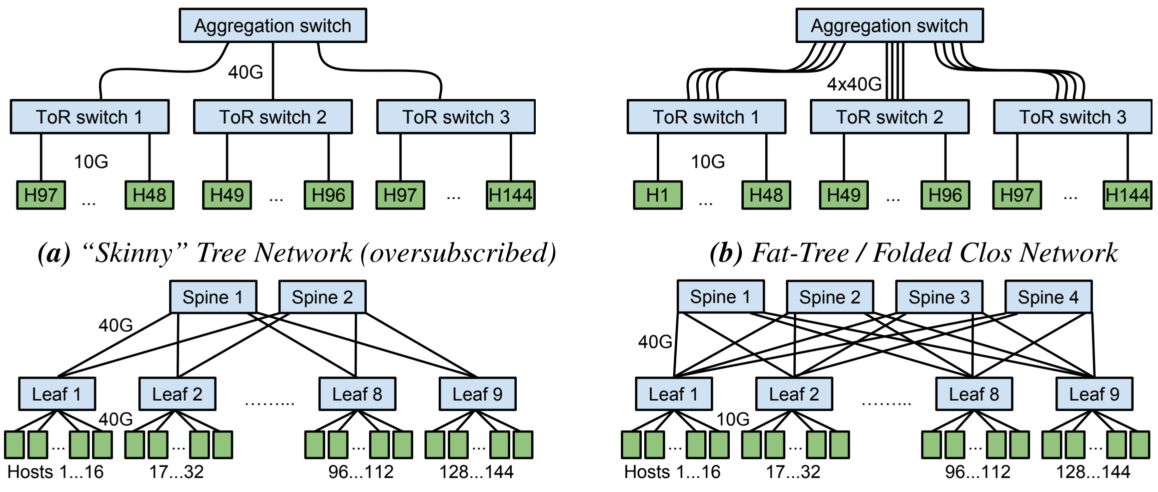
3.4.6.3 Throughput Conservation

Section 3.2.2 requires that the switch scheduler be throughput conserving. In it I argue that variable sized packets can be used with a Leaky Token Bucket regulator if back-to-back packets can be treated as if they were large packets of the same size. Section 3.2.4 also relies on this requirement to construct mutli-switch delay bounds. To analyse the behaviour of the switch when subject to “trains” of back-to-back packets, I reconfigured the experiment above. In this second experiment, each host generates 10 back-to-back packets destined for a single destination host. Once again, packets are generated by all hosts in response to a trigger message and triggers are spaced by 1ms. The results are shown in Figure 3.14d.

Again the same box and whisker plots described in Section 3.4.6.1 are used. In this case the Eq. 3.8 is used to model the worst case behaviour. The modelled worst case uses the parameters $P_{max} = $ 1518B, $R = $ 10Gb/s, $\epsilon_k = $ 588ns and $n = [10, 20, 30, .., 100]$. The plot shows the latency across the switch as a function of the number of sending hosts. Once again the measured behaviour in Figure 3.14d tracks closely with the modelled behaviour. Since the packet trains are used instead of single packets, there is a statistically higher probability of achieving the near worst conflicts which is reflected in the graph. In this graph there is a closer alignment

[27]Endace 9.2X2 datasheet. See Appendix A.8.1.

[28]Since the size of this experiment is small, ϵ_k cannot be ignored.



(a) *"Skinny" Tree Network (oversubscribed)*

(b) *Fat-Tree / Folded Clos Network*

(c) *Resilient / Path Diverse (oversubscribed)*

(d) *Non-Blocking Clos*

Figure 3.15: *Datacenter network topology examples.*

between the predicted worst case latency and the measured value. Overall, I can conclude from the measured results of a representative datacenter switching silicon, that my switch scheduling model presented in Section 2.2 is reasonable.

3.4.7 Network Topology

As described in Section 2.1.3, datacenter networks may be constructed from a variety of different network topologies. It is an operator specific decision which topology to use and which arising trade-offs are/are-not acceptable. Some example topologies are given in Figure 3.15. The figure depicts 4 network topologies, each servicing 144 hosts: (a) a simple tree network, (b) a fat-tree/folded Clos network, (c) a Facebook inspired "fabric" style network[29] and, (d) a 3-stage Clos network. Each of the different network designs have implicit trade-offs between the amount and complexity of the physical wiring, the amount of bandwidth/over subscription that each switch/host receives and the cost and complexity of the switches used. For example, the networks depicted in (b) and (d) provide full bisection bandwidth whereas the networks depicted in (a) and (c) do not. Furthermore, the networks depicted in (a) and (b) provide only 1 route between any pair of servers whereas the networks depicted in (c) and (d) provide many.

The choice of topology also has implications for how the equations derived in §3.2 can be applied and delay bounds that they can deliver. In each case, Equation 3.9 and Equation 3.10 must be used to calculate the senders' delay bound and sending rate respectively. For example, using the skinny tree network from Figure 3.15a, assuming that all hosts participate (e.g. $n = 144$), and that the switch latency (ϵ_K) is negligible, the delay bound is given by:

$$D \leq \sum_{k=1}^{K} \frac{n_k P_{max}}{r_k} = \frac{48P_{max}}{40\text{Gb/s}} + \frac{96P_{max}}{40\text{Gb/s}} + \frac{143P_{max}}{10\text{Gb/s}} = \frac{179P_{max}}{10\text{Gb/s}} \qquad (3.22)$$

[29]See Appendix A.11.3.

Topol.	ToR up (ns/b)	Core (ns/b)	ToR down (ns/b)	Total (ns/b)	Rate (Mb/s)
a	4.8	9.6	14.3	**28.7**	69.9
b	0.1	0.8	14.3	**15.2**	69.9
c	0.4	3.2	3.6	**6.8**	279.7
d	0.4	3.2	14.3	**17.9**	69.9
d@40G edge	0.4	3.2	3.6	**7.2**	297.7
b@40G edge	0.1	0.8	3.6	**4.5**	297.7

Table 3.2: *Delay bounds for each switch, total delay bound and average rate limit for each host using the topologies given in Figure 3.15, expressed as nanoseconds per bit.*

The first term ($48P_{max}/10$) accounts for the maximum fan-in at the ToR switches (48 ways), the second term ($96P_{max}/40$) accounts for the maximum fan-in (96 ways) at the core switches and the final term ($143P_{max}/10$) accounts for the maximum fan-in at the final ToR switch ($n-1$ ways = 143). The coefficient of 143 instead of 144 in the final term accounts for the fact that the destination host does not send to itself over the network. This term also gives the average and effective rate-limits which are 46.5Mb/s and $(P_{max}, 215P_{max}/10Gbps)$ respectively. Note that the average rate-limit does not depend on P_{max} which is precisely why the effective rate-limit is required.

To further explore the relationship between topology and delay bound see Table 3.2. This table shows the delays accumulated at each switch and the total delay bound for the each topology in Figure 3.15. Each delay value is given in nanoseconds per bit (ns/b), making them independent of P_{max}. To obtain the actual delay bound a suitable P_{max} value must be chosen. For example, the total delay bound for topology b with $P_{max} = 128B = 1024b$ is given by $15.2 \times 1024 = 15565ns \approx 15.5\mu$s

Interestingly, Table 3.2 shows that the largest component of overall network latency is always the final top-of-rack switch (ToR down) on the path. In topology (b), over 94% of the total delay is added by the final hop. There are two reasons for this. First, all hosts in the network need to pass through the final switch to be able to communicate with the destination. Thus the longest (potential) queues will be at this maximum fan-in point. Second, the slowest links in the network are typically at the edge of the network. This compounds the above delays. The result is that, not only are the queues potentially longest, but the service rate is also the slowest leading to the longest potential delays.

Counterintuitively, Table 3.2 also shows that oversubscription of bandwidth is not always negative. Topologies b and d have full bisection bandwidth while topology c is 8:1 over subscribed. But, topology c has the fastest edge speed at 40Gb/s. This makes topology c around $2.5\times$ faster than topology b/c. By setting the edge speed in topology d to 40Gb/s, the result improves it to parity with topology c. By setting the edge speed in topology b to 40Gb/s, the best result (4.5ns/b) is obtained. The important consideration here is that full bisection bandwidth is not sufficient. In general, having high path diversity with slow link speeds harms the latency bound unless all paths can be used in a non-blocking way. If the operator can use ECMP [111] or similar to bind flows to a given path in topology so that they do not interfere with each other, then the aggregate service rate would increase and the delay bound could be reduced. Once again, network topology and traffic engineering considerations are an operator specific decision. Such decisions involve resource/cost trade-offs (e.g.the number of 10Gb/s and/or 40Gb/s switches), physical complexity and failover trade-offs (e.g. cabling and failure domain concerns) as well as operational and latency delay bound concerns. Each an operator will wish to weigh these considerations for their own environments.

3.4.8 Host Latency Sources

The analysis and discussion in Chapter 3 has so far avoided host latency/interference issues. Hosts may contribute to end-to-end latency variance through a variety of mechanisms [146]; scheduling and interrupts from the operating system, the BIOS and even the hardware can preempt applications causing latency spikes. Likewise batching and queueing operations in the kernel and network card often exchange throughput performance for latency variance. Other sources of variation include cache and TLB misses as well as page table lookups. Finally resource sharing between applications and/or virtual machines can have adverse effects when applications are mixed.

In the commercial sector, High Frequency Traders (HFTs) and others whose applications are latency sensitive employ specifically designed low latency network cards (e.g. Solarflare Flareon Adapters[30], Exablaze ExaNIC X10[31]) and kernel bypass/userspace network stacks (e.g. Solarflare "Open"-Onload[32], Exablaze ExaSock[33], etc). In combination with scheduler priorities, memory pinning, CPU pinning, and interrupt pinning, these provide lower overall latency as well as better bounds on host based latency sources. These techniques are applicable to commodity operating systems and software. In my preceding and following experiments I used some or all of the above techniques where necessary. Furthermore, I ensure that applications are run on idle machines without any application sharing. Operators may be more or less constrained in the tools and techniques that they can apply. Once again, the specifics of the operational environment, the performance requirements, and the resources available will determine the acceptable trade-offs in this space.

While host latencies are, without a doubt, an important component of end-to-end delay bounds in datacenter networks, mitigating and controlling variability in them is a subject area of its own right. There is much previous and on-going work ([84, 146–151]) in the area. I therefore limit the scope of this dissertation to in-network latencies only and defer to these and other excellent works in the field for those interested in a more complete picture.

[30]According to Solarflare Flareon/"Open" Onload marketing material. See Appendix A.24.1.
[31]Exablaze X10 marketing material. See Appendix A.9.2.
[32]According to Solarflare Flareon/"Open"-Onload marketing material. See Appendix A.24.1.
[33]David, Exablaze Blog (14 July 2014). See Appendix A.9.3.

3.5 Conclusions

In this chapter I developed a model of a datacenter switch (§3.1) and showed that, in combination with a Leaky Token Bucket (LTB) (§3.2.3),it is possible to bound delay in datacenter networks (§3.2.4). Furthermore I showed that the above bounds are approximately the same as those offered by PGPS (§3.3). Following this discussion I went on to examine many of the practical issues associated with implementing these bounds in realistic datacenter settings (§3.4). In particular, I showed experimentally that a widely deployed switch silicon implements the minimum required properties of my model (§3.4.6). Finally, I showed that sensible values can be determined for many implementation specific concerns and trade-offs necessary to realise the preceding scheduler and regulator in a realistic datacenter network environment (§3.4.1–3.4.8).

3.6 Chapter Summary

- To control interference in a network it is necessary to bound delay. PGPS (see §2.5.1) offers bounded delay. However, it starts from a fluid-flow model governed by a token bucket regulator and works toward a packetised approximation with bounded delay.
- I pursue an alternative approach: I begin from a simple but realistic model of a datacenter switch (§3.1), derive a way to bound delay in it, and then derive a fluid-flow based regulator to maintain the delay (§3.2).
  - Switches are assumed to use a Virtual Output Queueing (VOQ) architecture and a switch scheduler that is (*i*) parallel and port-independent, (*ii*) work conserving and, (*iii*) throughput conserving (§3.1).
  - Delay can be bounded within a single switch in a single shot scenario according to the equation *R* $= d \leq m \times \frac{P}{r} + \epsilon_{sw}$ (§3.2.1).
  - If packets are limited to an effective rate limit described by (P, e_{sw}) this bound can be maintained indefinitely (§3.2.2).
  - I introduce Leaky Token Bucket (LTB) regulator. An LTB is a variant of a token bucket where tokens are added a rate described by (P, e_{sw}), one batch per time interval maintaining an average rate of $\rho = P_{max}/esw$ and; tokens leak out of the bucket at a constant (fluid-flow rate) which is given by $L = P_{max}/e_{sw}$. This maintains the effective rate limit but allows smaller packets to be sent (§3.2.3).
  - Multiple switches are cascaded together leading to the delay bound given by $d_{net} \leq \sum_{k=1}^{K} \frac{n_k P_{max}}{r_k} + \epsilon_k$ where the flow is LTB constrained with parameters (P_{max}, e_{net}) (§3.2.4).
- When arranged with the same parameters, the PGPS delay bound reduces to $\sum_{s=1}^{K} \frac{(n_s+1) \times P_{max}}{r_s}$ which is nearly identical to the bound derived using an LTB. PGPS uses a fluid-flow based scheduler and a discrete regulator while I use a discrete scheduler and a fluid-flow regulator (§3.3).

- Despite the simplicity of the mathematics, there are many practical considerations to address in order to implement the above schemes. Addressing these considerations is ultimately an operator's decision and will depend on operational trade-offs (§3.4).
  - The queue depth of the regulator determines how tolerant it is of bursts, but also how long packets can potentially wait, which extends the potential latency bound. I choose a queue depth of zero which is called "policing" regulator (§3.4.1).
  - Regulators may be placed at many places between the application and the network: in the application, between the application and the kernel, in the kernel, in the network card or in the switch. I find that the in-kernel placement offers the best compromise of features, fidelity, and overheads (§3.4.2).
  - Packet sizes are limited to a minimum of 64B which limits the minimum delay bound. The maximum size of packets is the TSO/GSO/LSO transfer size which is usually about 64kB. Beyond this size there is no gain to be had (§3.4.3).
  - A policing Leaky Token Bucket regulator can be implemented in 24 lines of high performance code. One optimisation is to perform calculations using cycles rather than using wall-clock time (§3.4.4).
  - Using TCP with an aggressive policing limiter can lead to poor performance. This can be mitigated by using transmit window sizing and smaller retransmit timeouts (§3.4.5).
  - At least one common variant of a popular datacenter switch chip (Trident II+) implements a (*i*) parallel and port-independent, (*ii*) work conserving and, (*iii*) throughput conserving scheduler (§3.4.6).
  - Datacenter network topologies have an impact on the obtainable bounds. The most significant limit comes from the speed at the edge of the network. The next most significant limit comes from amount of interference free throughput in the core of the network (§3.4.7).
  - Hosts can add latency and interference. This is out of scope for the dissertation (§3.4.8).

Chapter 4

Implementing Predictable Datacenter Networks

In the preceding chapter I described a scheduling model based on the operation of datacenter switches. From this model, I derived the source traffic constraints required to provide bounded latency in the network, and the Leaky Token Bucket (LTB) regulator to enforce these constraints. I showed that reasonable values can be determined for many implementation specific concerns and trade-offs required to realise the above mathematical constructions in a realistic datacenter network. I also demonstrated a potential issue with the approach (see §3.2.5): the average rate given to each host scales inversely proportionally to the number of hosts participating in the bounded delay network (N). For example, on a 10Gb/s network with $N = 5000$ hosts, each host will receive only 2Mb/s of average throughput (i.e. 99.98$\%$ network capacity is lost). The value of N is therefore an important practical consideration.

In this chapter I discuss a straightforward method to make the concepts from Chapter 3 practical. I embody this in a coordination free-system called QJUMP. QJUMP favours simplicity and deployability above all other concerns. Its design makes the convenient assumption that N is equal to the number of hosts in the network. This assumption is convenient because it implies that all hosts can participate in the bounded delay property of the network, but also that the delay bound is minimised and that average throughput value is maximised. Under this assumption, no runtime coordination or control is necessary, making QJUMP easy to implement and deploy.

4.1 Throughput vs. Latency

The problem demonstrated in Chapter 3 is that considerable throughput can be lost at the expense interference control. For some applications, particularly coordination and control applications, the trade-off between low throughput and interference control will lie deeply in favour of interference control. Nevertheless, low throughput is clearly not ideal for all applications. This situation can be improved upon by making two observations:

1. Equation 3.13 is pessimistic: it assumes that all hosts transmit to a single destination at the same time, taking the worst (combination of) paths. This is unlikely given a realistic network and traffic distribution.

2. Some applications (e.g. PTPd) are more sensitive to delays than others (e.g. memcached) whereas still other applications (e.g. Hadoop) are most sensitive to throughput restrictions.

From the first observation, we can relax the throughput constraints in Equation 3.14 by assuming that fewer than all hosts send to a single destination at the worst time. For example, we may assume that only half of the hosts concurrently send to a single destination (i.e. 72 out of 144). Substituting this in to Equations 3.9 and 3.14 results in:

$$D \leq \sum_{k=1}^{K} \frac{n_k P_{max}}{r_k} = \frac{16}{40\text{Gb/s}} + \frac{72}{40\text{Gb/s}} + \frac{71}{10\text{Gb/s}} = \frac{93}{10\text{Gb/s}} = 9.3\text{ns/bit} \tag{4.1}$$

$$R_{ave} \approx \frac{r_{edge}}{N} = \frac{10\text{Gb/s}}{72} = 138\text{Mb/s} \tag{4.2}$$

That is, assuming that only 72 hosts send at any one time, those hosts can all send at twice the rate (138Mb/s) and can receive almost half the latency guarantee (9.3ns/b vs 17.9ns/b – see Table 3.2). We can generalise the above idea by introducing a scaling factor f so that the number of *effective senders* in the network N' is given by:

$$N' = \frac{N}{f} \quad \text{where } 1 \leq f \leq N. \tag{4.3}$$

Intuitively, f is a "throughput factor": as the value of f grows, so does the amount of throughput available, but the effective number of hosts that can use the network (N') decreases. In the previous example, the total number of hosts in the network was $N = 144$, but the effective number of senders was $N' = 72$. The throughput factor was thus $f = 2$.

From the second observation above, some (but not all) applications can tolerate some degree of latency variance above the guaranteed bound. For these applications, we can aim for a statistical reduction in latency variance rather than bounded latency. This is achieved by assuming, but not enforcing the value for f. This re-introduces a degree of statistical multiplexing to the network albeit one that is more tightly controlled. When the guess for f is too optimistic, the actual number of senders is greater than N', then some queueing will occur and the latency bound will be violated. However, under reasonable circumstances, with reasonable traffic patterns, the resulting queueing will still yield an improvement in tail latencies.

The probability that interference occurs increases with increasing values of f. At the upper bound ($f = N$), the potential latency variance is equivalent to existing networks, but full network throughput ($R_{ave} = r_{edge}$) is available to applications. This configuration is essentially identical to networks as they are currently. Setting f to any value lower than N represents a decrease in the likelihood of interference and hence latency variability. At the lower bound ($f = 1$), latency is guaranteed, albeit with tightly controlled throughput. In essence, f quantifies the interference vs. throughput trade-off. Large values of f have high throughput with the cost of a high likelihood of interference. Small values of f have low throughput with the advantage of lower/bounded interference.

4.2 Jump the Queue with Prioritisation

It would be ideal to use multiple values of f concurrently, so that different applications can benefit from the interference/latency variability vs. throughput trade-off that suits them best. To achieve this, the network needs to be partitioned so that traffic from latency-sensitive applications (e.g. PTPd, memcached) can "jump-the-queue" over traffic from throughput intensive applications (e.g. Hadoop).

Datacenter switches support the IEEE 802.1Q [136] standard which provides eight (0–7) hardware enforced "service classes" or "priorities". IEEE 802.1Q is a strict priority system which means that high priority traffic will always be serviced in preference to low priority traffic.

Priorities are often hard to use in practice because priority selection can become a "race to the top". For example, PTPd users may assume that PTPd traffic is the most latency sensitive and should receive the highest priority. Meanwhile, Hadoop users may assume that Hadoop traffic is the most bandwidth sensitive, and should similarly receive the highest priority. Since there are a limited number of priorities, and neither can achieve an advantage and prioritisation loses its value. The QJUMP system is different.

In QJUMP, priority values are intentionally bound to rate-limit values. For each priority, I assign a value of f. The higher the priority the smaller the value of f. Since a small value of f implies an aggressive rate limit, priorities become useful because they are no longer "free". QJUMP users must choose between low latency variance/interference at low throughput (high priority) and high latency variance at high throughput (low priority).

I call the assignment of an f value to a priority a QJUMP *level*. The latency variance of a given QJUMP level is a function of the sum of the QJUMP levels above it. In Section 4.4, I discuss various ways of assigning f values to QJUMP levels.

4.3 Implementation

To implement QJUMP requires two components: (*i*) a leaky token bucket regulator (as described in §3.4.4), and (*ii*) a mechanism to configure applications to use QJUMP levels. The regulator can be deployed in many locations (see §3.4.2 for a fuller discussion). In a single-authority datacenter environments (e.g. Google, Facebook etc), the regulator is best deployed as an addition to the kernel network egress path. In a multi-tenant environment (e.g. Amazon AWS, Microsoft Azure etc), the regulator might best be deployed as a component in the hypervisor. In either case, these choices limit the changes to the system to a pluggable component with a well defined interface, yet offer the highest fidelity and configurability and the lowest runtime cost. In such a configuration, the applications and network hardware remain unchanged and inserting or removing a kernel module is a low overhead exercise. This aligns with QJUMP's goals of simplicity and deployability.

4.3.1 Application Utility

QJUMP requires that applications (or, specifically, sockets within applications) are assigned to QJUMP levels. In Linux and other UNIX like operating systems, this can easily be done in application code directly with a `setsockopt()` using the `SO_PRIORITY` option. The SO_PRIORITY option, in combination with the Linux IEEE 802.1Q driver module, allows hosts to issue packets tagged with a priority value that will be handled by the switch. To support unmodified applications without recompilation, I have implemented a utility that dynamically intercepts socket setup system-calls and alters their options. I inject the utility into unmodified executables via the Linux dynamic linker's `LD_PRELOAD` support (a similar technique to that used by ExaSock[1] and "Open" Onload[2]). The utility performs two tasks: (*i*) it configures socket priority values, and (*ii*) it sets socket send buffer sizes to avoid TCP retransmit timeouts occurring (as described in §3.4.1).

4.4 Configuration

A QJUMP deployment requires four parameters to be considered/configured: (*i*) P_{max}, the maximum packet size; (*ii*) r_k, the rate of the slowest edge link; (*iii*) ϵ, the cumulative switch processing delay; and (*iv*) f_i, the per-level throughput factors.

4.4.1 Configuring R and ϵ

Since the topology of a datacenter network is relatively static, the minimum link speed R and the cumulative switching delay ϵ do not vary much or often. Typical values are $R =$ 10Gb/s or 40Gb/s and ϵ = 0.1μs to 0.5μs. Setting the remaining parameters P, and f requires some consideration of the specific aims and objectives for the QJUMP deployment.

4.4.2 Configuring P_{max}

As discussed in Section 3.4.3, there are limits on the allowable sizes that P_{max} may take. In summary, the lower limit is 64B while the practical upper limit (for a kernel-based regulator) is around 64kB (see §3.4.3 for more details). In addition to these limitations, from Equation 3.10, the network epoch grows linearly with increasing P_{max}. The P_{max} value should therefore be kept small to keep the network epoch short. Incentives to decrease the packet size are usually aligned with incentives to decrease the latency bound. However, P_{max} should also be large enough to be useful. Benson *et al.* found that 30%–50% of packets in many datacenters contain fewer than 256 bytes [152]. This suggests that $\leq$256B packets are sufficient for some applications.

[1] Exablaze blog post (14 July 2014). See Appendix A.9.3.

[2] Solarflare Flareon/"Open"-Onload marketing material. See Appendix A.24.1.

For 1,000 hosts, setting P to 256 bytes results in a worst-case delay of $\approx 200\mu$s, which is low enough to be useful to applications.

4.4.3 Configuring f_i

The most difficult parameters to determine are the throughput factors f_i. Fortunately, each value of f_i is easily expressible as a rate-limit in Mb/s (e.g. $R'_{ave} \approx \frac{r_{edge}}{N'}$) which makes choosing values relatively intuitive (see §4.5 for examples). This is in contrast to other systems such as DCTCP which have complex parameters. And example from DCTCP is the parameter g. This is defined as follows: "$0 < g < 1$ is the weight given to new samples against the past in the estimation of α" [71]. The best value for f_i depends on the desired latency distribution and the workload. The simplest configuration is to use only two QJUMP levels: (*i*) guaranteed latency (f_1 = 1) and (*ii*) maximum throughput ($f_7 = N$). This configuration, however, lacks flexibility. Alternatively, a set of f_i values can be configured for a known application mix or for a known traffic distribution.

1. **Known Application Mix** Datacenter application mixes are often known, or information on application profiles can be obtained from users [153–155]. If application latency and throughput requirements can be estimated or measured, the QJUMP levels can be set to accommodate their needs. Likewise, applications can be assigned to existing QJUMP levels based on their profile. In practice, simple benchmarks at different rate limits make it easy to characterise an application. For an example of this method using memcached see Section 4.5.4.1. There may be more applications than QJUMP levels. In this case, either some levels will need to be shared between applications, or applications will need to be scheduled onto different hosts to avoid sharing (e.g. Silo [153]).

2. **Known Traffic Distribution** While the application mix in large datacenters can be complex, monitoring infrastructure can supply aggregate traffic statistics. An approximate distribution of flow sizes is often available [69, 71, 152]. For a known flow size distribution, f_i values can be configured to partition the traffic into a latency variance vs. throughput distribution. I applied this method on a flow size Cumulative Distribution Function (CDF) using a simple spreadsheet. This worked well in my experiments and simulations in §4.5.3. Alternatively, an exponential distribution of f_i values of e.g. f_0=10Gb/s, f_1=5Gb/s, f_2=2.5Gb/s, etc., can provide a wide range of choices within a limited number of QJUMP levels.

4.4.4 A note on configuring N and n_k

Equations 3.9 and 3.14 are parameterised over n_k and N respectively. From the total number of hosts N on the network, we need to select the subset that we wish to use with QJUMP. QJUMP makes the assumption that N is configured to be equal to the total number of hosts within a pod and n_k is a (topologically determined) function of N. The size of the QJUMP latency bound scales as a function of N. If all hosts in the network use QJUMP, and each host has one latency application socket, then N can take a value of between 1,000 and 4,000 hosts and maintain a bound of 100-500μs using small messages of 64–256B. Increasing the number of application sockets that use the bounded latency layer would require N to scale accordingly, resulting in either a higher latency bound or fewer hosts being able to participate. QJUMP could also be configured with N set as a subset of the hosts, provided that the remainder of hosts only use the lowest network priority and that priority is configured for best effort service (i.e. $f = N$).

Application-specific knowledge could be exploited to increase the number of hosts that can participate in a QJUMP network while retaining the latency bound. For example, a distribute/aggregate service may send requests to 10,000 hosts, but can be certain that fewer than 1,000 hosts will respond to each request. In this case, N can still be set to 1,000 hosts, but all 10,000 hosts could use QJUMP at the guaranteed level. The delay bound would still be met provided that no more than 1,000 hosts respond to any given request.

Finally, it is worth noting that QJUMP scales proportionally with the network edge speed. On a faster network (e.g. a 40Gb/s edge), the same delay bound can be maintained for larger values of N (e.g. 16,000) or as large as $40,000$ hosts (assuming the application specific knowledge above). This means that, although QJUMP is explicitly designed for pod-scale networks, it may scale as large as realistic datacenter networks depending on the configuration of the network and the use-case for which it is deployed.

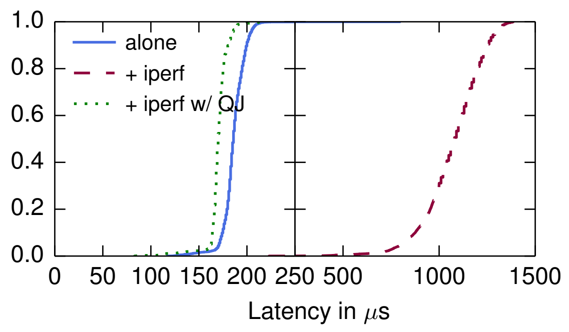
4.5 Evaluation

I have evaluated QJUMP on both a small deployment and in simulation. My evaluation shows that QJUMP:

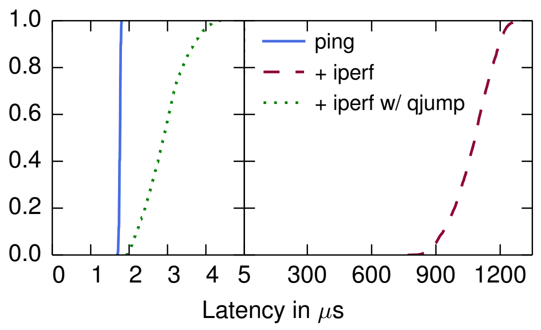
1. Resolves network interference for a collection of real-world datacenter applications (§4.5.1.1);
2. Outperforms Ethernet Flow Control (EFC)(IEEE 802.3x), Ethernet Flow Control (EFC) and DCTCP (§4.5.2);
3. Provides excellent flow completion times, close to or better than pFabric [94] (§4.5.3);
4. Is easily configurable, illustrated by examples of methods to determine QJUMP parameters (§4.4).

4.5.1 QJUMP Resolves Network Interference

My experiments in Section 2.3 and 2.4 showed that network interference degrades application performance. I now repeat those experiments with QJUMP enabled and show that QJUMP mitigates the network interference, resulting in near ideal performance. I also show that in a realistic, multi-application, setting QJUMP both resolves network interference and outperforms other readily available systems. As before, I execute these experiments on the topology shown in Figure 2.6.



***(a)** CDF of `ping` application measured latency.*



***(b)** CDF of `ping` packet latency across a switch.*

Figure 4.1: QJUMP *resolves network interference both at an application perceivable level (a) and across the switch (b) (note the change in x-axis scale in both cases)*

4.5.1.1 Low Latency RPC vs. Bulk Transfer

Remote Procedure Calls (RPCs) and bulk data transfers represent extreme ends of the latency-bandwidth spectrum. QJUMP resolves network interference at these extremes. As in Section 2.4, I emulate RPCs and bulk data transfers using `ping` and `iperf` respectively. In this experiment I measure in-network latency for the `ping` traffic both from the perspective of the application (`ping`) and directly using a high resolution Endace DAG[3] capture card and optical taps. The first measurement (Fig. 4.1a) shows that QJUMP resolves queueing latency from an application's perspective. In this case, QJUMP performs better than the idle case. QJUMP provides a median (50th percentile) latency of 175μs compared to 185μs in the idle case. This because prioritised packets are accelerated up through the host network card/stack.

The second measurement (Fig. 4.1b) verifies that the source of queueing latency (arising within shared switch queues) is resolved by QJUMP. By setting `ping` to the highest QJUMP level ($f_7 = 1$), the latency across the switch is reduced by over 300$\times$ (Figure 4.1b). The small difference between idle switch latency (1.6μs) and QJUMP latency (2–4μs) arises due to a small on-chip queue through which the switch processes packets in-order. No prioritisation is applied in this queue. The switch processing delay, represented as ϵ in Equations 3.10 and 3.9, is thus no more than 4μs. Without the re-scaled x-axis of Figure 4.1b, this difference would be imperceptible.

4.5.1.2 Memcached

QJUMP also resolves network interference between realistic datacenter applications such as memcached sharing a network with Hadoop. I show this by repeating the memcached experiments from Section 2.3.2. Once again I measure application-level request latency using `memaslap`. In this experiment, memcached is configured at an intermediate QJUMP level, rate-limited to 5Gb/s (see §4.5.4.1 for more details). Figure 4.2 shows the distribution plot (CDF) of memcached

[3]Endace DAG 9.2X2. See Appendix A.8.1.

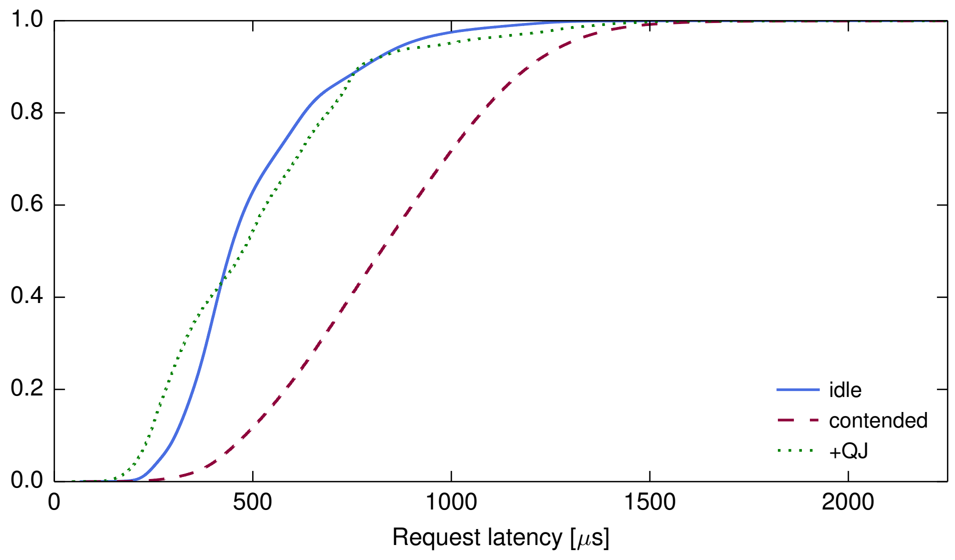


Figure 4.2: QJUMP *reduces memcached request latency: CDF of 9 million samples.*

request latencies when running on an idle network, a network shared with Hadoop, and a shared network with QJUMP enabled. With QJUMP enabled, the request latencies are close to the ideal. The median latency improves from 824μs in the shared case to 476μs: a (nearly) 2$\times$ improvement.[4] There is no measurable effect on Hadoop's performance (see §4.5.2 for more details).

4.5.1.3 Multi-application Environment

In real-world datacenters, a range of applications with different latency and bandwidth requirements share the same infrastructure. QJUMP effectively resolves network interference in these shared, multi-application environments. To demonstrate this I use the three representative applications discussed in Section 2.3.2: `ptpd` for time synchronisation, memcached for serving small objects, and Hadoop for batch data analysis. Once again, resolving on-host interference is outside the scope of my work, so I do not allow hosts to share applications in these experiments. More details on combating host based interference can be found in Section 3.4.8.

As demonstrated in Section 2.4.1, PTPd and memcached are easily perturbed by Hadoop's traffic. Figure 4.3 *(top)* shows a 1ms timeline of average request latencies for memcached, and synchronisation offsets for `ptpd`, each running alone on an otherwise idle network. Figure 4.3 *(middle)*, shows the two applications sharing the network with Hadoop. In this case, average latencies increase for both applications and visible latency spikes (corresponding to Hadoop's shuffle phases) emerge. With QJUMP deployed, I assign `ptpd` to $f_7 = 1$, Hadoop to $f_0 = N = 12$ and memcached to $T_5 = 5\text{Gb/s} \implies f_5 = 6$ (see §4.5.4 for further configuration details). Using QJUMP, the three applications are able to co-exist without interference (Figure 4.3

[4]The distributions for the idle network and the QJUMP case do not completely agree due to randomness in the load generated.

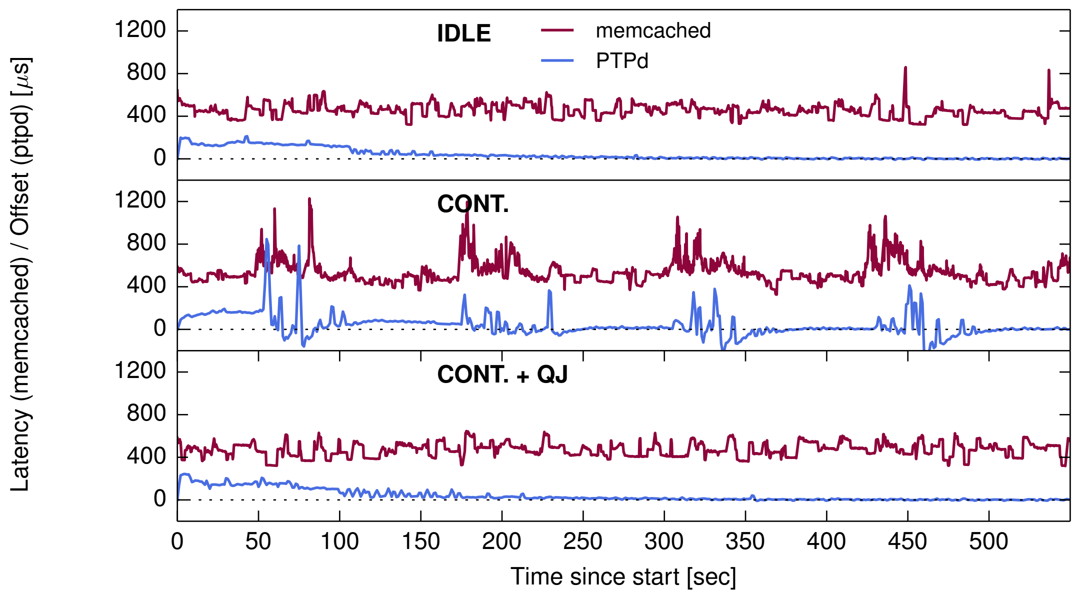


Figure 4.3: *PTPd and memcached in isolation* (top)*, with interfering traffic from Hadoop* (middle) *and with the interference mitigated by* QJUMP (bottom).

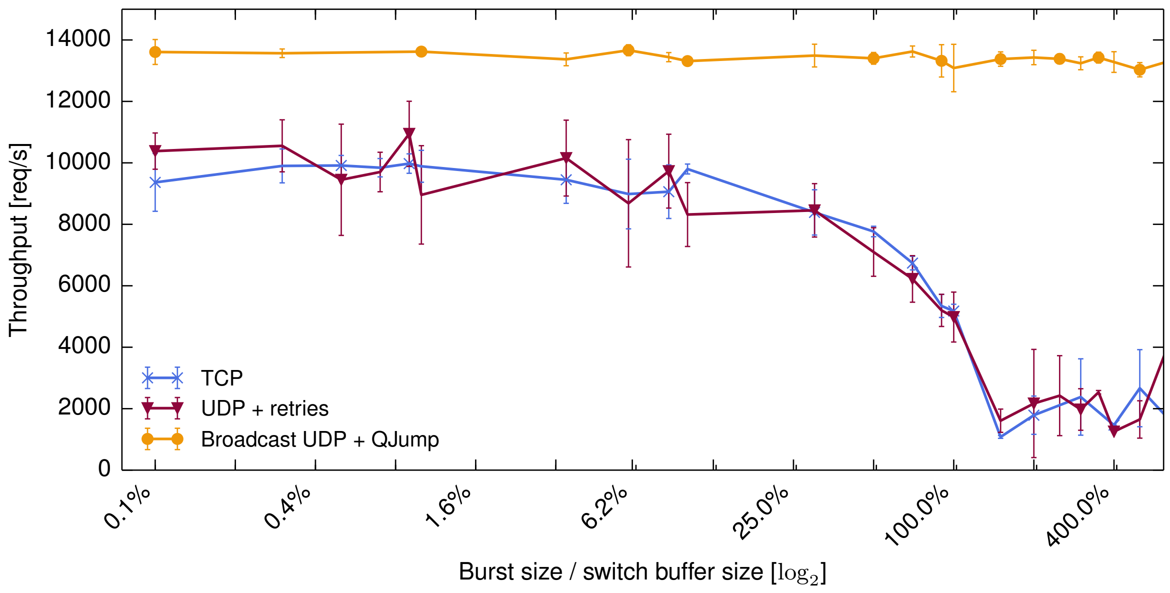


Figure 4.4: QJUMP *offers constant two-phase commit throughput even at high levels of network interference.*

(bottom)). Hadoop's performance is not noticeably affected by QJUMP (see §4.5.2 for more details).

4.5.1.4 Distributed Atomic Commit

One of QJUMP's features over traditional congestion control systems is its guaranteed latency level. Bounded latency enables interesting new designs for datacenter coordination software such as SDN control planes, fast failure detection, and distributed consensus systems. To demonstrate the usefulness of QJUMP's bounded-latency level, I built a simple distributed two-phase atomic-commit (2PC) application. 2PC like protocols are at the heart of many distributed agreement systems such as the Paxos [156] and Raft [157] protocol variants.

My 2PC application communicates over TCP or over UDP with explicit acknowledgements and retransmissions. Since QJUMP offers lossless delivery, the coordinator can send its messages by UDP broadcast when QJUMP is enabled. This optimisation yields a ≈30% throughput improvement over both TCP and UDP.

In Figure 4.4, I show the request rate for one coordinator and seven servers as a function of network interference. Interference is created with two traffic generators: one that generates a constant 10Gb/s of UDP traffic and another that sends fixed-size packet trains followed by a 25ms pause. I report interference as the ratio of the burst size to the internal switch buffer size. Beyond a ratio of 200%, permanent queues build up in the switch. At this point the impact of retransmissions degrade will throughput of the UDP and TCP implementations to 20% of the 10,000 requests per second observed on an idle network. By contrast, the UDP-over-QJUMP implementation does not degrade because its messages "jump the queue". At high interference ratios (>200%), two-phase commit over QJUMP achieves 6.5× the throughput of standard TCP or UDP. Furthermore, QJUMP's reliable delivery and low latency enables very aggressive timeouts to be used for failure detection. My 2PC system detects component failure within two network epochs (≈40μs on the network), several orders of magnitude faster than typical failure detection timeouts (e.g. 150 ms in RAMCloud [158, §4.6]).

4.5.2 QJUMP Compared

In Section 2.4 I discussed several contemporary approaches to congestion control in datacenter networks and their impact on network interference. Of those approaches, the first 4 (TCP CUBIC, Ethernet Flow Control, Explicit Congestion Notification and Datacenter TCP), were deployable in my testing environment (see §2.4.1–§2.4.4). Figure 4.5 shows a summary of the results from Section 2.4 on a unified axis and compared with QJUMP.

The figure shows the Root Mean Squared (RMS) value of each application specific metric of interest, normalised to the ideal case. RMS [159] is used because it provides a good typical value of a continuously varying quantity so that the different approaches can be compared easily. The application specific metrics of interest are synchronisation offset (PTPd), request latency (memcached) and runtime (Hadoop) (for more details see §2.3.2.1 to §2.3.2.3). The ideal case shows each application running on the network alone. Every other case shows all three applications running at the same time. The ECN case shows the 'optimal' [minimum,maximum] marking threshold of [40, 80] (see 2.4.3 for more details).

The figure shows that, of all the previously tested solutions, DCTCP does the best job of controlling network interference. However, even when using DCTCP, the RMS PTP synchronisation offset is still over 2× worse than the ideal case. This is because DCTCP still uses TCP's probing method to search for unused bandwidth, which has the effect of lengthening

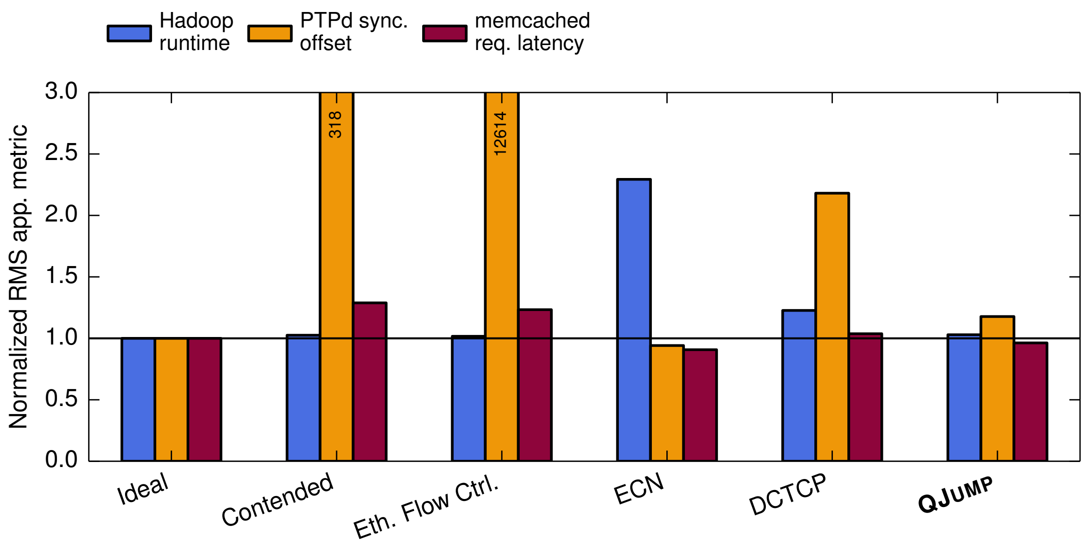


Figure 4.5: QJUMP *exhibits closest to ideal performance for all of Hadoop, PTPd and memcached.*

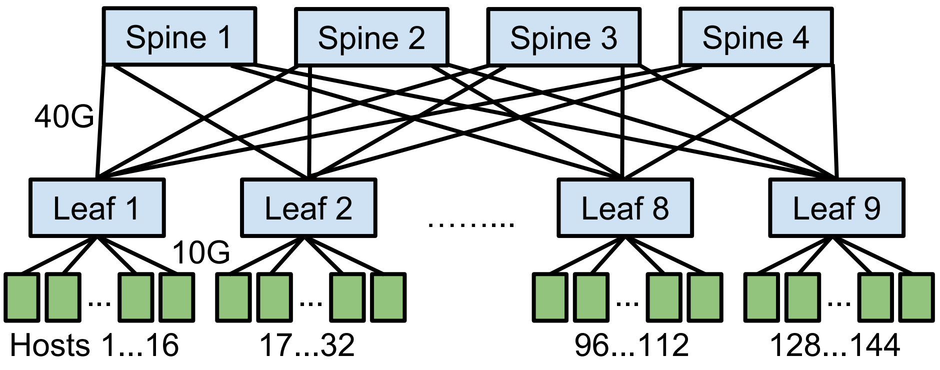
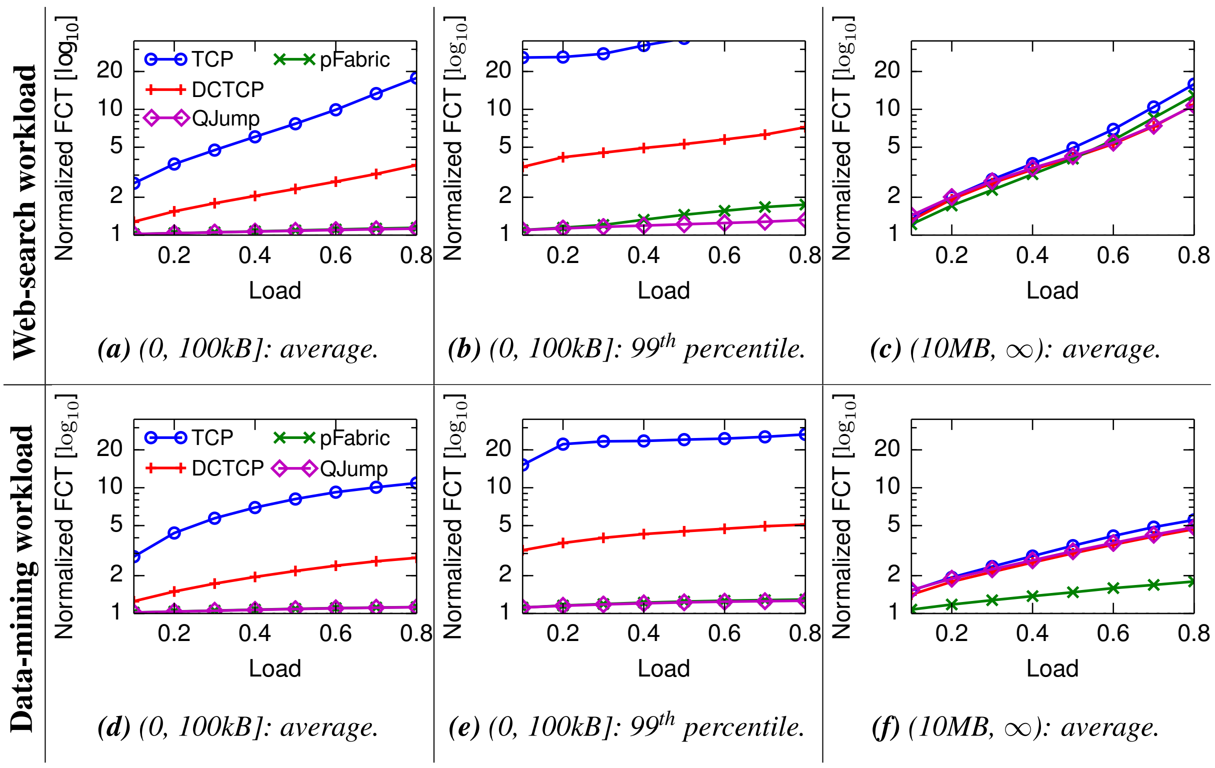


Figure 4.6: *144 node leaf-spine topology used for simulation experiments.*

queues within the network. By contrast, QJUMP is close to ideal for all metrics. The variance in Hadoop, PTPd and memcached performance is close to or slightly better than in the uncontended ideal case. This shows that QJUMP outperforms its competitors in a small practical datacenter applications test.

4.5.3 QJUMP Improves Flow Completion Times

While QJUMP performs well in comparison with other congestion control schemes, the QJUMP design has its roots in network scheduling (as described in §2.5 and §3–§3.3). It is therefore necessary to compare QJUMP against other network scheduling approaches. One such approach is pFabric [94] (see also §2.4.9). The pFabric architecture has been shown to schedule flows close to optimally. The authors of pFabric show that a "greedy scheduler [. . .] that prioritizes small flows over large flows end-to-end across the fabric can provide near-ideal average FCT" (Flow Completion Time). The pFabric greedy scheduler is an approximation of the shortest time remaining scheduling policy, which has been shown to be optimal [160].



(a) (0, 100kB]: average. *(b) (0, 100kB]: 99th percentile.* *(c) (10MB, ∞): average.*

(d) (0, 100kB]: average. *(e) (0, 100kB]: 99th percentile.* *(f) (10MB, ∞): average.*

Figure 4.7: *Normalised flow completion times in a 144-host simulation (1 is ideal):* QJUMP *outperforms TCP, DCTCP and pFabric for small flows. N.B.:* log-*scale y-axis;* QJUMP *and pFabric overlap in (a), (d) and (e).*

QJUMP's method of regulating and prioritising flows in order of increasing rate imposes an ordering on network traffic that also (crudely) approximates the shortest time remaining schedule. Flows with a higher rate allocation implicitly have a longer running time and are given a low priority. Flows with a low rate allocation implicitly have a shorter runtime and are given high priority. In this way, QJUMP is broadly similar to pFabric in its approach. The key difference between QJUMP and pFabric is that QJUMP levels are fixed. Flows do move between QJUMP levels as they near completion. This simplification makes QJUMP easier to implement and deploy.

Since pFabric is a "...clean-slate design [that] requires modifications both at the switches and the end-hosts ..." [94] it is only available in simulation. To compare the two, QJUMP was implemented in the simulation environment provided by the pFabric authors[5]. This environment also facilitates comparisons between QJUMP, a baseline TCP instantiation and a DCTCP implementation (also provided by the authors of pFabric). Comparing against TCP and DCTCP are useful to cross-validate the simulation with the experiments performed in Section 4.5.2.

Figure 4.6 shows the leaf-spine network topology used to evaluate pFabric. There are 144 hosts in total using 9 leaf switches and 4 four spine switches. The core of the network runs at

[5]The initial implementation was carried out by Ionel Gog, my coauthor and collaborator.

40Gb/s while the edge runs at 10Gb/s. To perform the evaluation, I used the same workloads as those used by pFabric. These are derived from web search [71, §2.2] and data mining [69, §3.1] clusters in Microsoft datacenters. I also show matching graphs in Figure 4.7 from the pFabric paper. As in pFabric, I normalise flows to their ideal flow completion times (FCTs) on an idle network.

Figure 4.7 reports the average and 99th percentile normalised FCTs for small flows (0kB, 100kB] and the average FCTs for large flows (10MB, ∞). For both workloads, QJUMP is configured with $P = 9$kB, $n = 144$, and $\{f_0 \ldots f_7\} = \{144, 100, 20, 10, 5, 3, 2, 1\}$. I chose this configuration based on the distribution of flow sizes in the web search workload. However, it also worked well for the data mining workload.

Despite its simplicity, QJUMP performs very well. As expected, it works best on short flows. On both workloads, QJUMP achieves average and 99th percentile FCTs close to or better than pFabric's. On the web-search workload, QJUMP beats pFabric by a margin of up to 32% at the 99th percentile (Fig. 4.7b). For larger flows, the results are mixed. On the web search workload, QJUMP outperforms pFabric by up to 20% at high load, but loses to pFabric by 15% at low load (Fig. 4.7c). On the data mining workload, QJUMP's average FCTs are between 30% and 63% worse than pFabric's (Fig. 4.7f).

In the data-mining workload, 85% of all *flows* transfer fewer than 100kB, but over 80% of the *bytes* are transferred in flows of greater than 100MB (less than 15% of the total flows). QJUMP's short epoch intervals cannot sense the difference between large flows and very large flow, so it does not apply any rate-limiting (scheduling) to them. This results in sub-optimal behaviour. It is possible that a combined approach might improve this. In such an approach QJUMP would regulate the interactions between large flows and small flows, while DCTCP is used to regulate the interactions between different large flows. I leave investigation of the avenue to future work.

4.5.4 QJUMP Configuration

4.5.4.1 Rate Selection (the f_i Parameter)

As described in Section 4.4, QJUMP levels can be determined in several ways. One approach is to tune the levels to a specific mix of applications. For some applications, it is clear that they perform best at guaranteed latency (e.g. `ptpd` at $f_7 = 1$) or high rate (e.g. Hadoop at $f_0 = N$). For others, their performance at different throughput factors is less straightforward. Memcached is an example of such an application.

Memcached requires low request latency variance as well as reasonable request throughput. To find the optimal point, I investigated the performance of memcached with different amounts of rate limiting applied. In this experiment, `memaslap` benchmarks a single server and there is no network interference. Figure 4.8 shows memcached's request throughput and latency as a function of rate-limiting. At very low rates, throughput is low due to rate limiting, and the

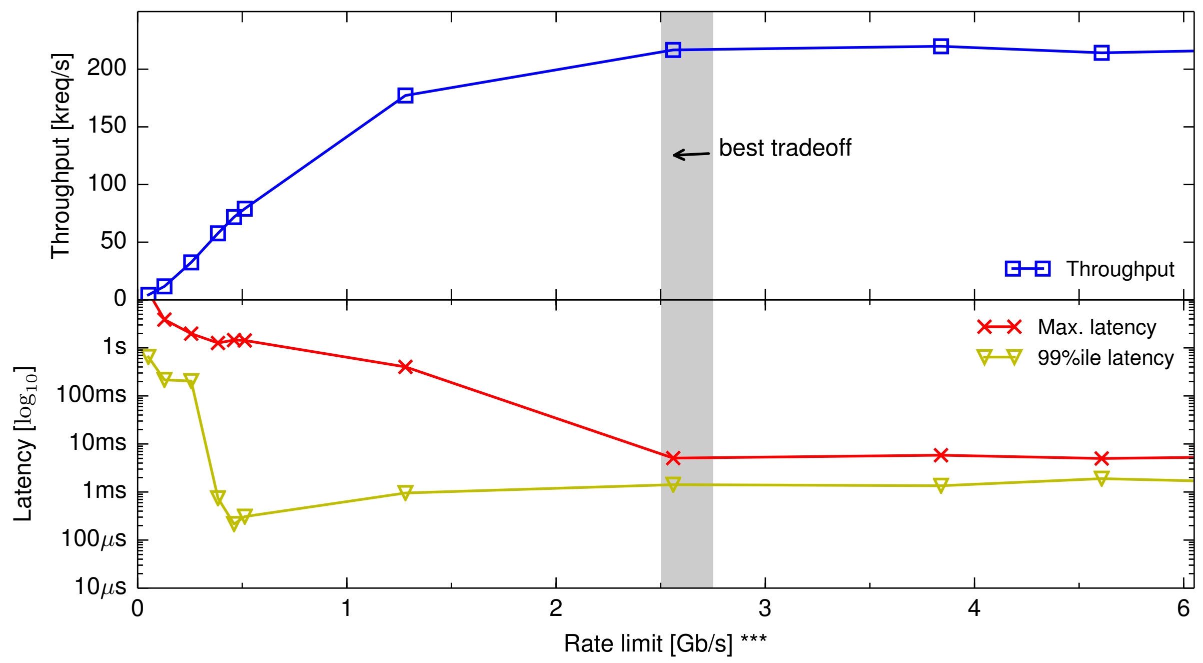


Figure 4.8: *Memcached throughput (top) and latency (bottom,* $\log_{10}$*) as a function of the* QJUMP *rate limit.*

tail latencies (99th percentile and maximum) are high because of constant TCP retransmission timeouts. As the rate increases to around 500Mb/s, the 99th percentile latency drops dramatically. At this point the server is underloaded and most requests can be completed quickly, although TCP timeouts still have an effect. Peak throughput is reached at a rate allocation of around 5Gb/s beyond which throughput improvements are marginal and the latency distribution is unchanged. At the same point, the request latency also stabilises. Hence, a rate-limit of 5Gb/s gives the best trade-off for memcached. This point has the strongest interference control from other systems without throughput restrictions to memcached itself.

A rate limit can easily be converted into a throughput factor using Equation 4.3 (pg. 96) and rearranging for f_i (e.g. $fi = \frac{nR'}{R_{edge}}$). On our test-bed $n = 12$ and R_{edge} =10Gb/s. Since R' =5Gb/s the throughput factor is $f = 6$. We can therefore choose a QJUMP level for memcached (e.g. f_4) and set it to a throughput factor ≥ 6.

4.5.4.2 Sensitivity to f

QJUMP claims to offer bounded latency level at throughput factor $f_7 = 1$. At this level, all packets admitted into the network should reach their destinations within the bound given by Equation 3.9. To verify that this is true, and to gauge the sensitivity of QJUMP to (mis)configuration of f_i, I performed the following scale-up experiment. In the experiment I arranged a 60-host virtualised topology running on ten physical machines (see Figure 4.9). Each machine runs a "hypervisor" (Linux kernel) with a 10Gb/s uplink to the network. Each hypervisor runs six "guests" (processes) each with a 1.6Gb/s network connection. Each guest runs two applications (threads): (*i*) an emulated "coordination service" that generates one 256 byte packet per time interval (at the

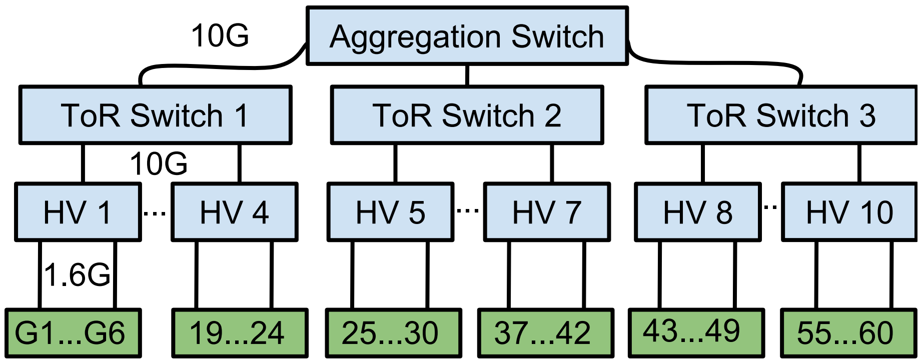


Figure 4.9: *Latency bound validation topology: 10 hypervisors (HV) and 60 guests (G1..60) and 120 apps.*

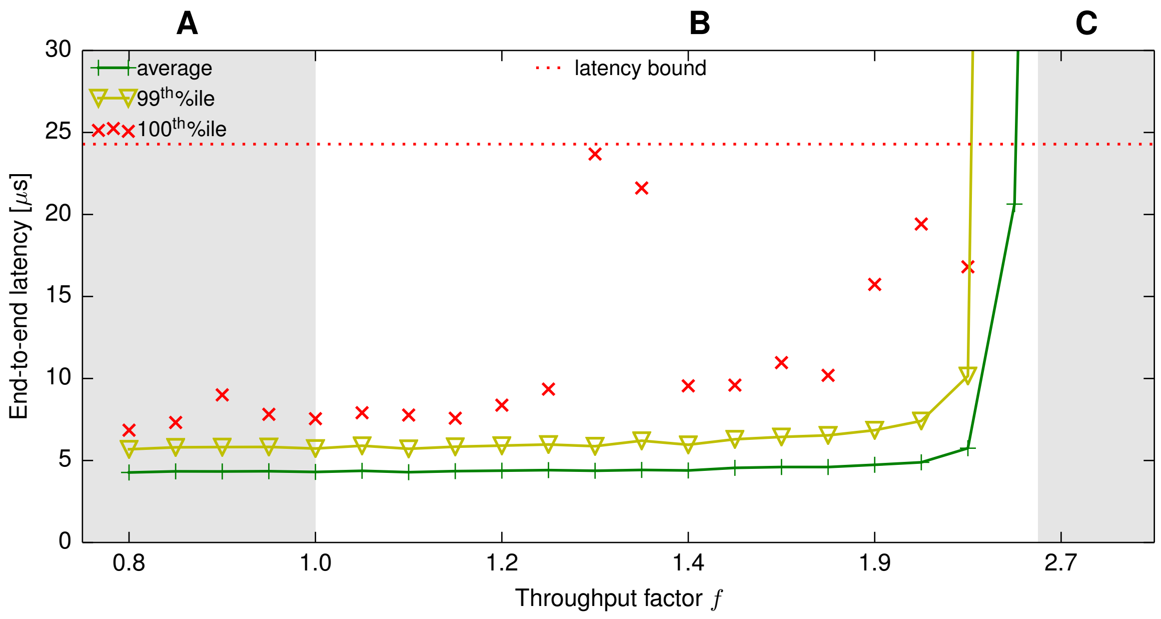


Figure 4.10: *Latency bound validation experiment using a 60 host fan-in of f_7 and f_0 traffic. 100 million samples per data point.*

highest QJUMP level), and (*ii*) an emulated "bulk sender" service that issues 1518 byte packets as fast as possible (at the lowest QJUMP level). The coordination messages are sent to a single destination. The time interval in-between coordination messages is configurable. For 60 hosts sending 256B packets, the correct interval is the network epoch which is 24.3μs. This is a throughput factor $f_i = 1$. To test the sensitivity of QJUMP to the throughput factor, I vary f_i between values of 0.8 and 3 (20% underloaded to 300% overloaded).

Figure 4.10 shows the latency distribution of coordination packets as a function of the throughput factor. If f_7 is set to less than 1.0 (region **A**), the latency bound is met (as we would expect). In region **B** (1.0 to 2.7), the 100th percentile outliers increase. Transient queueing affects some packets, but all requests make it within the latency bound. Beyond $f_7 = 2.7$ (region **C**), permanent queueing occurs and the 99th and 100th percentiles increase beyond the bound. This experiment indicates that QJUMP works as expected when configured correctly and can tolerate some degree of misconfiguration, in this case of over 200%, while still meeting the delay bound guarantee in a 100M data-point sample. Naturally, the system offers no actual guarantees beyond a throughput factor of $f = 1$, only a reduction in likelihood of interference occurring.

4.5.5 Experimental Limitations and Opportunities

The experiments in this section have demonstrated functional prototype implementations of QJUMP both on a small test-bed as well as in simulation. These experiments have verified that QJUMP resolves network interference both from an application perceived perspective and when measured directly in the network. QJUMP achieves good results in both cases. However, the experiments are limited in size and complexity to what can reasonably be achieved in a laboratory environment. Specifically, QJUMP has not been tested at a datacenter scale using thousands of hosts. It is quite possible that new, unforeseen, problems or issues may arise when operating at larger scale.

In addition to scale limitations, the combined application test (see §4.5.1.3) is not as compelling as it may at first appear. Specifically, there was no n-to-1 fan-in component present in the memcached/PTPd traffic patterns. This means that the traffic regulator, although operational, was not providing any useful protection and could easily have been disabled[6]. An interesting consequence of this limitation is that it demonstrates that QJUMP could potentially be deployed in a less rigorous manner. If operators simply prioritised traffic in order of decreasing rate requirements, a good network schedule may naturally result without the imposition of traffic regulators. Such a deployment would work best in underutilised networks where congestion is likely to cause transient effects but will not lead to significant periods of contention for network resources. Further investigation of these ideas is left to future work.

4.6 Conclusions

In this chapter I discussed QJUMP, a simple and direct approach to resolving the network interference problem described in Chapter 2. QJUMP implements the methods described in Chapter 3 but is expressly designed for simplicity and immediate deployability. It places minimal requirements on network infrastructure (prioritised packets §4.2), minimal requirements on the host infrastructure (loadable kernel modules §4.3) and minimal requirements on applications (runtime dynamic linker module to snoop on socket calls §4.3.1). These minimal requirements make QJUMP easy to deploy immediately. However, this minimalism also comes with several costs: (*i*) QJUMP only offers guaranteed latency messaging at very low throughput, (*ii*) QJUMP requires operators to manually measure and configure applications and; (*iii*) QJUMP is limited in scalability to pod-scale networks of a few thousand hosts.

[6]Note that this does not imply that the regulator is ineffective, §4.5.4.2 demonstrates that the regulator operates correctly, providing the expected latency bounds well beyond its expected effective range.

4.7 Chapter Summary

- I apply the model and regulator from Chapter 3 in a system called QJUMP. QJUMP takes a simple, direct and immediately deployable approach to implementing the equations.
- To do so, I make the simplifying assumption of setting N (the number of hosts/flows participating in the bound) to be equal to n, (the number of hosts in the network) (§4).
- The result of this decision is that the effective rate offered to each host scales inversely proportionally to the number of hosts. Over 99% of the available throughput is therefore lost.
- To resolve the (severe) throughput limitations, I take an approximate approach.
  - QJUMP uses a "throughput factor" f to provide relaxed throughput constraints in exchange for relaxed latency variance (§4.1)
  - Several throughput factor values are layered over each other using hardware enforced priorities (§4.2).
  - The guaranteed service layer is given the highest priority.
- The above approach is deployed on a small test cluster and tested with representative datacenter applications (§4.5). It is shown to:
  - Resolve interference for low latency RPC's in the presence of bulk transfer traffic (§4.5.1.1).
  - Resolve interference for datacenter applications sharing a network with Hadoop (§4.5.1.2).
  - Resolve interference in a mixed application environment with PTPd, memcached and Hadoop sharing the network infrastructure (§4.5.1.3).
  - Resolve interference and provide better runtime and failure detection performance for a two-phase commit coordination service (§4.5.1.4).
- QJUMP is also shown to outperform all of the deployable congestion control schemes discussed in Section 2.4 (TCP, EFC,ECN,DCTCP)(§4.5.2).
- The network scheduling methods used by QJUMP are similar to those used by pFabric. QJUMP is thus tested against pFabric in simulation and performs close to, and in many cases, better than pFabric (§4.5.3).
- Finally, I show that the configuration parameters are relatively easy to derive and that QJUMP shows minimal sensitivity to misconfiguration (§4.5.4.1–§4.5.4.2).

Chapter 5

Improvements and Future Work

In Chapter 4 I discussed QJUMP, a simple and direct approach to resolving the network interference problem described in Chapter 2, by implementing model and traffic regulator described in Chapter 3. QJUMP is expressly designed for simplicity and immediate deployability. However, QJUMP's minimalism also comes with several costs: QJUMP offers guaranteed latency messaging only at low throughput, it requires operators to manually measure and configure applications, and it is practically limited in scalability. In this chapter I discuss in detail several potential improvements that could be made to QJUMP to resolve these issues. I begin (§5.1) by discussing an approach to automatic application configuration that would make QJUMP easier to deploy and manage. I then (§5.2–§5.3) go on to discuss approaches to improving the scalability of QJUMP. Finally I present three (§5.5–§5.7) comprehensive system designs which would improve on all three issues with varying costs and complexity trade-offs.

5.1 Automatic Application Configuration

In Section 4.5.4 I showed that it is possible and practical to configure QJUMP levels. Furthermore, I showed that QJUMP is insensitive to some degree of misconfiguration. One problem with QJUMP is the requirement to manually classify applications into QJUMP levels. It would be ideal if applications could be automatically classified into QJUMP levels. This would require overcoming a several engineering challenges:

1. The regulator code would need to be extended to calculate an estimate of the instantaneous throughput for each application. The estimate could then be used to classify applications into appropriate QJUMP levels.
2. Applications that exceeded their throughput allocation would need to be moved to a lower QJUMP level. On the other hand, applications that underutilised their allocation would need be lifted to a higher QJUMP level. This is more complicated than it would initially appear because race-conditions and reordering can occur due to QJUMP's strict packet prioritisation. To implement this correctly, the priority elevation algorithm would need to be aware of packets in flight and wait a suitable time for these to complete.

3. Some applications (e.g. Naiad [52]) have latency-sensitive control traffic as well as throughput-intensive traffic mixed into the same sockets. This traffic needs to be treated separately [161], but to do so the regulator must be application aware. In my previously published work "Eden" [107] we showed that it was possible to describe application messages at a high level and compile that to a hardware or software interpreter inline with the data path. A similar approach would be needed, although the Eden system would need to be extended to understand Leaky Token Bucket (LTB) which it cannot easily express currently.

4. Some applications may have asymmetrical traffic patterns. For example, an Hadoop shuffle performed on a skewed dataset may transfer many gigabytes to some machines, and only a few kilobytes to others. Instances that move only a few kilobytes may stay at high QJUMP levels while instances that transfer many gigabytes might be downgraded. This would lead to application level race conditions, again as a result of network packet prioritisation. To resolve this, we would require some level of distributed coordination between applications at the same level and potentially an application aware message parser like Eden [107].

5.2 The problem of N

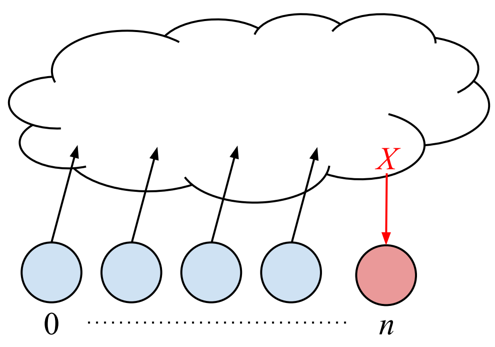
Further issues with QJUMP relate to choosing the value for N; the number of hosts participating in the guaranteed latency QJUMP level. For simplicity, QJUMP assumes that N is equal to the number of hosts in the network. For a pod-scale network this means that N is between 1000 and 5000 hosts. This choice has two limitations:

1. **Throughput** – Since the effective rate given to each application scales inversely proportionally to N, QJUMP offers only a fraction of the available bandwidth at the guaranteed layer (see §4.1 for more details). QJUMP resolves this by forcing applications to chose between either bounded latency at low throughput or high(er) throughput with (more) variable latency. It would be ideal to be able to achieve both high throughput *and* guaranteed latency bounds.
2. **Scalability** – QJUMP is targeted specially at pod-scale networks of 1000-5000 hosts. Pod-scale networks are 10-100$\times$ smaller than full-scale datacenter networks. It would be ideal to be able to offer bounded latency for entire datacenter networks.

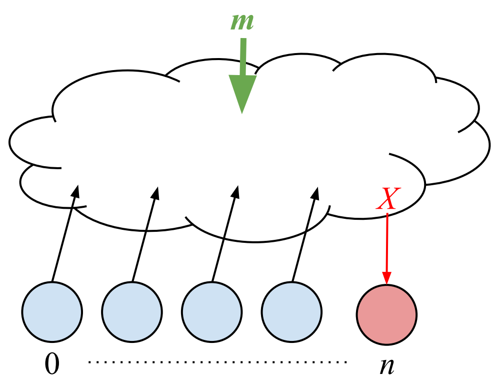
In following sections I discuss several future directions and trade-offs for resolving these limitations.

5.3 Scalability

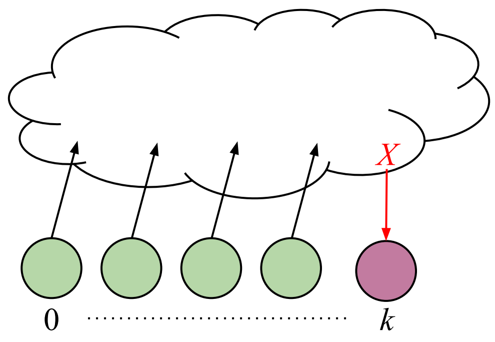
There are several potential approaches to resolving QJUMP's scalability issues. In this section, I describe an approach which requires minimal changes to the QJUMP system, but requires some changes to the datacenter network configuration. This change allows QJUMP to scale to larger networks, but retains the same throughput limitations. In Sections 5.5–5.7, I describe ways of



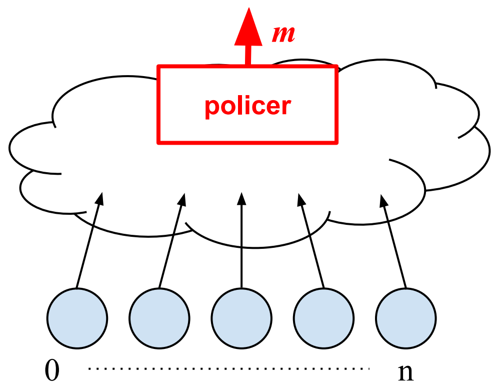
***(a)** An abstract view of a* QJUMP *pod*



***(b)** Incoming link of equivalent capacity m hosts connected to abstract pod*



***(c)** Abstract view of interconnect between pods*



***(d)** Outgoing link of equivalent capacity m hosts connected to abstract pod*

***Figure 5.1:** Abstracting the* QJUMP *pod to form groups of pods with a* QJUMP *domain inside each and bounded latency between them.*

instead modifying QJUMP to improve both its scalability and the throughput restrictions that it imposes.

One approach to resolving the scalability issues with QJUMP is to apply QJUMP to individual pods and then to implement latency bounds between the pods. That is, QJUMP could be used to bound latency inside of each pod and QJUMP could also be used to bound latency between each pod. This is a "QJUMP in QJUMP" approach. The result would be bounded latency between any node and any other node in the network.

To understand how this could work, consider Figure 5.1a. The figure shows an abstract depiction a of QJUMP pod. The pod has N hosts connected to each other via a network interconnect. The dominating source of delays in the network is the fan-in point (X) where all hosts converge (usually the last hop top-of-rack switch). The bound given by QJUMP at this point proportional to $N - 1$.

In Figure 5.1b the abstract view of a pod is extended to include an incoming link. This link is given the capacity of m hosts. The added link increases the potential delays at (X). The delays at (X) are now proportional to $N - 1 + m$ hosts. For example, if m is equivalent to one host, then the delays at (X) are proportional to N. This change allows hosts outside of the pod to act as if they were a single (or more with $m > 1$) machine within the pod. The difficulty now is ensuring that other pods cannot behave as any more than m machines.

Figure 5.1c shows an abstract view of interconnects between pods. The figure shows k pods connected to each via a network interconnect. It should be clear that this figure is nearly identical to Figure 5.1a. Once again, delays will occur at the fan-in point (X). Assuming a QJUMP deployment to control interference, these delays will be proportional to $k - 1$. As with QJUMP deployed in hosts, it is necessary to ensure that each pod is regulated so that it cannot overwhelm the destination pod. In Figure 5.1d I show a Leaky Token Bucket (LTB) see §3.2.3) applied to the output from a pod into the pod interconnect network. This regulator limits traffic out of the pod to a rate proportional to $\frac{m}{k-1}$ resulting in bounded delays throughout the network. Intuitively, this solution abstracts entire pods to act as if they were single hosts and allows multiple pods to be connected together, giving a "QJUMP in QJUMP" approach.

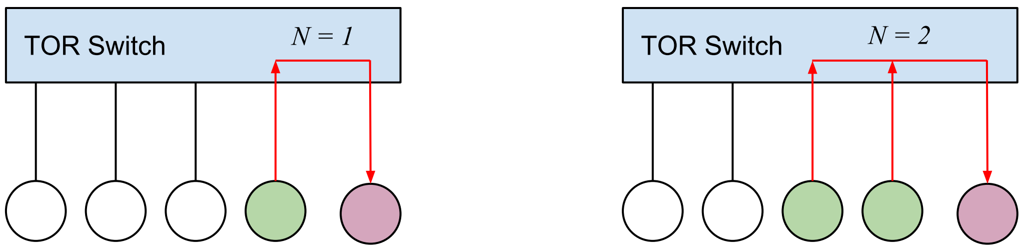
There are two potential problems with the solution: (*i*) it requires an LTB regulator to be inserted between pods in the network and; (*ii*) it further exacerbates the throughput limitations of QJUMP both within, and between pods.

Problem (*i*) is potentially resolvable by using the same technique described in Section 3.4.4. Instead of using a leaky token bucket regulator, a simpler policer could be used. Existing switches can provide traffic policing functions on egress ports[1]. Although this lacks some of the throughput availability of a true LTB regulator, the latency bounds would still hold. Another approach might be to place middle-boxes between pods to implement more complete LTB regulators. This would provide higher fidelity regulation, but would require the middle box to keep up with high throughput and would involve invasive changes to the network. Resolving problem (*ii*), QJUMP's throughput limitations, is the subject of the following sections (§5.5–§5.7).

5.4 The Dynamic Coordination Problem

To implement high-throughput and bounded latency simultaneously requires a system that can make dynamic choices about the value of N, the number of hosts sharing any given network path. For example, consider a datacenter application shown in Figure 5.2. The figure shows an application using several hosts and sharing a single switch. In Figure 5.2a the application is using only two hosts (3 and 4). In this scenario there is only one path through the network between the hosts. N thus has a value of 1 and the application can therefore enjoy line-rate throughput at bounded latency. If another component of the application were to join the system from host 2, N would need be set to 2 and each host could be given 50% of the available throughput, again, at bounded latency. In either case, this is significantly better than the 20% that would be allocated via QJUMP with a fixed N value of 5. The ideal scenario would be for this decision

[1]Cisco and Arista configuration guides. See Appendix A.4.6 and A.2.4.



***(a)** Two hosts on a single switch, the outgoing link not shared, thus $N = 1$* ***(b)** Three hosts on a single switch, the outgoing link is shared between 2 hosts, thus $N = 2$*

***Figure 5.2:** Dynamically assigning the value of N depending on the number of hosts sharing the outgoing link.*

	QJUMP (§4)	EYEQJUMP (§5.5)	FASTJUMP (§5.6)	R3CJUMP (§5.7)
Dynamic N	✗	✓	✓	✓
Coordination	none	end-host	centralised	distributed
Unmod. Application	✓	✗	✗	✗
Unmod. Kernel	✓	✗	✗	✗
Unmod. Hardware	✓	✓	✓	✓*
Topology independent	✓	✗	✓	✓
Source routing req.	✗	✗	✓	✓
Fully distributed	✓	✓	✗	✓
Scalability limit	N	net. topology	arbiter perf.	net. bandwidth

***Table 5.1:** Comparison of the features and trade-offs between* QJUMP, EYEQJUMP, FASTJUMP *and* R3CJUMP. **with caveats, see §5.7.2*

to be made dynamically as hosts/applications come and go. In order to implement this kind of dynamic system, it is necessary for all hosts to agree on the current value of N. Doing so requires some degree of coordination between hosts. The extent of the coordination required will depend on where the coordinator is run, and any assumptions that could be made about the network topology. In the following sections (§5.5–§5.7), I describe three potential dynamic system designs and some of the trade-offs involved in constructing them. These systems are called EYEQJUMP, FASTJUMP and R3CJUMP. Table 5.1 includes a summary of each of these systems compared with QJUMP and the important trade-offs between them. Each system is a potential direction for future work.

5.5 End-Host Network Coordination

In Section 2.4.10 I discussed EyeQ and pHost. Although subtly different in their goals and implementations, both systems make similar fundamental observations; that, in full or near-full bisection bandwidth networks, congestion occurs only at the last-hop Top of Rack (ToR) switch. As a result, both systems propose an end-host coordinated congestion control scheme. Although the details vary, both schemes essentially control the rate at which sources can issue packets into the network by pushing back on senders. There are some issues with each scheme: pHost implements a packet by packet schedule where requests are arbitrated at destination host arbiters. pHost therefore suffers from latency performance artefacts similar to Fastpass, which introduced a 3$\times$ latency penalty in the best case. On the other hand, EyeQ measures and issues rate allocation messages every 200μs and includes a 10% bandwidth overhead margin. This limits the throughput available as well as the period over which interference can be controlled. Despite their implementation shortcomings, both approaches show promising results.

Drawing inspiration from the promising results of EyeQ and pHost, and taking into account the lessons learned in QJUMP, I now propose EYEQJUMP, a host-based coordination system extension to QJUMP. Like EyeQ, EYEQJUMP would use an end host based arbitrator that issues rate allocation messages to sources. Unlike EyeQ, these messages would be based on explicit rate allocation requests issued by hosts rather than estimates over a measurement interval. Like pHost, a source would request (and most often be granted) permission to send before it can begin transmitting. Unlike pHost, only one message is required to reserve capacity for the lifetime of the flow. Once permission has been granted, the sender could continue to send, but its rate would be regulated by flow rate adjustment messages from the destination. Finally like QJUMP, EYEQJUMP would take into account bounded delay scheduling across the network by employing the network epoch calculation to limit the impact of interference. Unlike QJUMP, EYEQJUMP could dynamically vary the value of N at destination hosts which means that throughput can be as high as line-rate whilst retaining bounded delay.

5.5.1 Theory of Operation

QJUMP (§4) required only minimal information about the topology of the network. Specifically, it made the pessimal assumption that the worst case queueing delay would occur along a single path in the network shared by all hosts. By contrast, EYEQJUMP would make the assumption of a full bi-sectional bandwidth network such as those shown in Figure 3.15b and 3.15d. It would further assume that the network implements a fair load balancing mechanism across the links such as packet spraying [110]. This means that the aggregate bandwidth of multiple links could be taken as if they were one link. For example, the four 40Gb/s links shown in Figure 3.15b and 3.15d could be assumed to be effectively one 160Gb/s link. This alters the results of queuing equations (Eq. 3.22/3.10):

$$d_n et = \sum_{k=1}^{K} \frac{n_k P_{max}}{R_k} = \frac{16P_{max}}{160\text{Gb/s}} + \frac{112P_{max}}{160\text{Gb/s}} + \frac{143P_{max}}{10\text{Gb/s}} = \frac{2416P_{max}}{160\text{Gb/s}} = 15.1\text{ns/b} \qquad (5.1)$$

where:

- d_{net} is the network wide delay bound (in seconds)
- n_k is the maximum number of sources from all previous switches fanning-in to switch k

- P_{max} the maximum packet size (in bits)
- r_k is the output rate of the switch (in bits per second)

If we assume minimum packet size of 64B, then the maximum end-to-end D is:

$$D = 15.1\text{ns/b} \times 64\text{B} = 7.731\mu\text{s} \tag{5.2}$$

The network epoch e_{dcnet} is given by (Eq. 3.13):

$$e_{net} = \frac{n_k P_{max}}{r_k} = \frac{144 \times 64\text{B}}{10\text{GB/s}} = 7.373\mu\text{s} \tag{5.3}$$

which means that each host can transmit at an average rate of (Eq. 3.14):

$$R_{ave} = \frac{P_{max}}{e_{dcnet}} = \frac{64\text{B}}{7.373\mu\text{s}} = 69.44\text{Mb/s} \tag{5.4}$$

That is, each host could transmit at the maximum allowable fair share rate of (10Gb/s / 144 hosts = 69.44Mb/s) and achieve a guaranteed latency of $\leq 7.731\mu$s in the network.

The change in network assumptions would yield a limited improvement in latency (15.1ns/b vs 15.2ns/b see §3.4.7 for more details) over QJUMP. However, since both network designs are bottlenecked at the destination host, the change is not significant enough to be meaningful. To achieve a meaningful change, the number of hosts N participating in the system would needs to be made dynamic.

To vary the number of hosts dynamically, each host can be made aware of the network topology at start-time and keep a variable x, which is the number of connections to it. Since the host cannot know the extent of the queueing happening in the first two stages of the network, it would assume the worst case for the first two terms of equation 5.2. For the final (dominating) case, it would leave the fan-in parameter (x) as an unknown, resulting in the following (from Eq. 3.10):

$$D = \sum_{k=1}^{K} \frac{n_k P_{max}}{R_k} = \frac{16 P_{max}}{160\text{Gb/s}} + \frac{112 P_{max}}{160\text{Gb/s}} + \frac{x P_{max}}{10\text{Gb/s}} = \frac{(16x + 128) P_{max}}{160\text{Gb/s}} \tag{5.5}$$

The network epoch is thus given by (from Eq. 3.10):

$$e_{net} = \frac{x \times 64\text{B}}{10\text{GB/s}} \tag{5.6}$$

which means that each host can transmit at an average rate of (from Eq. 3.11):

$$R_{ave} = \frac{P_{max}}{e_{net}} = \frac{64\text{B}}{1} \times \frac{10\text{GB/s}}{64\text{B} \times x} = \frac{10\text{Gb/s}}{x} \tag{5.7}$$

When a new connection is made to the host, the host would increment the value of x and recalculate the correct transmission rates for all current connections. These recalculated rates would be returned as an update to all hosts which could then adjust their rates appropriately. The rate adjustment could be timed to happen one epoch before the new host begins transmitting so that interference in the network would be controlled, even as rates are adjusted.

Applying Equations 5.5 and 5.7 would result in a marked performance improvement over QJUMP. As an example, assuming that $P_{max} = 64B$ and that there is only source transmitting to the destination host, EYEQJUMP would provide a latency bound of 0.461μs at a rate of 10Gb/s (compared to 7.731μs at 0.069Gb/s).

One final improvement could be made to EYEQJUMP. In Chapter 4, I observed that not all applications require the maximum available throughput from the network. For example, PTPd only issues one packet every second. If PTPd shared a destination host with only one other application, then the value of x (from above) would be 2. That is, PTPd would be allocated 50% of the network bandwidth, giving it 5Gb/s of a 10Gb/s network. This is clearly overkill for an application that has a transfer rate of 1200b/s (bits per second!). Instead, if PTP used a minimum network allocation ($\frac{\text{10Gb/s}}{\text{144 hosts}}$) it would consume only 0.069Gb/s, and the second application could use the remaining 9.931Gb/s. In this toy example with only 144 hosts, the gains of this modification are substantial. Rather than consuming 50% of the network capacity, PTPd would consume less than 1%. In a pod sized network with 5000 machines, the gains would be even more significant (from 50% down to 0.2%).

The preceding improvement can be implemented if applications issue a "maximum throughput request" along with their connection requests. This request must be issued as a multiple of the minimum fair-share allocation so that bounded latency can still be implemented safely. This throughput request could then be used to provide a max-min fair share [162] allocation to users of the network. In a max-min allocation, all users receive either their maximum required share of the resource, or an equal share of whatever resource is remaining after maximum allocations have been made. As a further extension, users could also request their minimum throughput requirement. If this requirement cannot be met, the network could reject the request leading to more predictable latency *and* throughput performance. Such a scheme would emulate the behaviour of a circuit switched network over a commodity datacenter packet switched network infrastructure.

5.5.2 Architecture

QJUMP was expressly designed to minimise the changes to hosts and applications. Implementing EYEQJUMP would require more extensive modifications. It would require that applications specify their maximum/minimum rate requirements along with the usual BSD sockets calls to start a new connection. The regular sockets interface does not support this feature which means that both the interface and the applications would need to be modified to accept this. Implementing EYEQJUMP would require a more sophisticated source regulator implementation which:

- issues connection request messages before a new connection is established
- listens for new rate allocation messages and updates its parameters as they arrive
- issues periodic heartbeat messages to keep the destination informed of liveness
- issues shutdown messages when a flow/connection terminates

Finally, implementing EYEQJUMP would require that destination hosts be modified to:

- listen for new connection allocation messages
- keep track of the network state for allocating resources to sources and calculate max-min fair share network allocations
- issue rate allocations to sources as rates change and terminations to sources whose rate allocation cannot be satisfied.
- keep track of liveness in case a source crashes without terminating its flows safely

5.5.3 Scalability

EYEQJUMP primarily attempts to solve QJUMP's throughput limitations whilst retaining its latency bounds. EYEQJUMP works because senders are decoupled from each other through the network and are only coupled to at destination hosts, which determine a dynamic value of N. A further effect of this decoupling is that EYEQJUMP is no longer limited to networks of pod-scale. Provided that the network can sustain full-bisection bandwidth between all hosts, the scale of EYEQJUMP is essentially unlimited. Providing full-bisection bandwidth to very large networks is by no means impossible, but it is very expensive which is why large operators like Facebook and Google tend not to do this. As network capacities grow[2] it is likely that network capacity will begin to outstrip host capacities. It may then be possible to build large scale, undersubscribed, full-bisection bandwidth networks using cost effective components.

5.5.4 EYEQJUMP Limitations

By introducing end-host based coordination, allowing changes to the kernel and applications, and by assuming a full-bisection bandwidth network, EYEQJUMP could achieve bounded latency at high throughput and provide better scalability than QJUMP. The EYEQJUMP design has four major limitations:

1. The reservation scheme used by EYEQJUMP emulates circuit switching mechanisms over a packet switched network. These mechanisms are not work conserving. This drawback could be addressed by using a similar priority scheme to QJUMP. Applications that need bounded latency could do so by reserving capacity at a high-priority level (non work conserving), while applications that do not need bounded latency/throughput could continue to use a low-priority level as normal (work conserving). This strategy is similar to that used by QJUMP and would segregate the same network infrastructure into two virtual networks: (*i*) a high-priority bounded-latency circuit switched network, and; (*ii*) a low-priority work-conserving packet-switched network.
2. The assumption of a full bisection bandwidth network is not easy to justify in a datacenter network, pod-scale or otherwise. The existing and operational designs discussed in Section 2.1.3 are based on a 3:1 oversubscription ratio at the ToR switch. To be applicable to existing datacenter networks, a system could not make this assumption.

[2]Work has already begun on 400Gb/s Ethernet standards. See Appendix A.15.1.

3. The latency bound offered by EYEQJUMP is not optimal. End hosts have no knowledge of network utilisation and therefore would need to assume worst case delays through the intermediate switches in the network. The contribution of these delays can be significant when the number of hosts is small. For example, with only one host using a path, the worst case queueing delay offered is still 0.4μs (from Eq. 5.5) whereas it should be closer to 0.15μs.
4. A complete EYEQJUMP implementation would need to be optimised to operate at full line-rate. To do so, the traffic regulator implementation and, ideally, the reservation protocol would need to be integrated into the TCP protocol. This would require significant engineering efforts.

5.6 Centralised Network Coordination

In the preceding section (§5.5) I proposed an extension to QJUMP called EYEQJUMP. EYEQJUMP improved upon QJUMP by offering hosts the ability to dynamically negotiate channels through the network. These channels would have guaranteed latency bounds and could operate at up to line-rate throughput. The EYEQJUMP design has two major shortcomings. First, it is architected on the assumption of a full bi-section bandwidth network. As discussed in Section 2.1.3, deployed datacenter networks are typically oversubscribed. This means that a full bisection bandwidth network assumption is unlikely to hold. Secondly, the dynamic allocation scheme proposed by EYEQJUMP only operates at the final switch hop. EYEQJUMP must therefore assume worst case queueing delays in the core of the network. Both issues can be resolved by drawing inspiration from the Fastpass system, described more completely in Section 2.4.8.

In brief, Fastpass introduced a centralised mechanism for arbitrating access to the network. Hosts wishing to send a packet must first send a request to an arbitrator. The arbitrator calculates a maximal-matching for the network and allocates a timeslot to the host for transmission of the packet. The maximal-matching problem is computationally expensive, however, Fastpass shows that it is possible to schedule over 2Tb/s of traffic on an 8 core arbitrator[3]. Fastpass claims to offer a "zero-queue" network, although this claim is subtly misleading. While the system ensures that there are no collisions in the network, queuing still occurs in the arbitrator, and in hosts while they are waiting for their allocated time slots. To make the matching problem tractable, Fastpass requires a rearrangably non-blocking network design. This places similar constraints on the network to pHost, EyeQ (see §2.4.10) and EYEQJUMP.

In response to the problems with EYEQJUMP, I now introduce a further proposal to extend QJUMP, called FASTJUMP. The FASTJUMP design is modelled on a similar concept to Fastpass, but addresses many of its shortcomings. Like Fastpass, FASTJUMP would employ a centralised arbitrator. When a host wishes to send packets into the network, it would first contact the arbitrator to negotiate a channel through the network. Unlike Fastpass, the FASTJUMP

[3]The authors show that their allocator can schedule 2.21Tb/s of full sized frames, which is equivalent to 221 endpoints.

arbitrator would be flow based rather than packet based. Negotiation would only be triggered by the addition or removal of a flow. This means that FASTJUMP could operate on networks of arbitrary topology. Furthermore, the design would use the network scheduling model described in Chapter 3 and implemented in QJUMP. Unlike Fastpass, this scheduler uses of a model of network switches to provide bounded latency. Employing this model would make FASTJUMP less computationally expensive and therefore more scalable than Fastpass. The negotiation process used could be similar to that of EYEQJUMP. Like EYEQJUMP, the arbitrator would hold the state of the network in local memory. However, unlike EYEQJUMP, the arbitrator would hold the entire network state rather than just the state for a single destination. Like EYEQJUMP, FASTJUMP clients would send requests to the arbitrator which would respond by issuing rate adjustments to all hosts so that the bounded delay could be maintained.

5.6.1 Architecture

Architecturally, FASTJUMP would be very similar to EYEQJUMP (see §5.5 for further details). It would also require the same (extensive) modifications to applications and kernels. There are only two major differences between the FASTJUMP and EYEQJUMP architectures:

1. FASTJUMP hosts would issue flow start/stop requests to a (more sophisticated) centralised arbitrator, rather than the destination-host arbitrator design used in EYEQJUMP.
2. In addition to the assumptions made by EYEQJUMP, FASTJUMP would further assume that a source routing scheme such as MPLS [163] or GRE [164] is available in the network.

Like QJUMP and EYEQJUMP, FASTJUMP would rely on the assumption that datacenter networks have relatively static topologies. On this basis, the arbitrator could keep a graph in memory representing the entire network topology. Each edge in the graph would hold a list of the flows currently using the edge, along with their requested and allocated rates. Unlike EYEQJUMP this topology is not assumed to have full bisection bandwidth. To add a new flow to the network, the arbitrator would need to perform at least the following four operations:

1. Traverse the network graph to find a path/paths between the source and destination. Even for a large datacenter the number of paths/edges in the graph is relatively small. For a network of 50,000 hosts using the Facebook Fabric topology, the graph has less than 55,000 vertices and 150,000 edges[4].
2. Choose a path for the new flow to be added into the network. This choice could affect other flows that share any switches or links with the new flow.
3. Calculate a new max-min allocation (or otherwise) for all flows in the network. If the new allocation would cause a previously allocated flow to fall short of its minimum throughput requirements, the new flow allocation would need be to rejected.
4. Inform all hosts whose rate allocations have changed or inform the requester that the flow addition request has been rejected.

[4] In comparison, our previously published graph analytics work [27], the smallest workload had 3M vertices and 117M edges.

The major difference between the EYEQJUMP design and the FASTJUMP design is the need for a more sophisticated, network-wide, path planner and rate allocator. There is wide scope for future work on these path planners/allocators. To guide ongoing work, I now propose three potential approaches to the path planning / rate allocation problem. All approaches provide a max-min fair allocation across the network, however, each approach makes different trade-offs with respect to the cost of the implementation, and the allocation of network resources.

5.6.1.1 Optimal Planner

The optimal path planner would carry out three steps to add a new flow to the network:

1. It would use a graph search algorithm to find all candidate paths between the source and the destination. Datacenter network graphs are relatively small, so a simple depth/breadth-first search may be sufficient. The complexity of a depth/breadth-first search is $O(L + V)$ where L is the number of links in the network and V is the number of vertices. L and V are small ($< 200,000$) in a reasonable datacenter network comprising $\approx$50,000 hosts.

2. The impact of adding the additional flow to the candidate path would then need to be calculated for the entire network. This calculation could be performed by applying the "water filling" [165] max-min fair share allocation algorithm (or similar). The water-filling algorithm has a complexity of $O(FL + F^2)$, where F is the total number of flows in the network and L is the number of links [65]. It would need to be repeated P times, where P is the number of distinct path candidates found between the source and the destination. If the number of flows is very large, this step may limit the overall performance. At termination of the max-min allocation, each flow would be given a score to determine how much it was affected by the addition of the new flow. As an example, the fraction of the requested rate allocated to each flow could be used as a metric for scoring. These scores could then be aggregated to give a final, network-wide score for adding the new flow into the candidate path.

3. Finally, the planner would choose the candidate with the greatest score overall, thereby optimising for minimal impact on the network/other flows. This scheme would ensure that new flows admitted to the network would cause the least disruption and would maximise overall network utility.

5.6.1.2 Greedy Planner

The greedy planner is an heuristic algorithm that acts much like TCP when a new flow is added. In this planner, a modified Dijkstra's algorithm would be applied to search for an admissible path between the source and the destination. Dijkstra's algorithm has a complexity of $O((L + V)\log(L))$ where L is the number of links (edges), and V is the number vertices. A single link is considered at each step of the algorithm. The new flow would tentatively be added to this link, and a max-min allocation performed for all flows that share it. If the max-min allocation fails to find a solution that meets all flows' minimum rate requirements, then the link would be marked as inadmissible. Performing a max-min allocation on a single link is much simpler than the network-wide equivalent. If all flows sharing a link are stored in sorted order, the max-min allocation step can be performed in $O(f \log(f))$ time, where f is the number of flows on a single link. In general f is likely to be much smaller than F, the total number of flows in the network, and, L and V are likely to be small ($< 200,000$) in a reasonable datacenter network.

This modified Dijkstra's algorithm forms a path through the network by optimising for the maximum available rate allocation. Once a complete path has been found, the flow is assigned the maximum achievable rate common to all links that it traverses, thus maximising the flow's rate allocation. At this stage, the water-filling algorithm could then be applied to rebalance all other flows' rate allocations. This would ensure that all flows receive their maximum available rate and that no link has its capacity exceeded. As before, the water-filling algorithm has a complexity of $O(FL + F^2)$, where F is the total number of flows in the network, but this time it would only need to be run once. The F term is still likely to dominate the performance of the planner as a whole.

As a faster alternative, the water-filling step could be omitted. Instead, only those flows whose rate allocation decreased (as a result of the new flow being added) would have their rates adjusted. This would ensure that no link has its capacity exceeded, but no further steps would be taken. The network would not be globally rebalanced which means that some capacity might be wasted. However, if the rate at which flows are added/removed is high, the network may approximate an optimal allocation over time.

The greedy planner would act much like networks currently do when new TCP flows are added. These flows greedily consume as much available resource as they are able to on the path with no regard global optimality. We've become happy with this TCP approach in datacenters and in the Internet so this may be a suitable planner despite it's lack of global optimality. It is also likely to be faster than optimal planner because global optimality is not considered.

5.6.1.3 Hippocratic Planner

While the greedy planner would emulate TCP by maximising rate allocation for a newly added flow, this is not the only choice with a centralised planner. As an alternative, I propose the *hippocratic planner*. This planner is derived from famous greek philosopher's "do no harm" principle. As with the greedy planner, Dijkstra's algorithm would be applied to search for an admissible path between the source and sink. However, like the optimal planner, at each step, a cost metric would computed. Again, each flow could be given a score which is the fraction of its requested and actual rate allocations and these scores could be summed to give an estimate of the "goodness" of a given link. The new flow would then be tentatively added, a max-min allocation performed, and then all flows on the link would be re-scored. The "cost" of taking a given path would be calculated by the change in cost between the initial state and the tentative state. The hippocratic planner will have the same runtime cost as the greedy planner, but would attempt to minimise the "harm" done to other flows. It would thus achieve a compromise between the greedy planner and the optimal planner approaches. Like the greedy planner, it is an heuristic algorithm and is not guaranteed to be globally optimal. Also like the greedy planner, it likely to be faster than the optimal planner.

The preceding list of planner algorithms (§5.6.1.1–§5.6.1.3) are by no means definitive, nor can any strong statements be made about their relative performance. Further work would be needed to evaluate the efficacy of the approaches and validate their performance.

5.6.2 Scalability

The question of scalability for FASTJUMP is more complicated than that of EYEQJUMP. Like EYEQJUMP, the FASTJUMP design primarily attempts to solve QJUMP's throughput limitations whilst retaining its latency bounds. In a FASTJUMP implementation, senders would be decoupled from each other and from destination hosts which means that a network wide value of N can be determined dynamically. Once again, a further effect of this decoupling is that the system's performance is no longer globally limited by the number of hosts in the network. However, like Fastpass, FASTJUMP uses a centralised arbitrator. The scalability of the system would therefore be coupled to the performance of the arbitrator.

The existing Fastpass implementation is claimed to schedule over 2.2Tb/s of traffic on an 8 core arbitrator. Assuming 10Gb/s hosts, this implies a limit of 220 hosts and only 55 hosts at 40Gb/s. However, Fastpass is limited because it attempts to schedule every packet while the FASTJUMP design would only schedule flows. This makes FASTJUMP non work-conserving, but does imply orders of magnitude more scalability. Ultimately the question of scalability for FASTJUMP is an implementation specific one and cannot be answered without an adequate prototype to test with.

5.6.3 FASTJUMP Limitations

FASTJUMP is a proposal for an extension to the ideas presented in QJUMP and EYEQJUMP. The design retains the same benefits of bounded latency and interference isolation offered by QJUMP and EYEQJUMP. Like EYEQJUMP, it would improve over QJUMP by allowing applications to negotiate channels with bounded latency and high throughput. However, also like EYEQJUMP, FASTJUMP would achieve its higher throughput by eliminating the simplicity and immediate deployability of QJUMP.

The FASTJUMP architecture is inspired Fastpass (see §2.4.8) and, as a result, it also suffers from some of the same limitations. Firstly, both systems are reactive. Fastpass and FASTJUMP both wait for applications to contact them before making a scheduling decision. This means that significant latency can be incurred to establish a new connection. If the cluster scheduler could be made aware of an application's networking needs, scheduling/planning work could be done at cluster scheduling time. This is in principle similar to the Silo system [153]. Secondly, like any centralised coordination system, FASTJUMP would also introduce a central point of failure. The Fastpass authors suggested that this could be resolved by including a backup coordinator. However, to operate correctly, both controllers would need to implement a consistency protocol between them to ensure agreement over shared state and some mechanism to alert all hosts about leadership changes. These are both non-trivial distributed systems problems.

5.7 Distributed Network Coordination

The previous section (§5.6) described a proposal called FASTJUMP. FASTJUMP could provide better throughput opportunities than QJUMP over an arbitrary network topology which was not supported by the EYEQJUMP design. However, the FASTJUMP design is based on a centralised solution which also makes it a central point of failure. While the failure properties of FASTJUMP could be improved by the introduction of a backup coordinator, this would introduce further complications. The primary and backup systems would need to synchronise with each other and agree upon shared state in a consistent way. Failover would need to be handled carefully so that all hosts are made aware of the transition from primary to backup coordinator. During this transition period, in-flight requests would also need to be carefully handled. All of these problems fall into the domain of distributed consistency systems.

One of the benefits that QJUMP introduced was the ability to build simpler distributed systems using its bounded-latency property. A further benefit of QJUMP is that its latency bound applies to broadcast messages. I demonstrated these benefits in the construction and evaluation of a high performance two phase commit system (see §4.5.1.4). In the following section I outline a final proposed extension to QJUMP called R3CJUMP[5].

R3CJUMP would implement a fully distributed version of FASTJUMP built over the top of QJUMP. The changes between FASTJUMP and R3CJUMP would be minimal. R3CJUMP would use the same planner system as FASTJUMP but introduce an eventual-consistency system for distributing flow state to all hosts. When a host wishes to send a new flow, it would perform its rate and route calculations locally. Before beginning to transmit, the host would issue a broadcast message to all remote hosts. This would alert other hosts of the new flow and cause the remote hosts to adjust their rates for any flows which are affected. Collisions in this request could be handled by using a fixed priority order and timing information from the guarantees provided by QJUMP.

The R3CJUMP design shares many similarities with the Rack Routing and Congestion Control system (R2C2) [65][6]. R2C2 is specifically designed for deployment in Rack-Scale

[5]Resilient Rate, Routing and Coordination Jump

[6]Indeed, I worked as an intern at Microsoft Research Cambridge during its development and worked with many of its coauthors. It is likely that some degree of cross-pollination of ideas occurred and the R3CJUMP design was developed parallel with R2C2.

systems. The authors assume a 3D torus network topology where all nodes act as both hosts and switches. They argue that, the cost of broadcast is low in this network topology. They therefore employ broadcast as a mechanism for distributing state updates in the network. Flow initiation/termination events are broadcasted to all nodes. Each node, including the sender, calculates flow-rates locally in response these messages. The sender enforces this rate locally. In "Discussion and Future Work" the authors consider the possibility of using a broadcast mechanism in regular switched networks. This is precisely the proposal I explore with R3CJUMP.

5.7.1 Architecture

In R3CJUMP, all hosts would act like the coordinator used for FASTJUMP. Each host would store a complete network graph in local memory including all the flows on every link. The system would implement a simple eventual-consistency protocol to keep the state of all coordinators synchronised. This would have two benefits: (*i*) graph traversal computations are distributed and localised to each host, thereby improving performance and scalability, and reducing queueing time in the system, and; (*ii*) as in QJUMP, the system would again be fully distributed, thereby improving its fault tolerance properties.

The consensus protocol used by R3CJUMP would rely on the assumption of bounded-latency network provided by QJUMP for very fast convergence. Networks are lossy environments and network protocols are already designed to cope with this. Unlike bank account balances or airline reservations, inconsistencies or faults can be tolerated and even detected. This means that a coordination system can be "best-effort" rather than strict. This also simplifies the design and improves the potential performance.

In R3CJUMP, when a host wants to make a new connection, it would inspect its own local copy of the network graph and compute the best path from source to destination (potentially using one of the planner algorithms proposed in Section 5.6). Assuming that a path is found, it would then send a short broadcast message to all hosts that updates their graphs as well. The broadcast message would be sent at high priority over the guaranteed latency messaging layer offered by QJUMP. R3CJUMP would use the fact that broadcast messages are included in the latency guarantee offered by QJUMP. This means that after one network epoch (as defined in §3.2.1) sending hosts could assume that their message has arrived at all destinations[7]. Furthermore, after two network epochs, sending hosts could further assume that all conflicting messages that were in-flight have arrived. This would allow the conflict resolution algorithm to have a fixed upper bound on the time it waits before committing a new flow to the network state. Conflict resolution could be applied by any means, but for simplicity, each host could be given a fixed host-id (e.g. derived from the IP address) and conflicting messages could be resolved simply by taking the highest/lowest host-id value.

It is possible that some of the graph update messages may be lost or corrupted in the network and/or that some hosts may fail and lose their copy of the network state. To cope with

[7]or failed to arrive.

this, long lived connections could issue occasional broadcast "heartbeat" messages which include the connection state information. This would ensure that all hosts eventually see an up-to-date view of the network graph and that hosts can be added and removed dynamically. This would form a primitive, but fast, eventual-consistency coordination system.

In addition to heartbeats, individual flows could use aggressively set Explicit Congestion Notification functions in switches to quickly detect conflicting inconsistent states in the network. For example, consider the case where two hosts share a switch and wish to use a common a link of 10Gb/s. If one host fails to receive the message, then an aggregate of 15Gb/s would be pushed into the switch, but only 10Gb/s is available on the output link. Since the link cannot handle this rate, queueing would occur in the switch. Any queueing beyond the limit guaranteed by QJUMP indicates a problem. This queueing could be detected by monitoring ECN status bits and reacting by issuing broadcast state updates. Again, this would result in a fast eventual-consistency coordination system.

5.7.2 Limitations

In the R3CJUMP design, there is a natural limit to the performance. The distribution of the consensus messages is governed by the QJUMP epoch. This means that a host cannot issue more than one flow start/stop or heartbeat message per epoch. Assuming a pod-scale network of 1,000 hosts and a message size of 64B, Equation 3.13 gives a network epoch of 256μs. This means that each host can issue over 19,000 messages per second.

The obvious limitation of both R2C2 and the R3CJUMP design is that the broadcast coordination traffic consumes network bandwidth. Using broadcast messages means that costs scale with the number of destination hosts in the network. That is, a 64B message has an equivalent cost of 64B$\times$1,000 hosts $\approx$ 64kB on the network. The authors of R2C2 analysed this problem extensively. They concluded that the overhead of such a system, given a reasonable datacenter workload, is only 1.3%. However, the authors assumed a 16B broadcast message and a 512 node network. I have so far assumed that messages are 64B in size. Extending their analysis to the R3CJUMP design yields an overhead of 1.3% $\times 4$ $\times 2$ = 10.4%. Further extending to a 5,000 node pod yields an unacceptably high overhead of 52%. There are three potential mitigations to this problem:

1. The R2C2 authors demonstrate that a 16B message can be used. By scaling the message size down from 64B to 16B, the overheads would reduce to 2.6% and 13% for a 1000 and 5000 node cluster respectively. These are more tolerable, but would prevent using standard Ethernet networks.
2. R2C2 and R3CJUMP have different goals. R2C2 aims to be a complete network stack for rack-scale computers whereas R3CJUMP aims to add bounded latency, high throughput messaging to commodity datacenter networks. As such, R3CJUMP does not need to schedule all of the traffic in the network. Short flows could be sent using QJUMP mechanics, thus avoiding the broadcast setup costs, and longer flows that are not latency sensitive

could continue to use low priority, best effort services. This would reduce the fraction of flows that require broadcast messages and thus the resulting overhead.

3. The broadcast coordination messages could be offloaded into a separate, simple broadcast/aggregate network such as that described in my previous work R2D2 [166]. This network has no layer 2 switching capacity. It simply aggregates all messages and broadcasts them. The simplicity and speed of this network would make it ideal for this purpose and recent hardware products such as the Exablaze Fusion[8] and Metamako MetaMux[9] would make it realisable today.

5.8 Conclusions

This chapter has explored a number of potential modifications and extensions that could be made to the QJUMP system. These extensions would allow QJUMP to be easier to operate and/or offer bounded latency at higher throughput and/or higher scalability. However, each proposal also comes with further costs and complexities. All of the extensions would require more invasive changes to be made to the application and/or kernel of each host. Furthermore, each extension would add operator specific trade-offs and complexities. The EYEQJUMP proposal makes the assumption that the network has full-bisection bandwidth support. This is not true in many existing datacenter networks, but could be made true at additional cost/complexity. The FASTJUMP proposal adds a central point of failure to the network and may limit the rate of connections that can be made. Finally, the R3CJUMP proposal adds resilience, but makes the assumption that broadcast traffic can be used for coordination. This would reduce the effective size of the network that it could service, and add extra cost in terms bandwidth resources. Ultimately the choice of trade-off, if any, is an operator specific one which must be tailored to the circumstances of deployment. Further development, implementation, testing and evaluation of these extensions is left to future work.

5.9 Chapter Summary

- I discuss the two major flaws with the QJUMP system presented in Chapter 4: (*i*), that it requires application developers to specify the throughput requirements of their applications and; (*ii*), that it pessimistically assumes that all hosts participate in the bounded latency layer.

- To resolve (*i*), I propose a mechanism for automatically determining and setting the correct application rate and QJUMP level.

- To understand (*ii*), I discuss the problem of N, the number of hosts participating in the bounded latency layer (§5.2). QJUMP uses a static value for N, which is configured to be

[8]Exablaze ExaLink Fusion marketing materials. See Appendix A.9.1.
[9]Metamako MetaMux 48 marketing materials. See Appendix A.20.1.

equal to the total number of hosts. This limits the throughput available to hosts and limits potential scalability.

- I discuss a proposal for embedding “QJUMP in QJUMP” to provide scalability by linking multiple pods together (§5.3). While this method can be implemented with minimal changes, it exacerbates QJUMP’s throughput limitations.
- To resolve the throughput limitations a dynamic value for N is required, which implies the need for coordination. I then propose three potential coordination solutions which set dynamic values for N: EYEQJUMP, FASTJUMP, and R3CJUMP.
- Drawing inspiration from past work, EyeQ [89] and pHost [108], I show that by assuming that the network has full bisection bandwidth, only destination hosts are needed to make decisions regarding N (§5.5.1).
- I describe an architecture called EYEQJUMP for implementing this idea (§5.5.2), which requires significant changes to the QJUMP system.
- Taking inspiration from past work “Fastpass”, I show that by assuming a centralised coordinator the restrictions of EYEQJUMP can be lifted.
- I describe an architecture called FASTJUMP for a centralised coordinator (§5.6.1). This architecture assumes a network support for source routing and requires a more comprehensive route planner.
- I outline three potential route planner designs: (*i*) an optimal planner (§5.6.1.1), (*ii*) a greedy planner (§5.6.1.2) and; (*iii*) an heuristic “hippocratic” planner (§5.6.1.3).
- Drawing inspiration from past work R2C2 [65], I show that it is possible to implement fully distributed control to reduce the restrictions of QJUMP and EYEQJUMP, but overcome the central point of failure from FASTJUMP
- I describe an architecture called R3CJUMP for a distributed coordination system (§5.7.1).
- I describe a consistency protocol based on QJUMP to distribute state. This allows all hosts to operate as both sources and coordinators (§5.7.1).
- R3CJUMP relies on broadcast messaging which can be expensive. I discuss the limitations of this approach and provide an estimate of the costs for a reasonable datacenter workload (§5.7.2).

Chapter 6

Conclusions

IN this dissertation, I have thoroughly analysed, discussed, and addressed the problem of network interference in datacenter networks. I began by exploring the nature and function of modern, warehouse-scale datacenters. By collating a range of public and published information, I constructed a detailed picture of the internal operation of hyper-scale datacenter facilities. Each facility comprises 50,000–100,000 x86 hosts, which are networked together using commodity Ethernet / IP networks. These hosts run a collection of software including distributed coordination, distributed data access, and distributed applications/programming frameworks. I concluded that the tightly coupled, yet distributed nature of datacenter applications makes them susceptible to interference in the network.

To determine the potential impact of network interference, I constructed and ran a suite of micro-benchmark experiments using representative RPC and bulk transfer applications respectively. By doing so, I concluded that network interference is an application-measurable effect and validated the intuition that network interference occurs as a result of congestion in shared switch queues.

To understand the causes and potential solutions to network congestion, I reviewed the history of congestion control, from Van Jacobson's original scheme for TCP, to modern varieties such as Datacenter TCP. I also presented a range of academic contributions to congestion control, especially those with an emphasis on tail-latency and/or network interference in datacenters. Where possible, I tested these systems for their ability to control network interference using a selection of representative datacenter applications. I concluded that current congestion control schemes fail to resolve network interference because they do not take into account network multiplexor (switch) scheduling behaviour.

As a result, I reviewed several theoretical and practical approaches to network scheduling. I concentrated on systems that approximate Generalised Processor Sharing (GPS) because GPS provides theoretically perfect scheduling performance. I discussed several realisable schemes that approximate GPS including Fair Queueing (FQ), Weighted Fair Queueing (WFQ), Packet-by-packet Generalised Processor Sharing (PGPS), Worst Case Fair, Weighted Fair Queueing (WF^{2}Q), and others that fall into the class of latency rate (LR) servers. I concluded that network scheduling techniques can be used to provide approximate isolation in the network which would have the effect of bounding latency and thus resolving network interference.

Unfortunately, GPS approximation solutions require hardware support within switches/routers. This makes the above schedulers impossible to deploy in datacenter networks that are constructed out of commodity hardware components. In response, I developed a new scheduling system based on a simple model of commodity Virtual Output Queue (VOQ) datacenter switch architectures. Using this model, I derived the necessary traffic conditions required to provide bounded delay across a network of switches, and developed the Leaky Token Bucket (LTB) regulator to enforce these conditions. Combining the fluid-flow LTB regulator with a simple, discrete datacenter switch scheduler model results in similar performance to combining a token bucket regulator and PGPS/WFQ scheduler in the same environment. I thus concluded that it is possible to resolve network interference in datacenter networks using network scheduling.

To demonstrate that network scheduling is practical in datacenter networks, I implemented QJUMP. QJUMP applies the LTB regulator at each host and statically configures the regulator based on the total number of hosts in the network. QJUMP ensures that network latency remains bounded and that network interference is therefore tightly controlled. The problem with this static approach is that it limits the available throughput at each host to an unacceptably low level. To resolve this problem, QJUMP uses network enforced priorities and gives users the opportunity to trade latency variability for throughput. I concluded that the solution to QJUMP's throughput problems is viable for a variety of datacenter applications, but nevertheless lacks the ability to offer users bounded latency at high throughput.

To resolve QJUMP's throughput limitations, I proposed three new designs for future work: (*i*) EYEQJUMP, (*ii*) FASTJUMP and, (*iii*) R3CJUMP. The EYEQJUMP design improved on QJUMP's static configuration by introducing host-based dynamic channel negotiation. This allowed EYEQJUMP to offer bounded latency and at potentially line-rate throughput. The downside of this approach is, that it requires network topologies with full-bisection bandwidth. Alternatively, I proposed FASTJUMP which employed a centralised arbitrator for dynamic configuration. FASTJUMP could operate on any network topology but introduced a central point-of-failure. Finally, I proposed R3CJUMP that implemented a best-effort consensus system for fully distributing rate allocation decisions. The disadvantage of this approach is, that it would generate broadcast traffic on the network which could limit its scalability. I concluded that the choice between QJUMP, and its proposed extensions EYEQJUMP, FASTJUMP or R3CJUMP is an operator specific decision.

In conclusion, I have demonstrated that network interference is an application-measurable effect and a problem in datacenter networks. To address this problem, I have developed novel theoretical and practical solutions using a latency-first, network scheduling approach. I therefore conclude that, it is both possible and practical to control network interference in datacenter networks, using network scheduling techniques.

List of Acronyms

LR	latency rate
2PC	two-phase atomic-commit
AQM	Active Queue Management
ARP	Address Resolution Protocol
BA	behaviour aggregate
BGP	Border Gateway Protocol
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
CTS	Clear To Send
D^{2}TCP	Deadline-aware Datacenter TCP
D^3	Deadline Driven Delivery
DCB	Datacenter Bridging
DCTCP	Datacenter TCP
DSCP	Differentiated Services Code Point
DS	Differentiated Services
ECN	Explicit Congestion Notification
EDF	Earliest Deadline First
EFC	Ethernet Flow Control
FCT	Flow Completion Time
FIFO	first in, first out

FQ	Fair Queueing
GFS	Google File System
GPS	Generalised Processor Sharing
GSO	Generic Segment Offload
HDFS	Hadoop Distributed File System
HOL	Head of Line
HULL	High bandwidth Ultra Low Latency
ICMP	Internet Control Message Protocol
IFG	Inter-frame Gap
IPv4	IP version 4
IPv6	IP version 6
IP	Internet Protocol
ISP	Internet Service Provider
IS	Integrated Services
LAN	Local Area Network
LSO	Large Segment Offload
LTB	Leaky Token Bucket
MAC	Media Access Controller
MTU	Maximum Transfer Unit
NIC	Network Interface Controller
OCP	Open Compute Project
OSPF	Open Shortest Path First
PDQ	Preemptive Distributed Quick
PFC	Priority Flow Control
PGPS	Packet-by-packet Generalised Processor Sharing
PHB	Per-hop Behaviour

PS	Processor Sharing
PTPd	Precision Time Protocol daemon
PTP	Precision Time Protocol
QSFP28	Quad 25Gb/s Small Form-factor Pluggable
QoS	Quality of Service
RED	Random Early Detection
RMS	Root Mean Squared
RPC	Remote Procedure Call
RSVP	Resource reSerVation Protocol
RTO	Retransmit Timeout
RTS	Request To Send
RTT	Round-trip Time
RU	Rack Unit
SJR	Shortest Job Remaining
SLA	Service Level Agreement
SRPT	Shortest Remaining Processing Time
SoC	System on Chip
TCP	Transmission Control Protocol
TC	Traffic Control
TDMA	Time Division Multiple Access
TSC	Time Stamp Counter
TSO	TCP Segment Offload
ToR	Top of Rack
ToS	Type of Service
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network

VM	Virtual Machine
VOQ	Virtual Output Queue
WF^2Q	Worst Case Fair, Weighted Fair Queueing
WFQ	Weighted Fair Queueing
WRED	Weighted Random Early Detection
minRTO	minimum RTO

List of Figures

List of Tables

Bibliography

[1] The Wikimedia Foundation. Wikipedia: The Free Encyclopeida. `https://en.wikipedia.org/`; accessed 26/10/2015. See p. 11.

[2] Facebook Corporation. Facebook. `https://www.facebook.com/`; accessed 26/10/2015. See p. 11.

[3] HSB PLC. HSBC Bank Online. `https://www.hasbc.co.uk/`; accessed 26/10/2015. See p. 11.

[4] UKForex PLC. UK Foreign Exchange Services. `https://www.ukforex.co.uk/`; accessed 26/10/2015. See p. 11.

[5] Amazon Inc. Amazon Prime. `https://www.amazon.co.uk/gp/prime/`; accessed 26/10/2015. See p. 11.

[6] Apple Inc. iTunes . `http://www.apple.com/uk/itunes/`; accessed 26/10/2015. See p. 11.

[7] Amazon Inc. Amazon. `https://www.amazon.co.uk/`; accessed 26/10/2015. See p. 11.

[8] eBay Inc. eBay. `https://www.ebay.com/`; accessed 26/10/2015.

[9] Tesco Plc. Tesco Online Shopping. `https://www.tesco.com/`; accessed 26/10/2015. See p. 11.

[10] Apple Inc. Apple. `https://www.amazon.com/`; accessed 5/8/2016. See p. 11.

[11] Facebook Inc. Google . `https://www.google.com/`; accessed 5/8/2016. See p. 11.

[12] Microsoft Inc. Microsoft. `https://www.microsoft.com/`; accessed 5/8/2016. See p. 11.

[13] Jeffrey Dean and Luiz André Barroso. The Tail at Scale: Managing Latency Variability in Large-Scale Online Services. *Commun. ACM*, 56(2), February 2013. See pp. 11, 12, 26, and 27.

[14] Jeffrey Dean. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 1–1, New York, NY, USA, 2009. ACM. See p. 11.

[15] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864. See pp. 11 and 22.

[16] IEEE Standard for Ethernet. *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)*, pages 1–3747, Dec 2012. See pp. 11, 21, 62, and 81.

[17] S. Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. See p. 11.

[18] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of CHI*, pages 181–186, 1991. See pp. 12 and 27.

[19] Jake Brutlag. Speed Matters for Google Web Search. Technical report, Google. Available at: `http://goo.gl/1qF8xt`; accessed 24/09/2014. See pp. 12 and 27.

[20] Abhay K. Parekh and Robert G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-node Case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, June 1993. See pp. 12, 49, 50, 51, 53, and 75.

[21] Abhay K. Parekh and Robert G. Gallagher. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. *IEEE/ACM Trans. Netw.*, 2(2):137–150, April 1994. See pp. 12, 51, 53, 71, and 75.

[22] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, August 1989. See pp. 48, 49, and 57.

[23] J. C. R. Bennett and Hui Zhang. WF2Q: Worst-Case Fair Weighted Fair Queueing. In *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 1, pages 120–128 vol.1, Mar 1996. See pp. 12, 52, and 71.

[24] Jon Crowcroft, Steven Hand, Richard Mortier, Timothy Roscoe, and Andrew Warfield. QoS's Downfall: At the bottom, or not at all! In *Proceedings of the ACM SIGCOMM Workshop on Revisiting IP QoS*, 2003. See p. 12.

[25] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter when You Can JUMP THEM! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and*

Implementation, NSDI'15, pages 1–14, Berkeley, CA, USA, 2015. USENIX Association. See pp. 13 and 16.

[26] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, and Andrew W. Moore. Jump the Queue to Lower Latency. In *;login: The USENIX Magazine*, volume 40, pages 6–10. 2015. See pp. 13 and 16.

[27] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. Musketeer: All for One, One for All in Data Processing Systems. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 2:1–2:16, New York, NY, USA, 2015. ACM. See pp. 14, 18, 31, 32, and 123.

[28] Albert Greenberg. SDN for the Cloud (keynote speech) . In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 0, New York, NY, USA, 2015. ACM. See pp. 17, 20, 21, and 23.

[29] Albert Greenberg. SDN for the Cloud (slides for keynote speech). `http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p0.pdf`, August 2015. See pp. 17, 20, 21, and 23.

[30] Yuval Bachar. Disaggregation: The New Way to Build Mega (and Micro) Data Centers. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 1–1, Washington, DC, USA, 2015. IEEE Computer Society. See pp. 17 and 19.

[31] Amazon Web Services Inc. AWS re:Invent 2014 | (SPOT301) AWS Innovation at Scale. `https://www.youtube.com/watch?v=JIQETrFC_SQ`; accessed 26/10/2015. See pp. 17 and 19.

[32] Luyuan Fang, Fabio Chiussi, Deepak Bansal, Vijay Gill, Tony Lin, Jeff Cox, and Gary Ratterree. Hierarchical SDN for the Hyper-scale, Hyper-elastic Data Center and Cloud. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 7:1–7:13, New York, NY, USA, 2015. ACM. See p. 17.

[33] Alexey Andreyev. Introducing data center fabric, the next-generation Facebook data center network. `https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-cen` accessed 04/10/2016. See p. 17.

[34] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special*

Interest Group on Data Communication, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM. See pp. 17, 20, 21, 22, 23, 24, and 38.

[35] Facebook Corporation. Open Compute Project - Mission and Principles. `http://www.opencompute.org/about/mission-and-principles/`; accessed 26/10/2015. See pp. 17 and 20.

[36] L.A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*. Synthesis lectures in computer architecture. Morgan & Claypool, 2009. See pp. 18, 20, and 27.

[37] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–269, July 2008. See p. 18.

[38] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. See p. 18.

[39] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. See p. 18.

[40] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM. See p. 18.

[41] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM. See p. 18.

[42] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. See pp. 18 and 31.

[43] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010. See pp. 18 and 31.

[44] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed

Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. See p. 18.

[45] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011. See p. 18.

[46] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google&Rsquo;s Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013. See pp. 18 and 30.

[47] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM. See p. 18.

[48] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of NSDI*, pages 385–398, 2013. See pp. 18 and 30.

[49] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. See pp. 18 and 31.

[50] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient Big Data Processing in Hadoop MapReduce. *Proc. VLDB Endow.*, 5(12):2014–2015, August 2012. See pp. 18 and 31.

[51] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. See p. 18.

[52] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of SOSP*, pages 439–455, 2013. See pp. 18, 26, and 114.

[53] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. *SIGPLAN Not.*, 45(6):363–375, June 2010. See p. 18.

[54] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009. See p. 18.

[55] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. See p. 18.

[56] *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume Volume 3A: System Programming Guide, Part 1, page 16.12.1. Intel Corporation, 2011. See pp. 18 and 82.

[57] Robtert Mezger. The Relay Rack in Amateur Construction. *QST American Radio Relay League.*, 18, 1934. See p. 20.

[58] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal, The*, 32(2):406–424, March 1953. See p. 21.

[59] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*, page 111. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. See p. 21.

[60] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*, page 48. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. See p. 21.

[61] Reiss, C. and Tumanov, A. and Ganger, G.R. and Katz, R.H. and Kozuch, M.A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SoCC*, 2012. See p. 22.

[62] James R Hamilton. An Architecture for Modular Data Centers. In *Proceesings of the Conference on Innovative Data Systems Research*, 2007. See p. 22.

[63] Alexandros Daglis, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. Manycore Network Interfaces for In-memory Rack-scale Computing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 567–579, New York, NY, USA, 2015. ACM. See p. 22.

[64] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond Processor-centric Operating Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 17–17, Berkeley, CA, USA, 2015. USENIX Association.

[65] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2C2: A Network Stack for Rack-scale Computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 551–564, New York, NY, USA, 2015. ACM. See pp. 22, 124, 127, and 131.

[66] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (INTERNET STANDARD), October 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. See pp. 22, 35, and 84.

[67] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. See p. 22.

[68] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (INTERNET STANDARD), November 1982. Updated by RFCs 5227, 5494. See p. 22.

[69] 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 SIGCOMM*, pages 51–62, 2009. See pp. 22, 23, 44, 99, and 107.

[70] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFCs 6286, 6608, 6793, 7606, 7607, 7705. See p. 23.

[71] 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 SIGCOMM*, pages 63–74, 2010. See pp. 23, 24, 25, 33, 37, 38, 44, 99, and 107.

[72] LATEST HEADER FORMATS. IEN 44 (Proposed Standard), June 1978. See p. 23.

[73] V. Jacobson. Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM. See p. 23.

[74] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008. See pp. 24 and 33.

[75] Qian Zhang Murari Sridharan Kun Tan, Jingmin Song. A Compound TCP Approach for High-speed and Long Distance Networks. Technical report, July 2005. See pp. 24 and 25.

[76] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582 (Proposed Standard), April 2012. See p. 24.

[77] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782 (Experimental), January 2007. See p. 24.

[78] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. Updated by RFCs 4301, 6040. See pp. 25, 33, and 36.

[79] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, December 2006. See p. 25.

[80] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 24–35, New York, NY, USA, 1994. ACM. See p. 25.

[81] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *Queue*, 10(5):20:20–20:34, May 2012. See p. 25.

[82] Jim Gettys and Kathleen Nichols. Bufferbloat: dark buffers in the internet. *Commun. ACM*, 55(1):57–65, January 2012. See p. 25.

[83] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings of WREN*, pages 73–82, 2009. See pp. 25 and 31.

[84] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of NSDI*, pages 329–342, 2013. See pp. 26 and 92.

[85] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. *SIGCOMM Comput. Commun. Rev.*, 42(4):139–150, August 2012. See pp. 26, 33, and 46.

[86] J. Postel. Internet Control Message Protocol. RFC 792 (INTERNET STANDARD), September 1981. Updated by RFCs 950, 4884, 6633, 6918. See p. 27.

[87] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *Proceedings of SIGCOMM*, pages 307–318, 2014. See pp. 30, 33, 42, and 45.

[88] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011)*, pages 1–40, Sept 2011. See pp. 33 and 36.

[89] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of NSDI*, pages 297–311, 2013. See pp. 33, 46, and 131.

[90] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). *SIGCOMM Comput. Commun. Rev.*, 42(4):115–126, August 2012. See pp. 33 and 40.

[91] 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 NSDI*, pages 253–266, 2012. See pp. 33, 37, 39, and 40.

[92] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings SIGCOMM*, pages 50–61, 2011. See p. 33.

[93] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of SIGCOMM*, pages 127–138, 2012. See p. 33.

[94] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of SIGCOMM*, pages 435–446, 2013. See pp. 33, 42, 45, 100, 105, and 106.

[95] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. Practical TDMA for Datacenter Ethernet. In *Proceedings of Eurosys*, pages 225–238, 2012. See pp. 33, 35, 36, and 41.

[96] J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980. See p. 33.

[97] S. A. Reinemo, T. Skeie, and M. K. Wadekar. Ethernet for High-Performance Data centers: On the New IEEE Datacenter Bridging Standards. *IEEE Micro*, 30(4):42–51, July 2010. See p. 35.

[98] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, August 1993. See p. 36.

[99] Hao Jiang and Constantinos Dovrolis. Why is the Internet Traffic Bursty in Short Time Scales? In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’05, pages 241–252, New York, NY, USA, 2005. ACM. See p. 40.

[100] Lixia Zhang, Scott Shenker, and Daivd D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-way Traffic. *SIGCOMM Comput. Commun. Rev.*, 21(4):133–147, August 1991. See p. 40.

[101] Rishi Kapoor, Alex C. Snoeren, Geoffrey M. Voelker, and George Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 133–138, New York, NY, USA, 2013. ACM. See p. 40.

[102] T. Shanley, J. Winkles, and Inc MindShare. *InfiniBand Network Architecture*. PC system architecture series. Addison-Wesley, 2003. See p. 41.

[103] Nick McKeown. The iSLIP Scheduling Algorithm for Input-queued Switches. *IEEE/ACM Trans. Netw.*, 7(2):188–201, April 1999. See pp. 42 and 61.

[104] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed Switch Scheduling for Local-area Networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, November 1993. See p. 42.

[105] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association. See pp. 43 and 89.

[106] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks*, page 52. Morgan Kaufmann, 2002. See p. 43.

[107] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. Enabling End-Host Network Functions. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 493–507, New York, NY, USA, 2015. ACM. See pp. 44, 80, and 114.

[108] Gautam Kumar Rachit Agarwal Sylvia Ratnasamy Peter Xiang Gao, Akshay Narayan and Scott Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *Proceedings of the 2015 ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2015. See pp. 45 and 131.

[109] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks*, page 476. Morgan Kaufmann, 2002. See p. 45.

[110] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2130–2138, April 2013. See pp. 45, 47, and 118.

[111] IEEE Standard for Local and metropolitan area networks - Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks - Amendment 22: Equal Cost Multiple Path (ECMP). *IEEE Std 802.1Qbp-2014 (Amendment to IEEE Std 802.1Q-2011)*, pages 1–127, April 2014. See pp. 47 and 92.

[112] Nathan Farrington, Erik Rubow, and Amin Vahdat. Data Center Switch Architecture in the Age of Merchant Silicon. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects*, HOTI '09, pages 93–102, Washington, DC, USA, 2009. IEEE Computer Society. See p. 47.

[113] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896 (Historic), January 1984. Obsoleted by RFC 7805. See pp. 48 and 79.

[114] J. Nagle. On Packet Switches with Infinite Storage. *IEEE Transactions on Communications*, 35(4):435–438, Apr 1987. See pp. 48 and 49.

[115] Leonard Kleinrock. Analysis of A time-shared processor. *Naval Research Logistics Quarterly*, 11(1):59–73, 1964. See p. 50.

[116] Leonard Kleinrock. Time-shared Systems: A Theoretical Treatment. *J. ACM*, 14(2):242–261, April 1967. See p. 50.

[117] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM '94. Networking for Global Communications., 13th Proceedings IEEE*, pages 636–646 vol.2, Jun 1994. See p. 52.

[118] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. In *Proceedings of the ACM Symposium on Communications Architectures &Amp; Protocols*, SIGCOMM '90, pages 19–29, New York, NY, USA, 1990. ACM. See pp. 52 and 53.

[119] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, June 1996. See p. 52.

[120] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted Round-robin Cell Multiplexing in a General-purpose ATM Switch Chip. *IEEE J.Sel. A. Commun.*, 9(8):1265–1279, September 2006. See p. 52.

[121] Dimitrios Stiliadis and Anujan Varma. Latency-rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *IEEE/ACM Trans. Netw.*, 6(5):611–624, October 1998. See pp. 52 and 71.

[122] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Conference Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '92, pages 14–26, New York, NY, USA, 1992. ACM. See p. 53.

[123] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633 (Informational), June 1994. See pp. 53, 55, and 61.

[124] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205 (Proposed Standard), September 1997. Updated by RFCs 2750, 3936, 4495, 5946, 6437, 6780. See p. 54.

[125] J. Wroclawski. Specification of the Controlled-Load Network Element Service. RFC 2211 (Proposed Standard), September 1997. See p. 54.

[126] S. Shenker, C. Partridge, and R. Guerin. Specification of Guaranteed Quality of Service. RFC 2212 (Proposed Standard), September 1997. See pp. 54 and 55.

[127] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210 (Proposed Standard), September 1997. See p. 55.

[128] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475 (Informational), December 1998. Updated by RFC 3260. See p. 55.

[129] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), December 1998. Updated by RFCs 3168, 3260. See p. 55.

[130] P. Almquist. Type of Service in the Internet Protocol Suite. RFC 1349 (Proposed Standard), July 1992. Obsoleted by RFC 2474. See p. 55.

[131] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. RFC 2597 (Proposed Standard), June 1999. Updated by RFC 3260. See p. 56.

[132] D. C. Stephens, J. C. R. Bennett, and Hui Zhang. Implementing scheduling algorithms in high-speed networks. *IEEE Journal on Selected Areas in Communications*, 17(6):1145–1158, Jun 1999. See p. 56.

[133] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*, page 400. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. See p. 60.

[134] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed Switch Scheduling for Local-area Networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, November 1993. See p. 60.

[135] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz. Tiny Tera: a packet switch core. *IEEE Micro*, 17(1):26–33, Jan 1997. See p. 60.

[136] IEEE. Standard for local and metropolitan area networks, Virtual Bridged Local Area Networks. *IEEE Std. 802.11Q-2005*, 2005. See pp. 61 and 97.

[137] Ben Pfaff, Justin Pettit, and Scott Shenker. Extending Networking into the Virtualization Layer. In *Proceedings of the 2009 8th ACM Workshop on Hot Topics in Networks*, 2009. See p. 62.

[138] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3(4):267–286, 1979. See p. 63.

[139] IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery. *IEEE Std 802.1AB-2016 (Revision of IEEE Std 802.1AB-2009)*, pages 1–146, March 2016. See p. 63.

[140] IEEE Standard for Local and metropolitan area networks: Media Access Control (MAC) Bridges. *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)*, pages 1–277, June 2004. See p. 63.

[141] Bradner, Scott. Benchmarking Terminology for Network Interconnection Devices. RFC 1242, July 1991. See p. 63.

[142] 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 Conference on Networked Systems Design and Implementation*, NSDI'14, pages 475–488, Berkeley, CA, USA, 2014. USENIX Association. See pp. 79 and 80.

[143] Sameer Seth and M. Ajaykumar Venkatesulu. *TCP/IP Architecture, Design and Implementation in Linux*, pages 591–633. Wiley-IEEE Computer Society Press, 2008. See p. 79.

[144] Robert Winter, Rich Hernandez, Gaurav Chawla, Anthony Faustini, Carl Solder, Thomas Scheibe, David Law, Siamick Ayandeh, Brad Booth, Blaine Kohl, Charlie Lavacchia, Subi Krishnamurthy, Raja Karthikeyan, Eric Multanen, and Manoj Wadekar. Ethernet Jumbo Frames. *Ethernet Alliance*, 2009. See p. 81.

[145] *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume Volume 3C: System Programming Guide, Part 3, page 18.15. Intel Corporation, 2011. See p. 83.

[146] Noa Zilberman, Matthew P. Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W. Moore. Where has my time gone? In *Passive and Active Measurement - 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings*, pages 201–214, 2017. See p. 92.

[147] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[148] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of EuroSys*, pages 4:1–4:14, 2014.

[149] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of OSDI 14*, pages 1–16, October 2014.

[150] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary-Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of ISCA*, 2011.

[151] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward Predictable Performance in Software Packet-processing Platforms. In *Proceedings of NSDI*, pages 141–154, 2012. See p. 92.

[152] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of IMC*, pages 267–280, 2010. See pp. 98 and 99.

[153] Keon Jang et al. Silo: Predictable Message Completion Time in the Cloud. Technical report, Microsoft Research, 2013. MSR-TR-2013-95. See pp. 99 and 127.

[154] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the Tenant-provider Gap in Cloud Services. In *Proceedings of SoCC*, pages 10:1–10:14, 2012.

[155] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of SIGCOMM*, pages 242–253, 2011. See p. 99.

[156] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. See p. 104.

[157] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association. See p. 104.

[158] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of SOSP*, pages 29–41, 2011. See p. 104.

[159] Oxford Reference: root-mean-square value. `http://www.oxfordreference.com/10.1093/oi/authority.20110803100428422`, 2016. See p. 104.

[160] Linus Schrage. A Proof of the Optimality of the Shortest Remaining Processing Time Discipline. *Operations Research*, 16(3):687–690, 1968. See p. 105.

[161] Rebecca Isaacs. Tuning the performance of Naiad. Part 1: the network. Big Data at SVC blog, `http://bit.ly/1gl5Cjk`; accessed 25/09/2014. See p. 114.

[162] S. Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*, page 215. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. See p. 120.

[163] V. Sharma and F. Hellstrand. Framework for Multi-Protocol Label Switching (MPLS)-based Recovery. RFC 3469 (Informational), February 2003. Updated by RFC 5462. See p. 123.

[164] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). RFC 1701 (Informational), October 1994. See p. 123.

[165] Bozidar Radunović and Jean-Yves Le Boudec. A Unified Framework for Max-min and Min-max Fairness with Applications. *IEEE/ACM Trans. Netw.*, 15(5):1073–1083, October 2007. See p. 124.

[166] Matthew P. Grosvenor, Malte Schwarzkopf, and Andrew W. Moore. R2D2: Bufferless, Switchless Data Center Networks Using Commodity Ethernet Hardware. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 507–508, New York, NY, USA, 2013. ACM. See p. 130.

Appendices

Appendix A

Web Sources

The motivation for this dissertation arrises from the hyperscale networks used by today's giant Internet companies. Unsurprisingly, several of the references used in this dissertation were available from web sources only. These sources include consumer focused technical websites such as Wired and CNet, special interest websites such datacenterknowledge.com, press releases and marketing material from companies like Microsoft and Intel, blogs such as those run by Facebook and the OpenCompute project and statistics from company websites/corporate home pages. Although these sources provide a wealth of material not available from traditional (academic) sources, there is no guarantee that web based sources will remain hosted/available for the lifetime of this dissertation. To ensure that these valuable references remain accessible, I have included extracts from the original sources in the following appendix.

A.1 Amazon

A.1.1 AWS Documentation, Elastic Load Balancing, Classic Load Balancers, Internet-Facing Classic Load Balancers

EC2-Classic

Load balancers in EC2-Classic support both IPv4 and IPv6 addresses. The console displays the following public DNS names:

```
name-123456789.region.elb.amazonaws.com
ipv6.name-123456789.region.elb.amazonaws.com
dualstack.name-123456789.region.elb.amazonaws.com
```

The base public DNS name returns only IPv4 records. The public DNS name with the `ipv6` prefix returns only IPv6 records. The public DNS name with the `dualstack` prefix returns both IPv4 and IPv6 records. We recommend that you enable IPv6 support by using the DNS name with the`dualstack` prefix to ensure that clients can access the load balancer using either IPv4 or IPv6.

Clients can connect to your load balancer in EC2-Classic using either IPv4 or IPv6. However, communication between the load balancer and its back-end instances uses only IPv4, regardless of how the client communicates with your load balancer.

A.2 Arista

A.2.1 Arista Networks Delivers Open Management Infrastructure API Support for Microsoft

Press Releases

Arista Networks Delivers Open Management Infrastructure API Support for Microsoft

Furthering Automation and Programmability for Today's Data Center

Santa Clara, CA -- June 3, 2013 –-Arista today announced full support for Microsoft Open Management Infrastructure (OMI) across all Arista platforms through Arista EOS (extensible operating system) software version 4.12. This expands Arista's offerings in Software Defined Networking (SDN) by continuing to provide the broadest set of interfaces to management platforms for open programmability and orchestration.

"Microsoft's Cloud OS is built on Windows Server, System Center, and Windows Azure and aligns with Microsoft's commitment to openness, supporting standards (CIM + WS-Man) using OMI," said Chris Phillips, Partner Director PM, Windows Server, Microsoft. "Arista's manageable switch's support for Microsoft OMI gives customers flexibility and cost effectiveness as they implement networking across public and private cloud environments."

The growth of cloud-based computing is driving demand for more automation, which requires a solid foundation built upon management standards. Arista switches, widely deployed in Windows Azure data centers, drove Arista's implementation of OMI in close collaboration with Microsoft, and enabling OMI standards-based management to satisfy Windows Azure's cloud management requirements.

"Our customers are asking for open, standards-based API's to support next generation cloud architectures," said Anshul Sadana, senior vice president of Customer Engineering for Arista Networks. "Implementing OMI with its tight integration into Microsoft's System Center 2012 allows us to integrate Arista's industry leading cloud networking solutions and deliver easy and efficient management for our mutual customers' datacenters, virtual workloads, and hybrid cloud IT environments."

OMI uses the DMTF Common Information Model (CIM) object model and protocol to manage servers and network switches. More information about CIM may be found at DMTF (http://www.dmtf.org).

Availability

Arista EOS support of Microsoft OMI is available now.

About Arista Networks

The company was founded to deliver software defined cloud networking solutions for large data center and computing environments. Arista's award-winning 10/40/100 GbE switches redefine scalability, robustness, and price–performance, with more than one million cloud networking ports being deployed worldwide. At the core of Arista's platform is EOS, the world's most advanced network operating system. Arista Networks products are available worldwide through distribution partners, systems integrators and resellers.

Additional information and resources on today's announcement can be found at: http://www.arista.com

A.2.2 Arista 100G Transceivers and Cables: Q&A

Source: `https://www.arista.com/assets/data/pdf/Arista100G_TC_QA.pdf`

What is the difference between QSFP28 and QSFP100?

They are the same. “QSFP” form factor was originally defined for <10G speeds. When it was adopted for 40G, the name became QSFP+ to denote the higher aggregate performance. The same “QSFP” form factor was later adopted for 100G but the electrical interface had to be upgrade to handle 25Gbps/lane. The electrical interface for 100G can handle up to 28Gbps, hence the engineering and industry name is QSFP28. Arista refers to the 100G form factor as QSFP100 to ensure it is clear that this is a 100G optic, not a 28G optic.

What are the different speeds supported by Arista 100G ports?

All Arista products with 100G ports support multiple speeds. The table below is a summary of the range of port combinations permitted, with the correct optics:

Product	Interface Form Factor	100G Capable	100G Operating Mode	40G Capable	25/50G Capable	10G Capable
7500R-36CQ	QSFP100	Yes	4 Lanes of 25GbE	Yes	Yes	Yes
7500R-48S2CQ	QSFP100	Yes	4 Lanes of 25GbE	Yes	Yes	Yes
7500E-72S	MXP	Yes - XSR10	10 Lanes of 10GbE	Yes - XSR4	No	Yes - SR
7500E-12CM	MXP	Yes - XSR10	10 Lanes of 10GbE	Yes - XSR4	No	Yes - SR
7500E-12CQ	QSFP100	Yes	4 Lanes of 25GbE	Yes	No	Yes
7500E-6C2	CFP2	Yes	4 Lanes of 25GbE	Yes	No	Yes
7280TR-48C6	QSFP100	Yes	4 Lanes of 25GbE	Yes	Yes	Yes
7280SR-48C6	QSFP100	Yes	4 Lanes of 25GbE	Yes	Yes	Yes
7280QR-C36	QSFP100	Yes	4 Lanes of 25GbE	Yes	Yes	Yes
7280CR-48	QSFP100	Yes	4 Lanes of 25GbE	Yes	Yes	Yes
7280SE-72	MXP	Yes - XSR10	10 Lanes of 10GbE	Yes - XSR4	No	Yes - SR
7280SE-68	QSFP100	Yes	4 Lanes of	Yes	No	Yes

A.2.3 Arista 7150 Series

The Arista 7150 Series is the leading ultra low latency 1RU platform providing a unique combination of performance, advanced features and a balanced set of resources for low latency financial markets, HPC clusters and virtualized data centers.

Arista 7150 Series Video Datasheet

The Arista 7150S Advantages:

- First SDN Switch architected for leading-edge applications including Big Data, Cloud Networks, Financial Trading, HPC and Web 2.0 environments
- Industry leading accurate and predictable performance in a range of densities (deterministic high performance, lowest latency and jitter for all traffic)
- Unprecedented balanced resources and deployment flexibility for "Any Application” suitability
- First network–wide virtualization platform for next generation cloud bursting support with wire speed VXLAN hardware-based Tunnel Endpoint termination
- First platform to support sub-microsecond Network Address Translation (NAT)
- First deterministic latency and high-performance 10GbE/ 40GbE switch and IEEE 1588 Platform
- First integrated switch for precision data analysis and capture (DANZ) - redefining instrumentation, automation and analysis of high-end infrastructure

Arista 7150 Series Model Comparison

	7150 Switches		
	7150S-24	**7150S-52**	**7150S-64**
Description	7150 Switch 24-Port SFP+	7150 Switch 52-Port SFP+	7150 Switch 48-Port SFP+ 4 QSFP+
Total Ports	24	52	64
SFP+ Ports	24	52	48
L2/3 Throughput	480 Gbps	1.04 Tbps	1.28 Tbps
L2/3 PPS	360 Mpps	780 Mpps	960 Mpps
Latency	350ns	380ns	380ns
Typical Power Draw	191 W	191 W	224 W

Robust any-purpose data center switch with comprehensive feature sets:

High Performance, Low Latency, 10 and 40Gb Switching with Large Resources

A.2.4 Switch User Manual

Source: `https://www.arista.com/docs/Manuals/ConfigGuide.pdf`

- Section 22.4.3: CoS Rewrite – Petra Platform Switches
- Section 22.5.3: CoS and DSCP Rewrite – Trident Platform Switches
- Section 22.6.3: CoS and DSCP Rewrite – Trident-II Platform Switches

22.1.1.4 Traffic Classes

Data stream distribution is based on their traffic classes. Data stream management varies by switch platform. Traffic classes are derived from these data stream, inbound port, and switch attributes:

- CoS field contents
- DSCP field contents
- Inbound port trust setting
- CoS default setting (Arad, FM6000, Trident, and Trident-II platform switches)
- DSCP default setting (Arad, FM6000, Trident, and Trident-II platform switches)
- Traffic class default setting (Petra platform switches)

When a port is configured to derive a data stream's traffic class from the CoS or DSCP value associated with the stream, the traffic class is determined from a conversion map.

- A CoS-traffic class map derives a traffic class from a CoS value.
- A DSCP-traffic class map derives a traffic class from a DSCP value.

Map entries are configurable through CLI commands. Default maps determine the traffic class value when CLI map entry commands are not configured. Default maps vary by switch platform.

These sections describe traffic class configuration procedures:

- Section 22.2.2: Traffic Class Derivations – Arad Platform Switches
- Section 22.3.2: Traffic Class Derivations – FM6000 Platform Switches
- Section 22.4.2: Traffic Class Derivations – Petra Platform Switches
- Section 22.5.2: Traffic Class Derivations – Trident Platform Switches
- Section 22.6.2: Traffic Class Derivations – Trident-II Platform Switches

22.1.2 Transmit Queues and Port Shaping

Transmit queues are logical partitions of an Ethernet port's egress bandwidth. Data streams are assigned to queues based on their traffic class, then sent as scheduled by port and transmit settings. Support varies by switch platform. A queue's label determines its priority: Tx-queue 0 has lowest priority.

Parameters that determine transmission schedules include:

- Traffic class-transmit queue mapping: One set of traffic class-transmit queue maps is defined for all switch ports. The map determines the schedule for transmitting data streams based on traffic class. The set of available transmit maps vary by switch platforms:

  — Arad, FM6000, and Trident-II platforms: one map for all unicast and multicast traffic.
  — Trident platform: one map for unicast traffic and one map for multicast traffic.
  — Petra platform: one map for unicast traffic. Queue shaping is not available for multicast traffic.

- Port shaping: Port shaping specifies a port's maximum egress bandwidth.

- Queue shaping: Queue shaping specifies a transmit queue's maximum egress bandwidth.

  FM6000 platform switches do not support simultaneous port shaping and queue shaping. Enabling port shaping on an FM6000 switch disables queue shaping, regardless of the previous configuration.

- Queue priority: Queue priority specifies the transmission scheduling algorithm from the transmit queues. The switch defines two queue priority types:

  - — Strict Priority: Strict priority queues are serviced in the order of their priority rank - subject to each queue's configured maximum bandwidth. Data is not handled for a queue until all queues with higher priority are emptied or their transmission limit is reached. These queues typically carry low latency real time traffic and require highest available priority.
  - — Round Robin: Round robin queues are serviced simultaneously subject to assigned bandwidth percentage and configured maximum bandwidth. All round robin queues have lower priority than strict priority queues. Round robin queues can be starved by strict priority queues.

    Round robin priority is not available on Trident-II platform switches.
- Queue bandwidth allocation: Queue bandwidth allocation specifies the time slice (percentage) assigned to a round robin queue, relative to all other round robin queues.

These sections describe transmit queue and port shaping configuration procedures:

- Section 22.2.4: Transmit Queues and Port Shaping – Arad Platform Switches
- Section 22.3.4: Transmit Queues and Port Shaping – FM6000 Platform Switches
- Section 22.4.4: Transmit Queues and Port Shaping – Petra Platform Switches
- Section 22.5.4: Transmit Queues and Port Shaping – Trident Platform Switches
- Section 22.6.4: Transmit Queues and Port Shaping – Trident-II Platform Switches

22.1.3 Explicit Congestion Notification (ECN)

Explicit Congestion Notification (ECN) is an IP and TCP extension that facilitates end-to-end network congestion notification without dropping packets. ECN recognizes early congestion and sets flags that signal affected hosts. Trident platform switches extend ECN support to non-TCP packets.

ECN usage requires that it is supported and enabled by both endpoints. Although only unicast flows are modified by ECN markers, the multicast, broadcast, and unmarked unicast flows can affect network congestion and influence the indication of unicast packet congestion.

22.1.3.1 ECN Conceptual Overview

ECN uses DiffServ field bits 6 and 7 (IPv4 or IPv6 header) to advertise ECN capabilities:

- 00: Router does not support ECN.
- 10: Router supports ECN.
- 01: Router supports ECN.
- 11: Congestion encountered.

Networks typically signal congestion by dropping packets. After an ECN capable router negotiates ECN, it signals impending congestion by marking the IP header of packets encountering the congestion instead of dropping the packets. The recipient echoes the congestion indication back to the sender, which reduces its transmission rate as if it had detected a dropped packet.

Switches support ECN for unicast queues through Weighted Random Early Detection (WRED), which is an active queue management (AQM) algorithm that extends Random Early Detection (RED) to define multiple thresholds for an individual queue. WRED determines congestion by comparing average queue size with queue thresholds. Average queue size depends on the previous average and current queue size:

average queue size = (old_avg * (1-2 ^ (-weight))) + (current_queue_size * 2 ^ (-weight))

where weight is the exponential weight factor used for averaging the queue size.

Packets are marked based on WRED as follows:

- If average queue size is below the minimum threshold, packets are queued as in normal operation without ECN.

A.2.5 Arista 7060X Series

The Arista 7060X and 7260X Series are a range of 1RU and 2RU high performance 40GbE and 100GbE high density, fixed configuration, data center switches with wire speed layer 2 and layer 3 features, and advanced features for software driven cloud networking. The Arista 7060X and 7260X deliver a rich choice of interface speed and density allowing networks seamlessly evolve from 10GbE and 40GbE to 25GbE and 100GbE. With support for advanced EOS features these switches are ideal for traditional or fully virtualized data centers.

The 7060X and 7260X support a flexible combination of speeds including 10G, 25G, 40G and 100G in compact form factors that allows customers to design networks to accommodate the myriad different applications and east-west traffic patterns found in modern data centers whilst providing investment protection.

	7060CX-32	7260QX-64	7260CX-64
Switch Height	1RU	2RU	2RU
Ports	32 x QSFP100 2 x SFP+	64 x QSFP+ 2 x SFP+	64 x QSFP100 2 x SFP+
Max. 10GbE Density	130	2	258
Max. 25GbE Density	128	--	256
Max. 40GbE Density	32	64	64
Max. 50GbE Density	64	--	128
Max. 100GbE Density	32	--	64
Max. I/O Rate (Tbps)	6.4Tbps	5.12Tbps	12.8Tbps
Max. Forwarding Rate	3.3Bpps	3.3Bpps	9.52Bpps
Latency	450ns	550ns	550 to 1500ns
Packet Buffer Memory	16MB		64MB
Airflow Direction	Front-to-Back or Back-to-Front		

High Performance 40/100GbE leaf-spine

- High Density wirespeed 40GbE and 100GbE

A.3 Broadcom

A.3.1 High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series, Product Code: BCM56960 Series

High-Density 25/100 Gigabit Ethernet StrataXGS® Tomahawk Ethernet Switch Series

Product Code: **BCM56960 Series**

Broadcom's BCM56960 Series, also known as the StrataXGS® Tomahawk switch series, can help build highly scalable, feature-rich, top-of-rack (ToR), blade or aggregation switches to enable cloud-scale networking. StrataXGS Tomahawk is the first series of switches to offer 25 Gbps SerDes in support of 25/50/100GbE. It provides a total switching capacity of 3.2 Tbps ranging from 32 ports of 100 GbE to 128 ports of 25 GbE, with the flexibility to configure port type and speed to suit any high-performance networking application.

The StrataXGS Tomahawk switch series includes BroadView™ instrumentation, which provides operators the telemetry to troubleshoot large-scale networks, apply controls for optimal performance, respond to potential problems before they happen and drive down OPEX. This includes extensive application flow and debug statistics, link health and utilization monitors, streaming network congestion detection and packet-tracing capabilities.

FEATURES

- Support for up to 32 × 100 GbE, 64 × 40/50 GbE or even 128 × 25 GbE ports with an aggregate switching bandwidth of 3.2 Tbps
- Integrated low-power 25Ghz SerDes
- Authoritative support for 25G and 50G Ethernet Consortiumspecification
- Configurable pipeline latency enabling sub-400 ns port-to-port operation
- Supports high-performance storage/RDMA protocols including RoCE and RoCEv2
- BroadView instrumentation provides switch- and network-level telemetry
- High-density FleXGS™ flow processing for configurable forwarding/match/action capabilities
- OpenFlow 1.3+ support using Broadcom OF-DPA™
- Comprehensive overlay and tunneling support including VXLAN, NVGRE, MPLS, SPB
- Flexible policy enforcement for existing and new virtualization protocols
- Enhanced Smart-Hash™ load-balancing modes for leaf-spine congestion avoidance
- Integrated Smart-Buffer™ technology with 5x greater performance vs. static buffering
- Single-chip and multichip HiGig™ solutions for top-of-rack and scalable chassis applications

A.3.2 High-Capacity StrataXGS Trident II Ethernet Switch Series

High-Capacity StrataXGS® Trident II Ethernet Switch Series

Product Code: **BCM56850 Series**

With support for up to 100+ 10-Gigabit Ethernet (GbE) ports and full flexibility in configuring 10GbE/40GbE ports, the StrataXGS® Trident II switch series can be used to build highly scalable, feature-rich, blade switch, top-of-rack (ToR) switches and aggregation equipment to enable cloud-scale networking.

As server interfaces transition to higher Ethernet speeds and as virtualization continues to increase link utilization, data center networks are demanding switches with dense 10GbE and 40GbE connectivity at the access and aggregation layers. In addition, fabrics of BCM56750 with BCM56850 devices can be interconnected via the HiGig2™ protocol to support multiterabit chassis designs for large-scale data center, enterprise and service provider applications. The StrataXGS Trident II switch with integrated SmartSwitch™ has been designed to address performance, capacity and service requirements for next-generation data centers, cloud computing applications, enterprise campus backbone equipment and high-density fabrics for access and mobile core networks.

FEATURES

Single-chip solution for common fixed top-of-rack (ToR), aggregation and line-card switching applications

Single design meets the needs of multiple markets including enterprise and cloud data centers as well as carrier-access applications

First switch to support VMWare® VXLAN and Microsoft® NVGRE tunneling protocols supported by SmartNV™ technology

Enables spanning-tree-free and CLOS-style network topologies through TRILL, SPB and ECMP with SmartHash™ technology

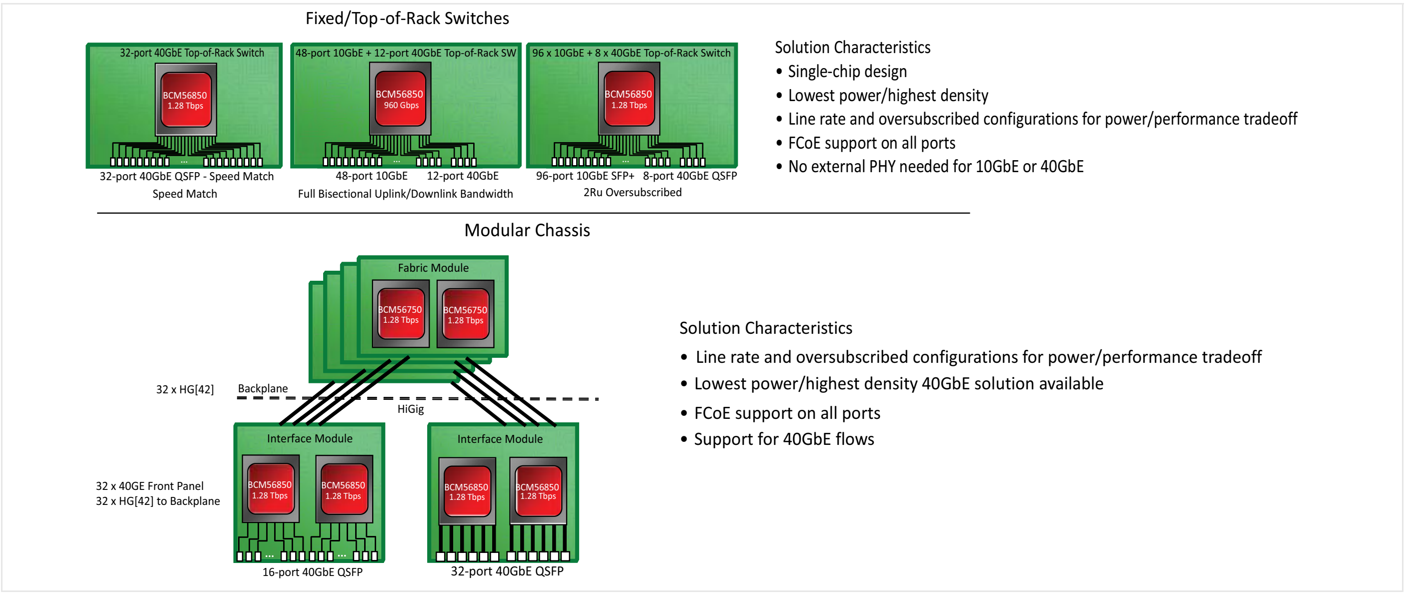
SmartTable and SmartBuffer technologies enable large-scale data centers with 10,000+ end user nodes

Up to 128x 10G integrated SerDes with Energy Efficient Ethernet for maximum port density per RU

Standards-compliant 10GbE/40GbE switch with support for up to 32 ports of 40GbE or up to 100+ ports 1GbE/10GbE

A.3.3 BCM56850 StrataXGS Trident II Switching Technology

Source: `https://www.broadcom.com/collateral/pb/56850-PB03-R.pdf`



Solution Characteristics
- Single-chip design
- Lowest power/highest density
- Line rate and oversubscribed configurations for power/performance tradeoff
- FCoE support on all ports
- No external PHY needed for 10GbE or 40GbE

Solution Characteristics
- Line rate and oversubscribed configurations for power/performance tradeoff
- Lowest power/highest density 40GbE solution available
- FCoE support on all ports
- Support for 40GbE flows

Deployment Scenarios

Features (Cont.)

- Power-efficient, fully integrated switching architecture.
- Low pin-to-pin latency in Cut-Through and Store-and-Forward Modes.
- Full IPv4 and IPv6 unicast and multicast routing support.
- Smart-NV technology featuring support for VMware VxLAN, Microsoft NVGRE overlays, 802.1Qbg Edge Virtual Bridging, 802.1BR Bridge Port extension, and per-virtual machine traffic shaping.
- SmartTable technology to maximize L2 and L3 forwarding database capacities for maximum deployment flexibility.
- Advanced SmartHash engine for optimal and resilient load distribution across HiGig™, LAG, and ECMP trunk groups.
- Integrated SmartBuffer supporting lossless performance guarantees and high-burst absorption using innovative traffic load awareness and dynamic allocation schemes.
- ContentAware™ Engine for scalable, high-density ingress and egress packet classification.
- Data Center Bridging support: PFC, QCN, ETS.
- Dedicated FCoE forwarding engine enabling FC-BB-5 and FC-BB-6 deployment models.
- Tagged and tagless virtual machine switching support for leading server and server virtualization switching vendors.
- Hardware-based tunneling services including MPLS, VPLS, ISATAP, MAC-in-MAC, TRILL, SPB, and Q-in-Q.
- Energy Efficient Ethernet support including customizable, low-power idle (LPI) control policies.
- Per-port configurable oversubscription to reduce peak power.
- Integrated 1588v2 processor for Precision Time Protocol and IEEE 802.1AS for Timing and Synchronization.

Ordering Information	I/O BW	Part Number
104 x 10GbE/32 x 40GbE	1280G	BCM56850A0KFSBG
104 x 10GbE/32 x 40GbE	1280G	BCM56851A0IFSBG
96x 10GbE/24 x 40GbE	960G	BCM56852A0KFSBG
72 x 10GbE/18 x 40GbE	720G	BCM56854A0IFSBG

About Broadcom

Broadcom Corporation (NASDAQ: BRCM), a FORTUNE 500® company, is a global leader and innovator in semiconductor solutions for wired and wireless communications. Broadcom® products seamlessly deliver voice, video, data, and multimedia connectivity in the home, office, and mobile environments. With the industry's broadest portfolio of state-of-the-art system-on-a-chip and embedded software solutions, Broadcom is changing the world by Connecting everything®. For more information, go to www.broadcom.com.

A.4 Cisco

A.4.1 Diffserv – The Scalable End-to-End Quality of Service Model

- The 3 bits restrict the number of possible priority classes to eight. Further, the network control and Internetwork control classes are usually reserved for router-generated packets such as routing updates, ICMP messages, etc. This is done to protect the packets that are necessary for the health of the network. However, this cuts down the usable classes for production traffic to six.
- IP-precedence and DTS bits (bits 3,4,5—the original type of service subfield) are not implemented consistently by network vendors today. In addition, RFC-1349 redefines the type of service subfield, by utilizing bits 3,4,5, and 6, and eliminating the DTS concept.

All of the above reduce the chances of successfully implementing end-to-end QoS using this scheme.

THE SOLUTION

The Differentiated Services Architecture

The IETF completed the Request for Comments (RFCs) for DiffServ toward the end of 1998. As stated in the DiffServ working group objectives [Ref-C], "There is a clear need for relatively simple and coarse methods of providing differentiated classes of service for Internet traffic, to support various types of applications, and specific business requirements. The differentiated service approach to providing quality of service in networks employs a small, well-defined set of building blocks from which a variety of aggregate behaviors may be built. A small bit-pattern in each packet, in the IPv4 ToS octet or the IPv6 traffic class octet, is used to mark a packet to receive a particular forwarding treatment, or per-hop behavior, at each network node. A common understanding about the use and interpretation of this bit-pattern is required for inter-domain use, multi-vendor interoperability, and consistent reasoning about expected aggregate behaviors in a network. Thus, the working group has standardized a common layout for a six-bit field of both octets, called the DS field. RFC 2474 and RFC 2475 define the architecture, and the general use of bits within the DS field (superseding the IPv4 ToS octet definitions of RFC 1349)."

In order to deliver end-to-end QoS, this architecture (RFC-2475) has two major components—packet marking using the IPv4 ToS byte and PHBs.

Packet Marking

Unlike the IP-precedence solution, the ToS byte is completely redefined [Figure3]. Six bits are now used to classify packets. The field is now called the Differentiated Services (DS) field, with two of the bits unused (RFC-2474). The six bits replace the three IP-precedence bits, and is called the Differentiated Services Codepoint (DSCP). With DSCP, in any given node, up to 64 different aggregates/classes can be supported (2^6). All classification and QoS revolves around the DSCP in the DiffServ model.

Per Hop Behaviors

Now that packets can be marked using the DSCP, how do we provide meaningful CoS, and provide the QoS that is needed? First, the collection of packets that have the same DSCP value (also called a codepoint) in them, and crossing in a particular direction is called a Behavior Aggregate (BA). Packets from multiple applications/sources could belong to the same BA. Formally, RFC-2475 defines a PHB as the externally observable forwarding behavior applied at a DS-compliant node to a DS BA. In more concrete terms, a PHB refers to the packet scheduling, queuing, policing, or shaping behavior of a node on any given packet belonging to a BA, and as configured by a Service Level Agreement (SLA) or policy. To date, four standard PHBs are available to construct a DiffServ-enabled network and achieve coarse-grained, end-to-end CoS and QoS:

The Default PHB (Defined in RFC-2474)

The default PHB specifies that a packet marked with a DSCP value (recommended) of '000000' gets the traditional best effort service from a DS-compliant node (a network node that complies to all the core DiffServ requirements). Also, if a packet arrives at a DS-compliant node and its DSCP value is not mapped to any of the other PHBs, it will get mapped to the default PHB.

A.4.2 Cisco Nexus 5000 Series Architecture: The Building Blocks of the Unified Fabric

10 Gigabit Ethernet and Unified Fabric Features

The Cisco Nexus 5000 Series is first and foremost a family of outstanding access switches for 10 Gigabit Ethernet connectivity. Most of the features on the switches are designed for high performance with 10 Gigabit Ethernet. The Cisco Nexus 5000 Series also supports FCoE on each 10 Gigabit Ethernet port that can be used to implement a unified data center fabric, consolidating LAN, SAN, and server clustering traffic.

Nonblocking Line-Rate Performance

All the 10 Gigabit Ethernet ports on the Cisco Nexus 5000 Series Switches can handle packet flows at wire speed. The absence of resource sharing helps ensure the best performance of each port regardless of the traffic patterns on other ports. The Cisco Nexus 5020 can have 52 Ethernet ports at 10 Gbps sending packets simultaneously without any effect on performance, offering true 1.04-Tbps bidirectional bandwidth.

Single-Stage Fabric

The crossbar fabric on the Cisco Nexus 5000 Series Switches is implemented as a single-stage fabric, thus eliminating any bottleneck within the switch. Single-stage fabric means that a single crossbar fabric scheduler has full visibility of the entire system and can therefore make optimal scheduling decisions without building congestion within the switch. With a single-stage fabric, the bandwidth you see is the bandwidth you get, and congestion becomes exclusively a function of your network design; the switch does not contribute to it.

Low Latency

The cut-through switching technology used in the Cisco Nexus 5000 Series ASICs enables the product to offer a low latency of 3.2 microseconds, which remains constant regardless of the size of the packet being switched. This latency was measured on fully configured interfaces, with access control lists (ACLs), quality of service (QoS), and all other data path features turned on. The low latency on the Cisco Nexus 5000 Series enables application-to-application latency on the order of 10 microseconds (depending on the network interface card [NIC]). These numbers, together with the congestion management features described next, make the Cisco Nexus 5000 Series a great choice for latency-sensitive environments.

Congestion Management

Keeping latency low is not the only critical element for a high-performance network solution. Servers tend to generate traffic in bursts, and when too many bursts occur at the same time, a short period of congestion occurs. Depending on how the burst of congestion is smoothed out, the overall network performance can be affected. The Cisco Nexus 5000 Series offers a full portfolio of congestion management features to minimize congestion. These features, described next, address congestion at different stages and offer maximum granularity of control over the performance of the network.

Virtual Output Queues

The Cisco Nexus 5000 Series implements virtual output queues (VOQs) on all ingress interfaces, so that a congested egress port does not affect traffic directed to other egress ports. But virtual output queuing does not stop there: every IEEE 802.1p class of service (CoS) uses a separate VOQ in the Cisco Nexus 5000 Series architecture, resulting in a total of 8 VOQs per egress on each ingress interface, or a total of 416 VOQs on each ingress interface. The extensive use of VOQs in the system helps ensure maximum throughput on a per-egress, per-CoS basis. Congestion on one egress port in one CoS does not affect traffic destined for other CoSs or other egress interfaces, thus avoiding head-of-line (HOL) blocking, which would otherwise cause congestion to spread.

Lossless Ethernet (Priority Flow Control)

By default, Ethernet is designed to drop packets when a switching node cannot sustain the pace of the incoming traffic. Packet drops make Ethernet very flexible in managing random traffic patterns injected into the network, but they effectively make Ethernet unreliable and push the burden of flow control and congestion management up at a higher level in the network stack.

IEEE 802.1Qbb Priority Flow Control (PFC) offers point-to-point flow control of Ethernet traffic based on IEEE 802.1p CoS. With a flow control mechanism in place, congestion does not result in drops, transforming Ethernet into a reliable medium. The CoS granularity then allows some CoSs to gain a no-drop, reliable, behavior while allowing other classes to retain traditional best-effort Ethernet behavior. A networking device implementing PFC makes an implicit agreement with the other end of the wire: any accepted packet will be delivered to the next hop and never be locally dropped. To keep this promise, the device must signal the peer when no more packets can reliably be accepted, and that, essentially, is the flow control function performed by PFC. The benefits are significant for any protocol that assumes reliability at the media level, such as FCoE.

A.4.3 Cisco Nexus 5548P Switch Architecture

Cisco Nexus 5500 Platform Features

The Cisco Nexus 5500 Series is the second generation of a family of outstanding access switches for 10 Gigabit Ethernet connectivity. The Cisco Nexus 5500 platform provides a rich feature set that makes it well suited for top-of-rack (ToR), middle-of-row (MoR), or end-of-row (EoR) access-layer applications. It protects investments in data center racks with standards-based 1 and 10 Gigabit Ethernet and FCoE features, and virtual machine awareness features that allow IT departments to consolidate networks based on their own requirements and timing. The combination of high port density, lossless Ethernet, wire-speed performance, and extremely low latency makes the switch family well suited to meet the growing demand for 10 Gigabit Ethernet that can support unified fabric in enterprise and service provider data centers, protecting enterprises' investments. The switch family has sufficient port density to support single and multiple racks fully populated with blade and rack-mount servers.

- High density and high availability: The Cisco Nexus 5548P provides 48 1/10-Gbps ports in 1RU, and the upcoming Cisco Nexus 5596 Switch provides a density of 96 1/10-Gbps ports in 2RUs. The Cisco Nexus 5500 Series is designed with redundant and hot-swappable power and fan modules that can be accessed from the front panel, where status lights offer an at-a-glance view of switch operation. To support efficient data center hot- and cold-aisle designs, front-to-back cooling is used for consistency with server designs.
- Nonblocking line-rate performance: All the 10 Gigabit Ethernet ports on the Cisco Nexus 5500 platform can handle packet flows at wire speed. The absence of resource sharing helps ensure the best performance of each port regardless of the traffic patterns on other ports. The Cisco Nexus 5548P can have 48 Ethernet ports at 10 Gbps sending packets simultaneously without any effect on performance, offering true 960-Gbps bidirectional bandwidth. The upcoming Cisco Nexus 5596 can have 96 Ethernet ports at 10 Gbps, offering true 1.92-terabits per second (Tbps) bidirectional bandwidth.
- Low latency: The cut-through switching technology used in the application-specific integrated circuits (ASICs) of the Cisco Nexus 5500 Series enables the product to offer a low latency of 2 microseconds, which remains constant regardless of the size of the packet being switched. This latency was measured on fully configured interfaces, with access control lists (ACLs), quality of service (QoS), and all other data path features turned on. The low latency on the Cisco Nexus 5500 Series together with a dedicated buffer per port and the congestion management features described next make the Cisco Nexus 5500 platform an excellent choice for latency-sensitive environments.
- Single-stage fabric: The crossbar fabric on the Cisco Nexus 5500 Series is implemented as a single-stage fabric, thus eliminating any bottleneck within the switches. Single-stage fabric means that a single crossbar fabric scheduler has full visibility into the entire system and can therefore make optimal scheduling decisions without building congestion within the switch. With a single-stage fabric, the congestion becomes exclusively a function of your network design; the switch does not contribute to it.
- Congestion management: Keeping latency low is not the only critical element for a high-performance network solution. Servers tend to generate traffic in bursts, and when too many bursts occur at the same time, a short period of congestion occurs. Depending on how the burst of congestion is smoothed out, the overall network performance can be affected. The Cisco Nexus 5500 platform offers a full portfolio of congestion management features to reduce congestion. These features, described next, address congestion at different stages and offer granular control over the performance of the network.
- Virtual output queues: The Cisco Nexus 5500 platform implements virtual output queues (VOQs) on all ingress interfaces, so that a congested egress port does not affect traffic directed to other egress ports. Every IEEE 802.1p class of service (CoS) uses a separate VOQ in the Cisco Nexus 5500 platform architecture, resulting in a total of 8 VOQs per egress on each ingress interface, or a total of 384 VOQs per ingress interface on the Cisco Nexus 5548P, and a total of 768 VOQs per ingress interface on the Cisco Nexus 5596. The extensive use of VOQs in the system helps ensure high throughput on a per-egress, per-CoS basis. Congestion on one egress port in one CoS does not affect traffic destined for other CoSs or other egress interfaces, thus avoiding head-of-line (HOL) blocking, which would otherwise cause congestion to spread.
- Separate egress queues for unicast and multicast: Traditionally, switches support 8 egress queues per output port, each servicing one IEEE 802.1p CoS. The Cisco Nexus 5500 platform increases the number of egress queues by supporting 8 egress queues for unicast and 8 egress queues for multicast. This support allows separation of unicast and multicast that are contending for system resources within the same CoS and provides more fairness between unicast and multicast. Through configuration, the user can control the amount of egress port bandwidth for each of the 16 egress queues.
- Lossless Ethernet with priority flow control (PFC): By default, Ethernet is designed to drop packets when a switching node cannot sustain the pace of the incoming traffic. Packet drops make Ethernet very flexible in managing random traffic patterns injected into the network, but they effectively make Ethernet unreliable and push the burden of flow control and congestion management up to a higher level in the network stack.

A.4.4 Cisco Nexus 5548P Switch Architecture

Cisco Nexus 5548P Architecture

The Cisco Nexus 5548P control plane runs Cisco NX-OS Software on a dual-core 1.7-GHz Intel Xeon Processor C5500/C3500 Series with 8 GB of DRAM. The supervisor complex is connected to the data plane in-band through two internal ports running 1-Gbps Ethernet, and the system is managed in-band, or through the out-of-band 10/100/1000-Mbps management port. Table 1 summarizes the control-plane specifications.

:isco Nexus 5548P Control Plane Components

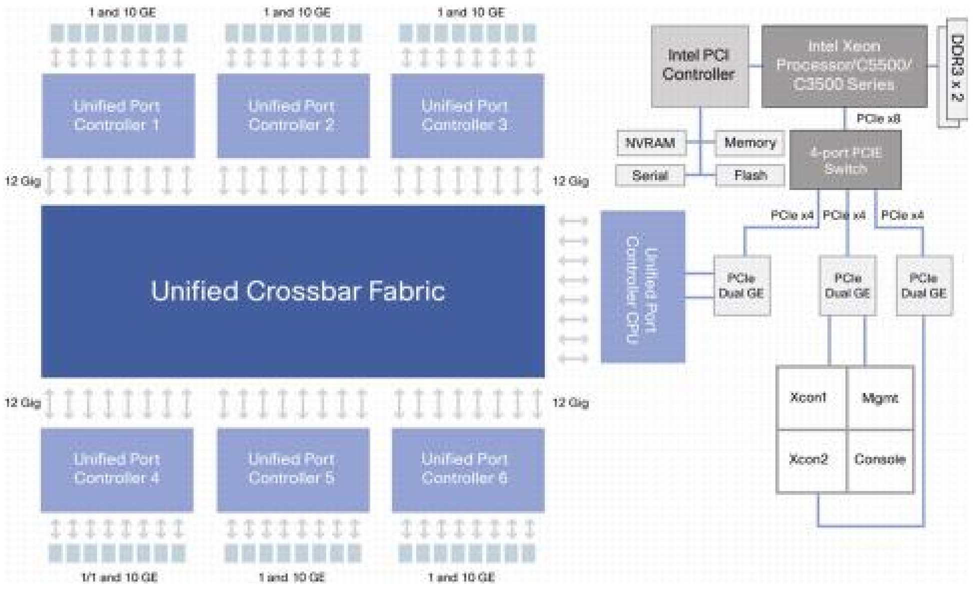
Component	Specification
CPU	1.7 GHz Intel Xeon Processor C5500/C3500 Series (dual core)
DRAM	8 GB of DDR3 in two DIMM slots
Program storage	2 GB of eUSB flash memory for base system storage
Boot and BIOS flash memory	8 MB to store upgradable and golden image
On-board fault log	64 MB of flash memory to store hardware-related fault and reset reasons
NVRAM	6 MB of SRAM to store syslog and licensing information
Management interface	RS-232 console port and 10/100/1000BASE-T mgmt0

The Cisco Nexus 5500 platform data plane is primarily implemented with two custom-built ASICs developed by Cisco: a set of unified port controllers (UPCs) that provides data-plane processing, and a unified crossbar fabric (UCF) that cross-connects the UPCs.

The UPC manages eight ports of 1 and 10 Gigabit Ethernet or eight ports of 1/2/4/8-Gbps Fibre Channel. It is responsible for all packet processing and forwarding on ingress and egress ports. Each port in the UPC has a dedicated data path. Each data path connects to UCF through a dedicated fabric interface at 12 Gbps. This 20 percent over-speed rate helps ensure line-rate throughput regardless of the internal packet headers imposed by the ASICs. Packets are always switched between ports of UPCs by the UCF.The UCF is a single-stage high-performance 100-by-100 crossbar with an integrated scheduler. The scheduler coordinates the use of the crossbar between inputs and outputs, allowing a contention-free match between I/O pairs. The scheduling algorithm is based on an enhanced iSLIP algorithm. The algorithm helps ensure high throughput, low latency, and weighted fairness across inputs, and starvation- and deadlock-free best-match policies across variable-sized packets.

The Cisco Nexus 5548P is equipped with seven UPCs: six to provide 48 ports at 10 Gbps, and one used for connectivity to the control plane. Figure 6 shows the connectivity between the control plane and the data plane.

Cisco Nexus 5548P Data Plane and Control Plane Architecture


1 and 10 GE
Unified Port Controller 1
Unified Port Controller 2
Unified Port Controller 3
12 Gig
Unified Crossbar Fabric
Unified Port Controller CPU
Unified Port Controller 4
Unified Port Controller 5
Unified Port Controller 6
1/1 and 10 GE
Intel PCI Controller
Intel Xeon Processor/C5500/C3500 Series
DDR3 x 2
PCIe x8
NVRAM
Memory
Serial
Flash
4-port PCIE Switch
PCIe x4
PCIe Dual GE
Xcon1
Mgmt
Xcon2
Console

Unified Port Controller Details

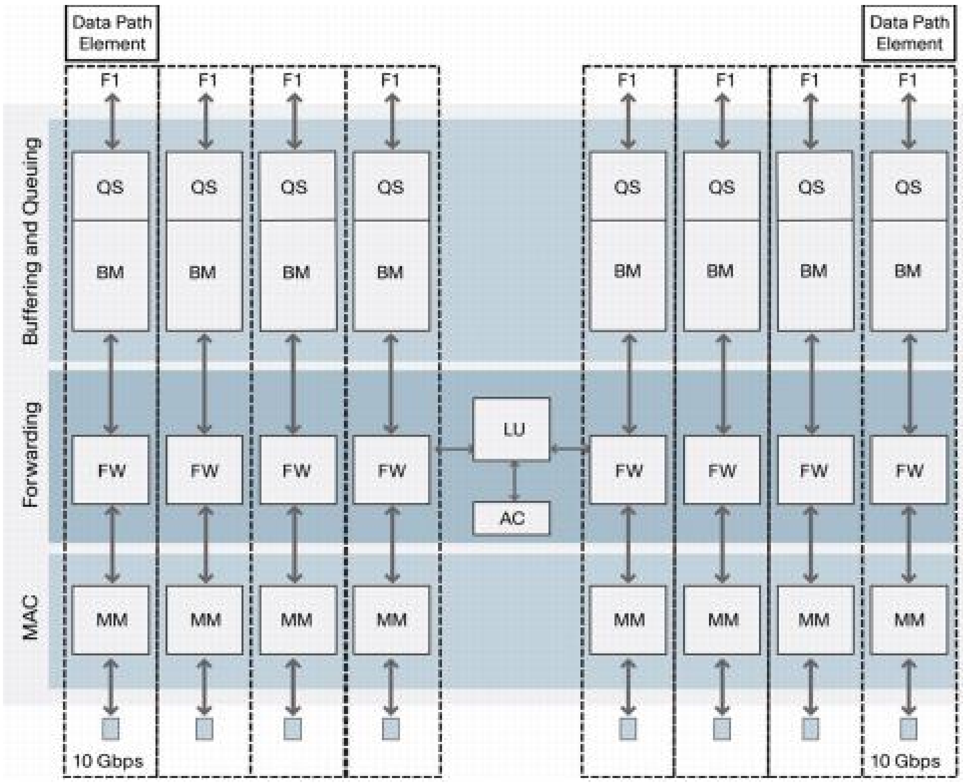
The UPC has three major elements: media access control (MAC), forwarding control, and the buffering and queuing subsystem.

The multimode MAC is responsible for the network interface packet protocol and flow-control functions. It consists of encoding-decoding and synchronization functions for the physical medium, frame cyclic redundancy check (CRC), and length check. The flow-control functions are IEEE 802.3x Pause, IEEE 802.1Qbb PFC, and Fibre Channel buffer-to-buffer credit. The multimode MAC supports 1 and 10 Gigabit Ethernet and 1/2/4/8-Gbps Fibre Channel.

The forwarding controller is responsible for the parsing and rewrite function (FW), lookup (LU), and access control list (ACL). Depending on the port mode, the parsing and editing element parses packets to extract fields that pertain to forwarding and policy decisions; it buffers the packet while waiting for forwarding and policy results and then inserts, removes, and rewrites headers based on a combination of static and per-packet configuration results from the forwarding and policy decision. The lookup and ACL receive the extracted packet fields, synthesize the lookup keys, and search a series of data structures that implement Fibre Channel, Ethernet, FCoE, Cisco FabricPath, TRILL forwarding modes, QoS, and security policies.

The buffering and queuing components consists of bulk memory (BM) and the queue subsystem (QS). The bulk memory is responsible for data buffering, congestion management, flow control, policing, ECN marking, and Deficit Weighted Round-Robin (DWRR) link scheduling. Packets are sent from bulk memory to the crossbar fabric through the fabric interface (FI). The queue subsystem is responsible for managing all queues in the system. At ingress, it manages the VOQ and multicast queues. At egress, it manages the egress queues. Figure 7 shows the UPC block. Each dedicated data path element has its own components except the lookup and ACL, which are shared among data path elements within the UPC.

Unified Port Controller

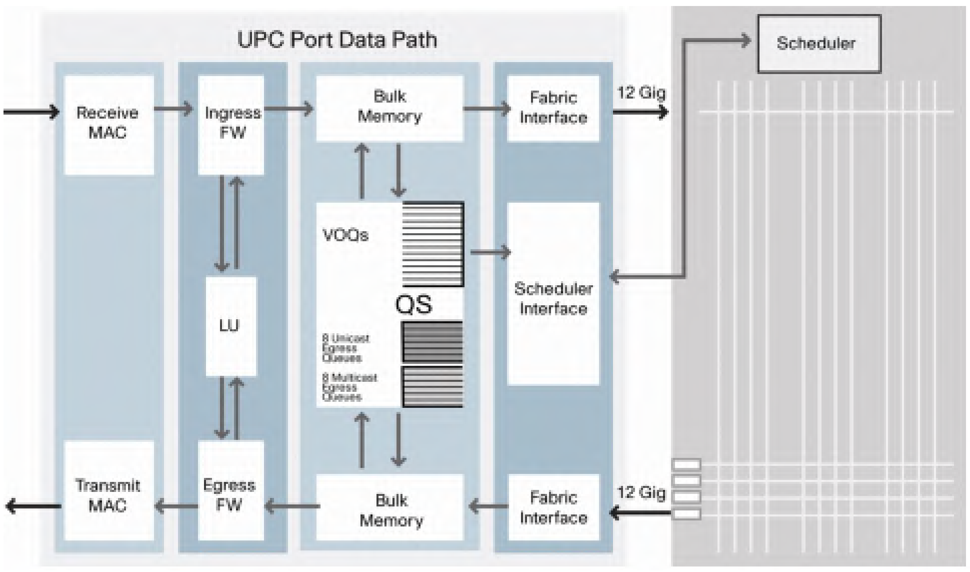


On ingress, a packet received through the MAC (Figure 8) goes through the parsing and editing element that is responsible for parsing and editing fields out of the incoming packets. The parsed fields are then fed to the lookup engine in the UPC for a forwarding decision. After the forwarding decision is received, the frame is edited based on the forwarding decision result and sent to bulk memory. The parsing and editing logic understands Ethernet, IPv4 and IPv6, IP Layer 4 transports (TCP and User Datagram Protocol [UDP]), Fibre Channel, FCoE, Cisco FabricPath, and TRILL. The parsing and editing block feeds inputs to the forwarding engine as soon as the relevant frame header fields have been extracted, enabling true cut-through switching.

When a frame is present in bulk memory, the frame is queued in a unicast VOQ or multicast queue, and a request is sent to scheduler to gain access the crossbar fabric. For unicast, each VOQ represents a specific CoS for a specific egress interface, giving high flexibility to the unicast scheduler in selecting the best egress port to serve an ingress at each scheduling cycle and completely eliminating head-of-line blocking. For multicast, there are 128 queues on every ingress port; each multicast queue can be used by one or more multicast fanout. When a grant is received from the scheduler, the packet is sent through the fabric interface to the crossbar fabric.

On egress, a packet received from the crossbar fabric is sent to bulk memory through the fabric interface. The packet is queued in one of the 16 egress queues, allowing complete separation between unicast and multicast traffic even within the same CoS. The packet then goes through the same forwarding and lookup logic before it is transmitted out of the port.

Unified Port Controller Data Path

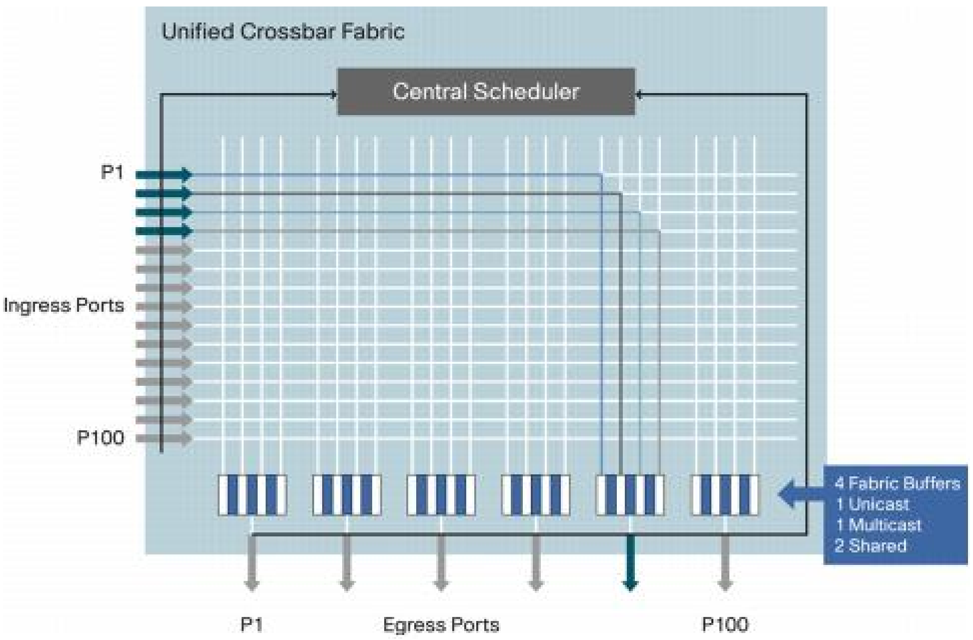


Unified Crossbar Fabric Details

The UCF is a single-stage, high-performance 100-by-100 nonblocking crossbar with an integrated scheduler (Figure 9). The single-stage fabric allows a single crossbar fabric scheduler to have full visibility into the entire system and therefore make optimal scheduling decisions without building congestion within the switch.

The crossbar provides the interconnectivity between input ports and output ports. Each row in the crossbar is associated with an input port, and each group of four columns is associated with an egress port; thus, there are four cross-points per egress port. In addition, there are four fabric buffers with 10,240 bytes of memory buffer per egress port. The four fabric buffers and four cross-points per egress interface allow four different ingress ports to simultaneously send packets to an egress port, allowing up to a 300 percent speed-up rate for unicast or multicast traffic. The four fabric buffers are shared between unicast and multicast traffic, with one reserved fabric buffer for unicast, one reserved fabric buffer for multicast, and two fabric buffers shared.

Unified Crossbar Fabric



The scheduler coordinates the use of the crossbar between input and output ports. The original iSLIP algorithm, which is based on iterative round-robin scheduling, has been enhanced to accommodate cut-through switching of different packet sizes. There is a separate scheduler for unicast and a separate scheduler for multicast. The scheduler uses a credit system when allocating bandwidth to each egress port. The credit system monitors fabric buffer and egress buffer use per egress port before a grant is sent to an ingress port to give access to the fabric. This approach helps ensure that the crossbar fabric is lossless, and it enables flow control to ingress ports when congestion occurs.

A.4.5 Cisco Nexus 3548 Switch

Exceptional Performance with Flexible Deployments

Get wire-rate, layer 2 and layer 3 switching with a comprehensive feature set, including Cisco Algo Boost technology, with the lowest latency in the industry. The Cisco Nexus 3548/3548-X Switch, part of the Unified Fabric Family, is well suited for mainstream top-of-rack, (ToR) data center deployments. It comes in a compact, 1-rack-unit (1RU) form factor.

Video Data Sheet

Learn more about the Cisco Nexus 3548-X Switch.

Read Data Sheet

Features and Capabilities

Algo Boost technology allows the Cisco Nexus 3548/3548-X to achieve exceptionally low latencies of 250 nanoseconds (ns) or less for all workloads. That includes unicast and multicast, and layer 2 and 3 switching, regardless of the features applied.

Benefits include:

- Warp mode to further reduce latency to 190 ns for small-to-midsize layer 2 and 3 deployments
- Warp SPAN to help enable stock market data delivery to trading servers in as little as 50 ns
- Active Buffer Monitoring to help you take advantage of market volatility and handle micro bursts
- Technology that supports Network Address Translation (NAT) for trade execution on any venue, without a latency penalty
- Embedded Remote SPAN with nanosecond timestamps to help you monitor your traffic with superior precision
- Multicast NAT and latency monitoring leading to simplified co-location integration and enhanced traffic visibility and troubleshooting (only 3548-X model).
- Next generation 3548-X offers lower power consumption further lowering operational cost

Comprehensive Feature Set with Cisco Nexus Operating System (NX-OS)

The Cisco Nexus 3548/3548-X is powered by the data-center-class, Cisco NX-OS. Features include:

- Full-featured unicast protocols, including Border Gateway Protocol (BGP), Open Shortest Path First (OSPF), Enhanced Interior Gateway Routing Protocol (EIGRP), and Routing Information Protocol (RIP)
- Multicast protocols, including Protocol Independent Multicast-Sparse Mode (PIM-SM), PIM-Source-Specific Multicast (PIM-SSM), and Multicast Source Discovery Protocol (MSDP)
- Full support for access control lists (including port, VLAN, and routed) and quality of service (queueing and marking)
- Full support for troubleshooting tools such as Switched Port Analyzer (SPAN) and Ethanalyzer
- Switch management by Cisco Prime Data Center Network Manager (DCNM)

Specifications at a Glance

- 48 fixed 1/10 Gbps small form factor pluggable plus (SFP+) ports
- Line-rate layer 2 and layer 3 throughput of up to 480 Gbps
- Compact 1RU form factor
- Dual, redundant, color-coded power supplies
- Four redundant color-coded fans

Low Latency, More Visibility

Learn the benefits of Cisco Algo Boost technology. (4:59 min)

Watch Video

Pushing the Performance Envelope

A.4.6 Cisco Nexus 3000 Configuration Guide

Information About Traffic Shaping

Traffic shaping allows you to control the traffic going out an interface in order to match its flow to the speed of the remote target interface and to ensure that the traffic conforms to policies contracted for it. Thus, traffic adhering to a particular profile can be shaped to meet downstream requirements, thereby eliminating bottlenecks in topologies with data-rate mismatches.

Traffic shaping regulates and smooths out the packet flow by imposing a maximum traffic rate for each port's egress queue. Packets that exceed the threshold are placed in the queue and are transmitted later. This is similar to traffic policing; however, the packets are not dropped. Because packets are buffered, traffic shaping minimizes packet loss (based on the queue length), thereby providing a better traffic behavior for TCP traffic.

Using traffic shaping, you can control access to available bandwidth, ensure that traffic conforms to the policies established for it, and regulate the flow of traffic in order to avoid congestion that can occur when the sent traffic exceeds the access speed of its remote, target interface. For example, you can control access to bandwidth when policy dictates that the rate of a given interface should not, on average, exceed a certain rate even though the access rate exceeds the speed.

The traffic shaping rate can be configured in Kilobits per second (Kbps) or packets per second (PPS) and is applied to unicast queues. Queue length thresholds are configured using WRED configuration.

Traffic shaping can be configured at the system level or the interface level. System level queuing policies can be overridden by interface queuing policies.

A.4.7 Cisco IOS Quality of Service Solutions Configuration Guide, Release 12.2

Traffic Policing

Traffic policing allows you to control the maximum rate of traffic sent or received on an interface, and to partition a network into multiple priority levels or class of service (CoS).

The Traffic Policing feature manages the maximum rate of traffic through a token bucket algorithm. The token bucket algorithm can use the user-configured values to determine the maximum rate of traffic allowed on an interface at a given moment in time. The token bucket algorithm is affected by all traffic entering or leaving (depending on where the traffic policy with Traffic Policing configured) and is useful in managing network bandwidth in cases where several large packets are sent in the same traffic stream.

The token bucket algorithm provides users with three actions for each packet: a conform action, an exceed action, and an optional violate action. Traffic entering the interface with Traffic Policing configured is placed in to one of these categories. Within these three categories, users can decide packet treatments. For instance, packets that conform can be configured to be transmitted, packets that exceed can be configured to be sent with a decreased priority, and packets that violate can be configured to be dropped.

Traffic Policing is often configured on interfaces at the edge of a network to limit the rate of traffic entering or leaving the network. In the most common Traffic Policing configurations, traffic that conforms is transmitted and traffic that exceeds is sent with a decreased priority or is dropped. Users can change these configuration options to suit their network needs.

The Traffic Policing feature supports the following MIBs:

- CISCO-CLASS-BASED-QOS-MIB
- CISCO-CLASS-BASED-QOS-CAPABILITY-MIB

This feature also supports RFC 2697, *A Single Rate Three Color Marker.*

For information on how to configure the Traffic Policing feature, see the chapter"Configuring Traffic Policing" in this book.

Benefits

Bandwidth Management Through Rate Limiting

Traffic policing allows you to control the maximum rate of traffic sent or received on an interface. Traffic policing is often configured on interfaces at the edge of a network to limit traffic into or out of the network. Traffic that falls within the rate parameters is sent, whereas traffic that exceeds the parameters is dropped or sent with a different priority.

Packet Marking Through IP Precedence, QoS Group, and DSCP Value Setting

Packet marking allows you to partition your network into multiple priority levels or classes of service (CoS), as follows:

- Use traffic policing to set the IP precedence or differentiated services code point (DSCP) values for packets entering the network. Networking devices within your network can then use the adjusted IP Precedence values to determine how the traffic should be treated. For example, the DWRED feature uses the IP Precedence values to determine the probability that a packet will be dropped.
- Use traffic policing to assign packets to a QoS group. The router uses the QoS group to determine how to prioritize packets.

Restrictions

The following restrictions apply to the Traffic Policing feature:

- On a Cisco 7500 series router, traffic policing can monitor CEF switching paths only. In order to use the Traffic Policing feature, CEF must be configured on both the interface receiving the packet and the interface sending the packet.
- On a Cisco 7500 series router, traffic policing cannot be applied to packets that originated from or are destined to a router.
- Traffic policing can be configured on an interface or a subinterface.
- Traffic policing is not supported on the following interfaces:
  - Fast EtherChannel
  - Tunnel
  - PRI
  - Any interface on a Cisco 7500 series router that does not support CEF

Prerequisites

On a Cisco 7500 series router, CEF must be configured on the interface before traffic policing can be used.

For additional information on CEF, refer to the *Cisco IOS Switching Services Configuration Guide.*

Traffic Shaping

Cisco IOS QoS software has three types of traffic shaping: GTS, class-based, and FRTS. All three of these traffic shaping methods are similar in implementation, though their CLIs differ somewhat and they use different types of queues to contain and shape traffic that is deferred. In particular, the underlying code that determines whether enough credit is in the token bucket for a packet to be sent or whether that packet must be delayed is common to both features. If a packet is deferred, GTS and Class-Based Shaping use a weighted fair queue to hold the delayed traffic. FRTS uses either a custom queue or a priority queue for the same, depending on what you have configured.

This section explains how traffic shaping works, then it describes the Cisco IOS QoS traffic shaping mechanisms. It includes the following sections:

- About Traffic Shaping
- Generic Traffic Shaping
- Class-Based Shaping
- Distributed Traffic Shaping
- Frame Relay Traffic Shaping

For description of a token bucket and explanation of how it works, see the section "What Is a Token Bucket?" earlier in this chapter.

About Traffic Shaping

Traffic shaping allows you to control the traffic going out an interface in order to match its flow to the speed of the remote target interface and to ensure that the traffic conforms to policies contracted for it. Thus, traffic adhering to a particular profile can be shaped to meet downstream requirements, thereby eliminating bottlenecks in topologies with data-rate mismatches.

Why Use Traffic Shaping?

The primary reasons you would use traffic shaping are to control access to available bandwidth, to ensure that traffic conforms to the policies established for it, and to regulate the flow of traffic in order to avoid congestion that can occur when the sent traffic exceeds the access speed of its remote, target interface. Here are some example reasons why you would use traffic shaping:

A.5 CNet

A.5.1 Google uncloaks once-secret server

Google uncloaks once-secret server

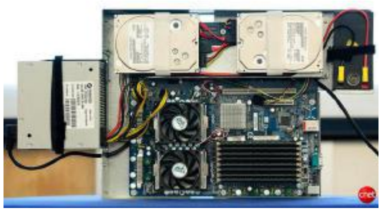
Unusually, the search giant designs its own servers. For the first time, Google unveils one publicly, showing a surprise built-in battery.

Tech Industry

December 11, 2009
11:37 AM PST

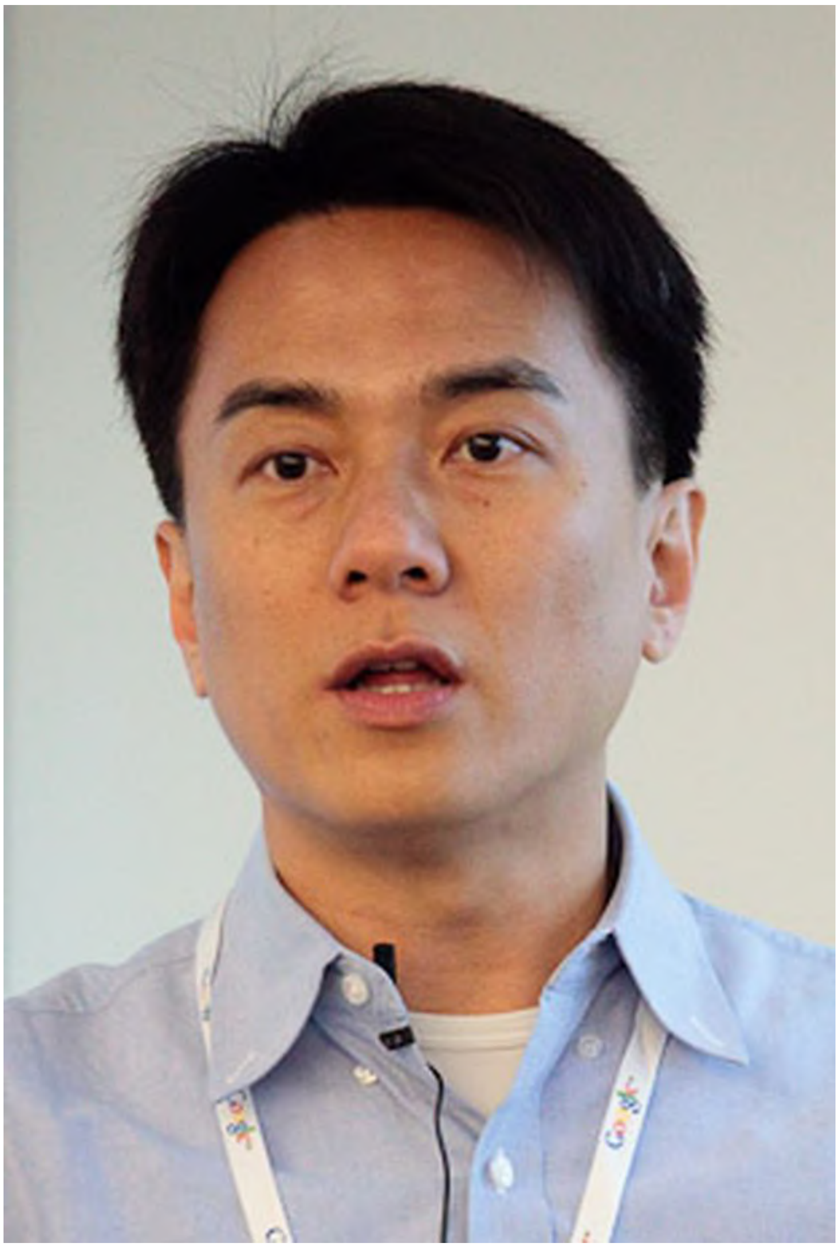
by Stephen Shankland
@stshank

Updated at 4:08 p.m. PDT April 1 with further details about Google's data center efficiency and shipping containers modules and 6:30 a.m. April 2 to correct the time frame of efficiency statistics.


Google for the first time showed off its server design. (Click to enlarge)
Stephen Shankland/CNET

MOUNTAIN VIEW,
Calif.--Google is tight-lipped about its computing operations, but the company for the first time on Wednesday revealed the hardware at the core of its Internet might at a conference here about the increasingly prominent issue of data center efficiency.

Most companies buy servers from the likes of Dell, Hewlett-Packard, IBM, or Sun Microsystems. But Google, which has hundreds of thousands of servers and considers running them part of its core expertise, designs and builds its own. Ben Jai, who designed many of Google's servers, unveiled a modern Google server before the hungry eyes of a technically sophisticated audience.



Google server designer Ben Jai
Stephen Shankland/CNET

Google's big surprise: each server has its own 12-volt battery to supply power if there's a problem with the main source of electricity. The company also revealed for the first time that since 2005, its data centers have been composed of standard shipping containers--each with 1,160 servers and a power consumption that can reach 250 kilowatts.

It may sound geeky, but a number of attendees--the kind of folks who run data centers packed with thousands of servers for a living--were surprised not only by Google's built-in battery approach, but by the fact that the company has kept it secret for years. Jai said in an interview that Google has been using the design since 2005 and now is in its sixth or seventh generation of design.

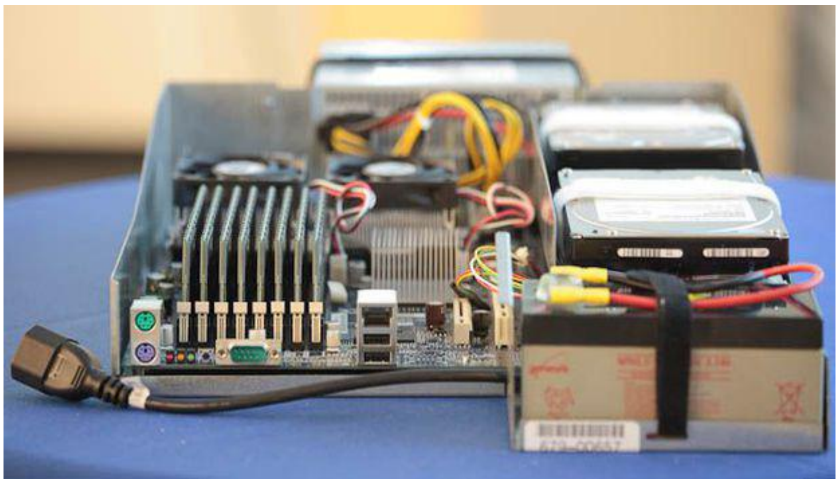
"It was our Manhattan Project," Jai said of the design.

Google has an obsessive focus on energy efficiency and now is sharing more of its experience with the world. With the recession pressuring operations budgets, environmental concerns waxing, and energy prices and constraints increasing, the time is ripe for Google to do more efficiency evangelism, said Urs Hoelzle, Google's vice president of operations.

"There wasn't much benefit in trying to preach if people weren't interested in it," said Hoelzle, but now attitudes have changed.

The company also focuses on data center issues such as power distribution, cooling, and ensuring hot and cool air don't intermingle, said Chris Malone, who's involved in the data center design and efficiency measurement. Google's data centers now have reached efficiency levels that the Environmental Protection Agency hopes will be attainable in 2011 using advanced technology.

"We've achieved this now by application of best practices and some innovations--nothing really inaccessible to the rest of the market," Malone said.



The rear side of Google's server.

Stephen Shankland/CNET

Why built-in batteries?
Why is the battery approach significant? Money.

Typical data centers rely on large, centralized machines called uninterruptible power supplies (UPS)--essentially giant batteries that kick in when the main supply fails and before generators have time to kick in. Building the power supply into the server is cheaper and means costs are matched directly to the number of servers, Jai said.

"This is much cheaper than huge centralized UPS," he said. "Therefore no wasted capacity."

Efficiency is another financial factor. Large UPSs can reach 92 to 95 percent efficiency, meaning that a large amount of power is squandered. The server-mounted batteries do better, Jai said: "We were able to measure our actual usage to greater than 99.9 percent efficiency."

Urs Hoelzle, Google's vice president of operations

Stephen Shankland/CNET

The Google server was 3.5 inches thick--2U, or 2 rack units, in data center parlance. It had two processors, two hard drives, and eight memory slots mounted on a motherboard built by Gigabyte. Google uses x86 processors from both AMD and Intel, Jai said, and Google uses the battery design on its network equipment, too.

Efficiency is important not just because improving it cuts power consumption costs, but also because inefficiencies typically produce waste heat that requires yet more expense in cooling.

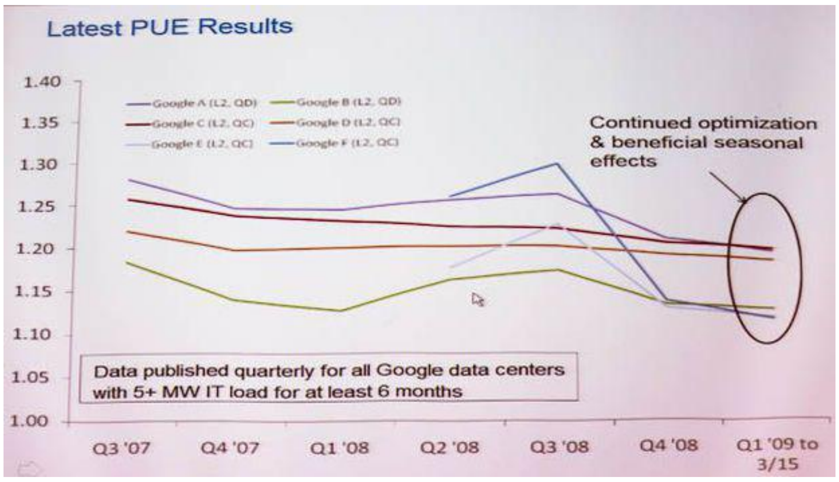
Costs add up

Google operates servers at a tremendous scale, and these costs add up quickly.

Jai has borne a lot of the burden himself. He was the only electrical engineer on the server design job from 2003 to 2005, he said. "I worked 14-hour days for two and a half years," he said, before more employees were hired to share the work.

Google has patents on the built-in battery design, "but I think we'd be willing to license them to vendors," Hoelzle said.

Another illustration of Google's obsession with efficiency comes through power supply design. Power supplies convert conventional AC (alternating current--what you get from a wall socket) electricity into the DC (direct current--what you get from a battery) electricity, and typical power supplies provide computers with both 5-volt and 12-volt DC power. Google's designs supply only 12-volt power, with the necessary conversions taking place on the motherboard.



Google's data center efficiency has been improving gradually.
Stephen Shankland/CNET

That adds $1 or $2 to the cost of the motherboard, but it's worth it not just because the power supply is cheaper, but because the power supply can be run closer to its peak capacity, which means it runs much more efficiently. Google even pays attention to the greater efficiency of transmitting power over copper wires at 12 volts compared to 5 volts.

Google also revealed new performance results for data center energy efficiency measured by a standard called power usage effectiveness. PUE, developed by a consortium called the Green Grid, measures how much power goes directly to computing compared to ancillary services such as lighting and cooling. A perfect score of 1 means no power goes to the extra costs; 1.5 means that ancillary services consume half the power devoted to computing.

Google's PUE scores are enviably low, but the company is working to lower them further. In the third quarter of 2008, Google's PUE was 1.21, but it dropped to 1.20 for the fourth quarter and to 1.19 for the first quarter of 2009 through March 15, Malone said.

Older Google facilities generally have higher PUEs, he said; the best has a score of 1.12. When the weather gets warmer, Google notices is that it's harder to keep servers cool.

An excerpt from a video tour Google presented of its data center containers. Like conventional data centers, Google's shipping containers have raised floors.
Stephen Shankland/CNET

Shipping containers
Most people buy computers one at a time, but Google thinks on a very different

scale. Jimmy Clidaras revealed that the core of the company's data centers are composed of standard 1AAA shipping containers packed with 1,160 servers each, with many containers in each data center.

Modular data centers are not unique to Google; Sun Microsystems and Rackable Systems both sell them. But Google started using them in 2005.

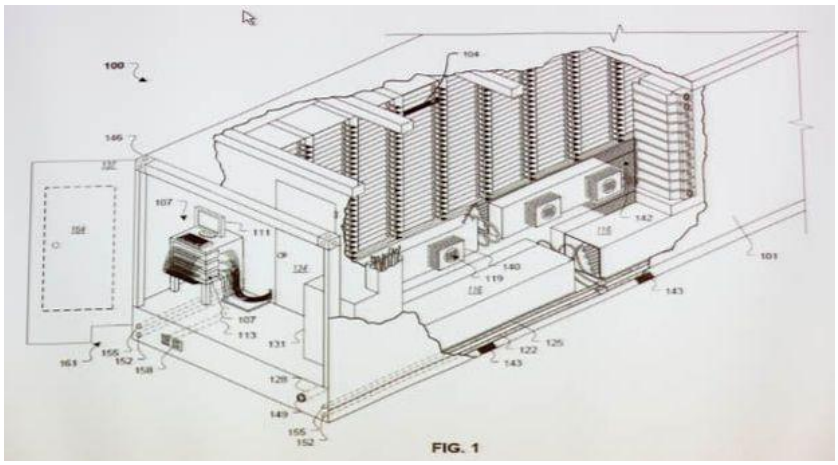
Google's first experiments had some rough patches, though, Clidaras said--for example when they found the first crane they used wasn't big enough to actually lift one.

Overall, Google's choices have been driven by a broad analysis on cost that encompasses software, hardware, and facilities.

"Early on, there was an emphasis on the dollar per (search) query," Hoelzle said. "We were forced to focus. Revenue per query is very low."

Mainstream servers with x86 processors were the only option, he added. "Ten years ago...it was clear the only way to make (search) work as free product was to run on relatively cheap hardware. You can't run it on a mainframe. The margins just don't work out," he said.

Operating at Google's scale has its challenges, but it also has its silver linings. For example, a given investment on research can be applied to a larger amount of infrastructure, yielding return faster, Hoelzle said.



A diagram of a Google modular data center

Stephen Shankland/CNET

Tags: Tech Industry, Tech Culture

DISCUSS: GOOGLE UNCLOAKS ONCE-SECRET SERVER

2 Comments Log In

Show Comments

Featured Video

Autoplay: ONAutoplay: OFF

A.6 Datacenter Knowledge

A.6.1 Google Unveils Its Container Data Center

GOOGLE

Google Unveils Its Container Data Center

Google's containers are real. And it's not an April Fool's Joke.

Four years after the first reports of server-packed shipping containers lurking in parking garages, Google today confirmed its use of data center containers and provided a group of industry engineers with an overview of how they were implemented in the company's first data center project in the fall of 2005. "It's certainly more fun talking about it than keeping it a secret," said Google's Jimmy Clidaras, who gave a presentation on the containers at the first Google Data center Efficiency Summit today in Mountain View, Calif.

The Google facility features a "container hanger" filled with 45 containers, with some housed on a second-story balcony. Each shipping container can hold up to 1,160 servers, and uses 250 kilowatts of power, giving the container a power density of more than 780 watts per square foot. Google's design allows the containers to operate at a temperature of 81 degrees in the cold aisle. Those specs are seen in some advanced designs today, but were rare indeed in 2005 when the facility was built.

Google's design focused on "power above, water below," according to Clidaras, and the racks are actually suspended from the ceiling of the container. The below-floor cooling is pumped into the hot aisle through a raised floor, passes through the racks and is returned via a plenum behind the racks. The cooling fans are variable speed and tightly managed, allowing the fans to run at the lowest speed required to cool the rack at that moment.

"Water was a big concern," said Urs Holzle, who heads Google's data center operations. "You never know how well these couplings (on the water lines) work in real life. It turns out they work pretty well. At the time, there was nothing to go on."

Google was awarded a patent on a portable data centerin a shipping container in October 2008, confirming a 2005 report from PBS columnist Robert Cringley that the company was building prototypes of container-based data centers in a garage in Mountain View. Containers also featured prominently in Google's patent filing for a floating data center that generates its own electricity using wave energy.

Holzle said today that Google opted for containers from the start, beginning its prototype work in 2003. At the time, Google housed all of its servers in third-party data centers. "Once we saw that the commercial data center market was going to dry up, it was a natural step to ask whether we should build one," said Holzle.

The data center facility, referred to as Data Center A, spans 75,000 square feet and has a power capacity of 10 megawatts. The facility has a Power Usage Effectiveness (PUE) of 1.25, and when the container load is measured across the entire hangar floor space, it equates to a density of 133 watts per square foot. Google didn't identify the facility's location, but the timeline suggests that it's likely one of the facilites at Google's three-building data center complex in The Dalles, Oregon.

Data center containers have been used for years by the U.S. military. The first commercial product, Sun's Project Blackbox, was announced in 2006. We noted at the time that the Blackbox "extends the boundaries of the data center universe, and gives additional options to managers of fast-growing enterprises."

It turns out that containers have developed as key weapons in the data center arms race between Google and Microsoft, which last year announced its shift to a container model. Microsoft has yet to complete its first container data center in Chicago,

OTHER COMMERCIAL CONTAINERS:

- Rackable ICE Cube
- HP POD (Portable Optimized Container)
- Verari Forest Container
- IBM Portable Modular Data Center (PMDC)
- Sun MD S20 (Project Blackbox)

Feedback

A.6.2 Intel Designs Custom Chips for AWS' New C4 Instances

Intel Xeon Processor E5-1600 v3 die shot (Photo: Intel)

Intel Xeon Processor E5-1600 v3 die shot (Photo: Intel)

AMAZON, BLADES, CLOUD COMPUTING, INTEL

Intel Designs Custom Chips for AWS' New C4 Instances

Intel has designed custom Xeon processors for **Amazon Web Services** that will power the cloud provider's new server instances optimized for high-octane computing. The chips will provide the highest level of CPU performance EC2 has ever seen.

Amazon previewed the new type of instance, which is not yet available, at its re:Invent conference in Las Vegas Thursday. Called C4, it comes in five different configurations, ranging from two to 36 virtual CPU cores and from 3.75 Gigabytes to 60 Gigabytes of RAM.

The instances will use hardware virtualization, which is as close to bare-metal cloud as AWS gets, and run within Virtual Private Cloud environments only, Jeff Barr, chief evangelist at AWS, wrote in a blog post.

The custom AWS CPU, called Intel Xeon E5-2666 v3, is based on the chipmaker's Haswell architecture and built using its smallest-yet 22 nanometer process technology. The processor runs at base speed of 2.9 GHz, but with "Turbo boost" can go up to 3.5 GHz, according to Amazon.

This is not the first time Intel has customized a processor for a big customer. Making tailored chips for cloud service providers, Internet companies, and hardware vendors has grown into a big business for the company in recent years.

Another recent custom job was for Oracle's massive database machines that came out in July.

Diane Bryant, general manager of Intel's data center group, described a new approach the company had taken to tailoring chips for hyper-scale customers using Field-Programmable Gate Arrays.

An FPGA is a reconfigurable semiconductor typically used to give a user the ability to test different configurations before they commit to a volume purchase of non-programmable chips. Intel plans to include an FPGA in a single Xeon package and offload some of the CPU workload to the FPGA.

The chipmaker gives the customer the option of testing different configurations and then order static System-on-Chips that would use the configuration that works best for them. Another option is to deploy Xeon packages with the FPGAs at scale so you can reconfigure them in the future for different workloads.

Feedback

When Bryant talked about the offering in June, it was not yet available, and she did not say when it would hit the market. It wasn't clear whether Intel used the approach in designing the latest custom Oracle or AWS CPUs.

In June, Bryant said Intel had designed 15 custom CPUs in 2013 for different customers, including Facebook and eBay. More than double that amount was in the pipeline for 2014, she said.

A.7 Datacenter Frontier

A.7.1 Inside Amazon's Cloud Computing Infrastructure

As cloud computing has emerged as the new paradigm for computing at scale, Amazon has firmly established itself as the dominant player. After effectively creating the public cloud market with the launch of Amazon Web Services in 2006, the retailer has built AWS into a $6 billion a year business.

Along the way, Amazon's infrastructure has become critical to the uptime of more than 1 million customers. That's why an outage at Amazon can create ripples across popular sites like Netflix, Reddit, Tinder and IMdB, which was the case on Sunday when Amazon experienced problems at a data center in Virginia.

This week we'll look at Amazon's mighty cloud infrastructure, including how it builds its data centers and where they live (and why).

Amazon operates at least 30 data centers in its global network, with another 10 to 15 on the drawing board. Amazon doesn't disclose the full scope of its infrastructure, but third-party estimates peg its U.S. data center network at about 600 megawatts of IT capacity.

Leading analysts view Amazon as the dominant player in public cloud. "AWS is the overwhelming (cloud computing) market share leader, with more than five times the compute capacity in use than the aggregate total of the other 14 providers," writes IT research firm Gartner in its assessment of the cloud landscape.

Lifting the Veil of Secrecy … A Bit

Amazon has historically been secretive about its data center operations, disclosing far less about its infrastructure than other hyperscale computing leaders such as Google, Facebook and Microsoft. That has begun to change in the last several years, as Amazon executives Werner Vogels and James Hamilton have opened up about the company's data center operations at events for the developer community.

"There's been quite a few requests from customers asking us to talk a bit about the physical layout of our data centers," said Werner Vogels, VP and Chief Technology Office for Amazon, in a presentation at the AWS Summit Tel Aviv in July. "We never talk that much about it. So we wanted to lift up the secrecy around our networking and data centers."

A key goal of these sessions is to help developers understand Amazon's philosophy on redundancy and uptime. The company organizes its infrastructure into 11 regions, each containing a cluster of data centers. Each region contains multiple Availability Zones, providing effect" of outages whenever AWS experiences problems indicates that this feature remains underutilized.

Scale Drives Platform Investment

In its most recent quarter, the revenue for Amazon Web Services was growing at an 81 percent annual rate. That may not translate directly into a similar rate of infrastructure growth, but one thing is certain: Amazon is adding servers, storage and new data centers at an insane pace.

"Every day, Amazon adds enough new server capacity to support all of Amazon's global infrastructure when it was a $7 billion annual revenue enterprise," said James Hamilton, Distinguished Engineer at Amazon, who described the AWS infrastructure at the Re:Invent conference last fall. "There's a lot of scale. That volume allows us to reinvest deeply into the platform and keep innovating."

Amazon's data center strategy is relentlessly focused on reducing cost, according to Vogels, who noted that the company has reduced prices 49 times since launching Amazon Web Services in 2006.

Amazon CTO Werner Vogels (Image: YouTube)

"We do a lot of infrastructure innovation in our data centers to drive cost down," Vogels said. "We see this as a high-volume, low-margin business, and we're more than happy to keep the margins where they are. And then if we have a lower cost base, we'll hand money back to you."

Amazon's Werner Vogels: We see (cloud computing) as a high-volume, low-margin business.

CLICK TO TWEET

A key decision in planning and deploying cloud capacity is how large a data center to build. Amazon's huge scale offers advantages in both cost and operations. Hamilton said most Amazon data centers house between 50,000 and 80,000 servers, with a power capacity of between 25 and 30 megawatts.

"In our view, this is around the right number, and we've chosen to build this number for an awfully long time," said Hamilton. "We can build bigger. The thing is, the early advantages of scale are huge, but there's a point where these advantages go down. A really huge data centers is only marginally less expensive per rack than a medium sized data center."

How Big is Too Big?

As data centers get bigger, they represent a larger risk as a component of the company network.

"It's undesirable to have data centers that are larger than that due to what we call the 'blast radius'," said Vogels, noting the industry term for assessing risk based on a single destructive regional event. "A data center is still a unit of failure. The larger you built your data centers, the larger the impact such a failure could have. We really like to keep the size of data centers to less than 100,000 servers per data center."

So how many servers does Amazon Web Services run? The descriptions by Hamilton and Vogels suggest the number is at least 1.5 million. Figuring out the upper end of the range is more difficult, but could range as high as 5.6 million, according to calculations by Timothy Prickett Morgan at the Platform.

Amazon's Werner Vogels: We like to keep the size of data centers to less than 100,000 servers

CLICK TO TWEET

Amazon leases buildings from a number of wholesale data center providers, including Digital Realty Trust and Corporate Office Properties Trust. In the past the company typically leased existing properties such as warehouses and then renovated them for data center use. In recent

in 572 209 439 G+ 89 106 1K SHA

Amazon has used pre-fabricated "modular" data center components to accelerate its expansion.

An interesting element of Amazon's approach to data center development is that it has the ability to design and build its own power substations. Tha specialization is driven by the need for speed, rather than cost management.

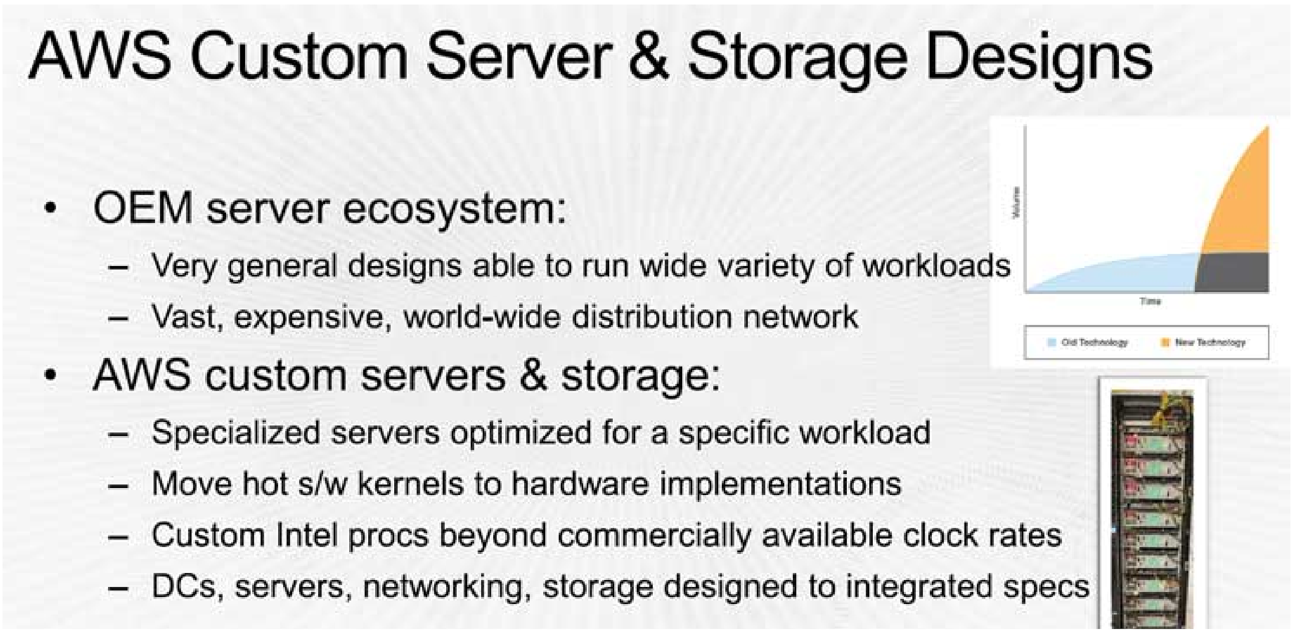
"You save a tiny amount," said Hamilton. "What's useful is that we can build them much more quickly. Our growth rate is not a normal rate for utility companies. We did this because we had to. But it's cool that we can do it."

Custom Servers and Storage

In the early days of its cloud platform, Amazon bought its servers from leading vendors. One of its major providers was Rackable Systems, an early player in innovative cloud-scale server designs. Amazon bought $86 million in servers from Rackable in 2008, up from $56 million a year earlier.

But as its operations grew, Amazon followed the lead of Google and began creating custom hardware for its data centers. This allows Amazon to fine-tune its servers, storage and networking gear to get the best bang for its buck, offering greater control over both performance and cost.

"Yes, we build our own servers," said Vogels. "We could buy off the shelf, but they're very expensive and very general purpose. So we're building custom storage and servers to address these workloads. We've worked together with Intel to make household processors available that run at much higher clockrates. It allows us to build custom server types to support very specific workloads."


Volume
Time
Old Technology
New Technology

AWS Custom Server & Storage Designs

- OEM server ecosystem:
  - Very general designs able to run wide variety of workloads
  - Vast, expensive, world-wide distribution network
- AWS custom servers & storage:
  - Specialized servers optimized for a specific workload
  - Move hot s/w kernels to hardware implementations
  - Custom Intel procs beyond commercially available clock rates
  - DCs, servers, networking, storage designed to integrated specs

572 209 439 89 106 1K SHA

(Image: James Hamilton, Amazon Web Services)

Amazon offers several EC2 instance types featuring these custom chips, a souped-up version of the Xeon E5 processor based on Intel's Haswell architecture and 22-nanometer process technology. Different configurations offer optimizations for compute intensive, memory intensive of IOPS intensive applications.

"We know how to build servers to a certain specification, and as a consequence the processors can be pushed harder." said Hamilton.

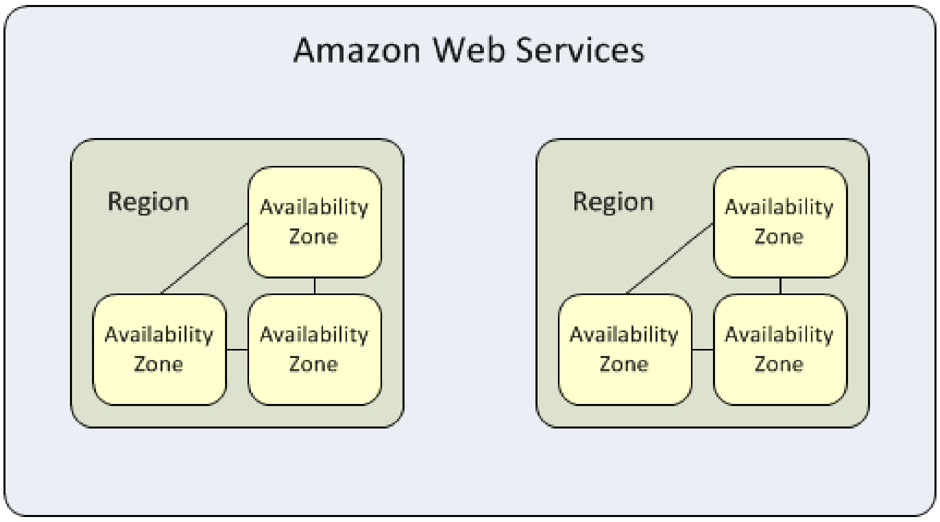
AWS uses designs its own software and hardware for its networking, which is perhaps the most challenging component of its infrastructure. Vogels said servers still account for the bulk of data center spending, but while servers and storage are getting cheaper, the cost of networking has gone up.

The Speed of Light Versus the Cloud

The "speed of light factor" in networking plays a significant role in how Amazon designs its infrastructure.

"The way most customers work is that an application runs in a single data center, and you work as hard as you can to make the data center as reliable as you can, and in the end you realize that about three nines (99.9 percent uptime) is all you're going to get," said Hamilton. "As soon as you get a high-reliability app, you run it in two data centers. Usually they're a long way apart, so the return trip is very long. It's excellent protection against a rare problem."

"Building distributed development across multiple data centers, especially if they're geographically further away, becomes really hard," said Vogels.



The answer was Availability Zones: clusters of data centers within a region that allow customers to run instances in several isolated locations to avoid a single point of failure. If customers distribute instances and data across multiple Availability Zones (AZs) and one instance fails, the application can be designed so that an

A.8 Endace

A.8.1 Endace DAG 9.2X2 Datasheet

Source: `https://www.endace.com/dag-9.2x2-datasheet.pdf`

DAG 9.2X2 – Technical Specifications

Monitoring interfaces	2x SFP+ transceivers
Network type	IEEE 802.3ae LAN IEEE 802.3ae WAN IEEE 802.3ab
Packet encapsulations	Ethernet
Hardware packet processing	Enhanced Packet Processing v2
Time synchronization	External: IEEE-1394 connector for RS-422 PPS and IRIG-B signal from GPS, CDMA or TDS (using adapter) Internal: Host PC clock Other DAG cards
Packet timestamping	7.5ns
PCI interface	x8 lane PCIe 2.0
Operating system supported	Endace software is supported on the following operating systems: Linux, FreeBSD, and Windows Server 2003 and 2008
Power requirements	Less than 20W
Operating temperature	0 to 55°C (32 to 131°F)
Airflow requirements	200 LFM (@50°C Ambient)
Operating humidity	5 to 95% non condensing
Physical dimensions	Half Height, Half Length Height 64.25mm (2.53") Length 167.5mm (6.6")

RoHS Compliant

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the Federal Communications Commission [FCC] Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction document, may cause harmful interference to radio communications.

Companion Products

Transceivers

10GBase-SR optical SFP+ transceiver 850nm, Multi-mode with LC connectors	TXR-10G-850-MM-SFP+
10GBase-LR optical SFP+ transceiver 1310nm, Single-mode with LC connectors	TXR-10G-1310-SM-SFP+
10GBase-ER optical SFP+ transceiver 1550nm, Single-mode with LC connectors	TXR-10G-1550-SM-SFP+
1000Base-SX optical Ethernet SFP transceiver 850nm, Multi-mode with LC connectors	TXR-1000SX
10GBase-ZR optical SFP+ transceiver 1550nm, Single-mode with LC connectors	TXR-10G-1550-SM-HS-SFP+
1000Base-LX optical Ethernet SFP transceiver 1310nm, Single-mode with LC connectors	TXR-1000LX
1000Base-ZX optical Ethernet SFP transceiver 1550nm, Single-mode with LC connectors	TXR-1000ZX
1/10 Gigabit LR (10km) SFP+ transceiver 1310nm, Single-mode	TXR-10G-1G-SWCH-850-MM-SFP+
1/10 Gigabit SR SFP+ transceiver 850nm, Multi-mode	TXR-10G-1G-SWCH-1310-SM-SFP+

Time Measurement Accessories

Trimble Acutime™ Gold GPS receiver	GPS-2
Endace 2-port Time Distribution Server, accepts serial input from GPS/CDMA sources	TDS-2
Endace 6-port expansion module for TDS-2, shares common reference time source	TDS-6
Endace 24-port Time Distribution Server, accepts serial input from GPS/CDMA sources	TDS-24

For more information on the Endace portfolio of products, visit: endace.com/products

For further information, email: info@endace.com

A.9 Exablaze

A.9.1 ExaLink Fusion

Source: `http://exablaze.com/downloads/pdf/ExaLINK_Fusion_Brochure.pdf`

EXALINKFUSION

Rethink your network architecture and application development using the ExaLINK Fusion.

Latency, reduced.

High frequency trading customers use the ExaLINK Fusion to seamlessly reduce the end to end latency in their networks.

Share a single network resource such as an order line by taking advantage of the Fusion's 95ns aggregation latency, many times lower than a traditional switch.

On the return path, optionally deliver order acknowledgements to clients by using layer 1 data distribution, replicating feeds at 5ns. In addition, use layer 1 fanout to distribute multicast to consumers with ultra low latency and negligible jitter.

This mode of operation reduces the full round trip time at the edge of your network to 100ns.

Patch and Tap

5 nanoseconds tapping and patching. Dynamically tap the feed on any input or output to any other port, replacing conventional optical taps. Dynamically patch any port to any other port to create bidirectional layer 1 links. Clock and data recovery circuitry ensures signal integrity is maintained throughout your network.

Aggregate

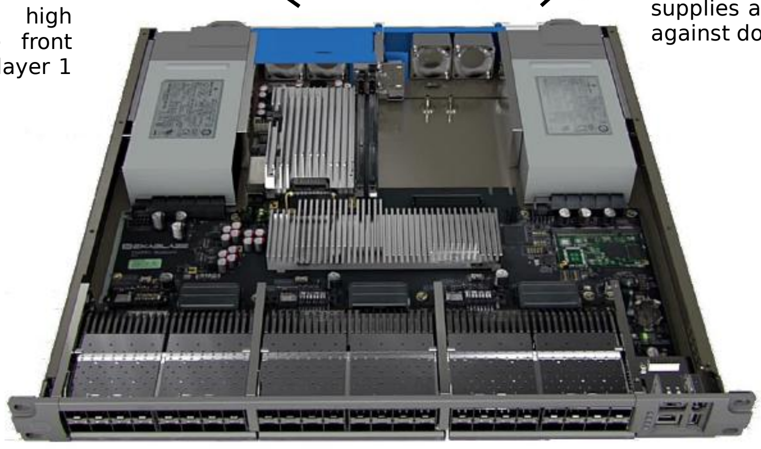
95 nanosecond aggregation. Aggregate multiple streams with extremely low and deterministic latency by bypassing MAC address lookups, sharing a single network resource fairly. Optionally, use layer 1 on the return path to further reduce latency and jitter and guarantee response times.

Switch

110 nanoseconds port to port latency with full layer 2 switching. The industry's lowest latency cut-through layer 2 switch. Supports ultra-low latency multicast and broadcast.

Redefine

It's your network, build it your way. The modular ExaLINK Fusion can be shipped with a Xilinx Ultrascale FPGA module that allows you to completely redefine the way your network operates. The layer 1 technology in the Fusion allows you to create dynamic circuits between any and all front panel ports and the FPGA. Build custom packet processing engines. Filter traffic. Switch based on packet content. Experiment with new protocols. The Fusion makes all of this and more possible.



Dual module bays

Fit two FPGA modules or x86 processors. Connect any high speed transceiver to the front panel by creating dynamic layer 1 circuits.

Redundant power supplies

Maximum reliability. Dual power supplies and fan modules protect against downtime.

Reconfigurable front panel

Three line card bays provide front a flexible front panel, allowing for up to 48 SFP+ ports.

Layer 1 crosspoint

The heart of the Fusion. A layer 1 crosspoint allows you to dynamically create circuits between any front panel port or any internal module bay

Smart management

With flexibility comes great complexity. The management interface on the Fusion provides an intuitive way to configure the device.

SPECIFICATIONS

Latency

Measured front panel port to port:

- 5 ns (Layer 1 tap/patch)
- 95 ns (Layer 2 aggregation)
- 110ns (Layer 2 switching)

Timestamping

- Any flow through the device can be timestamped to nanosecond resolution
- Pulse per second input for synchronization
- 0.2 parts per billion holdover drift with loss of external timing (with optional timing upgrade)
- Local time synchronized by NTP or PTP

Connectivity

- 3 x 16 SFP+ line cards
- SFP+ Fiber (10GBASE-SR, 10GBASE-LR, 10GBASE-LRM, 1000BASE-SX)
- SFP+ Copper Direct Attach

Physical, power, cooling

- 19” 1RU, rack mount
- Weight 11kg (24lbs)
- Dual, hot-swappable supplies
- Standard: AC 90-264V, 47-64 Hz, included IEC C13-C14 cables
- Optional: DC 40-72V
- Typical consumption: 150W
- Dual hot-swappable fan modules
- Port side intake (front-to-back) or port side exhaust (back-to-front) – specify when ordering

FPGA development

- FPGA board installable in both module bays
- Xilinx Ultrascale KU115 FPGA with 52 high speed transceivers
- 48 transceivers can be directly connected to front panel ports via the internal layer 1 crosspoint
- 4 transceivers available on internal header, for communication with adjacent module bay using optional cable
- 288 Mbit QDR4 SRAM per FPGA board
- 2 x DDR4 DIMM slots (low profile)
- Thermally controlled oscillator (TCXO) standard fit. Lower drift options available
- Optional TPM module for cryptographic applications

Management

- Ethernet 10/100M management port
- Industry standard serial port
- Intuitive command line interface
- External JSON RPC scripting interface
- SSH
- Telnet
- SNMP
- Local and remote syslog
- PTP, NTP and pulse-per-second interfaces for timing
- Firmware upgradeable via SFTP, TFTP or USB
- Custom FPGA firmware loaded and managed by onboard processor

EXABLAZE

A.9.2 ExaNIC X10

ExaNIC X10

SUB-MICRO TCP HALF RTT* DUAL-PORT 10GBE NETWORK INTERFACE CARD

Exablaze's ExaNIC X10 is an ultra low latency network interface card which delivers the lowest latency in the industry: 780ns application to application for small Ethernet frames (using the native API) and under a microsecond for small TCP and UDP payloads (using the transparent socket acceleration library).

The ExaNIC X10 also has built in bridging functionality which allows it to function as a miniature switch, avoiding the extra latency of a switch in some common use cases. Additionally, with hardware–based time stamping of every packet to 6 nanoseconds resolution, users can rethink their approach to measurement and latency management.

The card features an SMA connector which can accept a PPS in or drive PPS out to further improve the accuracy of timestamps taken throughout the system.

PERFORMANCE

Typical latency, raw frames:
60 bytes: 780ns
300 bytes: 1µs

Typical latency, raw frames with pre-loaded TX buffer:
60 bytes: 710ns
300 bytes: 930ns

Typical latency, UDP:
14 bytes: 880ns
300 bytes: 1.2µs

Typical latency, TCP:
14 bytes: 930ns
300 bytes: 1.2µs

A.9.3 Reducing latency with Exasock

Reducing latency with Exasock

We've been working really hard lately to bring you our kernel bypass sockets library for the ExaNIC, and we're really pleased with the results. We call it ExaNIC sockets - “exasock” - and it's a library that allows you to transparently improve the latency of your existing applications without requiring a rebuild.

So just what is a kernel bypass sockets library and how could it help you? Lets say, like many of our customers, you have an existing TCP or UDP networking application that is latency critical. In some cases, either the source code for that application isn't available or the effort required to port it to a custom networking API is prohibitive. For these customers, an easy approach to get instant performance gains is to use a library that intercepts regular socket calls, providing faster alternatives. This is exactly what exasock does, and the latency boost is impressive.

For those in finance, the industry standard way of measuring latency is via benchmarking through STAC and we've certainly done that for exasock, but the results are only available to STAC members. For those with access I'd recommend you check them out, but in this post I'd like to show you the performance of exasock using sockperf, a fairly standard sockets benchmarking utility. At the same time I'd like to show you how easy it is to get started with exasock. Let's get going!

First things first, lets start sockperf with exasock acceleration. To do this we simply prefix the application with exasock, like this:

```
$ exasock taskset -c 5 ./sockperf pp -i 192.168.4.11 -t 5 -m 12
```

Additionally, I'm using taskset to pin the sockperf process to an isolated CPU, which prevents the process from being interrupted by the scheduler and allows us to get more consistent results. In this case, we're doing a UDP ping-pong latency test, running for 5 seconds with a message payload of 12 bytes. By prefixing sockperf with exasock, all socket calls are transparently accelerated. We have a second machine running at 192.168.4.11 which runs sockperf in server mode, also accelerated by exasock. The results look like this:

A.10 Extreme Tech

A.10.1 Facebook, ARM, x86, and the future of the data center

Facebook, ARM, x86, and the future of the data center

By Joel Hruska (http://www.extremetech.com/author/jhruska) on January 28, 2013 at 8:43 am
5 Comments (http://www.extremetech.com/extreme/146850-facebook-arm-x86-and-the-future-of-the-data-center#disqus_thread)

0
shares

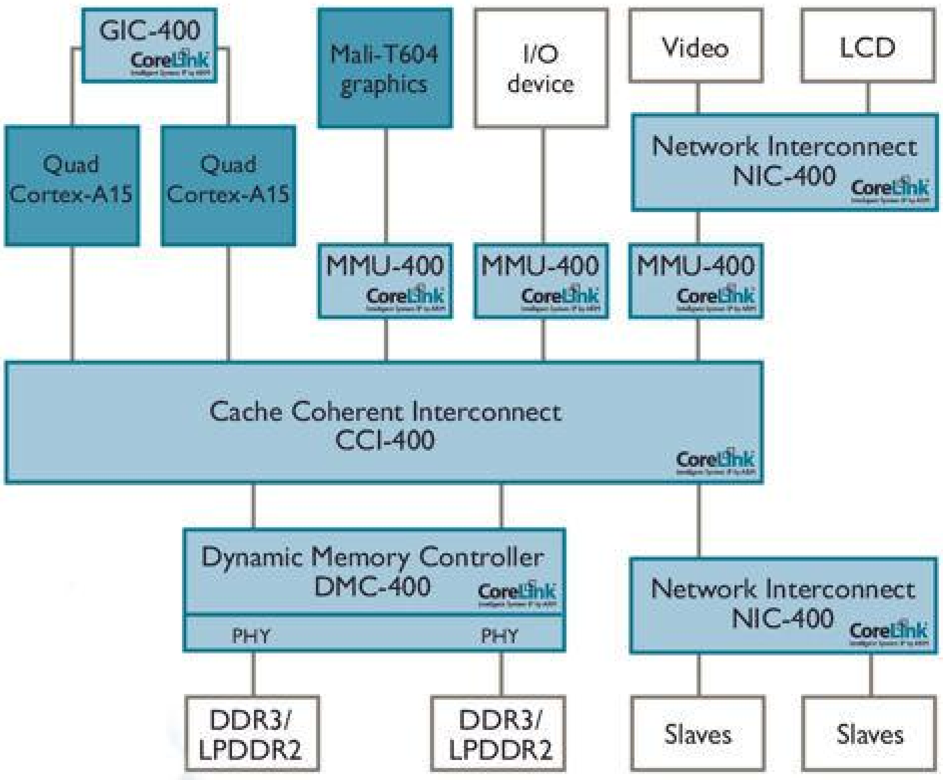
Last week, Facebook announced a new motherboard/daughtercard design it dubbed "Group Hug." Thanks to innovative work from the Open Compute Platform, the new daughtercard allows CPUs from ARM, Intel, or AMD to be plugged into a single motherboard. At least, that's the plan — for now, the hardware interconnects are still in the design phase, as is the framework required to manage disparate CPUs from multiple vendors.

There are, in other words, huge questions left to settle. Adding different daughtercards to the same server to optimize CPU usage is a great idea, but writing software that understands how to manage the various assets is still a huge task. In the rush to herald the advent of a bold new era for server commoditization and the advent of ARM servers, some significant factors are being ignored.

(http://www.extremetech.com/wp-content/uploads/2013/01/OpenComputeRack.jpg)

Here's the biggest: Contrary to what others have said, mobile SoCs will *not* "remake the server world (http://www.wired.com/wiredenterprise/2013/01/facebook-arm-chips/)." Wired attempts to draw a parallel between Facebook's experiments with turning off chip cache and the idea of replacing "brawny" CPU cores with "wimpy" ones. But it's not that simple.

Take a look at ARM's next-generation IP block for connecting up to 16 processors, their caches, and a variety of additional system devices, as compared to the CCI-400 that's currently shipping for the Cortex-A15 (http://www.extremetech.com/computing/141873-cortex-a15-posts-impressive-performance-threat-intel-amd).



(http://www.extremetech.com/wp-content/uploads/2013/01/CoreLink_400_Series.jpg)

The CoreLink 400 runs at half CPU speed

Here's the server variant.

A.11 Facebook

A.11.1 Facebook Company Information

introduced.

2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 | 2016

Stats

Headquarters

1 Hacker Way, Menlo Park, California 94025

Employees

14,495 employees as of June 30, 2016

Statistics

- 1.13 billion daily active users on average for June 2016
- 1.03 billion mobile daily active users on average for June 2016
- 1.71 billion monthly active users as of June 30, 2016
- 1.57 billion mobile monthly active users as of June 30, 2016
- Approximately 84.5% of our daily active users are outside the US and Canada

US offices

Atlanta, Austin, Boston, Chicago, Dallas, Detroit, Denver, Los Angeles, Menlo Park, Miami, New York, Reno, Seattle, Washington D.C.

International offices

Amsterdam, Auckland, Berlin, Brasilia, Brussels, Buenos Aires, Dubai, Dublin, Gurgaon, Hamburg, Hong Kong, Hyderabad, Jakarta, Johannesburg, Karlsruhe, Kuala Lumpur, London, Madrid, Melbourne, Mexico City, Milan, Montreal, Mumbai, New Delhi, Paris, Sao Paulo, Seoul, Singapore, Stockholm, Sydney, Tel Aviv, Tokyo, Toronto, Vancouver, Warsaw

Data Centers

Altoona, Forest City, Lulea, and Prineville, with a co-located facility in Ashburn

Our Culture

A.11.2 Introducing “Yosemite”: the first open source modular chassis for high-powered microservers

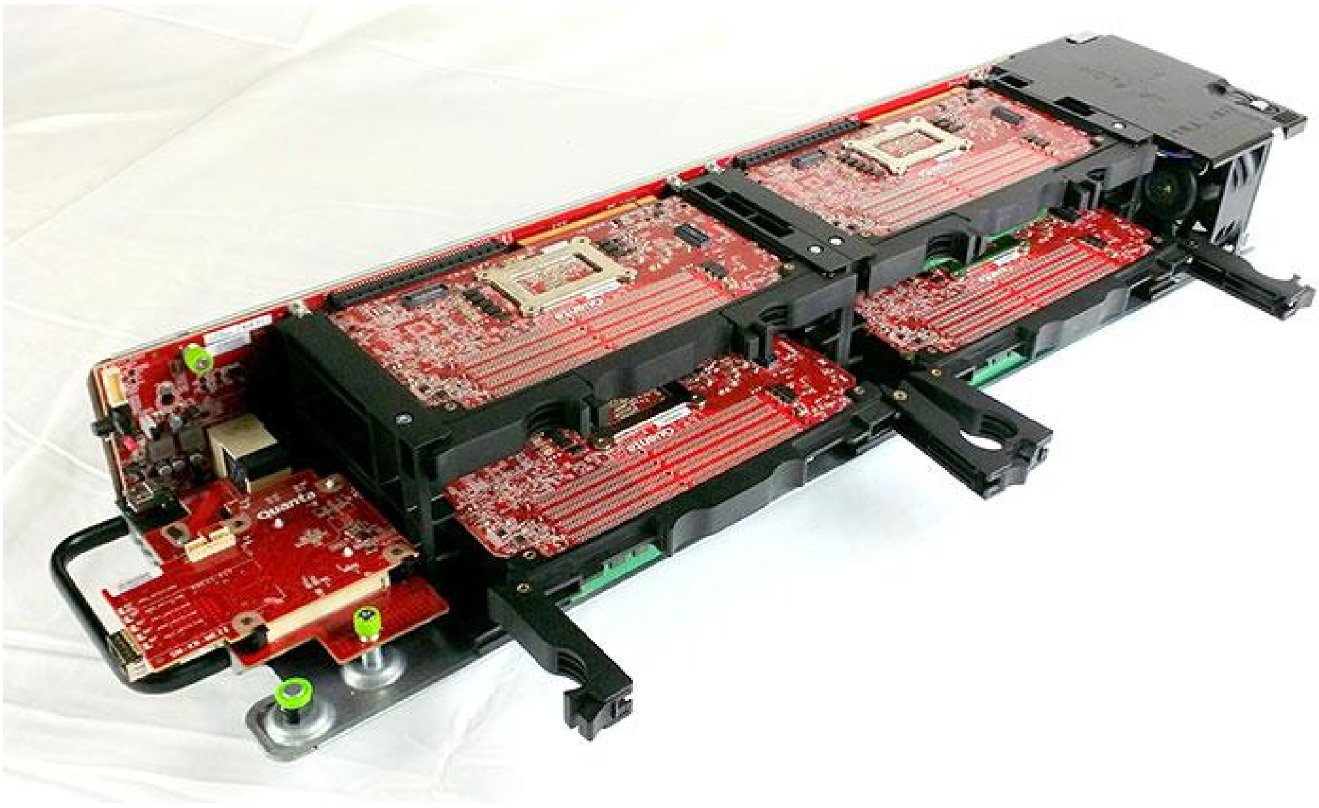
10 March 2015 INFRA · DATA · PERFORMANCE · OPEN COMPUTE · OPEN SOURCE · COMPUTE · DATA CENTERS · HARDWARE

Introducing "Yosemite": the first open source modular chassis for high-powered microservers

Hu Li

In hardware design, there are two approaches to solving the vast computing needs of a site like Facebook. There's the approach of “scale up” — building ever-increasing amounts of computing power in a given system. Or you can “scale out,” building an ever-increasing fleet of simple systems, each with a moderate amount of computing power.

This especially applies to two-socket (2S) computing platforms, which have become scale-up systems. 2S has been the mainstream server architecture for a long time for good reason. With multiple high-performance processors, it's strong and versatile, but it's also bulky and power-hungry. In other words, it's not optimized for scale-out uses. As we continued to evolve our infrastructure, we realized 2S was the wrong tool for some of our needs. To provide our infrastructure with capacity that scales out with the demand, we designed a modular chassis that contains high-powered system-on-a-chip (SoC) processor cards, code-named “Yosemite.” Today, we're proposing the Yosemite design as a contribution to the Open Compute Project for all members of the community to build on and deploy.



We started experimenting with SoCs about two years ago. At that time, the SoC products on the market were mostly lightweight, focusing on small cores and low power. Most of them were less than 30W. Our first approach was to pack up to 36 SoCs into a 2U enclosure, which could become up to 540 SoCs per rack. But that solution didn't work well because the single-thread performance was too low, resulting in higher latency for our web platform. Based on that experiment, we set our sights on higher-power processors while maintaining the modular SoC approach.

For Yosemite, we defined each server node as a pluggable module. Each module holds one SoC targeting up to 65W TDP, multiple memory channels with standard DDR DIMM slots, at least one local SSD interface, and a local management controller. We also standardized the module interface such that compliant cards and systems can interoperate. This interface is an extension from the original “Group Hug” OCP microserver interface, extended to provide more I/O through an additional PCI-E x16 connector. The Yosemite system holds four SoC cards consuming up to 400W total power, which provides about 90W for each SoC card. In order to simplify the external connectivity for this modular server system, we specify one shared network connection providing both data and management traffic.

Diving into the Yosemite design, we consider the following design elements to be important to the system:

- A server-class SoC with multiple memory channels, which provides high-performance computing in 65W TDP for SoC and 90W for the whole server card.
- A standard SoC card interface to provide a CPU-agnostic system interface.

- A platform-agnostic system management solution to manage the system and these 4 SoC server cards, regardless of vendor.
- A multi-host network interconnect card following OCP Mezzanine Card 2.0 specification, which connects up to 4 SoC server cards through a single Ethernet port.
- A cost-effective, flexible, and easy-to-service system structure.

This system will be fully compatible with Open Rack, which can accommodate up to 192 SoC server cards in a single rack. We're happy to say that Mellanox has already enabled the multi-host support in its next generation ConnectX®-4 OCP Mezzanine Card. With the design proposed as a contribution for OCP, we're excited to see what the rest of the community builds and deploys based on this submission.

Like

More to Read

Facebook Open Switching System ("FBOSS") and Wedge in the open

Recommended

Reflections on the Open Compute Summit

Introducing “6-pack”: the first open hardware modular switch

A.11.3 Introducing data center fabric, the next-generation Facebook data center network

Introducing the fabric

For our next-generation data center network design we challenged ourselves to make the entire data center building one high-performance network, instead of a hierarchically oversubscribed system of clusters. We also wanted a clear and easy path for rapid network deployment and performance scalability without ripping out or customizing massive previous infrastructures every time we need to build more capacity.

To achieve this, we took a disaggregated approach: Instead of the large devices and clusters, we broke the network up into small identical units – server pods – and created uniform high-performance connectivity between all pods in the data center.

There is nothing particularly special about a pod – it's just like a layer3 micro-cluster. The pod is not defined by any hard physical properties; it is simply a standard "unit of network" on our new fabric. Each pod is served by a set of four devices that we call fabric switches, maintaining the advantages of our current 3+1 four-post architecture for server rack TOR uplinks, and scalable beyond that if needed. Each TOR currently has 4 x 40G uplinks, providing 160G total bandwidth capacity for a rack of 10G-connected servers.

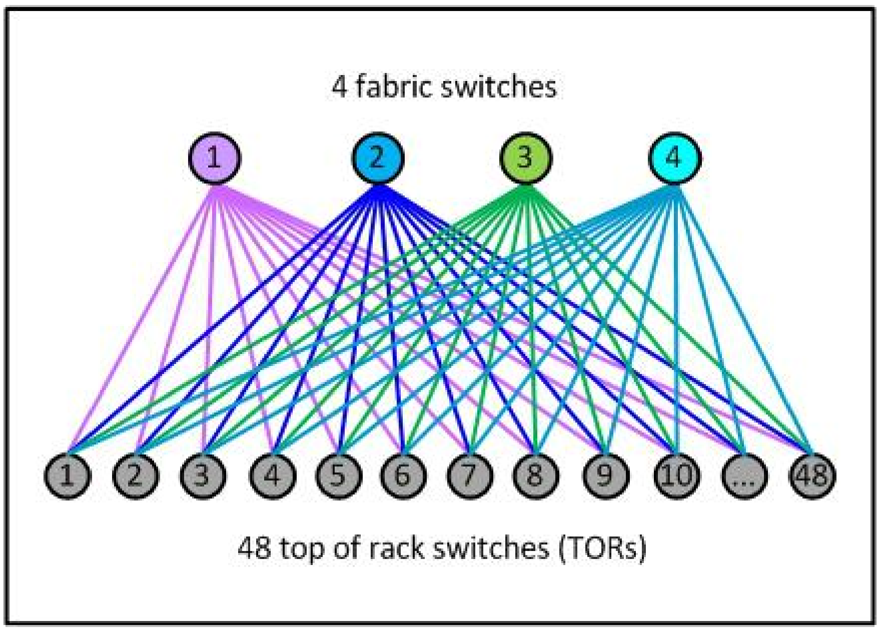


Figure 1: A sample pod – our new unit of network

What's different is the much smaller size of our new unit – each pod has only 48 server racks, and this form factor is always the same for all pods. It's an efficient building block that fits nicely into various data center floor plans, and it requires only basic mid-size switches to aggregate the TORs. The smaller port density of the fabric switches makes their internal architecture very simple, modular, and robust, and there are several easy-to-find options available from multiple sources.

Another notable difference is how the pods are connected together to form a data center network. For each downlink port to a TOR, we are reserving an equal amount of uplink capacity on the pod's fabric switches, which allows us to scale the network performance up to statistically non-blocking.

To implement building-wide connectivity, we created four independent "planes" of spine switches, each scalable up to 48 independent devices within a plane. Each fabric switch of each pod connects to each spine switch within its local plane. Together, pods and planes form a modular network topology capable of accommodating hundreds of thousands of 10G-connected servers, scaling to multi-petabit bisection bandwidth, and covering our data center buildings with non-oversubscribed rack-to-rack performance.

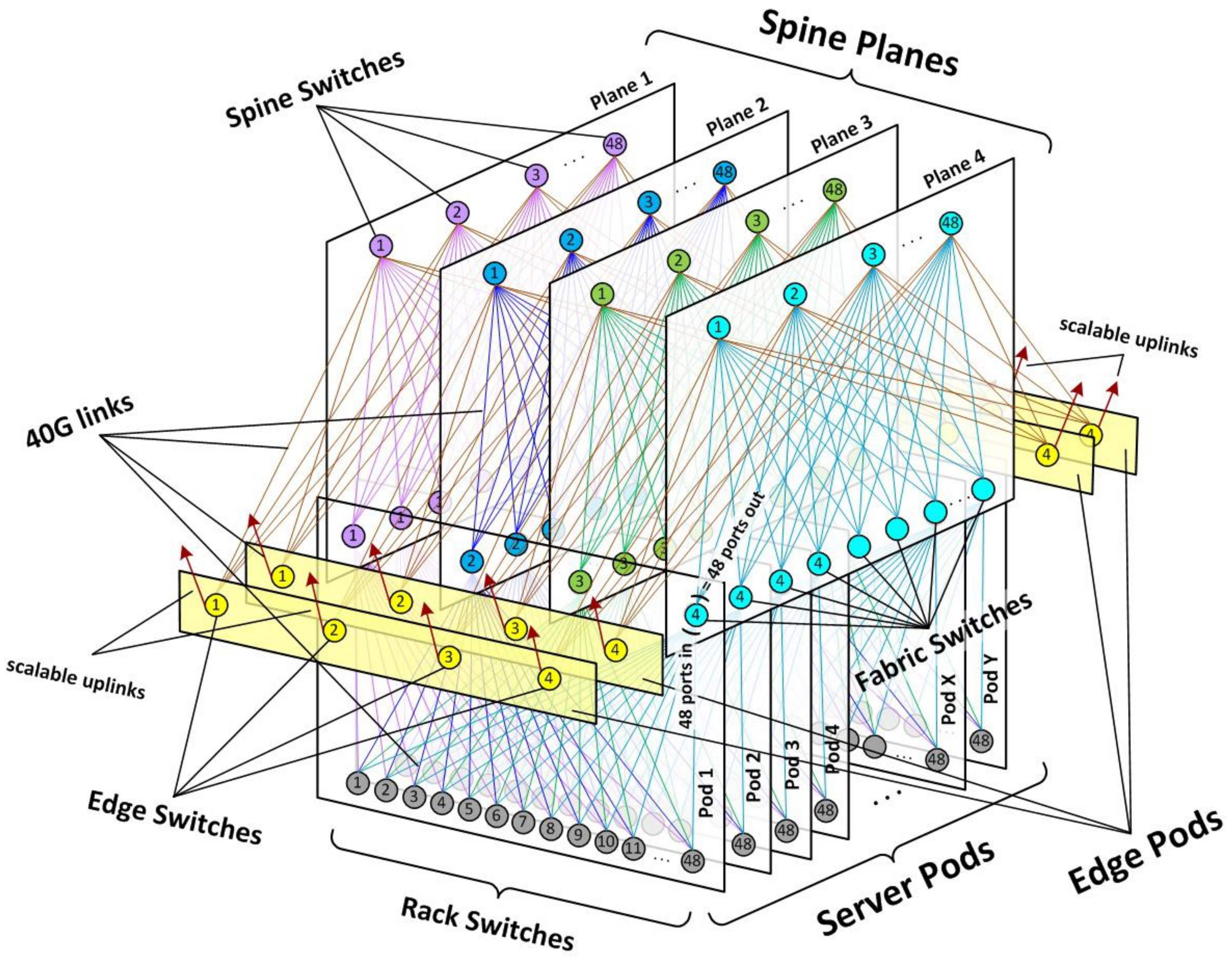


Figure 2: Schematic of Facebook data center fabric network topology

For external connectivity, we equipped our fabric with a flexible number of edge pods, each capable of providing up to 7.68Tbps to the backbone and to back-end inter-building fabrics on our data center sites, and scalable to 100G and higher port speeds within the same device form factors.

This highly modular design allows us to quickly scale capacity in any dimension, within a simple and uniform framework. When we need more compute capacity, we add server pods. When we need more intra-fabric network capacity, we add spine switches on all planes. When we need more extra-fabric connectivity, we add edge pods or scale uplinks on the existing edge switches.

How we did it

When we first thought about building the fabric, it seemed complicated and intimidating because of the number of devices and links. However, what we were able to achieve ended up being more simple, elegant, and operationally efficient than our customary cluster designs. Here's how we got there.

Network technology

We took a "top down" approach – thinking in terms of the overall network first, and then translating the necessary actions to individual topology elements and devices.

We were able to build our fabric using standard BGP4 as the only routing protocol. To keep things simple, we used only the minimum necessary protocol features. This enabled us to leverage the performance and scalability of a distributed control plane for convergence, while offering tight and granular routing propagation management and ensuring compatibility with a broad range of existing systems and software. At the same time, we developed a centralized BGP controller that is able to override any routing paths on the fabric by pure software decisions. We call this flexible hybrid approach “distributed control, centralized override.”

The network is all layer3 – from TOR uplinks to the edge. And like all our networks, it’s dual stack, natively supporting both IPv4 and IPv6. We’ve designed the routing in a way that minimizes the use of RIB and FIB resources, allowing us to leverage merchant silicon and keep the requirements to switches as basic as possible.

For most traffic, our fabric makes heavy use of equal-cost multi-path (ECMP) routing, with flow-based hashing. There are a very large number of diverse concurrent flows in a Facebook data center, and statistically we are seeing almost ideal load distribution across all fabric links. To prevent occasional “elephant flows” from taking over and degrading an end-to-end path, we’ve made the network multi-speed – with 40G links between all switches, while connecting the servers on 10G ports on the TORs. We also have server-side means to “hash away” and route around trouble spots, if they occur.

Gradual scalability

While we need a clear and predictable path to scale our capacity, we don’t necessarily require a non-blocking network in every deployment from day one.

To achieve the seamless growth capability, we’ve designed and planned the whole network as an end-to-end non-oversubscribed environment. We’ve allocated all necessary physical resources for full fabric device park, and we pre-built all the time-consuming passive infrastructure “skeleton” components. But our current starting point is 4:1 fabric oversubscription from rack to rack, with only 12 spines per plane, out of 48 possible. This level allows us to achieve the same forwarding capacity building-wide as what we previously had intra-cluster.

When the need comes, we can increase this capacity in granular steps, or we can quickly jump to 2:1 oversubscription, or even full 1:1 non-oversubscribed state at once. All we need to do is add more spine devices to each of the planes, and all physical and logical resources for that are already in place to make it a quick and simple operation.

Physical infrastructure

Despite the large scale of hundreds of thousands of fiber strands, fabric’s physical and cabling infrastructure is far less complex than it may appear from the logical network topology drawings. We’ve worked together across multiple Facebook infrastructure teams to optimize our third-generation data center building designs for fabric networks, shorten cabling lengths, and enable rapid deployment. Our Altoona data center is the first implementation of this new building layout.

A.11.4 Open networking advances with Wedge and FBOSS

CPU, and soon enough we are no longer even able to keep up BGP peering sessions, exacerbating the situation.

Now that the CPU has been protected, any situation that tries to send too much traffic to the switch CPU — including this one — is no longer a problem.

Looking ahead

As exciting as the Wedge and FBOSS journey has been, we still have a lot of work ahead of us. Eventually, our goal is to use Wedge for every top-of-rack switch throughout our data centers.

We've been working with a number of vendors and operators to help them start using Wedge and to share our experience with deployment of OCP switches in order to create a bigger open source ecosystem in networking. For example, Big Switch Networks now provides an Open Network Linux image that includes everything you need to run FBOSS on a Wedge and start programming it: **http://opennetlinux.org/wedge**.

We also don't plan to stop with the rack switch. Work is already underway in scaling our software to operate at higher speeds and handle higher complexity. We are working on Wedge 100, a 32x100G switch that we are looking forward to sharing with the networking community soon. This is in addition to our work adapting Wedge to a much bigger aggregation switch called **6-pack**, which uses Wedge as its foundation and stacks 12 of these Wedges in a modular and nonblocking aggregation switch. FBOSS as a software stack runs across our growing platform of network switches: Wedge, 6-pack, and now Wedge 100.

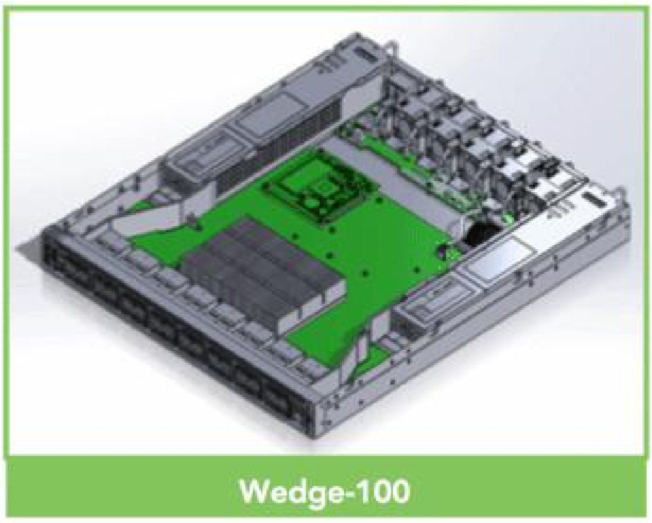
Thanks to the entire Wedge and FBOSS teams for all their hard work getting us to this scale so quickly. We're looking forward to sharing even more with the community.

Like

More to Read

mention-bot

A.11.5 Opening designs for 6-pack and Wedge 100

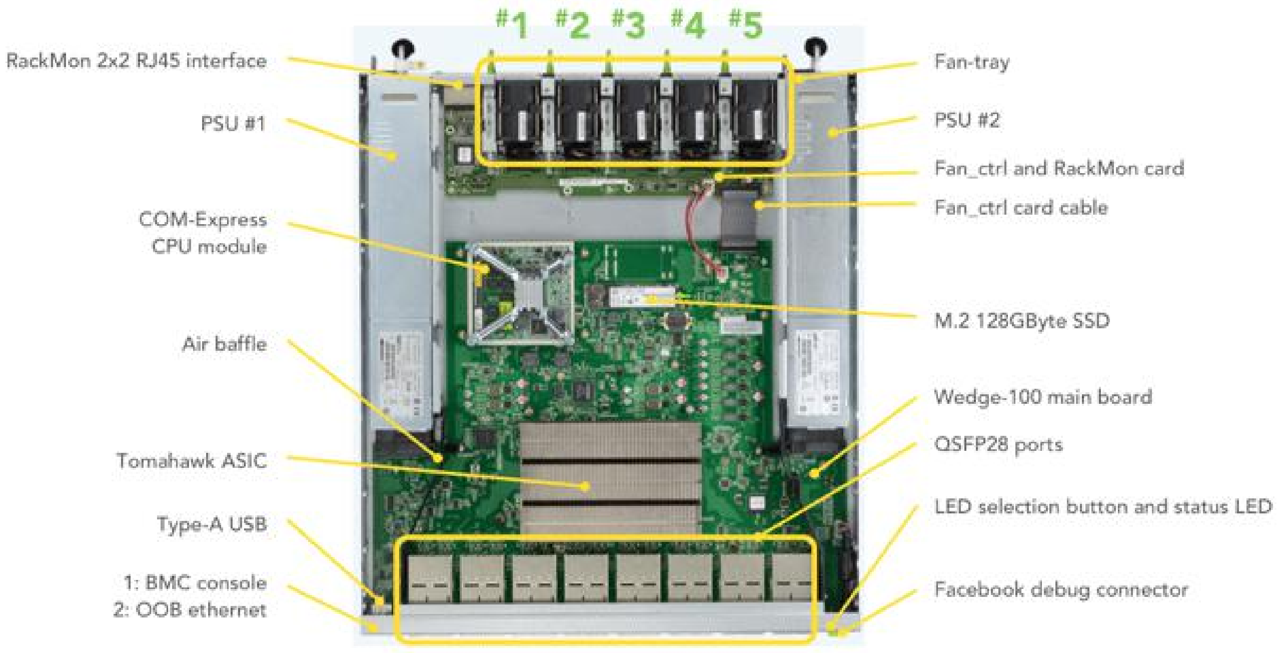


Wedge 100 – a 32x100G TOR switch

Wedge 100 is Facebook's second-generation TOR switch:

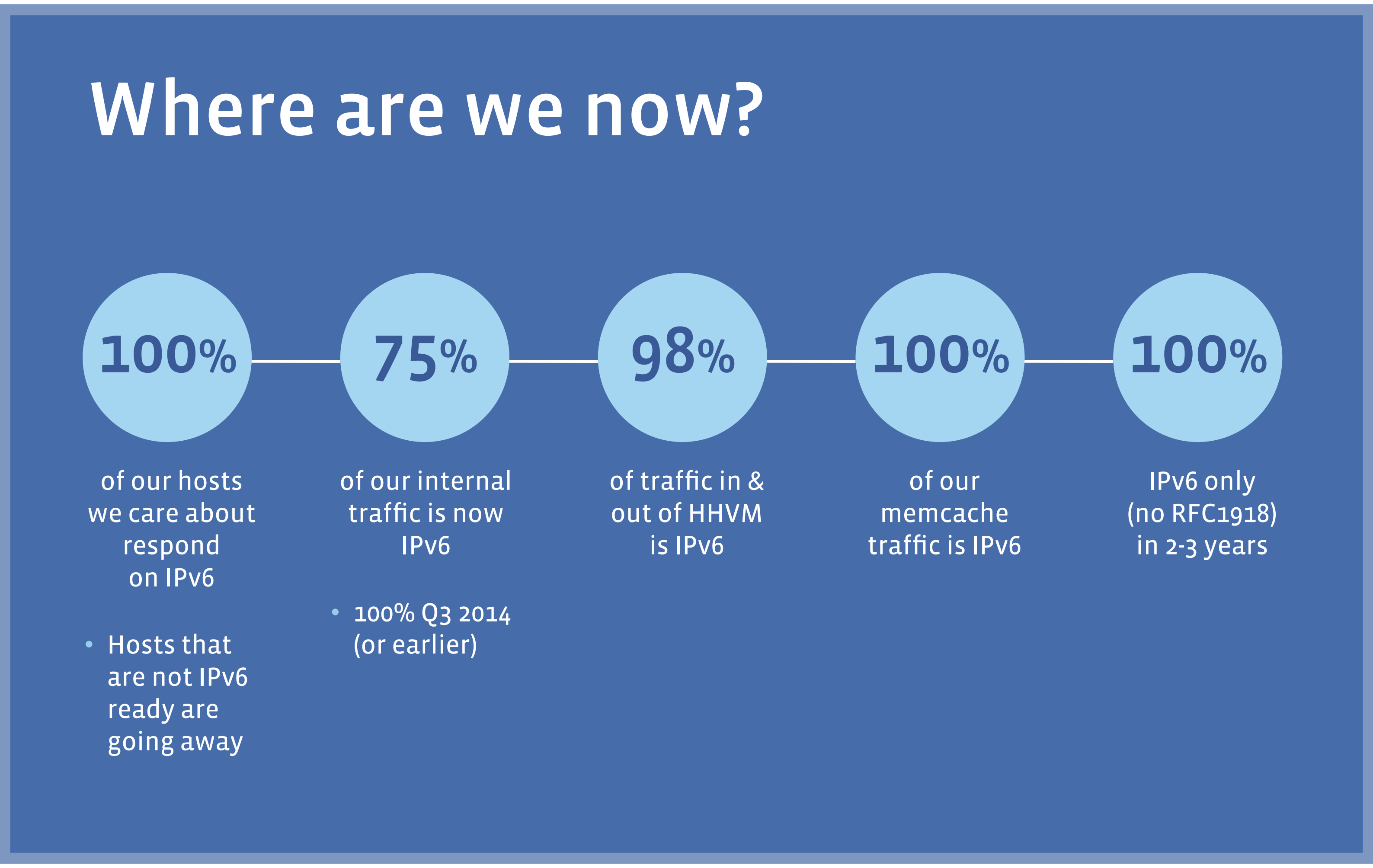
- Uses Broadcom's Tomahawk ASIC
- Supports the OpenRack v2 bus bar
- Now supports COM-E as the CPU module
- Supports the 100G QSFP28 DAC cables and 55C CWDM4 optic transceivers

We did this as a critical step toward supporting 100G throughout our data centers, especially for our new **Yosemite servers**.



A lot of what we've done in Wedge 100 is to accommodate 100G connectivity and support the 55C optic at an ambient 35C environment by having the five-fan tray, avoiding recycling air, and having multiple on-board temperature sensors.

A.11.6 Facebook IPv6 Public Group



Where are we now?

100% of our hosts we care about respond on IPv6

- Hosts that are not IPv6 ready are going away

75% of our internal traffic is now IPv6

- 100% Q3 2014 (or earlier)

98% of traffic in & out of HHVM is IPv6

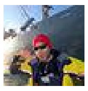
100% of our memcache traffic is IPv6

100% IPv6 only (no RFC1918) in 2-3 years

A.11.7 The Road To IPv6: Bumpy – slides

WorldIPv6Congress-IPv6_LH v2.pdf

RECENT ACTIVITY



Paul Saab uploaded a file.
21 March 2014

By popular demand, here's the presentation I did at the world v6 congress yesterday.

WorldIPv6Congress-IPv6_LH v2.pdf · version 1
Portable Document Format

Download Preview

Like Share

A.12 Frank McSherry

A.12.1 The impact of fast networks on graph analytics, part 1

The impact of fast networks on graph analytics, part 1

Jul 8, 2015

This is a joint post with Malte Schwarzkopf, cross-blogged here and at the CamSaS blog.

***tl;dr:* A recent NSDI paper argued that data analytics stacks don't get much faster at tasks like PageRank when given better networking, but this is likely just a property of the stack they evaluated (Spark and GraphX) rather than generally true. A different framework (timely dataflow) goes 6x faster than GraphX on a 1G network, which improves by 3x to 15-17x faster than GraphX on a 10G network.**

I spent the past few weeks visiting the CamSaS folks at theUniversity of Cambridge Computer Lab. Together, we did some interesting work, which we – Malte Schwarzkopf and I – are now going to tell you about.

Recently, a paper entitled "Making Sense of Performance in Data Analytics Frameworks" appeared at NSDI 2015. This paper contains some surprising results: in particular, it argues that data analytics stacks are limited more by CPU than they are by network or disk IO. Specifically,

> *"Network optimizations can only reduce job completion time by a median of at most 2%. The network is not a bottleneck because much less data is sent over the network than is transferred to and from disk. As a result, network I/O is mostly irrelevant to overall performance, even on 1Gbps networks." (§1)*

The measurements were done using Spark, but the authors argue that they generalize to other systems. We thought that this was surprising, as it doesn't match our experience with other data processing systems. In this blog post, we will look into whether these observations do indeed generalize.

A.13 Google

A.13.1 Efficient Data Center Summit 2009

Google

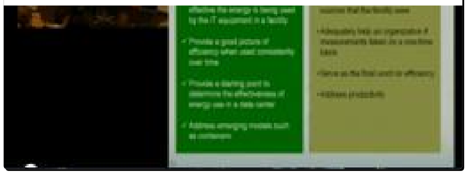
Data Centers

Efficient Data Center Summit 2009

On April 1, 2009, we hosted leaders of the IT industry to discuss best practices for improving data center efficiency. We spent the day discussing how to significantly reduce resource use while meeting service requirements. Not only is saving electricity and water good for the environment—it also makes good business sense. Being greener reduces operating costs and keeps our industry competitive.

For those who could not attend in person, we've published slide decks and videos of the entire day here:

1: Green Grid, Google PUE, Whats next



2: Best practices, data center tour, Q&A

VP & Distinguished Engineer, Amazon Web Services
Olivier Sanche
Senior Director Data Center Services, eBay
William Tschudi
PI, Applications Team, Environmental Energy Technologies Division, LBNL
Luiz Barroso, (moderator)
Distinguished Engineer, Google

3: Sustainability, water management

Google Data Center Video Tour

Presenter: Jimmy Clidaras, Google

Summary: A video tour of Google's high-efficiency power distribution architecture and cooling systems, traced from the equipment yard to the container. Technological improvements such as Google's high-efficiency UPS are shown in a production environment, as well as an explanation of close-coupled cooling, tight air flow management, and temperature control.

Additional Vitals

- 45 containers, approx. 40000 servers
- Single and 2-story on facing sides of hangar
- Bridge crane for container handling

| Slide 16 |

Slides

A.13.2 Compute Engine, Documentation, Networking and Firewalls

Networking and Firewalls

Contents

Google Compute Engine offers a configurable and flexible networking system that enables you to permit connections between the outside world and your instances. You can manage your Compute Engine network by configuring the network, firewall, and instance settings.

Networks

Your Cloud Platform Console project can contain multiple networks, and each network can have multiple instances attached to it. A network allows you to define a gateway IP and the network range for the instances attached to that network. By default, every project is provided with a `default` network with preset configurations and firewall rules. You can choose to customize the `default` network by adding or removing rules, or you can create new networks in that project. Generally, most users only need one network, although you can have up to five networks per project by default.

A network belongs to only one project, and each instance can only belong to one network. All Compute Engine networks use the IPv4 protocol. Compute Engine currently does not support IPv6. However, Google is a major advocate of IPv6 and it is an important future direction.

A.14 Hewlett Packard

A.14.1 The Machine is coming

Research The Machine

The Machine is coming

The Machine is a new kind of computer that allows you to do things you can't even conceive today. Imagine a computer with hundreds of petabytes of fast memory that remembers everything about your history, helps inform real time situational decisions, and enables you to predict, prevent, and respond to whatever the future brings.

For the past 60 years, we have been using the same computer systems and the only thing changing has been the massive amounts of data growing exponentially from our online world. By 2020, 30 billion connected devices will generate unprecedented amounts of data our legacy systems cannot keep up.

Hewlett Packard Labs is committed to revolutionizing the computer from the ground up, enabling computers of all sizes to take a quantum leap in performance and efficiency. It's all about turning all your big data into secure, actionable intelligence, using less energy and lowering costs.

The Machine puts the data first. Instead of processors, we put memory at the core of what we call "Memory-Driven Computing". Memory-Driven Computing collapses the memory and storage into one vast pool of memory called universal memory. To connect the memory and processing power, we're using advanced photonic fabric. Using light instead of electricity is key to rapidly accessing any part of the massive memory pool while using much less energy.

With The Machine, we believe we can broaden and impact technical innovations and develop new ways to extract knowledge and insights from large, complex collections of digital data with unprecedented scale and speed, allowing us to collectively help solve some of the world's most pressing technical, economic, and social challenges.

A.15 IEEE 802 LAN/MAN Standards Committee

A.15.1 400GBASE-SR16

Source: `http://www.intel.co.uk/content/dam/www/public/us/en/documents/product-briefs/xeon-processor-d-brief.pdf`. Accessed: 12 October 2016

Background

- 400G Ethernet (1st generation) is likely to use a 16 lane 25 Gb/s electrical interface (in each direction).
  - 16x25G most favoured electrical interface in a straw poll at 29th April 2014 Ethernet Alliance 400Gb/s Subcommittee meeting.
- Lowest cost, lowest power, PMDs tend to have a 1:1 mapping of electrical lanes to optical lane.
- Low initial volume for 400G MMF modules, probably dominated by breakout applications.
- 2x16 fibre connector: physical contact and expanded beam ferrules are in development.
  - "400G Optical Interconnection Options", Nathan Tracy, 29th April 2014, Ethernet Alliance 400Gb/s Subcommittee meeting.

A.16 Intel

A.16.1 Chip Shot: Custom Intel Xeon Sever Chip Boost Cloud Processing Power for Amazon Web Services Customers

Chip Shot (https://newsroom.intel.com/chip-shots/)

June 12, 2015

CHIP SHOT: CUSTOM INTEL® XEON® SEVER CHIP BOOST CLOUD PROCESSING POWER FOR AMAZON WEB SERVICES CUSTOMERS

Today, Amazon Web Services announced (https://aws.amazon.com/blogs/aws/the-new-m4-instance-type-bonus-price-reduction-on-m3-c4/) it is using another customized Intel® Xeon® server processor (http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e5-family.html) to deliver M4 instances. M4 instances are a new Amazon Elastic Compute Cloud (Amazon EC2 (http://aws.amazon.com/ec2/)) instance designed for small and mid-size databases and memory-intensive data processing tasks. The custom Intel Xeon processor E5-2676 v3 builds on the Amazon and Intel collaboration to develop and deliver optimal experiences for customers of Amazon EC2, a web service providing resizable compute capacity in the cloud. The custom Intel Xeon processor E5-2676 v3 for AWS runs at a base frequency of 2.4 GHz and can deliver clock speeds as high as 3.0 GHz with Intel® Turbo Boost. For more information on the AWS EC2 instances powered by Intel Xeon processors, visit Amazon EC2 Instances (http://aws.amazon.com/ec2/instance-types/).

Tags: Amazon (https://newsroom.intel.com/tag/amazon/), Cloud (https://newsroom.intel.com/tag/cloud/), Xeon (https://newsroom.intel.com/tag/xeon/)

Other News

August 25, 2016

A.16.2 Product brief, Intel Xeon Processor D-1500 Product Family, Extending Intelligence to the Edge

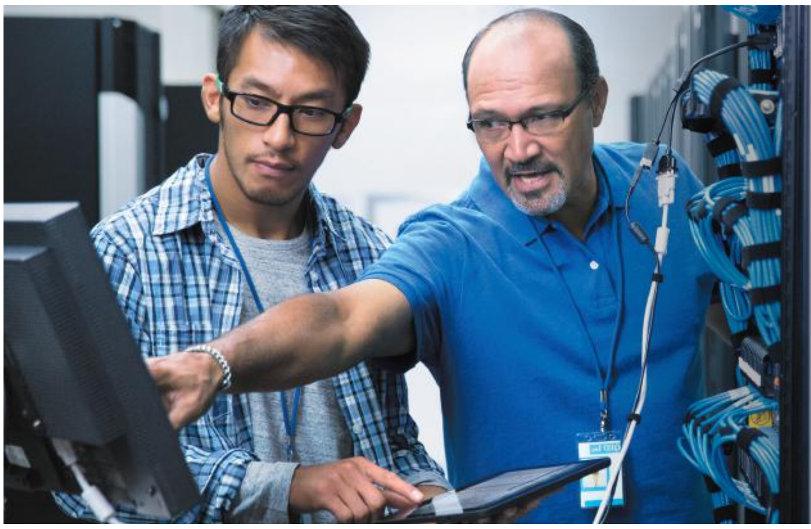
Source: `http://www.intel.co.uk/content/dam/www/public/us/en/documents/product-briefs/xeon-processor-d-brief.pdf`.

INTEL® XEON® PROCESSOR D PRODUCT FAMILY OVERVIEW

Intel® Xeon® Processor Intelligence in a Low-Power SoC	Up to 5.4x the networking performance[8,3] and up to 6.06x the storage performance[8,6] of the Intel® Atom™ processor C2750. Includes up to 16 cores (coming first quarter 2016), two integrated ports of 10 Gigabit Intel® Ethernet, plus support for up to 128 GB of memory. Also includes Intel® 64-bit software support,[9] L1 cache (32K data, 32K instructions per core), L2 cache (256K per core), LLC cache (1.5 MB per core), Intel® Turbo Boost Technology,[9] and Intel® Hyper-Threading Technology.[9]
Industry-Leading 14 nm Process Technology	Enables dense, low power system designs with thermal design points of ~20W to 45W and system level performance per watt of up to 1.7x that of Intel Atom processor C2750-based solutions[7,10].
Built-In Intel® Virtualization Technology	Delivers near-native compute and I/O performance in virtualized data centers, network infrastructure, and cloud computing, with advanced monitoring of cache and memory bandwidth for better service level and infrastructure management.
Server-Class Reliability, Availability, and Serviceability (RAS)	Provides high system reliability and data integrity with support for error correction code (ECC) memory, single device data correction (SDDC), memory demand and patrol scrubbing, and much more.
Hardware-Enhanced Security and Compliance	Intel Advanced Encryption Standard New Instructions (Intel AES-NI) provide integrated support for fast, low-overhead encryption and Intel® Trusted Execution Technology (Intel® TXT) provides platform verification (through authenticated boot) to enable strong security with reduced performance impact.
Server-Class Manageability	Includes Intel® Node Manager Base for adaptive power management.
Intel® Platform Storage Extensions	Enables fast data movement and high availability through integrated support for non-transparent bridging (NTB), asynchronous DRAM self-refresh (ADR), and Intel® QuickData technology, which provides a direct memory access (DMA) engine within the SoC.

INTEL® XEON® PROCESSOR D-1500 PRODUCT FAMILY SKU LIST

Processor Number (Standard SKUs)	CPU Cores	Memory Speed	CPU Speed	Max. DRAM Capacity	Intel Ethernet	Power
Intel® Xeon® processor D-1577	16	DDR4-2400	1.30 GHz	128 GB	2 x 10 GbE	45 W
Intel® Xeon® processor D-1571	16	DDR4-2400	1.30 GHz	128 GB	2 x 10 GbE	45 W
Intel® Xeon® processor D-1567	12	DDR4-2400	2.10 GHz	128 GB	2 x 10 GbE	65 W
Intel® Xeon® processor D-1557	12	DDR4-2400	1.50 GHz	128 GB	2 x 10 GbE	45 W
Intel® Xeon® processor D-1548	8	DDR4-2400	2.00 GHz	128 GB	2 x 10 GbE	45 W
Intel® Xeon® processor D-1541	8	DDR4-2400	2.10 GHz	128 GB	2 x 10 GbE	45 W
Intel® Xeon® processor D-1537	8	DDR4-2133	1.70 GHz	128 GB	2 x 10 GbE	35 W
Intel® Xeon® processor D-1531	6	DDR4-2133	2.20 GHz	128 GB	2 x 10 GbE	45 W
Intel® Xeon® processor D-1528	6	DDR4-2133	1.90 GHz	128 GB	2 x 10 GbE	35 W
Intel® Xeon® processor D-1527	4	DDR4-2133	2.20 GHz	128 GB	2 x 10 GbE	35 W
Intel® Xeon® processor D-1521	4	DDR4-2133	2.40 GHz	128 GB	2 x 10 GbE	45 W
Intel® Xeon® processor D-1518	4	DDR4-2133	2.20 GHz	128 GB	2 x 10 GbE	35 W



MORE INFORMATION

On the Intel® Xeon® processor D product family, visit **www.intel.com/xeond**.

A.16.3 The Intel Rack Scale Design Vision

Source: `http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design/rsd-vision-brochure.html`.

INTEL ® RACK SCALE DESIGN

The acceleration of time-to-market means getting your product into the market sooner by reducing the barriers to product development and scaling. To enable this, Intel® Rack Scale Design was designed to closely match the needs and workflows of the Agile Development, DevOps-driven organization. Intel® Rack Scale Design will introduce capabilities to speed time-to-market in a stepwise fashion, supporting companies as they learn how to be more flexible and adaptable over time.

» **Dynamic Resource Allocation** is the first step. Having the ability to fluidly assign and change storage, network, and compute resources—without having to physically rewire racks and servers—is a big benefit for the enterprise.

» **Agile Infrastructure** is what comes next. A big part of this agility comes when rack scale tools alert operators to bottlenecks or hotspots in the infrastructure, forecast and predict where these hotspots may occur, and recommend changes to the allocation of particular resources.

» **Automated Infrastructure** is the fulfillment of this part of the rack scale vision. Imagine routine alterations to the configuration of the data center—like responses to increased demand or graceful recovery from partial failure—being conducted automatically, with minimal oversight by staff. Better yet, imagine rack scale tools learning how to take action by monitoring how the staff adapts and changes the infrastructure, and then replicating best practices and even improvising better solutions, saving the staff's time and energy for the most interesting problems.

EVOLUTION

Greater transparency and control is all about creating a unified view of applications and resources, and this too is a process that increases in scope and capability over time as Intel® Rack Scale **Design**'s product roadmap evolves.

» **Glass Box Visibility** is the where it starts. The current version of Intel® Rack Scale Design will provide standards-based interfaces that enable a real-time view into how and what storage, network, and compute resource are being used across the data center.

» **Elastic Security** will transform how applications and networks are secured. Intel® Rack Scale Design's ability to dynamically reconfigure and reallocate resources will enable vendors to instantly deploy virtual security measures that will keep attackers out of any application or service they might try to compromise.

» **Intelligent Policy Pro iles** will be the standard way to express security and performance profiles, along with any critical nonfunctional requirements, like measures of service availability and response levels. These profiles will allow businesses to specify how they'd like applications to behave, and have Intel® Rack Scale Design comply automatically.

» **Enhanced Operations** is the culmination of the vision for transparency and control, because it will merge Glass Box Visibility, Elastic Security, and Intelligent Policy Profiles with the ability to manage and operate assets that exist outside of Intel® Rack Scale Design as well. This will radically simplify operations because staff will have one place to go to see what's happening and make the infrastructure do what they need it to do. On top of that, this capability will be open source, allowing unlimited extensibility and inclusion of any vendor's platform.

A.16.4 Intel Ethernet Controller XL710 Datasheet

Source: `http://www.intel.co.uk/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf`.

7.5.3.1.1 Initial setting of a linked list

Associating interrupt causes to an interrupt signal can be programmed by the software by any arbitrary order. As long as interrupts are not generated, the linked list is considered as static any programming order is acceptable.

7.5.3.1.2 Adding an interrupt cause to an active interrupt

Once an interrupt is active, its linked list is considered as dynamic resource. Special programming ordering is required when adding an interrupt cause as follow:

- Program the cause register as required while the NEXTQ_INDX and NEXTQ_TYPE parameters point to the next cause in the linked list.
- Update the NEXTQ_INDX and NEXTQ_TYPE parameters in the previous cause pointing to the added one.
- Note that there is no need to disable the interrupt before these two steps.

7.5.3.1.3 Removing an interrupt cause from an active interrupt

Removing an interrupt cause from an interrupt linked list can be done in one of the following 2 cases:

Case 1 - the interrupt is disabled and it is guaranteed that this interrupt does not have any possible pending interrupts. In this case, the linked list is considered static and there are no special programing ordering. The software should simply update the NEXTQ_INDX and NEXTQ_TYPE parameters in the previous cause to the next cause, skipping the cause that is removed from the linked list. At this point, the software can modify the cause register of the removed interrupt cause if required.

Case 2 - the interrupt is active. In this case, the linked list is considered dynamic resource and the software should follow a strict flow.

- Update the NEXTQ_INDX and NEXTQ_TYPE parameters in the previous cause to the next cause, skipping the cause that is removed from the linked list. In case it is the first cause in the linked list then update the FIRSTQ_INDX and FIRSTQ_TYPE parameters in the matched xxINT_LNKLSTx register, skipping the cause that is removed from the linked list.
- Clear the CAUSE_ENA flag in the matched xINT_xQCTL register.
- The software should wait "long enough" time till it is guaranteed that the hardware fetched the updated value of the previous cause. Waiting "long enough" time could be done by scheduling a software interrupt and waiting for that interrupt that follows.
- At this point, the software can modify the cause register of the removed interrupt cause if required.
- Note that there is no need to disable the interrupt before starting the above sequence.

7.5.4 Interrupt Moderation

The XL710 is able to throttle interrupts in two layered methods: Interrupt Throttling (ITR) and Interrupt Rate limiting (INTRL). These methods are detailed in the following subsections.

7.5.4.1 Interrupt Throttling (ITR)

Interrupt throttling (ITR) is a mechanism that guarantees a minimum gap between two consecutive interrupts (other than possible jitter caused by handling the interrupts). The XL710 counts the time since the last interrupt is scheduled and compares it against the ITR setting. If an event associated with this ITR happens before the ITR expires, the interrupt assertion is delayed until the ITR expires. If the ITR expires before any event associated with this interrupt, the interrupt logic is armed and the interrupt can be asserted the moment the event happens. The ITR intervals per vector are programmed by the xxINT_ITRx registers (xx stands for PF or VF and x stands for 0 or N). The ITR is measured in units of 2 μs.

The XL710 supports 3 ITRs per MSI-X vector as well as a NoITR option. The interrupt causes are mapped to one of the ITRs by the ITR_INDX field (per cause). The ITR intervals can be programmed directly to the xxINT_ITRx registers or via the xxINT_DYN_CTLx registers ('xx' stands for PF or VF and 'x' stands for '0' or 'N'). It might be useful to set the initial values using the xxINT_ITRx registers and dynamic update by the xxINT_DYN_CTLx registers as explained in step #4 of the interrupt sequence explained in Section 7.5.1.3. When any ITR interval of an interrupt with pending event is expired and the INTRL(*) credit is positive, the hardware follows the following steps:

- Clear the other ITRs of the same interrupt
- Process all causes of the same interrupt (associated to all ITRs) as defined by the linked list (described above in Section 7.5.3).
- (*) See next section for description of the interrupt rate limiting (INTRL)

7.5.4.2 Interrupt rate limiting (INTRL)

Interrupt rate limiting (INTRL) is a credit based mechanism that limits the maximum average number of interrupts per second. The PF controls its interrupt rate limit by the PFINT_RATE0 and PFINT_RATEN registers. The PF also controls its VF's interrupt rate limit by the VPINT_RATE0 and VPINT_RATEN registers. The control parameters of these registers are detailed below:

- INTRL_ENA: Enable / Disable option for the INTRL scheme. When disabled, interrupts can be generated without rate limiting control.
- INTERVAL: The INTRL is a 6 bit interval defined in 4 usec units that controls the time gap on which new interrupt credit is gained.

A.16.5 Intel Ethernet Switch FM5000/FM6000 Datasheet

Source: `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/Ethernet-switch-fm5000-fm6000-datasheet.pdf`.

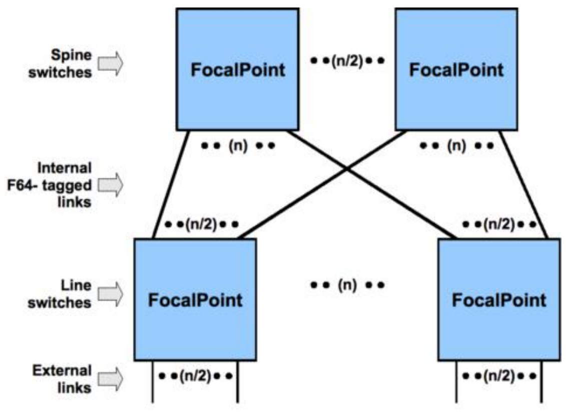


Figure 5-3 FM5000/FM6000 in a Tightly Coupled Clos Topology

5.3 Frame Processor

The FM5000/FM6000 series frame processor is designed to handle wire-speed L2/L3/L4 switching in the context of a single-chip or a multi-chip solution in a variety of topologies. The features offered are:

- Global Layer 3 (L3) routing over multiple devices in fat tree, ring or meshed topologies, with support for Equal Cost Multipath (ECMP) route selection
- L2 switching with optional automatic address learning and security
- Tunneling support for protocols such as TRILL, MPLS, VPWS, VPLS, Q-in-Q, MAC-in-MAC
- DCB support for PFC, ETS, QCN and DCBx
- Server virtualization support for protocols such as VEPA+
- Basic and extended Access Control Lists (ACLs) for L2/L3/L4 and deep packet inspection
- Snooping of IGMP v1, v2, and v3
- Link aggregation across multiple links using various sources of information from the frame header to derive the hashing function
- Trapping special frames
- Egress filtering and redirections including mirroring and logging
- Traffic policing with tricolor marking
- Congestion management
- IEEE 802.1ad provider bridging support
- Jumbo packet support (up to 15864 bytes)
- Cut through switching

The frame header pipeline is designed to process the following frame data:

- Source Port (7 bits)
- Source MAC Address (48 bits)

A.16.6 Intel 82599 10 GbE Controller Datasheet

Source: `http://www.intel.co.uk/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`.

1.4 Overview of New Capabilities Beyond the 82598

1.4.1 Security

The 82599 supports the IEEE P802.1AE LinkSec specification. It incorporates an inline packet crypto unit to support both privacy and integrity checks on a packet by packet basis. The transmit data path includes both encryption and signing engines. On the receive data path, the 82599 includes both decryption and integrity checkers. The crypto engines use the AES GCM algorithm, which is designed to support the 802.1AE protocol. Note that both host traffic and Manageability Controller (MC) management traffic might be subject to authentication and/or encryption.

The 82599 supports IPsec offload for a given number of flows. It is the operating system's responsibility to submit (to hardware) the most loaded flows in order to take maximum benefits of the IPsec offload in terms of CPU utilization savings. Main features are:

- Offload IPsec for up to 1024 Security Associations (SA) for each of Tx and Rx
- AH and ESP protocols for authentication and encryption
- AES-128-GMAC and AES-128-GCM crypto engines
- Transport mode encapsulation
- IPv4 and IPv6 versions (no options or extension headers)

1.4.2 Transmit Rate Limiting

The 82599 supports Transmit Rate Scheduler (TRS) in addition to the Data Center Bridging (DCB) functionality provided in the 82598. TRS is enabled for each transmit queue. The following modes of TRS are used:

- Frame Overhead — IPG is extended by a fixed value for all transmit queues.
- Payload Rate — IPG, stretched relative to frame size, provides pre-determined data (bytes) rates for each transmit queue.

1.4.3 Fibre Channel over Ethernet (FCoE)

Fibre Channel (FC) is the predominant protocol used in Storage Area Networks (SAN). Fibre Channel over Ethernet (FCoE) enables a connection between an Ethernet storage initiator and legacy FC storage targets or a complete Ethernet connection between a storage initiator and a device.

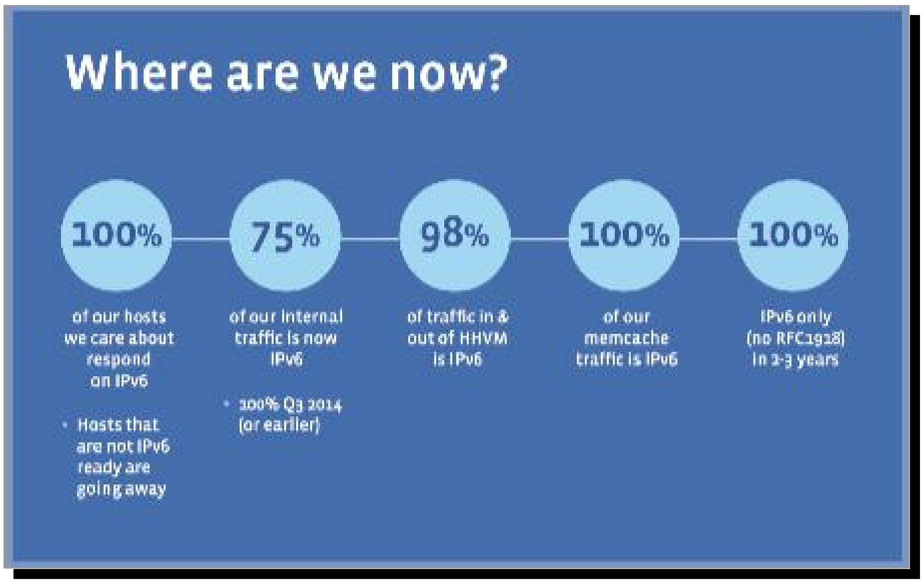
Existing FC Host Bus Adapters (HBAs) used to connect between FC initiator and FC targets provide full offload of the FC protocol to the initiator that enables maximizing storage performance. The 82599 offloads the main data path of I/O Read and Write commands to the storage target.

A.17 Internet Society

A.17.1 Facebook’s Extremely Impressive Internal Use of IPv6

Facebook’s Extremely Impressive Internal Use of IPv6

Wow! At the v6 World Congress this week in Paris (where Chris and Jan were), Facebook’s Paul Saab gave a very impressive presentation about what Facebook has gone through to convert its internal network over to IPv6. Paul has now posted his presentation online (in the IPv6 Group on Facebook, of course) and the story he relays with all the bumps and issues is great to see. Here’s the key slide at the end showing where they are at:



Where are we now?

100% of our hosts we care about respond on IPv6
- Hosts that are not IPv6 ready are going away

75% of our internal traffic is now IPv6
- 100% Q3 2014 (or earlier)

98% of traffic in & out of HHVM is IPv6

100% of our memcache traffic is IPv6

100% IPv6 only (no RFC1918) in 2-3 years

UPDATE: To view the slides at the link above, you need a Facebook login because the slides were posted to the IPv6 Group inside of Facebook. For those who don’t have a Facebook login, here is a copy of the slides stored on our server.

Those statistics are:

100% of hosts they care about respond on IPv6 (Hosts that are not IPv6 ready are going away.)
75% of internal traffic is now IPv6 with a goal to be at 100% by Q3 2014 or earlier
98% of traffic in and out of HHVM is IPv6
100% of our memcache traffic is IPv6
A goal of being 100% IPv6-only in 2-3 years

VERY impressive! Paul’s entire presentation is worth a read as he outlines a good number of the challenges they ran into, from vendors equipment not supporting IPv6 to engineers always writing in IPv6 to some of the problems they had with software. It’s all great info and good to have out there as a case study and for others to learn from.

I love that he ends noting that engineers are asking if they can start writing IPv6-only code today! (I also enjoy that the “solution” to stopping engineers from writing IPv4-only code was simple: take away IPv4 on development systems! 🙂)

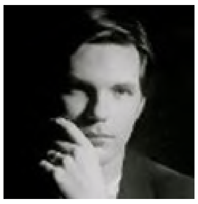
So… Facebook is going to be out in front of most other companies with having made the transition over to IPv6. What are *you* waiting for? Check out our IPv6 resources and let us know if there is anything more we can do to help you!

A.18 IPv6 World Congress 2014

A.18.1 THURSDAY 20 MARCH 2014 | CONFERENCE DAY TWO

IPv6 Content Session

10.30 Linking in with IPv6 at Linkedin



Franck Martin | LINKEDIN

10.50 The Road to IPv6: Bumpy

Paul Saab | FACEBOOK

Panel/Debate

11.10 **IPv6 for Enterprise: at the Edge and Inside**

MODERATOR

Yanick Pouffary | HP

PARTICIPANTS

Franck Martin | LINKEDIN
Paul Saab | FACEBOOK
Khalid Jawaid | CISCO

A.19 Linux

A.19.1 Kernel Newbies – Linux 3.18 release notes

1. Prominent features

1.1. Overlayfs

An overlay filesystem combines two filesystems, an 'upper' filesystem and a 'lower' filesystem, into a single file system namespace, modifications will be done to the upper filesystem. It has many uses, but it is most often used for live CDs, where a read-only OS image is used as lower filesystem and a writeable RAM-backed filesystem is used as the upper one. Any modifications will be done in the upper filesystem, thus allowing users to run the OS image provided normally. Overlayfs differs from other "union filesystem" implementations in that after a file is opened all operations go directly to the underlying, lower or upper, filesystems. This simplifies the implementation and allows native performance in these cases.

It is possible for both directory trees to be in the same filesystem and there is no requirement that the root of a filesystem be given for either upper or lower. The lower filesystem can be any filesystem supported by Linux and does not need to be writable. The lower filesystem can even be another overlayfs. The upper filesystem will normally be writable and if it is it must support the creation of trusted.* extended attributes, and must provide valid d_type in readdir() responses, so NFS is not suitable.

Documentation: commit Code: commit

1.2. Radeon: mapping of user pages into video memory

Linux 3.16 added the ability to map users addresses into the video memory for Intel hardware. In this release, AMD Radeon has also gained support for this feature. Normal application data can be used as a texture source or even as a render target (depending upon the capabilities of the chipset). This has a number of uses, with zero-copy downloads to the GPU and efficient readback making the intermixed streaming of CPU and GPU operations fairly efficient. This ability has many widespread implications from faster rendering of client-side software rasterisers (chromium), mitigation of stalls due to read back (Mozilla Firefox) and to faster pipelining of texture data (such as pixel buffer objects in OpenGL or data blobs in OpenCL).

Code: commit

1.3. bpf() syscall for eBFP virtual machine programs

bpf() syscall is a multiplexor for a range of different operations on eBPF which can be characterized as "universal in-kernel virtual machine". eBPF is similar to original Berkeley Packet Filter used to filter network packets. eBPF "extends" classic BPF in multiple ways including ability to call in-kernel helper functions and access shared data structures like eBPF maps. The programs can be written in a restricted C that is compiled into eBPF bytecode and executed on the eBPF virtual machine or JITed into native instruction set.

eBPF programs are similar to kernel modules. They are loaded by the user process and automatically unloaded when process exits. Each eBPF program is a safe run-to-completion set of instructions. eBPF verifier statically determines that the program terminates and is safe to execute. The programs are attached to different events. These events can be packets, tracepoint events and other types in the future. Beyond storing data the programs may call into in-kernel helper functions which may, for example, dump stack, do trace_printk or other forms of live kernel debugging.

Recommended LWN article: The BPF system call API, version 14

ebfp() man page and design documentation can be read on the merge commit: commit

Code: commit 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

1.4. TCP: Data Center TCP congestion algorithm

This release adds the Data Center TCP (DCTCP) congestion control algorithm. DCTCP is an enhancement to the TCP congestion control algorithm for data center networks. DCTCP has been designed for workloads typical of data center environments to provide/achieve: high burst tolerance, low latency and high throughput.

For more details about DCTCP, see the DCTCP web page

Code: commit

A.19.2 Linux Programmer's Manual - LD.SO(8)

LD_PRELOAD

A list of additional, user-specified, ELF shared objects to be loaded before all others. The items of the list can be separated by spaces or colons. This can be used to selectively override functions in other shared objects. The objects are searched for using the rules given under DESCRIPTION. In secure-execution mode, preload pathnames containing slashes are ignored, and shared objects in the standard search directories are loaded only if the set-user-ID mode bit is enabled on the shared object file.

Within the pathnames specified in **LD_PRELOAD**, the dynamic linker understands the strings *$ORIGIN*, *$LIB*, and *$PLATFORM* (or the versions using curly braces around the names) as described above in *Rpath token expansion*.

A.19.3 sendto(2) - Linux man page

ENOBUFS
The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion. (Normally, this does not occur in Linux. Packets are just silently dropped when a device queue overflows.)

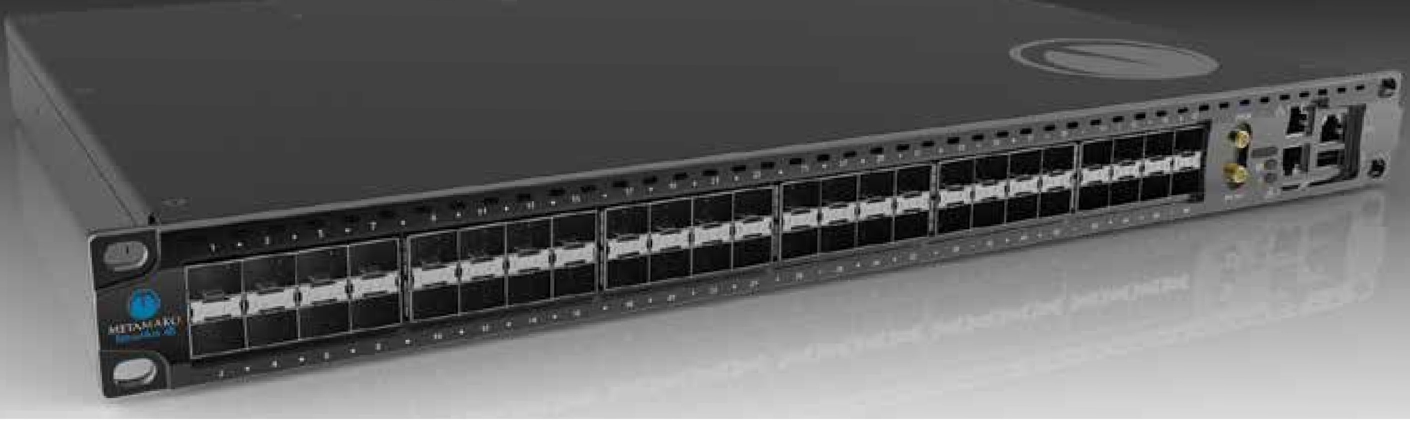
A.20 Metmako

A.20.1 MetaMux 48

Source: `https://www.metamako.com/Data%20Sheets/MetaMux48_data_sheet.pdf`

MetaMux 48

The hybrid ultra low-latency switch and FPGA platform. Applications include 80 ns packet aggregation, high resolution timestamping, fast filtering or 5 ns packet-aware layer 1 switching.



MetaMux 48 is a hybrid multi-layer switch designed for the most latency critical networks such as trading environments.

It is multiple switches in one – performing layer 1 switching in only 5 ns, multiplexing/aggregation in 80 ns, and layer 3 routing protocols, making it extremely useful for connecting trading machines directly to exchanges, or tapping and aggregating timestamped packets[1].

MetaMux is built on the same foundation as MetaConnect, so its layer 1 features include dynamic patching, tapping, one-to-many replication, media conversion, packet stats and precise timestamping. That means that the delay on downstream packets is barely detectable.

When using aggregation, packets can be multiplexed from many to one in 80 ns assuming no congestion, otherwise the packets are queued. That is less than half the latency and much more deterministic, compared with the best conventional switches.

MetaMux's hybrid approach allows for high performance layer 1 switching in addition to the flexibility of an applications platform with 80 ns packet aggregation, high resolution timestamping or fast filtering.

FEATURE	BENEFIT
Fast 48:1 multiplexing	Aggregate streams from multiple sources into a single stream for hand-off to exchanges, microwave links, or WAN links. Also configurable as multiple many-to-one multiplexers.
Integrated layer 1 switching	Patching, media conversion, tapping, replication and packet statistics. Wire-once to reduce visits to the data centre. Use Layer 1 broadcast to implement a return path with a latency of just 5 ns and virtually no jitter.
Deterministic	Know and rely upon your system's latency for fairness, or to get the best execution environment for your orders. Without contention, the MetaMux aggregation latency varies by +/- 7 ns.
Precision timestamping with synchronisation	Precisely timestamps packets on ingress using PPS or PTP synchronisation.
Flexible SFP/SFP+ support	Allows the use of less expensive modules, including direct attached copper cables, that are boosted by MetaMux's high performance signal recovery and regeneration.
Protocol flexibility	Protocol agnostic at layer 1 while layer 2 packet features support 1GbE and 10GbE with 100M. 100M Ethernet and rate conversion are planned for a future software release.
Fast filtering	Cut-through filtering based on specific ACLs.
Packet statistics	Advanced monitoring. Captures high-level packet statistics across all ports. Supports detailed switch statistics via SNMP or CLI. Provides tcpdump and LLDP on every port.
FPGA Development	Flexible platform with all 48 ports connected to the onboard FPGA for custom applications.
x86-64 Linux management	Uses open standards platform and MOS operating software to provide user extensible solutions.

A.21 Microsoft

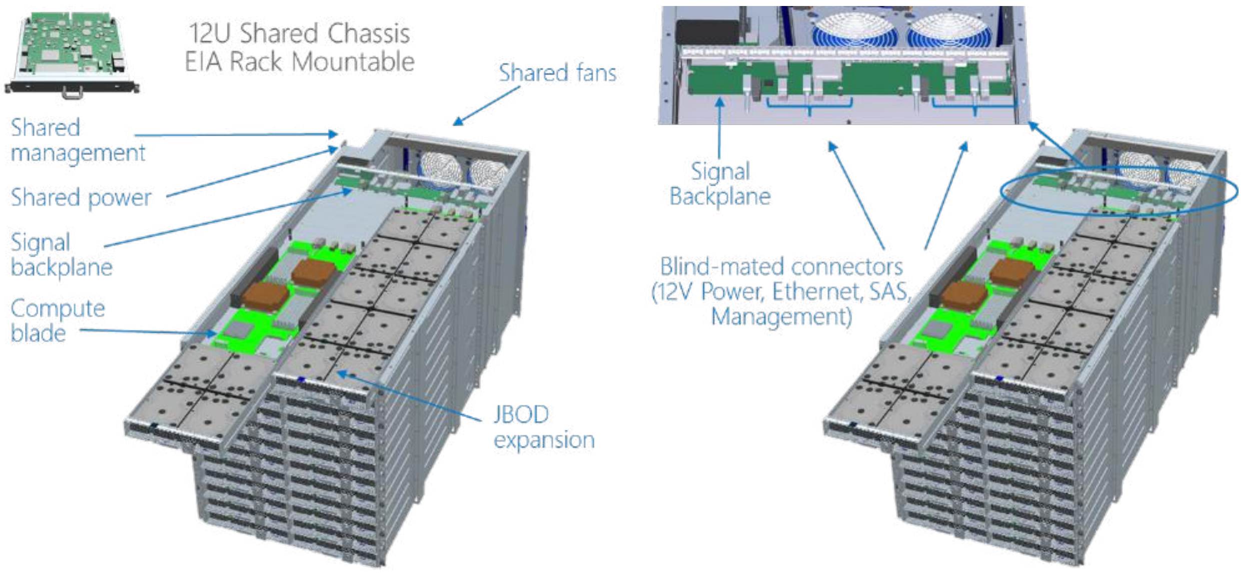
A.21.1 Microsoft's Open CloudServer

Source: `http://download.microsoft.com/download/B/1/7/B179029E-7AE8-447A-B8C9`
`Microsofts_Open_CloudServer_Strategy_Brief.pdf`

Microsoft's CloudServer

Microsoft has been deploying online services for over two decades and we brought our experience and learnings to develop the Open CloudServer V1, which we shared with the Open Compute Foundation and industry in early 2014. This new architecture for hardware and software converged all of our key cloud services, such as Bing, Office 365, and the Microsoft Azure platform on a common platform framework. Reducing total cost of ownership (TCO) was a key aspect. We strived to keep the acquisition costs low and reduce operational expenses, and we took a holistic look at the full server lifecycle– from architecture design through eventual decommissioning.

The result of this effort was a fully integrated design from the silicon, to the rack, and all the way to the datacenter level. It incorporates the blade, storage, network, systems management, and power mechanicals. And it comes together in a highly efficient single modular design. This cloud server design has been optimized for managing and operating an installed base of more than one million servers across a global footprint of Microsoft's datacenters.



Performance results

The servers built against this design are currently in production in Microsoft datacenters and are yielding significant advantages over the traditional enterprise servers they replace:

- Up to **40% cost savings** and **15% power efficiency** benefits vs. traditional enterprise servers
- Up to **50% improvement** in deployment and service times
- Up to **75% improvement** in operational agility vs. traditional enterprise servers
- Is expected to save **10,000 tons of metal** and **1,100 miles of cable** for a deployment of one million servers

Open CloudServer v2

In a sign of commitment to the Open Compute project, we released the V2 specifications in October 2014. As with our original design specification, we donated the hardware specifications for the chassis, the PCBA board, gerber files, and the mechanical CAD models for the metal chassis. In addition, we open sourced the management software and the tools we use during deployments, service, and repair. Vendors will be able to build blades that interoperate within the chassis and users will be able to deploy, manage, and service complete cloud systems.

The heart of the Open CloudServer V2 upgrade is a new compute blade. This blade supports the latest Intel processor, enabling 28 cores per blade. More cores enable more virtual machines, leading to a need to rebalance the system. The new design added capacity increases in all of the other subsystems on the blade.

The V2 blade design supports the transition from 10G networking to 40G networking. The new design utilizes RDMA over Converged Ethernet, or ROCE v2, to improve network efficiencies moving data to and from storage. It increases the SSD flash capacity by 4 times by transitioning away from SATA-based SSD flash storage to the PCI-Express based M.2 form factor. The M.2 flash cards are used in most laptops and tablets today, and are transitioning to the high performance NVMe interface. The M.2's small "stick" form factors enables thermal efficiency improvements, resulting in lower power consumption for cooling fans.

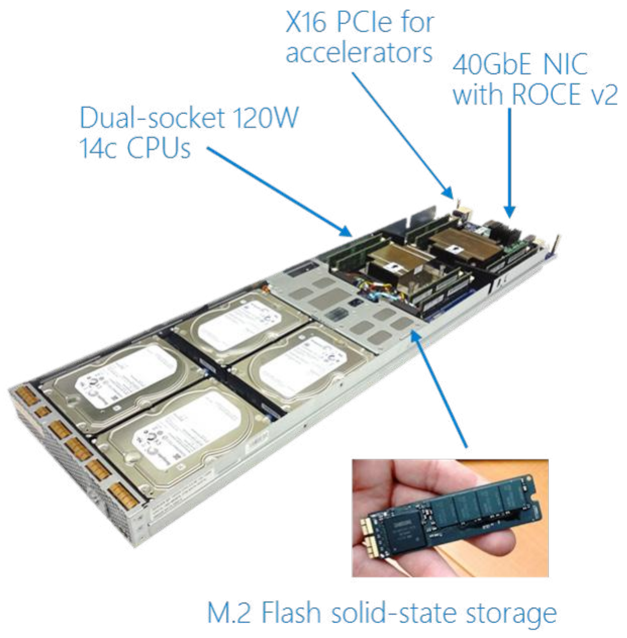
To improve the efficiencies of the processors, the design enables an expansion card that can be accommodate items such as GPU and FPGA accelerator cards.

In the chassis, the capacity of the power supplies was increased and optimized for our hyper-scale cloud datacenter configurations. Utilizing a standard 19" EIA rack was very important in creating a flexible platform. To increase overall efficiency, the management, cooling, and power is pooled and shared across 24 blades. The blades can be any combination of compute or storage to meet flexible requirements.

Management of the blades and chassis is through the Chassis Manager, essentially an x86 PC. The Chassis Manager is responsible for setting fan speeds, monitoring the health of power supplies and fans, gathering event logs, and for monitoring the minimum set of out of band features required for running the servers.

There are six shared power supplies that allow for balanced three-phase power at the datacenter, allowing full utilization. Six large fans are used to reduce power consumption and enable fault redundancy. The assembly is designed for low-cost manufacturing and pre-rack assembly before arriving at the datacenter.

The overall result is that a full 52U rack with 96 servers can support 2688 cores, 48TB memory, 2.3 PB disk storage, 3/4ths PB flash storage, and 3.8 Tbps network bandwidth.


X16 PCIe for accelerators
40GbE NIC with ROCE v2
Dual-socket 120W 14c CPUs
M.2 Flash solid-state storage

Microsoft Open CloudServer V2 enhancements

A.21.2 Azure Purchase FAQ

Azure Purchase FAQ

Does Azure support IPv6?

Microsoft has played a leading role in helping customers to smoothly transition from IPv4 to IPv6 for the past several years. To date, Microsoft has built IPv6 support into many of its products and solutions like Window 8 and Windows Server 2012 R2. Microsoft is committed to expanding the worldwide capabilities of the Internet through IPv6 and enabling a variety valuable and exciting scenarios, including peer-to-peer and mobile applications. The foundational work to enable IPv6 in the Azure environment is well underway. However, we are unable to share a date when IPv6 support will be generally available at this time. For more information on IPv6 technologies and IPv6 support available in the Windov operating system today, see Microsoft's IPv6 information site which include business, technical, and developer resources: http://technet.microsoft.com/en-us/network/bb530961

A.21.3 Data Center Transmission Control Protocol (DCTCP)

Data Center Transmission Control Protocol (DCTCP)

Updated: May 9, 2012

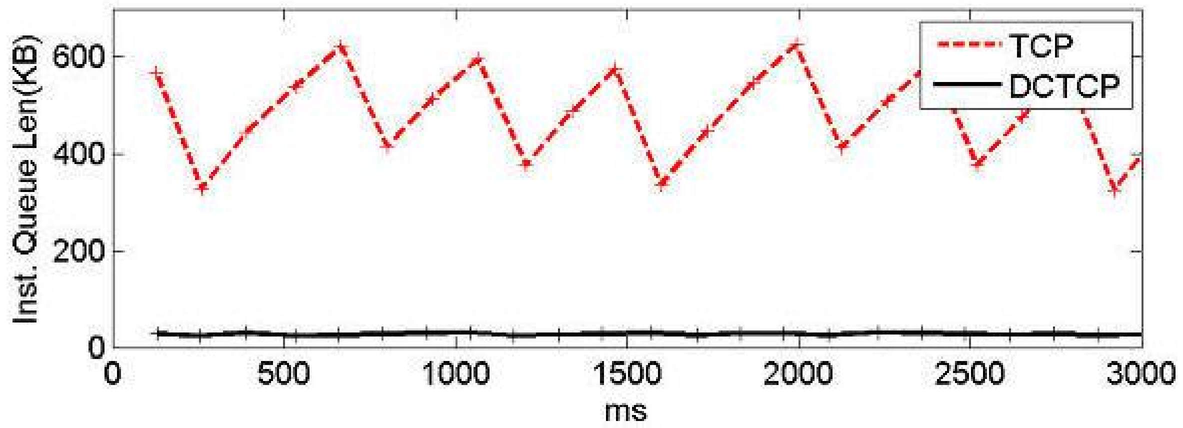
Applies To: Windows Server 2012

Data Center Transmission Control Protocol (DCTCP)

Data centers host diverse applications, mixing on the same network a variety of workflows that require small, predictable latency, while other applications require large, sustained throughput. In this environment, today's state-of-the-art Transmission Control Protocol (TCP) congestion control mechanisms do not provide sufficiently detailed congestion control settings. This results in queue formations in network switches leading to delays, fluctuations in latency, and timeouts.

To reduce this problem, Windows Server 2012 introduces DCTCP, which uses Explicit Congestion Notification (ECN) to estimate the extent of the congestion at the source, and reduce the sending rate only to the extent of the congestion. This provides a more detailed control over network traffic, allowing DCTCP to operate with very low buffer occupancies while still achieving high throughput.

The following illustration demonstrates the effectiveness of DCTCP in achieving full throughput while taking up a very small footprint in an Ethernet switch packet buffer, as compared to traditional TCP. The graph depicts the queue length in the network switch when using DCTCP and TCP. Two separate 1 gigabits per second (Gbps) TCP/IP streams are directed into two separate switch ports, and are combined into a single outgoing 1 Gbps port.



With traditional TCP, long-lived, large volume TCP flows cause the length of the bottleneck queue to grow until packets are dropped, resulting in the sawtooth pattern of the TCP traffic. When long and short flows traverse the same queue, two impairments occur. First, packet loss on the short flows may occur as described above. Second, there is a queue buildup even when no packets are lost. The short flows experience increased latency as they are queued behind packets from the large flows.

DCTCP, however, provides earlier, more detailed responses to congestion, which effectively fine-tune the sending rates at each source to operate with much smaller message queue build-up in the switch, while maintaining the same aggregate throughput. The much lower queue lengths exhibited with DCTCP avoid the latency and variation in latency that occur with TCP.

When used with commodity, shallow-buffered switches, DCTCP delivers the same or better throughput than TCP, while using 90% less buffer space in the network infrastructure. Unlike TCP, it also provides high burst tolerance and low latency for short flows. While the limitations of TCP cause the mistransmission of traffic sent

A.22 Netcraft

A.22.1 September 2016 Netcraft Survey

September 2016 Web Server Survey

In the **September 2016** survey we received responses from **1,285,759,146** sites and **6,118,785** web-facing computers, reflecting large gains in both metrics: 132 million additional sites, and 138,000 more computers.

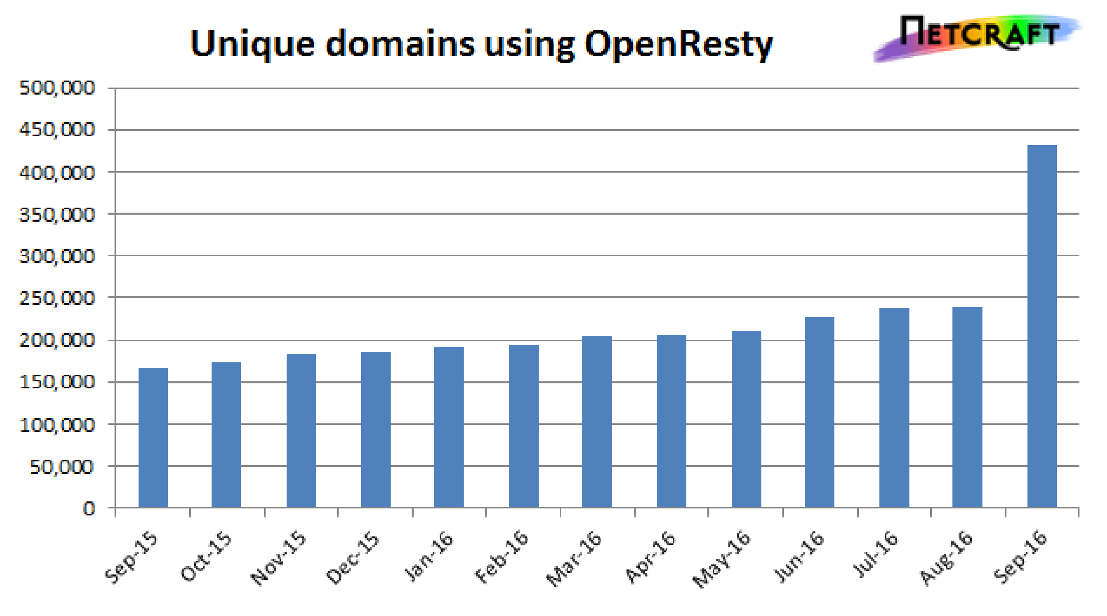
Microsoft made up the majority of this month's website growth, with the largest gain of 97 million sites, although it showed only modest increases of 5,200 web-facing computers and 693,000 active sites.

Apache was responsible for most of this month's additional web-facing computers, increasing its count by 87,000 to 2.8 million (+3.2%). Similarly, nginx made a 3.0% gain of 30,000 computers. However, Microsoft's 0.3% gain was not enough to stop its share falling by half a percentage point to 25.3% as a result of the gains made by Apache and nginx.

Although nginx made a healthy gain in web-facing computers, it lost more than 5 million active sites and 5,600 sites within the top-million. 27.6% of the busiest million sites now use nginx (-0.56 pp from last month), while Apache retains its lead with a 42.5% share.

Along with nginx, all of the major web server vendors suffered losses within the top million sites, largely due to the growth of OpenResty this month. More than 10,000 of the top million sites are now using OpenResty, compared with fewer than 4,000 last month, after millions of Tumblr blogs switched from nginx. As well as tumblr.com, basecamp.com — the home of the Basecamp web-based project management tool — ranks amongst the most visited sites to use OpenResty.

Tumblr's adoption of OpenResty has caused the web server to leap up the rankings to become the seventh largest web server vendor by websites, and fifth by active sites. This month, 87% of all OpenResty sites appear under the tumblr.com domain.



Although most OpenResty sites reside under the tumblr.com domain, the number of unique domains using OpenResty also increased noticeably this month.

Switching from nginx to OpenResty is not such a paradigm shift as moving to, say, Apache or Microsoft IIS. The OpenResty web application platform is built around the standard nginx core, which offers some familiarity, as well

A.23 OpenCompute Project

A.23.1 Server/SpecsAndDesigns

Server/SpecsAndDesigns

From OpenCompute
(Redirected from Motherboard/SpecsAndDesigns)

Contents

Specs and Designs

This page contains links to the current Specs and Designs that have been contributed to OCP.

Link to Server Committee wiki home http://www.opencompute.org/wiki/Server

A template specification is located here (http://files.opencompute.org/oc/public.php?service=files&t=46e5ed20913f672865bb941c5c1f0285) . Be sure to ensure that you've got the correct license in place according to your company's submission policies.

Open CloudServer

A chassis system that utilizes EIA 310-D 19" racks has been contributed by Microsoft. The system has been built with operational knowledge gained by operating over a million servers. The 12U chassis has dedicated, hard-wired out of band management, phase-balanced power, and high efficiency cooling. All of these are accepted by IC

Open CloudServer (OCS) Chassis

Open CloudServer OCS V2.1 Specification	Version	Submit Date	Contributor	License	Notes
Microsoft OCS Updates Chassis Manager Update spec+collateral (aka CMv2) (http://files.opencompute.org/oc/public.php?service=files&t=111766962f86d4d65e20914917f756ca) Blade spec update (http://files.opencompute.org/oc/public.php?service=files&t=dcc019cf49972add62d90d58a783ee41)	v2.1	Feb 09, 2016	Microsoft	OWFa 1.0	Refresh for Future Intel® Xeon® Pr family

Open CloudServer OCS V2 Specification	Version	Submit Date	Contributor	License	Notes
Chassis (http://files.opencompute.org/oc/public.php?service=files&t=795ec754e6c94de697fd79a6404730f1)	v2.0	Oct 30, 2014	Microsoft	OWFa 1.0	
Chassis Management (http://files.opencompute.org/oc/public.php?service=files&t=7421ce14df325e8c48a2abd93d3d649c)	v2.0	Oct 30, 2014	Microsoft	OWFa 1.0	
Blade (http://files.opencompute.org/oc/public.php?service=files&t=1c2ed966035b8b83aaeadc80b4a5b356)	v2.0	Oct 30, 2014	Microsoft	OWFa 1.0	
Blade NIC Mezzanine (http://files.opencompute.org/oc/public.php?service=files&t=a3cc433a091c398b9b8c91be84b0e215)	v2.0	Oct 30, 2014	Microsoft	OWFa 1.0	
Tray Mezzanine (http://files.opencompute.org/oc/public.php?service=files&t=ee9283903f328a46731a0a480ea15c15)	v2.0	Oct 30, 2014	Microsoft	OWFa 1.0	
Chassis Manager Service Open Source Location https://github.com/MSOpenTech/ChassisManager OCS Operations Toolkit Open Source Location https://github.com/MSOpenTech/OCSOperationsToolKit OCS Mechanical Step Files OCS_Mechanical_120314.zip, 66MB (http://files.opencompute.org/oc/public.php?service=files&t=749387d0a9e9420d7973f5f0db33f832) OCS Tray backplane and power distribution board collateral (Chassis Manager, use v1) OCSv2_TBP_PDB.zip, 31MB (http://files.opencompute.org/oc/public.php?service=files&t=66ad36190d19eae3d3cf99e247d1acf8)	v2.0	Oct 30, 2014	Microsoft	OWFa 1.0	
OCS Open CloudServer Power Supply v2.0 (http://files.opencompute.org/oc/public.php?service=files&t=4bb682624927b315b4f3d8a30f022c19) OCS Open CloudServer Power Supply Mechanical v2.0 (http://files.opencompute.org/oc/public.php?service=files&t=1a3206d2892a699cdf8f02c05dd180dd)	v2.0	Jan 20, 2015	Microsoft	OWFa 1.0	Power Supply with integrated backu Accepted by IC 11/6/2015
OCS Open CloudServer Solid State Drive v2.0 (http://files.opencompute.org/oc/public.php?service=files&t=506cbcb9b88e35f335eb85c5a60be54a)	v2.0	Feb 28, 2015	Microsoft	OWFa 1.0	M.2 NVME Solid State Flash Drive Accepted by IC 11/6/2015
OCS Open CloudServer Solid State Drive v2.1 (http://files.opencompute.org/oc/public.php?service=files&t=b58eb742adc395c1ed86f7f768989918)	v2.1	Aug 14, 2015	Microsoft	OWFa 1.0	M.2 NVME Solid State Flash Drive Capacity raised to 960GB and only N Accepted by IC 11/6/2015

Open CloudServer OCS V1 Specification	Version	Submit Date	Contributor	License
Blade (http://files.opencompute.org/oc/public.php?service=files&t=f479e4ad3f6ad681befe7c15bd200aaa)	v1.0	Jan 28, 2014	Microsoft	OWFa 1.0
Chassis (http://files.opencompute.org/oc/public.php?service=files&t=15159413e31833aa7b51d1421d3e89c7)	v1.0	Jan 28, 2014	Microsoft	OWFa 1.0
Chassis Management (http://files.opencompute.org/oc/public.php?service=files&t=159797cbb4d43184213dec1fd6156e71)	v1.0	Jan 28, 2014	Microsoft	OWFa 1.0
JBOD Blade (http://files.opencompute.org/oc/public.php?service=files&t=b6ef7174fe22e054302341aa7cc17d37)	v1.0	Jan 28, 2014	Microsoft	OWFa 1.0
NIC Mezzanine (http://files.opencompute.org/oc/public.php?	v1.0	Jan 28, 2014	Microsoft	OWFa 1.0

A.23.2 Yosemite Platform Side Plane

Source: `http://files.opencompute.org/oc/public.php?service=files&t=7f5d8ba3c80744bc78a92d3cad46b7d2&download`

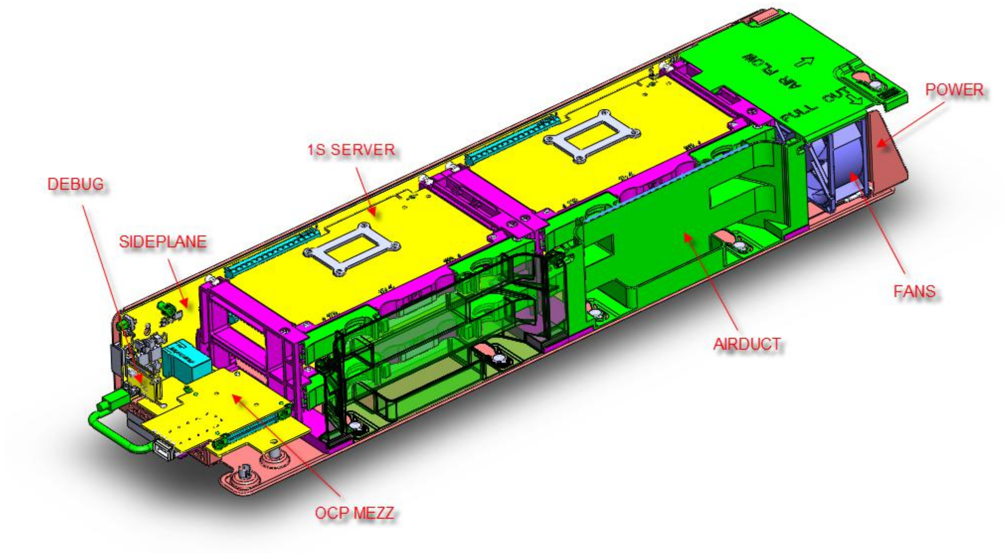


Figure 5-2: Yosemite Platform

5.2 Yosemite Platform Side Plane

The Yosemite Platform contains a side plane to hold all connectors and common infrastructure pieces, including the 1S server card connectors, OCP V2 Mezzanine card adapter card, a 12.5V inlet power connector (from the Cubby chassis), a BMC section, fan connectors, a hot-swap controller, and a front panel.

The side plane is installed vertically on the side of a Cubby chassis. OCP-compliant 1S server cards with height of 110mm or 160mm can be installed horizontally to the side plane.

The side plane shall be implemented as a low-maintenance, robust, platform to reduce the need of service. A BMC (ASPEED AST1250) is the main control unit on the side plane. The Yosemite Platform uses an adapter card at the front of the sled as a carrier board for OCP 2.0 Mezzanine cards. The OCP 2.0 Mezzanine connectors on the adapter cards have been carefully designed in a hybrid way to take a PCIe-based multi-host OCP V2 Mezzanine 40GbE/50GbE card, or a 40Gb capable KR retimer Mezzanine card or a 50GbE KR Aggregation Mezzanine that connects to the 1S server's built-in 10GBase-KR Network Interface Cards (NICs), as the Ethernet interface to the external world. Either way, the NIC will be used as a shared NIC, so that a BMC can be accessed via the OOB of the NIC, Network Controller Sideband Interface (NC-SI), or System Management Bus (SMBus).

By sampling the sensors on the Yosemite Platform periodically, the BMC continuously monitors the system's health status from function, power and thermal perspectives. The BMC shall implement sophisticated algorithms to control fans accordingly.

The BMC is connected to a hot-swap controller thru an Inter-Integrated Circuit (I^2C) bus so that it can get system-wide power consumption and maintain a healthy status. The BMC also controls 12.5V power to each 1S server. It is possible to let the BMC completely shut down 12.5V to a 1S server when the server needs a cold reboot.

A.23.3 OCP Summit IV: Breaking Up the Monolith

Open Rack and Open Vault for use in cold-storage environments and designs for a new, all-flash database server (codenamed Dragonstone) and the latest version of its web server (codenamed Winterfell).

But most exciting of all are a series of new developments that will enable us to take some big steps forward toward better utilization of these technologies. One of the challenges we face as an industry is that much of the hardware we build and consume is highly monolithic -- our processors are inextricably linked to our motherboards, which are in turn linked to specific networking technology, and so on. This leads to poorly configured systems that can't keep up with rapidly evolving software and waste lots of energy and material.

To fix this, we need to break up some of these monolithic designs -- to disaggregate some of the components of these technologies from each other so we can build systems that truly fit the workloads they run and whose components can be replaced or updated independently of each other. Several members of the Open Compute Project have come together today to take the first steps toward this kind of disaggregation:

- **Silicon photonics:** Intel is contributing designs for its forthcoming silicon photonics technology, which will enable 100 Gbps interconnects -- enough bandwidth to serve multiple processor generations. This technology also has such low latency that we can take components that previously needed to be bound to the same motherboard and begin to spread them out within a rack.

- **"Group Hug" board:** Facebook is contributing a new common slot architecture specification for motherboards. This specification -- which we've nicknamed "Group Hug" -- can be used to produce boards that are completely vendor-neutral and will last through multiple processor generations. The specification uses a simple PCIe x8 connector to link the SOCs to the board.

- **New SOCs:** AMD, Applied Micro, Calxeda, and Intel have all announced support for the Group Hug board, and Applied Micro and Intel have already built mechanical demos of their new designs.

Taken together, these announcements will enable data center operators to build systems that better fit the workloads they need to run and to upgrade through multiple generations of processors without having to replace the motherboards or the in-rack networking. This should in turn enable real gains in utilization and unlock even more efficiencies in data center construction and operations.

There's a lot of work left to do here, but it's never been more important for us to take these steps. As our lives become more connected, and as more devices and applications generate more data, we will face compute and storage challenges that existing technologies cannot handle efficiently.

But we have one big advantage, as we face these challenges: We are doing this work together, in the open, and everyone has a chance to contribute -- to help ensure that all the technologies we develop and consume are as scalable as possible, as efficient as possible, and as innovative as possible.

A.24 Solarflare

A.24.1 Hardware Platforms

Flareon Ultra and Flareon 10/40GbE Server I/O Adapters

Flareon 10/40GbE PCIe 3.1 server I/O adapters deliver industry-leading message rates with lowest latency and jitter over standard Ethernet along with low CPU utilization, enabling the industry's best performance and scalability for enterprise data center environments. Flareon server I/O adapters enable the deployment of more services to more users to fully leverage multi-core CPUs by providing hardware-assisted features to efficiently distribute I/O processing workloads, which eliminate bottlenecks, and optimize CPU utilization. Flareon server I/O adapters also provide application compatibility and protocol compliance, bypassing kernel networking overhead, while featuring binary compatibility with standard APIs and applications. Flareon server I/O adapters accelerate high frequency trading applications, high performance computing, Hadoop, cloud, grid, database, social networking and virtualized data centers.

Flareon Server I/O Adapters with AppFlex® technology provide a flexible platform for delivering multiple network services with a single server adapter. With patent pending technology Flareon adapters enable selectively adding and controlling a wide range of services such as innovative instrumentation and capture tools to detect network problems quickly, precision time, tamper-resistant security, and application acceleration when deploying servers.

More on Flareon Ultra and Flareon 10/40GbE Server I/O Adapters

Onload/Performant 10GbE Server Adapters

Solarflare Onload server adapters deliver the highest performance, lowest latency, and most scalable 10GbE solution in the industry. Onload server adapters are ideal for the most demanding, latency-sensitive applications in environments requiring the highest message rates. They also support OpenOnload® application acceleration middleware, which offers a unique combination of ultra-low latency and high message rate performance, with seamless application compatibility. Solarflare Onload PTP server adapters combine the lowest latency at highest message rates with the highest precision server clock accuracy – all in a single server slot operating over a single 10GbE network.

Solarflare Performant server adapters combine excellent performance with outstanding value to meet a broad range of 10GbE networking needs, such as data center, SMB, cloud computing, HPC, grid, and virtualized environments. Solarflare Performant adapter hardware and drivers are the same as those found in Solarflare Onload server adapters.

More on Onload/Performant 10GbE Server Adapters

A.25 Statistics Brain

A.25.1 Google Annual Search Statistics (2016)

STATISTIC BRAIN

Google Annual Search Statistics

Year	Annual Number of Google Searches	Average Searches Per Day
2015	2,834,650,000,000	7,766,000,000
2014	2,095,100,000,000	5,740,000,000
2013	2,161,530,000,000	5,922,000,000
2012	1,873,910,000,000	5,134,000,000
2011	1,722,071,000,000	4,717,000,000
2010	1,324,670,000,000	3,627,000,000
2009	953,700,000,000	2,610,000,000
2008	637,200,000,000	1,745,000,000
2007	438,000,000,000	1,200,000,000
2000	22,000,000,000	60,000,000
1998	3,600,000 **Googles official first year*	9,800

PREV DATA SET **NEXT DATA SET**

TRENDING STATISTICS

High School Graduation Rates by Country and State

Homeless Veterans Statistics

Crowdfunding Platform Statistics

Job Turnover Rates by Industry Statistics

Girl Scout Cookie Statistics

Vegetable Consumption and Production Statistics

Statistic Sources & References

Sources: Google Official History, Comscore, Statistic Brain Research Institute

Content Author: Statistic Brain

Date research was conducted: September 6, 2016

Google Annual Search Statistics

Digital Technology

ADVERTISEMENT

questions....
Click Here to Play

HISTORIC TIMELINES

Check out these visual timelines that take you from the beginning of a companies history to where they are today

FACEBOOK COMPANY TIMELINE

NIKE COMPANY TIMELINE

APPLE COMPANY TIMELINE

SEARS COMPANY TIMELINE

Related Statistic Brain Research

A.26 The Storage Networking Industry Association

A.26.1 Small Form Factor Committee - QSFP+ 28 Gb/s 4X Pluggable Transceiver Solution

Source: `ftp://ftp.seagate.com/sff/SFF-8665.PDF`

SFF Committee

SFF-8665

Specification for

QSFP+ 28 Gb/s 4X Pluggable Transceiver Solution (QSFP28)

Rev 1.9 June 29, 2015

Secretariat: SFF Committee

Abstract: This specification defines a 28 Gb/s QSFP+ pluggable transceiver solution popularly known as QSFP28. It gathers the appropriate/unique Base Electrical, Optical, Common Management, Module/Plug Formfactor, Host connector and cage specifications into a clearly delineated solution for users.

There are multiple generations of QSFP+

Forwarded to Standardization		
10 Gb/s	QSFP10	EIA-964/SFF-8436
Continuing Projects		
10 Gb/s	QSFP10	SFF-8635
14 Gb/s	QSFP14	SFF-8685
28 Gb/s	QSFP28	SFF-8665

Connectors compliant to SFF-8665 are also compliant to SFF-8685, and SFF-8635, but the reverse is not necessarily true.

POINTS OF CONTACT:

Jay Neer
Molex Incorporated
2222 Wellington Court
Lisle, IL 60532

Ph: 561-251-8016
Jay dot neer at molex dot com

I. Dal Allan
Chairman SFF Committee
14426 Black Walnut Court
Saratoga CA 95070

Ph: 408-867-6630
endlcom at acm dot org

A.27 Wired

A.27.1 How Facebook Changed the Basic Tech That Runs the Internet

CADE METZ BUSINESS 03.11.15 7:00 AM

HOW FACEBOOK CHANGED THE BASIC TECH THAT RUNS THE INTERNET

EVEN APPLE ADMITS the idea was a good one.

Back in 2011, in a cafeteria at the old Facebook headquarters in Palo Alto, California, Mark Zuckerberg revealed that his company was building all sorts of new computing hardware that could more efficiently run its vast online empire. But that wasn't the only surprise. Facebook, Zuckerberg said, would share its new hardware designs with the rest of the world. The Facebook social networking empire had grown so large—serving hundreds of millions of people across the globe—it only made sense that the company would want to streamline the vast server farms that underpin its operation. (Not to mention that Google had already done something similar). But for many, Facebook's decision to "open source" these hardware designs seemed overly idealistic, impractical, even pointless.

The idea was that others could use Facebook's designs to build their own online operations, create a broad market for the gear, and reduce Facebook's costs even further. But the skeptics saw it as little more a PR stunt: Facebook showing the word how "open" it was. After all, how many others were the size of a Google or a Facebook? How many others would want this gear enough to change the way they've always done things? And even if they did, how could it possibly help Facebook?

Four years on, this seemingly quixotic idea has played out much as Facebook said it would. A social-networking company has changed the way internet companies consume the hardware on which they run—and the way many of the world's hardware companies build and sell it.

On Tuesday, at the annual Silicon Valley gathering of the Open Compute Project, the non-profit that oversees Facebook's effort to share hardware across the tech industry, Project chairman and ex-Facebooker Frank Frankovsky announced that Apple has joined the effort, following in the footsteps of Microsoft, cloud computing giant Rackspace, and several of the country's biggest financial companies, including Goldman Sachs, Fidelity, and Bank of America.

Frank Frankovsky, chairman, Open Compute Foundation. Jon Snyder/WIRED

Like these others, Apple is a company that operates its own enormous online services—a company that needs the sort of hardware Facebook is sharing. Behind the scenes, Apple has long explored the use of Facebook's designs. And considering the intensely private nature of the company, its public involvement in the Open Compute project shows just how much it believes in this big idea.

Meanwhile, at the Open Compute Summit, two big-name American hardware companies revealed new products in direct response to Facebook's efforts—products that could help all sorts of other companies streamline their operations in much the same way Facebook has done. HP released a new line of computer servers based on Facebook's designs, and chip maker Intel unveiled a new streamlined server processor designed in tandem with Facebook. They believe in the idea too.

Bespoke to Benefit Everyone

Intel isn't releasing the design of its new "single socket" chip to the world at large. Others can't build their own, as they can with Facebook's open source server designs. But Intel will eventually sell the chip to anyone, not just to Facebook. "The important bit here is that Facebook and Intel are doing this in the open," says Facebook vice president of infrastructure engineering Jay Parikh, who helped oversee this 18-month effort.

Facebook Vice President of Engineering Jay Parikh. Ariel Zambelich/WIRED

It's not an easy situation to wrap your head around. But perhaps more than anything else, this partnership shows how effectively Facebook's foundation has brought together the companies building the world's online services to the benefit of everyone who uses the internet.

In the past, Intel would work with a Google or a Facebook to modify its chips to better suit such enormous operations. But these modifications were small, rare, and largely secret. Intel didn't want everyone asking for their own custom chips. Its business, after all, is based on mass production. Now, with a community of companies—from Apple to Microsoft and beyond—getting behind Facebook's basic ideas, Intel has reason to change its ways.

As Frankovsky puts it, a company like Intel is no longer building a custom part for just one company. It's building for "one anchor tenant who also has a voice within a broader community." Indeed, Kushagra Vaid—who oversees the design of the hardware that drives Microsoft's online services—says that his company is building a new computer server that requires a single-socket chip. The Intel-Facebook's creation, he says, could be that chip.

Open Source Apple

In short, the hardware market isn't what it once was. And odds are, with Apple involved, the landscape will soon change even more. Open Compute Project chairman Frankovsky tells WIRED that Apple has been involved in the Open Compute project "pretty darn close to day one," meaning it has likely used Facebook's designs in its own data centers. Microsoft's Vaid says that Apple has already contributed to some of the project's open source hardware projects. And Frankovsky indicates that Apple will eventually open source some of its own designs .

"They are building stuff that would be really cool for them to contribute to the community," he says. "Membership in Open Compute is the just the first phase, and can then lead to them collaborating out in the open an then contributing design work."

In the long run, this can help anyone else building modern online services. And that includes Facebook, whose selflessness might have turned out to be a little bit selfish after all.