Washington University in St. Louis
Washington University Open Scholarship

All Theses and Dissertations (ETDs)

Spring 4-30-2013

Delivering Consistent Network Performance in Multi-tenant Data Centers

Mart Albert Haitjema
Washington University in St. Louis

Follow this and additional works at: http://openscholarship.wustl.edu/etd

Part of the Computer Engineering Commons

Recommended Citation
Haitjema, Mart Albert, "Delivering Consistent Network Performance in Multi-tenant Data Centers" (2013). *All Theses and Dissertations (ETDs)*. 1077.
http://openscholarship.wustl.edu/etd/1077

This Dissertation is brought to you for free and open access by Washington University Open Scholarship. It has been accepted for inclusion in All Theses and Dissertations (ETDs) by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Jonathan S. Turner, Chair
Roger D. Chamberlain
Patrick J. Crowley
Ron K. Cytron
Jason E. Fritts
Robert E. Morley

Delivering Consistent Network Performance in Multi-tenant Data Centers

by

Mart Albert Haitjema

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2013
Saint Louis, Missouri

© Copyright 2013 by Mart Albert Haitjema.

All rights reserved.

Contents

List of Figures . . . . **vi**

List of Tables . . . . **viii**

Acknowledgments . . . . **ix**

Abstract . . . . **xi**

1 Introduction **1**
1.1 Objectives 2
1.2 Approach 3
1.3 Contributions 5
1.4 Methodology 6
1.5 Organization 7

2 Background **8**
2.1 Topology 8
2.1.1 Partitioning the network into virtual networks: 9
2.2 Routing 11
2.2.1 Routing in traditional data-center topologies 11
2.2.2 Oblivious flow-level routing 12
2.2.3 Adaptive flow-level routing 13
2.2.4 Flow-splitting 14
2.2.5 Packet-level routing 14
2.3 Flow control 15
2.3.1 Hardware-level mechanisms: 16
2.3.2 End-to-end protocols: 16
2.3.3 System-wide techniques: 17

3 Packet-level Routing **20**
3.1 The case for packet-level routing 21
3.1.1 Methodology 22
3.1.2 Oblivious flow-level routing 24
3.1.3 Understanding the performance of ECMP 28
3.1.4 Oblivious packet-level routing 32
3.2 Packet-level routing strategies in DCNs 32
3.2.1 Imbalances on small time-scales 33

3.2.2 Accounting for packet size . . . 35
3.2.3 Accounting for topology . . . 37
3.2.4 Comparison of approaches . . . 40
3.3 Performance in context . . . 42
3.3.1 Separating routing & flow-control . . . 42
3.3.2 Queueing theory model . . . 44
3.3.3 Evaluation . . . 45
3.3.4 **Partitioning the DCN into tenants** . . . 48
3.4 Resequencing packets at end hosts . . . 50
3.4.1 Dealing with out-of-order arrivals . . . 50
3.4.2 Design considerations . . . 51
3.4.3 Hybrid resequencer . . . 54
3.4.4 Evaluation . . . 57
3.5 Summary . . . 59

4 Isolating Tenants with Distributed Scheduling . . . 61
4.1 Introduction . . . 61
4.1.1 Objectives . . . 62
4.2 Scheduling Framework . . . 62
4.2.1 Scheduling layer . . . 62
4.2.2 Tenant virtual networks . . . 64
4.2.3 Constraints on rates . . . 65
4.2.4 Assigning rates on VOQs . . . 67
4.2.5 Assigning rates on bottleneck links . . . 68
4.3 Distributed Algorithm . . . 73
4.3.1 Link proxies . . . 73
4.3.2 Convergence to centralized rates: . . . 77
4.3.3 Accounting for control-overhead . . . 78
4.3.4 Related work . . . 79
4.4 Evaluation . . . 80
4.4.1 Isolation . . . 81
4.4.2 Distributed approach . . . 82
4.4.3 Flow control . . . 85
4.5 Discussion & future work . . . 87
4.5.1 Interactions with other protocols: . . . 87
4.5.2 Virtual machines . . . 88
4.5.3 Practical considerations . . . 88

5 Backlog scheduling . . . 90
5.1 Introduction . . . 90
5.2 Backlog scheduling problem . . . 91
5.2.1 Preliminary definitions: . . . 91
5.3 Initial-backlog problem: . . . 95

5.3.1 Problem definition: 95
5.3.2 Rate-assignment as a network flow: 96
5.3.3 Max-min is not optimal 97
5.3.4 Optimal algorithm: 98
5.3.5 Bounds on optimal 100
5.4 Deterministic backlog-schedule problem: 101
5.4.1 Problem definition: 102
5.4.2 Optimal bounds 102
5.4.3 Linear programming formulation 103
5.4.4 Proof of correctness 108
5.5 Online backlog-scheduling: 110
5.5.1 No online algorithm is optimal: 110
5.5.2 Any optimal initial-backlog algorithm is 2-competitive: 111
5.5.3 Any blocking algorithm is 2-competitive: 111
5.6 Evaluation 112
5.6.1 Experimental setup 113
5.6.2 Initial-backlog 115
5.6.3 Backlog scheduling stress test 118
5.7 Extending the results to oversubscribed trees 120
5.8 Summary 120

6 Conclusion 122
6.1 Summary 122
6.2 Future Directions 123

Appendix A FatTree DCN Simulator 126
A.1 Introduction 126
A.1.1 Motivation 126
A.1.2 OMNeT++ 127
A.1.3 INET framework 130
A.2 Simulator Overview 130
A.2.1 Modeling FatTree DCNs 132
A.2.2 BuildFatTree 134
A.3 Server components 134
A.3.1 Server 134
A.3.2 Control module 135
A.3.3 Application Layer 136
A.3.4 Transport Layer 137
A.3.5 Network Layer 137
A.3.6 Scheduling Layer 137
A.3.7 Resequencing Layer 138
A.3.8 Link Layer 139

Appendix B DCN Queueing Models **141**
B.1 Overview 141
B.2 Basic queueing theory concepts 141
B.2.1 M/M/1 Queue 142
B.2.2 M/M/1/K Queue 142
B.2.3 Modeling a network of queues 143
B.3 M/M/1/K FatTree 144
B.4 M/M/1/K LogicalTree 147

References **148**

List of Figures

2.1 A fully provisioned 4-port 3-level FatTree. 8
2.2 FatTree partitioned into tenant virtual networks. 10
2.3 High performance router interconnect 10
2.4 Distributed scheduling in high-performance routers 18

3.1 Under ideal routing, a FatTree is equivalent to a tree. 20
3.2 Performance of ECMP routing under the permutation traffic pattern. 25
3.3 Performance of ECMP as the network scales. 26
3.4 Flows colliding in an unfolded 3-level 4-port FatTree. 27
3.5 ECMP routing under the pod-permutation traffic pattern 31
3.6 Performance of packet-level VLB under the permutation pattern. . . 32
3.7 Comparison of VLB and Round Robin (RR) with maximum size packets. 34
3.8 Performance with packet sizes alternating between max and min size. 36
3.9 Load balancing with multiple flows in a simple two port network. . . 37
3.10 Separating flow control from load balancing. 43
3.11 Fraction of the network's capacity achievable as a function of the acceptable loss threshold. $l = 3$ levels, $k = 12$ ports, queue size $K = 50$ packets. 46
3.12 Per-port switch buffer size required to achieve given fraction of the network's capacity. $l = 3$ levels, $k = 12$ ports, loss threshold $= 10^{-3}$. . 47
3.13 Capacity vs loss threshold with servers partitioned into tenants of size m. $l = 3$ levels, $k = 12$ ports, loss threshold $= 10^{-3}$. 49
3.14 Fraction of traffic arriving out-of-order. 50
3.15 Average delays experienced by packets. 58
3.16 Fraction of end-to-end delay spent in resequencer (out-of-order packets). 58

4.1 Conceptual view of scheduling as layer a implemented in the networking stack of servers. 62
4.2 Two different tenant virtual network abstractions. 64
4.3 Effect of malicious traffic on the throughput of a "victim" tenant's flow. 81
4.4 Effect of different scheduling intervals on VOQs 84
4.5 Effect of assigning VOQs a minimum rate of 10 Mbps. 85
4.6 Capacity vs threshold with scheduling 86
4.7 Queue size vs capacity with scheduling 87

5.1 Feasible rate assignment as a feasible network flow. Flow values for an example solution are shown in red. 96
5.2 Initial backlog “two-phase” stress test 117
5.3 Limit on the rate of backlog entering the scheduling layer. 118
5.4 Performance with “stress test” backlog schedule. 119

A.1 The compound module representing a server. 135
A.2 The compound module representing the scheduling layer. 138

B.1 A simple M/M/1 queue. 142
B.2 Splitting and joining traffic at M/M/1 queues. 144

List of Tables

3.1 Imbalance in queueing at various stages in the network. 39
3.2 Throughput before maximum loss exceeds threshold - permutation traffic. 40
3.3 Throughput before maximum loss exceeds threshold - all-to-all traffic. 41
3.4 Throughput before average loss exceeds threshold - all-to-all traffic. . 41

Acknowledgments

I would like to thank my advisor, Jon Turner, for his guidance and patience in the research and final culmination of this thesis. He continually pushed me to make it the best it could be, and I feel truly privileged to have had the chance to work with him.

It has been a great pleasure working with many of my fellow students including Shakir James, Todd Sproull, Charlie Wiseman, Rohan Sen, Ben Wun, Haowei Yuan, Haraldur Thorvaldsson, Ritun Patney, and Joe Lancaster. I am especially indebted to Charlie Wiseman and Rohan Sen for their guidance in my early years at Washington University. I learned a great deal from them and their mentorship was invaluable. I would also like to thank Shakir James and Todd Sproull for their friendship and support. They made my life as a graduate student considerably more enjoyable.

I would also like to thank the staff of the CSE department: Kelli Eckman, Sharon Matlock, Myrna Harbison, and Madeline Hawkins. A special thanks to the staff members of the Applied Research Laboratory: John DeHart, Jytoi Parwatikar, Fred Kuhns, Ken Wong, and Dave Zar. In particular, I would like to thank John DeHart and Fred Kuhns for teaching me much of what I know about networking.

Finally, I would like to thank my parents, to whom this thesis is dedicated, as well as my brother, Charles, and my sister, Coraline. This thesis would not have been possible without their support and encouragement.

Mart Albert Haitjema

Washington University in Saint Louis
May 2013

Dedicated to my parents, Henk and Bieneke Haitjema.

ABSTRACT OF THE DISSERTATION

Delivering Consistent Network Performance in Multi-tenant Data Centers

by

Mart Albert Haitjema

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2013

Jonathan S. Turner, Chairperson

Data centers are growing rapidly in size and have recently begun acquiring a new role as cloud hosting platforms, allowing outside developers to deploy their own applications on large scales. As a result, today's data centers are multi-tenant environments that host an increasingly diverse set of applications, many of which have very demanding networking requirements. This has prompted research into new data center architectures that offer increased capacity by using topologies that introduce multiple paths between servers. To achieve consistent network performance in these networks, traffic must be effectively load balanced among the available paths. In addition, some form of system-wide traffic regulation is necessary to provide performance guarantees to tenants.

To address these issues, this thesis introduces several software-based mechanisms that were inspired by techniques used to regulate traffic in the interconnects of scalable Internet routers. In particular, we borrow two key concepts that serve as the basis for our approach. First, we investigate packet-level routing techniques that are similar to

those used to balance load effectively in routers. This work is novel in the data center context because most existing approaches route traffic at the level of flows to prevent their packets from arriving out-of-order. We show that routing at the packet-level allows for far more efficient use of the network's resources and we provide a novel resequencing scheme to deal with out-of-order arrivals.

Secondly, we introduce distributed scheduling as a means to engineer traffic in data centers. In routers, distributed scheduling controls the rates between ports on different line cards enabling traffic to move efficiently through the interconnect. We apply the same basic idea to schedule rates between servers in the data center. We show that scheduling can prevent congestion from occurring and can be used as a flexible mechanism to support network performance guarantees for tenants. In contrast to previous work, which relied on centralized controllers to schedule traffic, our approach is fully distributed and we provide a novel distributed algorithm to control rates. In addition, we introduce an optimization problem called backlog scheduling to study scheduling strategies that facilitate more efficient application execution.

Chapter 1

Introduction

Data center networks (DCNs) form an important part of the modern Internet, hosting many of the applications and services that we use online. These networks were traditionally constructed to support the services of large organizations. With the growth of cloud computing, however, they increasingly operate as pools of shared computing resources that serve the needs of multiple “tenants”. For example, cloud hosting platforms like EC2 [11], enable anyone to access a large volume of compute resources and deploy their own applications on a massive scale. As a result, many DCNs must support a diverse mix of tenants that often have a wide range of requirements for their applications.

One limitation facing tenants today is that many cloud environments provide little in the way of guarantees on the performance of the intra-data center network [42, 39]. While they are typically given abstractions for the computing resources of the servers they use, the network that connects the servers together is usually treated as a shared resource. The result is that tenants often face network performance that can be highly variable. This best effort network model is not ideally suited to meet the needs of all tenants. For example, poor network performance has been cited as a barrier to entry for high-performance scientific applications [30] and the lack of performance isolation raises concerns over denial of service attacks [53, 60, 13]. A wide range of existing applications also depend upon consistent network performance to perform well, making it difficult for them to operate in this environment [59, 12].

1.1 Objectives

For at least some tenants, it will be useful to have the ability to engineer their applications for consistent network performance. If data center networks can be made to support this, they will likely be more attractive to potential users. However, to retain the economic advantage that they provide, it will be important to make efficient use of the network's resources. Realizing this goal translates into the following objectives:

- **Provide tenants with virtual network abstractions:** To engineer their applications for consistent performance, tenants should be provided with guarantees on the bandwidth available between their servers. Ideally, these guarantees could come in the form of different virtual network abstractions. One example of such an abstraction is the virtual switch [26, 14, 13], which provides a tenant with the illusion of having all of its servers connected to the same non-interfering switch.

- **Achieve performance isolation among tenants:** Tenants should not be allowed to interfere with the bandwidth guarantees provided to other tenants. They should also experience a low degree of packet loss and delay provided they remain within the limits prescribed by their network abstractions. This means the network must provide a form of traffic isolation among tenants that is robust to the arbitrary traffic patterns they may produce.

- **Support flexible assignments of servers to tenants:** Maintaining a high level of server utilization is a key factor in the economic advantage that data centers provide. This makes *agility*, the ability to assign a tenant to any available server, an important property that must be preserved [25].

- **Operate in networks constructed from commodity switches:** Another aspect important to the success of data centers is their reliance on commodity off-the-shelf components. Because they typically use commodity Ethernet switches, they provide a cost advantage over high-performance computing clusters that employ more specialized networking technologies. These switches are cheap and easy to configure yet they provide speeds that are competitive with

the more expensive technologies. However, they currently lack the mechanisms necessary to provide the type of performance guarantees offered by virtual network abstractions. Developing custom hardware to address this issue may undercut the cost advantage provided by commodity switches and would only benefit the networks that adopt them. Therefore, an ideal solution would allow for operation in networks constructed from existing off-the-shelf switches.

In the past, these goals have been difficult to reconcile. The use of Ethernet has constrained the design of data centers to a tree-structured topology that cannot provide uniformly high capacity among servers because branches at higher levels of the tree are shared among more servers [27, 26, 9]. In these networks, realizing the guarantees offered by abstractions like the virtual switch would require assigning tenants to servers that are close to one another in the tree, where bandwidth is plentiful. However, this practice would limit network agility and lead to fragmentation of the data center's resources [26, 27].

Recently, researchers have proposed novel methods to construct new multi-path data center networks using existing switches [9, 26, 29, 28, 44, 40]. These networks can offer greatly increased capacity by providing multiple paths between servers yet they remain economical by leveraging inexpensive off-the-shelf components. This makes it feasible to provide tenants with separate virtual networks without sacrificing agility. For instance, a FatTree [38] with full bisection bandwidth provides enough capacity to allow all servers to send and receive at the full rate of their interfaces. In principle, this makes it possible to provide every tenant with the virtual switch abstraction regardless of how they are assigned to servers. However, the bandwidth that exists between servers can only be fully utilized if traffic is balanced evenly among the available paths. Moreover, additional mechanisms are still needed to provide traffic isolation among tenants.

1.2 Approach

The work in this thesis was inspired by the observation that the design of large high-performance Internet routers face many of the same issues. Such routers are

constructed using interconnection networks that are often similar to some of the multi-path data center topologies. The Clos network [20], for example, is a generalization of the topologies used by VL2 [26] and FatTree [9]. These routers must maintain consistent performance even under extreme traffic conditions; this requires that they load balance and regulate traffic effectively to make efficient use of their interconnect. Since network performance in the data center should also be robust to the arbitrary traffic produced by its tenants, we explore using similar techniques to regulate traffic in the data center.

Specifically, we draw upon two key techniques that we have adapted to the data center to serve as the basis for our approach. First, we use packet-level routing to make the most efficient use of the multi-path data center networks. Routers typically route the packets that arrive at an input separately through the interconnect and use resequencing mechanisms to ensure they are sent in-order at outputs. By leveraging simple routing techniques, such as Valiant Load Balancing (VLB) [56], they can achieve near optimal load balancing independent of the pattern of traffic arriving at the inputs [22]. We investigate applying similar techniques in the data center context. This work is novel because most of the current approaches route traffic at the level of flows to prevent their packets from arriving out-of-order. We have chosen to deal with the problem of out-of-order arrivals by resequencing them in software.

The second component to our approach is based on distributed scheduling in routers. Routers use distributed scheduling to control the rates between ports in order to move traffic efficiently through the interconnect [22]. We apply this concept to the data center network to control the rates that a tenant's servers can send to one-another. To do this, we introduce a scheduling layer in the networking stack of servers in a manner that is transparent to tenants. By coordinating the rates between a tenant's servers, scheduling provides a mechanism for traffic isolation that effectively uses a tenant's servers to police its own traffic. We show that it can be used to support abstractions such as the virtual switch as well as private tree structured networks like the virtual oversubscribed cluster [14].

1.3 Contributions

This thesis makes several key contributions.

Packet-level routing & resequencing
First, this work is among the first to seriously investigate the use of packet-level routing in data center networks. We perform an in-depth study on the limits of packet-level routing on FatTree data center networks and introduce several new routing techniques specifically adapted to the data center environment. In contrast to previous work, the emphasis of our investigation is to determine the bounds under which performance isolation can be maintained between tenants. To our knowledge, we are also the first to address the issue of out-of-order arrivals by resequencing packets in software at servers and we introduce a novel resequencing scheme that combines the benefits of using time stamps and sequence numbers to reorder packets.

Framework for distributed scheduling
Second, we present a framework for distributed scheduling to control traffic in the data center network. The use of scheduling with software rate limiters to enforce tenant virtual network abstractions has been studied previously [14, 36]. However, previous work has relied upon centralized controllers that schedule rates for the entire network. This limits the frequency at which rates can change. Our contribution is to allow rates to be set in a fully distributed manner, enabling rates to respond to changes in traffic on the order of milliseconds. We provide a novel distributed asynchronous algorithm that can assign rates between servers in max-min fair fashion as well as in "backlog-proportional" fashion, which assigns rates in proportion to the volume of traffic that servers have to send to one another. We evaluate the tradeoffs of our approach and demonstrate that, in concert with packet-level routing, distributed scheduling can provide performance guarantees to tenants regardless of how they are assigned to servers or what traffic patterns they produce.

Strategies for efficiently scheduling tenant traffic
Thirdly, we investigate which strategies may schedule the traffic between a tenant's servers most efficiently. In particular, we introduce the concept of backlog-scheduling, which defines the problem in terms of minimizing the time required to clear the backlog of data that servers have to send to one another. While backlog-scheduling

does not characterize the optimal way to assign rates in general, it may apply to an important class of applications, such as those based on MapReduce [23]. We provide a set of formal problem definitions and take an algorithmic approach to study the problem space. We prove that the backlog-proportional assignment of rates is optimal for a restricted form of the problem and provide a linear program that optimally solves the more general offline problem. To evaluate the performance of max-min and the backlog-proportional algorithm, we use competitive analysis to place bounds on their online performance. We show several examples of traffic patterns that can cause the performance to approach these bounds and we simulate several of these cases to demonstrate our results.

Packet-level simulator for data center networks
Finally, we develop a packet-level simulator based on the OMNeT++ discrete-event simulation framework [7] to study large data center networks. OMNeT++ is free for academic use and we have made our code publicly available for other researchers to use.

1.4 Methodology

This thesis is concerned with providing consistently high network performance to tenants in data center networks. Here we are concerned exclusively with the network fabric connecting servers together, not their connection to the outside world. Our target platforms are multi-path data center networks constructed from commodity Ethernet switches and we focus specifically on the FatTree for our investigation of packet-level routing. For our work on scheduling, we also assume that tenants have been given network abstractions that include bandwidth guarantees. We are not concerned with the allocation of virtual networks to tenants, however, and we do not propose any new abstractions of our own. Rather, we build upon the work of others to make these networks more practical by providing the necessary mechanisms to support these abstractions flexibly.

The mechanisms that we propose focus on performance at the network-level. To preserve agility, we assume that tenants may be assigned to servers arbitrarily. To be

robust to malicious traffic, we also assume that they may produce arbitrary traffic patterns. In line with these goals, we use similar performance measures and procedures to those used to evaluate interconnection networks. As a result, our evaluations typically focus on the worst case and use adversarial traffic patterns to probe the limits of performance.

Given the size of modern data centers, it is impractical to evaluate an implementation of our approach at scale. Instead, we rely on a combination of simulation and analysis to evaluate our work. In our investigation of packet-level routing, we use probabilistic analysis and queueing theory to validate our simulations and to determine the scaling behavior of our results. The queueing theory models that we develop also help us to examine the dependency between our routing and scheduling techniques and provide insight into the extent to which the issues they address can be separated. While we rely primarily on simulation to evaluate distributed scheduling, much of our work on scheduling is algorithmic in nature and we compare our analytical results with simulation.

1.5 Organization

The organization of this thesis is as follows. Chapter 2 provides additional background and related work and expands on the details of our approach. In chapter 3, we study packet-level routing and resequencing in FatTree data center networks. Chapter 4 introduces the scheduling framework and focuses on using scheduling to provide isolation between tenants. In chapter 5 we introduce backlog scheduling as a case study for examining scheduling strategies that improve the performance of tenant applications. Finally, we summarize our findings in chapter 6 and point to a number of interesting avenues for future work.

Chapter 2

Background

In this chapter we provide some background on the problem of providing network performance to tenants in data center networks and review some of the relevant related work.

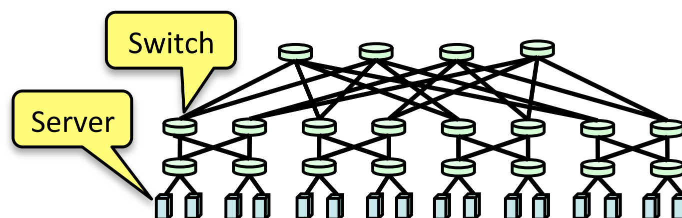


Figure 2.1: A fully provisioned 4-port 3-level FatTree.

2.1 Topology

FatTree

While a number of recent multi-path architectures have emerged, we will focus on the FatTree [38] in this work. FatTrees are a popular topology in high-performance computing interconnects and are also a natural choice for data center networks [61]. Figure 2.1, shows a simple 3-level FatTree constructed from 4-port switches. In this example all links have the same capacity which means the bandwidth available between servers is determined by the number of paths. Note that the network proposed by VL2 [26] is essentially the same as the FatTree shown here except that the switch-to-switch links have a higher capacity than the server-to-switch links.

A FatTree with l levels and k port switches can be constructed recursively by viewing each level as a *subtree*. We can think of a subtree at level 0 as consisting of just a single server. At level 1, a subtree contains a switch with $\frac{k}{2}$ "downward facing" ports connecting to level 0 subtrees (servers) and $\frac{k}{2}$ "upward facing" ports connecting to switches at level 2. A level 2 subtree contains all of the level 1 subtrees that connect to the same set of level 2 switches. Each of the downward facing ports of a level 2 switch connects to a separate subtree at level 1 so that each level 2 switch connects to each of the $\frac{k}{2}$ subtrees. Each of the upward facing ports, however, connects to a unique switch at level 3. This process repeats for each level until we reach the root of the tree at level l. At this stage, all ports are downward facing which means each switch connects to k subtrees causing the entire network to be encompassed. The number of servers in a subtree at some level $i < l$ is $(\frac{k}{2})^i$ and since there are k subtrees at level l, the FatTree has $k(\frac{k}{2})^{l-1}$ servers (e.g. 16 for the 4-port FatTree in Figure 2.1).

Bisection bandwidth:
A common metric for the network capacity of an interconnection network is the bisection bandwidth. A bisection is a cut that partitions the network into two evenly sized sets [1] and the bandwidth of a bisection is equal to the sum of the capacity on the links between the two sets. The network's bisection bandwidth is the minimum bandwidth between any two bisections. Notice that every level of the FatTree in Figure 2.1 has the same number of links and thus the same bandwidth. As a result, we could partition this network into any two sets and the bisection bandwidth will always match the total bandwidth of the servers' interfaces. This network is said to have *full bisection bandwidth*.

2.1.1 Partitioning the network into virtual networks:

With full bisection bandwidth, we could partition the servers arbitrarily into groups and provide each tenant with the virtual switch abstraction as illustrated in figure 2.2. This would provide tenants with full bandwidth between their servers without placing constraints on agility. Achieving the isolation provided by this abstraction would imply that servers connected to one virtual switch would be free to use the

[1] In the case of an odd number of endpoints, one set contains an extra endpoint.

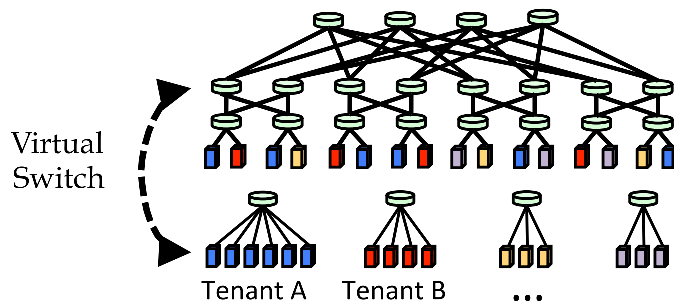


Figure 2.2: FatTree partitioned into tenant virtual networks.

network arbitrarily without being able to affect servers connected to other virtual switches.

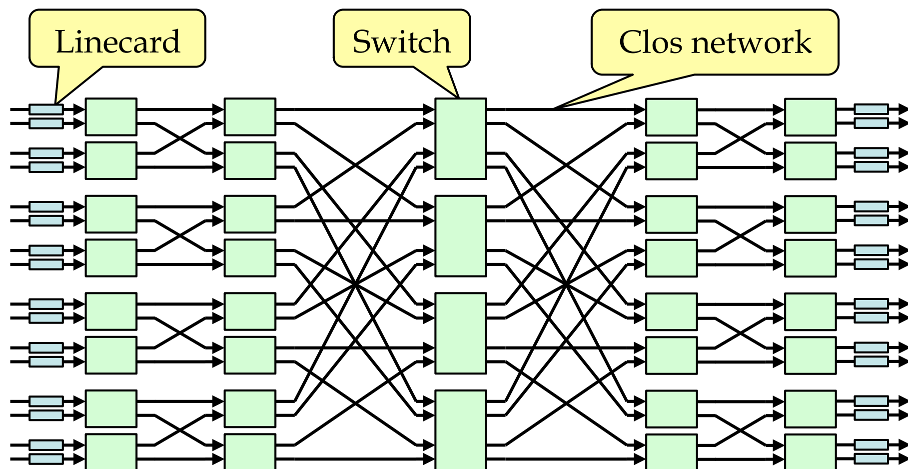


Figure 2.3: High performance router interconnect

Non-interference:
Realizing the performance isolation provided by the virtual switch abstraction is analogous to ensuring non-interference in an interconnection network, such as those used in the construction of high-performance Internet routers. Figure 2.3 shows an abstract representation of such a router interconnect. Ports are distributed across physically separate linecards that are interconnected with a network of smaller switch elements. Note that the Clos network [20] shown in this figure is an unfolded version of the FatTree (i.e., the links are unidirectional). The definition of non-interference provided by Dally and Towles [22], is a good one in this context. They state that non-interference means that "an excess in traffic destined for line-card A, perhaps due to a momentary overload, should not interfere with or 'steal' bandwidth from traffic

destined for a different line card B, even if messages destined to A and messages destined to B share resources throughout the fabric".

From this perspective, topology is one of three aspects needed to meet this requirement. The other two are:

- **Routing:** While the topology determines the physical capacity of the network, the actual capacity that can be achieved, or "effective bisection bandwidth" [31] depends on how well traffic is balanced among the paths. The coarse granularity of flow-level routing means that load cannot be balanced precisely among paths, making the performance of flow-level techniques dependent on the traffic produced by tenants. In fact, previous studies have shown that under many traffic patterns, only about half of the network's capacity is achievable [8, 32].
- **Flow control:** Even if full bisection bandwidth can be achieved, a malicious tenant could still create congestion by concentrating traffic onto one of its servers. By combining traffic from multiple servers it can exceed the capacity available to the destination server creating congestion in the network that can affect traffic to neighboring servers. In interconnection networks, this problem is known as "tree saturation" [22] and to provide consistent performance guarantees, the network must be robust to such pathological traffic patterns. In interconnection networks, the term flow control describes the mechanisms used to prevent congestion and provide guarantees to different classes of traffic.

In the remainder of this section we consider the key issues and approaches to routing and flow control and place related work within this framework.

2.2 Routing

2.2.1 Routing in traditional data-center topologies

Because data centers have traditionally been constructed using a tree-like topology, they generally have limited path diversity making load balancing less of a concern.

When multiple paths do exist, they are typically between IP routers in the backbone of the network. To balance load across these paths, routers equipped with Equal-Cost Multi-Path (ECMP) can be used. ECMP assigns flows to paths randomly by computing a hash based on the flow's header. While it is an IP routing protocol, many Ethernet switches support a variety of IP features, including EMCP. Since Ethernet provides no support for multiple paths, ECMP provides a convenient option for FatTrees built from commodity switches. In fact, VL2 [26], one of the early proposals for constructing FatTree DCNs, relies upon this approach.

2.2.2 Oblivious flow-level routing

ECMP is an example of oblivious flow-level routing. Oblivious routing refers to any approach that routes traffic randomly, rather than based upon the state of congestion in the network. In the FatTree network, the number of paths between servers is determined by the number of switches at the top level, which we refer to as "intermediate switches". This means load can be balanced between a source and destination by routing its traffic randomly through an intermediate switch. This approach is attractive because it is simple; we expect all paths to receive roughly equal traffic given that all paths have an equal probability of being chosen. As we will see in chapter 3, this approach works well when traffic is split at fine granularity (i.e. at the level of individual packets). When routing at the level of flows, however, it does not always perform well, particularly when most of the traffic belongs to a small number of large flows. Unfortunately this is often the type of traffic that we see in data center networks [15]. Intuitively, the problem is that with fewer large flows, the random assignment of flows to paths can cause some links to receive multiple flows while others are left idle. The resulting "collisions" of flows on links reduces their throughput. One study showed that this can be by as much as 60% of the network's capacity when the traffic consists of only one flow per server [8] and our simulations have confirmed this result.

2.2.3 Adaptive flow-level routing

To improve the performance of flow-level routing, adaptive techniques can be used. Adaptive techniques make routing decisions dynamically based on the state of congestion in the network. The literature on adaptive routing can be divided into two categories, centralized and distributed.

Centralized
In the first category, a central scheduler attempts to optimize the routing of flows in the network by periodically recomputing routes for flows. Some examples of this approach in the data center context are Hedera [8], DevoFlow [21], and MicroTE [16]. The basic idea is to leverage the fact that most of the traffic in data centers belongs to a smaller number of large flows. So by monitoring and rerouting only these flows, a central scheduler could avoid most of the imbalance between paths yet still be scalable. Several issues arise with this approach, however. First, to determine whether the mapping of flows creates an imbalance, the demands of flows must be estimated. That is, the amount of bandwidth that a given flow would use if it were not constrained by congestion along its path. This information must then be communicated to a central scheduler, which can then compute a global schedule of flows to paths and return updated routes to servers. This means that scheduling can only benefit large long-lived flows. While studies suggest that most packets belong to large flows [15] [26], large flows are not necessarily long lived with today's Ethernet. For example, at 10 gbps, even a 100 MB flow can complete within a second. This raises doubts whether a central scheduler can make decisions rapidly enough at large scales. In fact, it was shown through simulation that on traffic patterns based on real traces, Hedera provides little benefit over ECMP when scheduling occurs at realistic intervals (i.e. $>$ 100 ms)[49].

Distributed
The scheduling of flows to paths is an optimization problem that is not easily distributed. Instead, distributed approaches focus on routing new flows heuristically, based on the current levels of congestion detected in the network. That is, rather than rerouting existing flows, these techniques simply try to route new flows to the least congested paths. While routing decisions can be made in a distributed fashion, some of the approaches in this category require the support of switches to infer

congestion and assign routes [41] and [54]. Since existing switches do not have such features, these approaches are only useful if vendors were to adopt them in the near future. A purely software based approach was proposed by Mahapatra and Yuan [41]. When a sever has a new flow to send, it first probes the available paths by sending a packet along each path to determine the relative levels of congestion. They showed through simulation that while their techniques can outperform oblivious flow-level routing on average, it still performs significantly worse than packet-level routing.

2.2.4 Flow-splitting

There exists a small category of approaches that can be seen as lying in-between packet-level and flow-level routing. Most notably, Multipath TCP [49] is a recent development that allows flows to be broken into subflows which may then take separate paths. Thus increasing the number of sub-flows allows a flow's traffic to be spread across more paths. In the limit it would be possible to ensure a flow is split so that it has a sub-flow assigned to each path at which point the behavior approaches packet-level VLB, which we discuss below. This approach would require adopting this particular version of TCP, however, and creating subflows does add new overhead to TCP. This means tenants could not use other protocols such as UDP or deploy virtual servers that provide their own version of TCP.

2.2.5 Packet-level routing

While oblivious routing can perform poorly at the level of flows, it is a logical choice for packet-level routing in a FatTree topology. Randomly routing packets to intermediate nodes is also known as Valiant Load Balancing (VLB) and it has the benefit that its performance is independent of the traffic pattern [22]. Since all paths have equal cost in a FatTree, it exactly balances load over long time scales. While this approach is commonly used in interconnection networks in other contexts, it has received little attention in the data center context. This is primarily because of the assumption that packets within a flow should not be delivered out-of-order. While IP does not guarantee in-order arrival of packets, it is generally treated as an implicit requirement. One of the key concerns is that out-of-order arrivals can significantly

affect the performance of TCP because TCP treats out-of-order arrivals as a sign of congestion [24]. In section 3.4 we will examine the degree to which packets can arrive out of order and introduce a method to resequence out-of-order packets at servers to address this issue.

There is surprisingly little in the literature on the use of packet-level routing in data center networks. One recent paper, however, did challenge the assumption that packet-order must be maintained in a FatTree DCN [37]. The basic argument is that because all paths in a FatTree have an equal number of hops (equal cost), packets can only arrive out of order when the level of congestion on different paths differs significantly. By using packet-level VLB, however, paths should experience roughly an equal degree of congestion which limits the extent to which out-of-order delivery can occur. They performed some simulations to demonstrate that the increase in throughput from VLB could outweigh the drop in TCP performance due to out-of-order arrivals. Another study [24] investigated packet-level routing in data center networks and represents the work most closely related to our own. They investigated three-different switch-based mechanisms to route packets. These included randomly picking a port, round-robin, and a counter-based approach which keeps track of the total number of bytes transferred on each link. While these require hardware support, they turn out to be similar to several of the server-based strategies that we explore in the next chapter.

2.3 Flow control

In general, the job of flow control is to control congestion and provide guarantees to different classes of traffic. In this context, each tenant represents a separate traffic class and their guarantees are defined by their virtual network abstractions. To support these abstractions and provide traffic isolation among tenants requires a flow control mechanism to perform the following functions:

- Prevent congestion from occurring in the network.
- Enforce the bandwidth limits imposed by virtual network abstractions.

2.3.1 Hardware-level mechanisms:

Most interconnection networks employ hardware-level flow-control mechanisms designed to work in conjunction with the specific topology and routing mechanism. One of the objectives of our approach is to work in networks constructed from existing off-the-shelf Ethernet switches. However, Ethernet was not designed to support topologies such as the FatTree and provides few flow-control mechanisms that can support the types of guarantees offered to tenants. VLANs can be used to create separate virtual networks, effectively isolating tenants from one another. However, they do not provide the ability to enforce the bandwidth limits defined by virtual network abstractions. QCN [45] is an emerging standard that can provide hardware-level bandwidth guarantees. However, it cannot enforce these guarantees across the multiple paths that exist in topologies, such as the FatTree.

2.3.2 End-to-end protocols:

Currently, many data centers rely on tenants to use TCP to control congestion. For tenants to engineer their applications for consistent performance, they should not be constrained to using a particular protocol. Another issue with depending on end-to-end mechanisms like TCP, is that they can only react to congestion. This makes it difficult to provide isolation among tenants because it means that some level of congestion must first occur. For example, a TCP flow increases its rate until it detects congestion, typically through packet loss, at which point its sending rate will be reduced. Since it has no way to determine the appropriate rate at which to send, it immediately resumes increasing its sending rate to probe for available bandwidth. This cycle ensures that several long-lived TCP flows will keep the switch queue on a bottleneck link full. Alizadeh et al. call this “queue buildup” and identify it as a performance impairment for a number of reasons [10]. First, every packet experiences the maximum amount of queueing delay. Secondly, queue buildup means there is little room left to absorb a burst of new traffic which can cause new flows to experience packet loss and incur timeouts. Since TCP timeouts were designed for the round-trip times common on the Internet, they can be particularly expensive in the data center as evidenced by the TCP incasting phenomenon [19, 48, 43, 58]. Finally, on

shared memory switches, queue buildup on one port reduces the memory available for buffering on other ports which exacerbates these issues. Therefore, queueing can negatively affect performance for all tenants whose traffic passes through the switch.

Some Ethernet switches do include Explicit Congestion Notification (ECN), a feature which allows switches to provide feedback to servers about congestion before packets are lost due to full queues. By marking packets at a very low threshold, ECN can be used to keep queueing in switches to a minimum. In order to fully utilize a link, however, TCP's congestion control algorithm normally depends on some amount of queuing (typically equal to the bandwidth delay product). Data Center TCP [10] proposed some changes to the way that TCP reacts to ECN in order to fully utilize the link while allowing switches to keep queue levels to a minimum. Seawall [52] is another end-to-end mechanism that was designed to provide isolation between applications in the data center. It allows any protocol to be used but forces all traffic between a tenant's servers through TCP tunnels.

Even with ECN, end-to-end mechanisms face another limitation in that they can only provide a limited form of performance isolation. While they can enforce bandwidth limitations on individual flows, they cannot enforce limits on the aggregate bandwidth consumed by a tenant's servers on a link. This means that they cannot support more sophisticated network abstractions such as the Virtual Oversubscribed Cluster [14], which we will look at in section 4.2.2.

2.3.3 System-wide techniques:

In order to support these abstractions, aggregate rates must be enforced which requires a system-wide approach.

Scheduling in Data Centers

In the data center this means explicitly scheduling the rates that servers can send to one another and enforcing these rates at servers in software, typically by modifying the kernel or hypervisor. This approach is reasonable because data centers are under

single administrative control which means that the networking stack of the kernel or hypervisor is part of the trusted code base. By explicitly scheduling the rates between all of a tenant's servers, scheduling can ensure that a tenant does not use more than its share of a bottleneck link. In principle, this means it can be used to enforce the bandwidth constraints imposed by any arbitrary network topology. In addition, by coordinating rates, explicit scheduling makes it possible to *avoid* congestion.

Current approaches schedule rates centrally

There are a number of approaches that schedule rates and enforce them in software. XCo [50] proposes a general framework to schedule traffic in data center networks. Their current approach uses a central scheduler to collect traffic information from all of the servers in the network in order to create a global traffic matrix. Using this information, the scheduler periodically provides servers with a time schedule indicating the times during which each server can transmit. Ballani et al. describe "Oktopus" [14], which also enforces server-to-server rates at end hosts using a central controller. NetShare [36] also uses kernel-based rate limiting and a central controller to schedule rates between servers. While this approach can avoid congestion, it is unclear how well it can scale to manage large networks.

Distributed Scheduling in Interconnection Networks

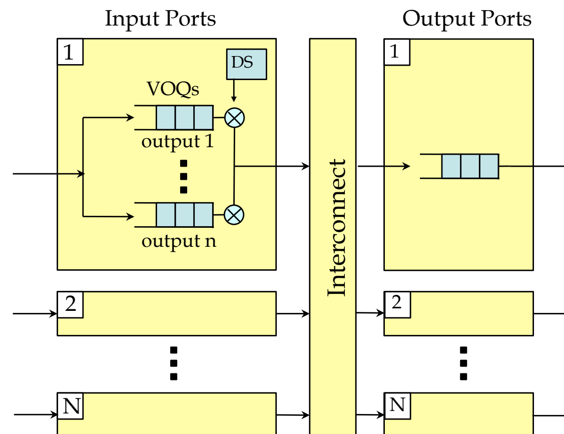


Figure 2.4: Distributed scheduling in high-performance routers

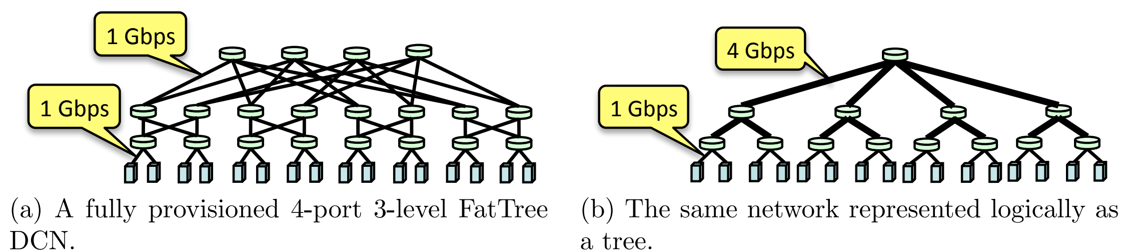
Our work draws upon the concept of distributed scheduling used in interconnection networks. Distributed scheduling is used as a flow control mechanism in large interconnection networks [22] such as those used in high performance routers [51] [47] [46]. It works by scheduling the rates that input ports can send to outputs through the interconnect. Figure 2.4 shows a simplified router diagram that helps illustrates the idea. Arriving packets are buffered at each input port in Virtual Output Queues (VOQs) corresponding to the output ports that they are destined for. By controlling the rate of traffic leaving each VOQ, the router can manage the rate that each input sends to each output. These rates will be assigned periodically by a controller, shown as DS in the figure, that resides at each port (or linecard). Controllers periodically exchange information about the state of their queues and use this information to independently assign rates to their VOQs. Congestion can be avoided by assigning rates at inputs so that the total rate sent from inputs does not overload any of the outputs. This approach can also be used to provide QoS guarantees by managing separate VOQs for each traffic class, effectively providing virtual networks within the interconnect [22].

While the scheduling that we propose uses the same basic idea, there are some important differences. In the router context, the network typically provides a speedup relative to the speed of the ports. To accommodate this, each output has an output queue which is where most of the queueing occurs. These queues are typically quite large (e.g. 100 ms worth of traffic) and the VOQs at inputs usually only begin to fill when there is an overload at an output. Maximizing throughput is often an objective of the scheduling algorithm used in these routers. For example, Distributed BLOOFA [47] attempts to remain work conserving by focusing traffic on the least occupied output queues. However, there is no sensible analogy for an output-side queue at servers in the data center network. Traffic arriving at a server may be destined for different transport-level ports which may process incoming messages at different rates. It is generally also not very practical to construct a data center network that provides a speedup relative to the speed of the server's interface. The scheduling that occurs in routers also occurs on a very small time scale compared to the end-to-end delays experienced by the traffic. In the data center network, control messages experience the same end-to-end delay as the traffic being scheduled which means that scheduling occurs on an inherently longer time scale.

Chapter 3

Packet-level Routing

In this chapter, we investigate the use of packet-level routing and resequencing in the context of multi-tenant data center networks. Here we focus specifically on FatTree networks constructed from commodity switches such as the simple example shown in figure Figure 3.1a. The purpose of the FatTree is to approximate the topology shown in Figure 3.1b since trees with such "fat" links are impractical to construct for large networks. The motivation for using packet-level routing is to be able to view the network by its logically equivalent tree. In this chapter, we argue that this is an important property to achieve in the context of providing performance isolation while maintaining agility.



(a) A fully provisioned 4-port 3-level FatTree DCN.

(b) The same network represented logically as a tree.

Figure 3.1: Under ideal routing, a FatTree is equivalent to a tree.

This chapter is divided into four sections.

- First, we make the case for packet-level routing by presenting a performance study comparing flow-level and packet-level routing in the data center.
- Second, we adapt packet-level routing to the data center context by exploring several ways to maximize performance.

- Third, we perform a thorough evaluation where we examine the tradeoff between isolation and network utilization and the degree to which performance is dependent on flow control and the amount of buffering available at the switches.

- Finally, we show how to cope with out-of-order arrivals by describing an efficient method for resequencing packets in software at servers.

3.1 The case for packet-level routing

There are several key reasons why we cannot rely on flow-level load balancing to provide strong isolation among tenants. When flows are large, any non-optimal arrangement of flows to paths will suffer from the same issue of "flows colliding" that we described in section 2.2.2. Such flows suffer from reduced throughput which leads to significant unfairness with those flows not experiencing congestion. To see why this is true, we can consider that FatTrees are a type of Clos Network that are known to be rearrangeably non-blocking in circuit switched contexts such as a telephone network. In a rearrangeably non-blocking switch, it is always possible to route a call between an input and a free output port but only if we can reroute existing calls. This mirrors the situation in a data center where each server sends at full rate to one other server. In order to achieve 100% throughput as traffic changes, this result means that flows would have to be routed adaptively.

Adaptive routing presents a fundamental problem, however. Any adaptive technique that we could use in a data center requires time to determine the state of traffic in the network and compute new routes. Of course, these routes are only useful as long as they match the current traffic pattern. When traffic changes more quickly than the time scale on which routes are computed, then adaptive routing is no better than oblivious routing. This was demonstrated in several previous studies [49] [54]. Moreover, adaptive techniques that do reroute flows quickly create another problem because every time a flow is rerouted, its packets have the potential to arrive out-of-order. Thus, even if adaptive techniques could react quickly enough to match the performance of packet-level routing, it would undermine the very reason for using flow-level routing in the first place, which is to avoid out-of-order arrivals.

In related work, average throughput is often the metric used to measure performance. This makes sense when maximizing the overall utilization of the network is the goal. However, emphasizing average throughput can hide the unfairness that can exist between flows. In this work we are concerned with traffic isolation among tenants and the degree to which this can be achieved depends directly on the degree to which we can provide fairness among flows. This is because if we are to maintain agility, we must assume that flows between different servers may belong to different tenants. To achieve strong isolation, we cannot tolerate significant unfairness among flows. Therefore, to properly evaluate load balancing in this context, we must emphasize the minimum throughput achievable under any traffic pattern. This is also consistent with the way throughput is measured in interconnection networks, where it is defined as the minimum throughput of any flow [22].

3.1.1 Methodology

Before we present the results of our study, we briefly detail the methodology used. We rely primarily on simulation but we devote a portion of this section to validate some of our results through probabilistic analysis.

Topology

In this study, we will focus on FatTrees that are fully provisioned and we will assume that all links have the same capacity. The reason for focusing on this type of network is that routing is most important when the oversubscription factor is 1. An undersubscribed network has more capacity than the servers can use which means congestion is less likely to occur as a result of poor load balancing. Such a network is also not very practical to construct in the data center context. Oversubscribed networks, by contrast, are more realistic and can be constructed by using fewer switches and thus providing fewer paths among servers. While our work does consider oversubscribed networks, the approach to routing does not fundamentally change when fewer paths are present. Choosing to focus on the case where all links have the same capacity also represents the case where routing is most important. Some FatTree topologies, such as VL2 [26], use higher capacity switch-to-switch links than server-to-switch links. These topologies provide the same capacity using fewer paths and are therefore physically already a step closer to a tree. A comparison of the relative importance

of routing in FatTree data centers as the oversubscription ratio and link speeds are varied can be found in [41].

Packet-level simulator

We used our own packet-level simulator written on top of the OMNeT++ framework [7]. The simulator is described in detail in Appendix A. In all of our experiments, we used 1 Gbps links. While our simulator can model propagation delay and bit errors on links, for these simulations we used an idealized representation for links that only captures transmission delay. Ethernet switches are modeled as idealized output-buffered switches and we do not model their processing delay because this is typically quite small (often less than a millisecond) on modern switches. We used switch queue sizes in the range of 32-1024 KB, which are typical per-port buffer sizes on commodity Ethernet switches [48]. Unless otherwise stated, switch queues in our experiments are 32 KB.

Traffic pattern

Since our goal is to evaluate the performance of load balancing, it is important to decouple the traffic source from the network. This means that the traffic pattern presented to the network should be independent of the traffic that the network delivers. In other words, the traffic produced by the servers should not be affected by loss or delay in the network. The specific traffic pattern used can be represented by a matrix where element (i, j) represents the rate server i sends to j. The sum of a row represents the total rate a server sends and the sum of a column represents the total rate of traffic destined for it. These rates are expressed as a fraction of the server's interface. Since the purpose is to measure the performance of routing, the traffic matrix should be normalized so that the rates do not exceed the limit of a server's interface. In other words, under ideal routing it should be possible to deliver any traffic pattern as long as the sum of any column or row in the traffic matrix is less than or equal to 1. In these simulations, each server sends at the same rate and this value corresponds to the offered load.

Measuring latency and throughput

The capacity of the network is the minimum throughput achieved by the network under any traffic pattern. We measure the average for each flow as well as the mean and maximum latency experienced by packets in the flow. As we mentioned earlier,

the throughput of the network is defined as the minimum throughput of any flow. However, at times we show both the average and the minimum throughput in our results to highlight the difference. Data was collected using the replication method. Each data point represents the mean of 30 iterations, where each iteration uses a different random number seed. We used a measurement interval of 100 ms with a 10 ms warmup interval to ensure the network reaches steady state prior to the measurement interval. This warmup interval was chosen conservatively to match the various network sizes and traffic patterns we used. By using the ensemble-average technique to measure throughput and latency measurements over increasing intervals of time, we found that the network actually reaches steady state much more quickly for most of our simulations. Unless otherwise noted, error bars represent the 95% confidence intervals where error was derived using Student's t-distribution.

Source queue

The standard setup for measuring interconnection networks includes a source queue in the terminal of each network [22]. With this setup, packets are counted and time stamped (for latency measurements) when they enter the source queue and not when they enter the network. The reason behind this approach is to measure the latency and loss that occurs as a result of the network's flow control mechanism. This is less meaningful in our simulations, however, because Ethernet uses dropping flow control. There is no link-layer flow control mechanism that prevents a server's packets from entering the network[2]. Servers in our simulation do have a source queue at their interfaces but packets are only queued here when the traffic rate temporarily exceeds the rate of the interface. Since the purpose is to measure the latency and loss that occurs in the network, we measure the packets as they leave this queue.

3.1.2 Oblivious flow-level routing

To highlight the limitations of flow-level routing we will focus on oblivious routing. Adaptive routing can perform much better on average than oblivious routing, but as we discussed earlier, it provides little benefit under rapidly changing traffic. While

[2]Ethernet does support link-layer flow control but we do not use it here because issues such as head-of-line blocking create complexities that limit its effectiveness in large multi-stage networks [35, 48, 58, 50].

the traffic patterns that we use here to stress oblivious routing are static, on small timescales, adaptive routing still exhibits the same worst-case behavior when the traffic is dynamic.

For flow-level routing, a demanding traffic pattern is one where each server sends to only one other server. This type of traffic pattern is called **permutation traffic** [22] since the traffic matrix can be expressed as a permutation matrix. The intuition behind why this represents a difficult case is simple. A flow can only be assigned to one path with flow-level routing, so each server sends all of its traffic along a single path. If a server were to split its outgoing bandwidth among multiple flows, it would be more likely to spread its load evenly over more paths.

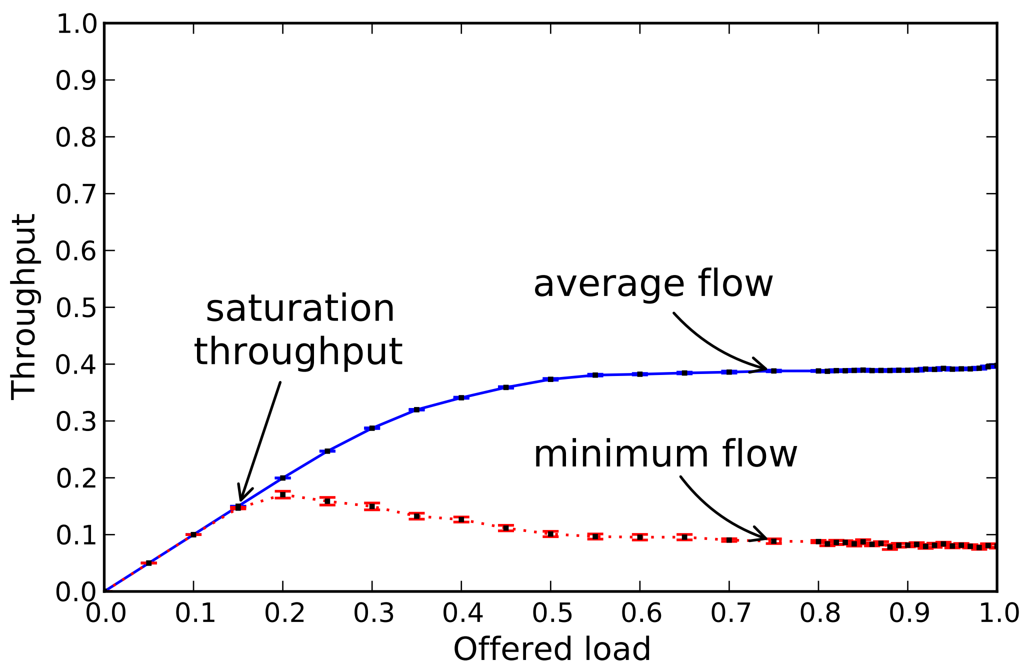


Figure 3.2: Performance of ECMP routing under the permutation traffic pattern.

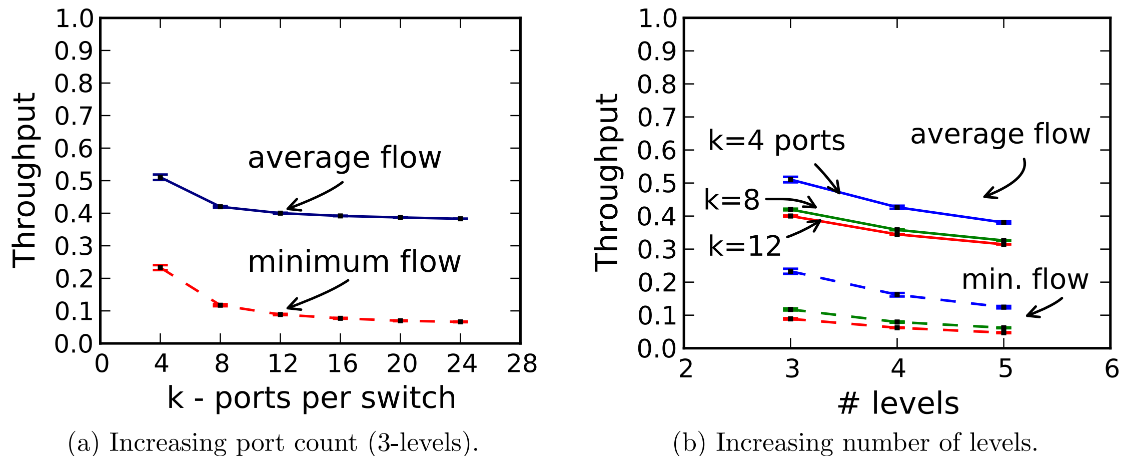
Figure 3.2 shows throughput vs offered load for a 3-level FatTree constructed from 16-port gigabit switches. This network has 1024 servers spread across 128 top of rack switches (switches at layer 1). There are 64 intermediate switches and thus 64 paths available between any two servers. The traffic pattern evaluated here matches the permutation pattern described above with the exception that we added the restriction that no server sends to another server within the same subtree. This helps to stress the network as it forces all traffic to be routed through the intermediate switches.

Without this restriction, traffic within a given subtree would not be routed through all three layers. As we can see, the average throughput over all server-to-server flows is only around 40% of the offered load. This result matches the 60% loss rate reported in VL2's evaluation of ECMP [26].

Minimum throughput

Measuring the average throughput across all flows gives an optimistic view of performance as it does not take into account fairness among flows. This is why it is important to measure the throughput of the minimum flow when reporting throughput. The line labeled "minimum flow" represents the minimum throughput among all flows averaged across all 30 repetitions. This reveals that some flows receive less than $\frac{1}{4}^{th}$ of the throughput of the average flow. More importantly, this shows that the network reaches saturation at around 15% of its capacity. This is the point at which the minimum and the average throughput separate and it indicates the point at which flows begin to affect each others throughput. Thus, if we relied on oblivious routing, we could not allocate more than $\frac{1}{7}^{th}$ of the network's resources to tenants and still guarantee isolation.

Scaling



(a) Increasing port count (3-levels).

(b) Increasing number of levels.

Figure 3.3: Performance of ECMP as the network scales.

The performance of oblivious flow-level routing becomes slightly worse as we scale to larger networks. Ethernet switches are only cost effective when they contain a limited

number of ports, which is, after all, the motivation behind constructing FatTree networks out of many smaller, inexpensive switches. To keep the oversubscription factor at 1, we can scale the network by keeping the number of levels in the FatTree fixed at 3 and increase k, the number of ports per switch. Figure 3.3a shows the performance for ECMP under the same permutation traffic pattern as we scale k from 4 to 24. At 24 ports, we begin to approach the upper limit on the size of the network that we can feasibly simulate on commodity PCs. At this size, however, it already becomes clear that the drop in average throughput begins to level off around 40% of the network's capacity.

We can also increase the size of the network by keeping the number of ports fixed and increasing the number of levels. We show the scaling characteristics as we increase the network from 3 to 5 levels for several different values of k in Figure 3.3b. This shows that increasing the number of levels also reduces throughput but that the rate of decline does begin to slow as we move beyond 5 levels. These results suggests that we can simulate 3-level networks of modest size (e.g. 3-levels, 12-16 ports) and still expect our results to extend to larger networks.

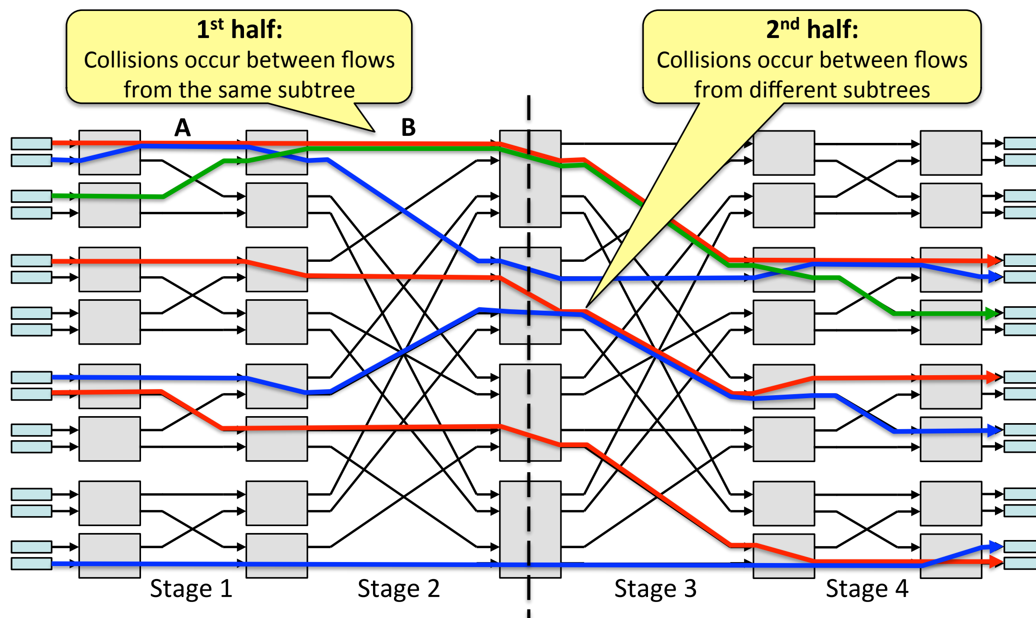


Figure 3.4: Flows colliding in an unfolded 3-level 4-port FatTree.

3.1.3 Understanding the performance of ECMP

To understand the poor performance of flow-level routing, we can use a probabilistic method to compute the throughput that we should expect to see with ECMP. Computing the throughput analytically allows us to validate the correctness of our simulations and helps provide some insight into the behavior of flow-level routing. The particular point of comparison we chose for this exercise was the performance of ECMP in a 3-level FatTree as the switch port count is varied. This is similar to what we presented in Figure 3.3a except that we used a restricted form of the permutation traffic pattern so that we could limit our analysis to the first half of the network.

To determine the throughput of a given flow, we need to consider the ways in which it may collide with other flows in the network. To aid in this discussion, we present an unfolded view of the 4-port FatTree network in Figure 3.4. The unfolded FatTree is essentially equivalent to the folded network shown earlier in Figure 3.1a with the main difference being that the network is drawn with unidirectional links so that all traffic flows form left to right. The unfolded view helps illustrate the ways in which flows can collide and allows us to divide the network into a set stages which we can analyze separately. This view does imply that all traffic must pass through the intermediate switches which is not necessary for traffic local to a subtree in the folded network.

For the traffic pattern we use here, all traffic is between servers in different pods. In the literature, a "pod" refers to the set of servers that share the same links to the intermediate switches (i.e. a subtree at level $l-1$). There are 4 such pods in Figure 3.4 and a FatTree with k port switches has k pods. In the first half of the network, flows can only collide if they are from servers in the same subtree. Also notice that collisions in the first half only depend on the paths that flows take and not their destinations. Flows from different pods can only collide in the second half of the network when they are destined for the same subtree. Notice that flows that collide on a link in the first half may diverge and subsequently reconverge on a link in the second half. However, they can no longer affect each others' throughput when this happens. We can leverage this observation by adding a requirement that the servers in one pod all choose destinations in the same pod. We call this the "pod-to-pod" permutation pattern and under this traffic pattern, no further loss can occur after the second stage.

Throughput in stage 1:

We begin by using the 4-port network as an example.

Let X be a random variable representing the number of flows on a link in stage 1, e.g link A. Since there are 2 servers per switch, the possible outcomes are 0, 1, 2 flows. Each flow is assigned randomly to a path independently of the other flows so a flow is routed through link A with probability $p = \frac{1}{2}$:

$$p(X = x) = \begin{cases} 1/4, & \text{if } x = 0 \text{ flows} \\ 1/2, & \text{if } x = 1 \text{ flows} \\ 1/4, & \text{if } x = 2 \text{ flows} \end{cases}$$

The throughput in stage 1 is equivalent to the expected load on link A. The load on link A is a function of X, which we can represent as another random variable $A = g(X)$. Since each server has one flow sending at full offered load, A has outcomes 1 if it carries one or more flows and 0 otherwise. So we have:

$$p_A(a) = \begin{cases} 1/4, & \text{if } a = 0 \\ 3/4, & \text{if } a = 1 \end{cases}$$

This gives us an expected load of $E[A] = 0.75$ in stage 1. Thus the average throughput of a flow in stage 1 is also 0.75.

We can generalize this for k port switches by viewing X as the number of successes in a series of n bernoulli trials where the probability of success is $p = \frac{1}{n}$ for each trial. This follows a binomial distribution $B(n, p)$ giving X the probability mass function (PMF):

$$P_X(x) = \binom{n}{x} p^x (1 - p)^{n-x} \tag{3.1}$$

Since the load on link A is 1 as long as $x \neq 0$ the expected throughput is equivalent to $1 - P(X = 0)$. Thus with $n = \frac{k}{2}$ the expected throughput in stage 1 is:

$$E[A] = 1 - (1 - \frac{1}{n})^n \tag{3.2}$$

Observe that $\lim_{n\to\infty}(1-\frac{1}{n}) \approx \frac{1}{e}$. This means as we scale the number of ports, the throughput in stage 1 converges to $1-\frac{1}{e}$, which is roughly 63%.

Throughput in stage 2:
Finding the expected load on a link in stage 2 is more complicated since flows that collide in stage 1 do not create the same load on links in stage 2. To find the total load on some link B in stage 2, we first find the fraction of traffic from link A that continues on to B. Let Y represent the flows that continue on to B from the X flows on link A. For each pair (x, y) we need to compute the probability that $P(X = x, Y = y)$. For the 4-port case this joint probability distribution is:

$$P_{X,Y}(x,y) = \begin{cases} \frac{1}{4}, & (0,0) \\ \frac{1}{4}, & (1,0) \\ \frac{1}{4}, & (1,1) \\ \frac{1}{16}, & (2,0) \\ \frac{1}{8}, & (2,1) \\ \frac{1}{16}, & (2,2) \end{cases}$$

We then calculate Z, the load from link A on link B:

$$Z = \begin{cases} Y/X & \text{if } X > 0 \\ 0 & \text{if otherwise} \end{cases} \tag{3.3}$$

For the 4-port case we have the PMF:

$$P_Z(z) = \begin{cases} \frac{9}{16}, & 0 \\ \frac{1}{8}, & \frac{1}{2} \\ \frac{5}{16}, & 1 \end{cases}$$

Z represents the load on link B from one of the switches in stage 1. Since there are two such switches, we need to consider the contributions of both. We can represent their combined load on B as Z_1+Z_2. Since the link has a capacity of one, $B = Z_1+Z_2$

if $Z_1 + Z_2 \leq 1$ and 1 otherwise. The PMF of B is then:

$$B(Z_1 + Z_2) = \begin{cases} \frac{81}{256}, & 0 \\ \frac{36}{256}, & \frac{1}{2} \\ \frac{139}{256}, & 1 \end{cases}$$

For this case where $k = 4$, the expected throughput in stage 2 is thus $\frac{157}{256} \approx 0.613$. Since each pod is independent, this also represents the throughput we expect to see under the "pod-to-pod" permutation pattern.

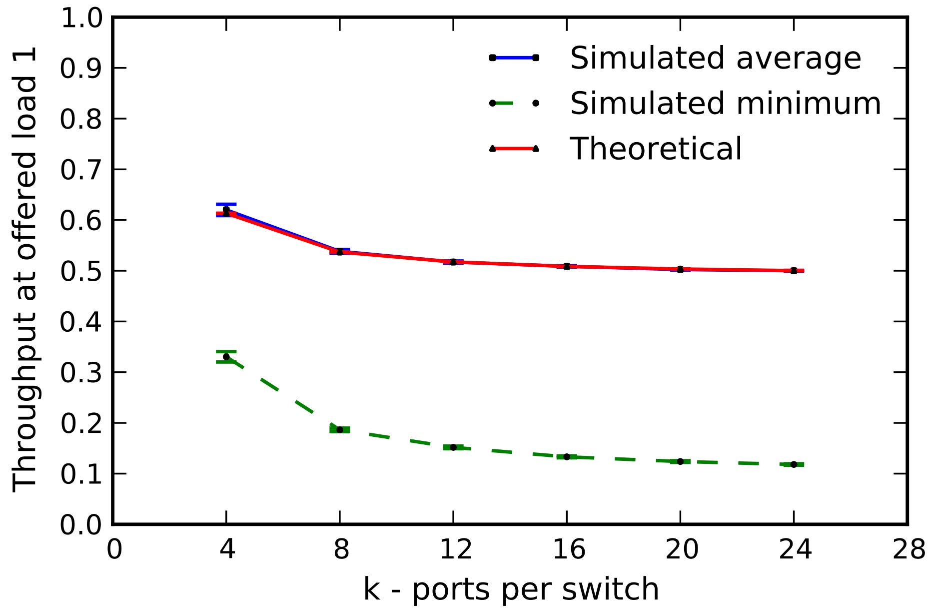
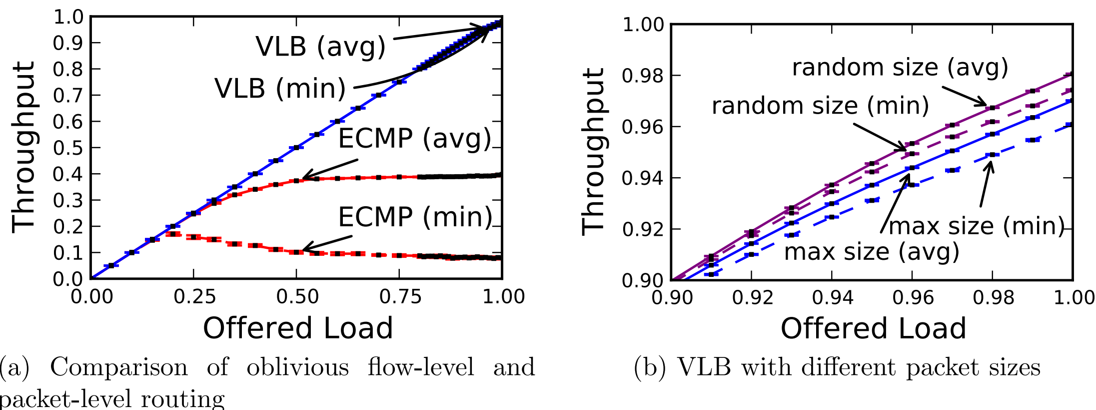


Figure 3.5: ECMP routing under the pod-permutation traffic pattern

In order to compare our simulations to the analysis, we simulated the pod-permutation traffic pattern for the values of k from 4 to 24 ports. We wrote a python script to carry out the analysis described above for each value of k. The results are shown in figure 3.5 and demonstrate that the simulated throughput closely matches the theoretical values.



(a) Comparison of oblivious flow-level and packet-level routing

(b) VLB with different packet sizes

Figure 3.6: Performance of packet-level VLB under the permutation pattern.

3.1.4 Oblivious packet-level routing

To compare the performance of oblivious packet-level and flow-level routing, we repeated the same experiment shown in Figure 3.2 using packet-level VLB. The results are shown in Figure 3.6a and shows that packet-level VLB clearly performs much better than flow-level VLB. The average throughput is above 95% of the network's capacity. The degree of unfairness between flows is also much smaller; the minimum throughput for any flow is only a few percent less than average. Figure 3.6b shows the performance of VLB under high offered load with the traffic consisting of different packet sizes. The line labeled "max size" refers to traffic consisting of 1500 byte Ethernet frames, which is the standard MTU. The line labeled "random size" represents sizes that are uniformly distributed between the minimum (84 bytes) and maximum size Ethernet frames. The performance is worse for maximum size packets because the limit on queue sizes at switches is in terms of bytes which means that load balancing effectively occurs at a coarser granularity with large packets.

3.2 Packet-level routing strategies in DCNs

In the previous section we compared randomized packet-level and flow-level routing and found that packet-level VLB performs significantly better than randomized routing at the flow-level. However, the results from Figure 3.6b indicate that there is

some room for improvement. In particular, there are three reasons to believe that a more sophisticated strategy could perform better.

- First, randomly spreading packets ensures that all paths receive an equal amount of traffic, on average, but when viewed over small time scales, significant imbalances can exist. Moreover, if an imbalance produces longer delays on some paths, these delays can persist for a fairly long time.
- Second, randomly spreading packets does not account for the varying size of packets. Thus, even if all paths receive an equal number of packets, some may receive significantly more load than others.
- Finally, routing based on paths does not account for the fact that paths leading to different subtrees share different links. This means that balancing a server's traffic evenly among the intermediate switches does not necessarily ensure its load is spread evenly across the links in each of the subtrees that it sends to.

In this section we take a closer look at these issues to examine how packet-level routing can be adapted to provide optimal performance in the data center environment. The purpose here is to identify which strategies can, in principle, provide the best performance. As such, we use a variety of traffic patterns and metrics that highlight differences in order to understand the factors that effect performance and guide our design choices. In the next section we follow a more rigorous methodology to evaluate how well we can expect these approaches to perform in practice.

3.2.1 Imbalances on small time-scales

We will begin by addressing the imbalances created by randomly spreading packets. To isolate the other factors, we will keep the sizes and destinations of packets fixed. Thus we use the same permutation traffic pattern as before using a fixed packet size. If we start by considering just the servers within a pod, then if there are p servers, they share p paths. VLB ensures that by choosing paths randomly, on average each path receives the same number of packets and thus the same load. On small time scales, however, we should not expect load to be balanced evenly. For example, let's

consider just the time it takes each server to send a single packet and let's call this a *packet interval.* For servers to balance load evenly over a single packet interval, each server would have to choose a unique path for its packet. Of course, this is unlikely to happen when paths are chosen at random. Balancing load evenly at this time scale would require coordinating the transmission of every packet which is clearly not practical in a data center network. Given that servers must make routing decisions independently, we cannot avoid imbalance at the level of a single packet interval.

While we cannot prevent servers from choosing the same path in a given interval, we can limit the number of times they do so. Let's define a *round* as consist of p packet intervals. Since there are p servers and p paths, load would be balanced evenly in a round if each path receives p packets. With VLB, each server picks a path at random at every interval. This means there is nothing to prevent all servers from sending to just a single path over the course of the round. This means in the worst case it is possible for one path to receive p^2 packets. We can reduce this unevenness if we prevent each server from using a given path more than one time per round. A simple way to accomplish this is for each server to compute a permutation on the available paths each round and send one packet to each path in a round robin fashion. We implemented this approach which we call the "Permutation Round Robin" (PRR) load balancer.

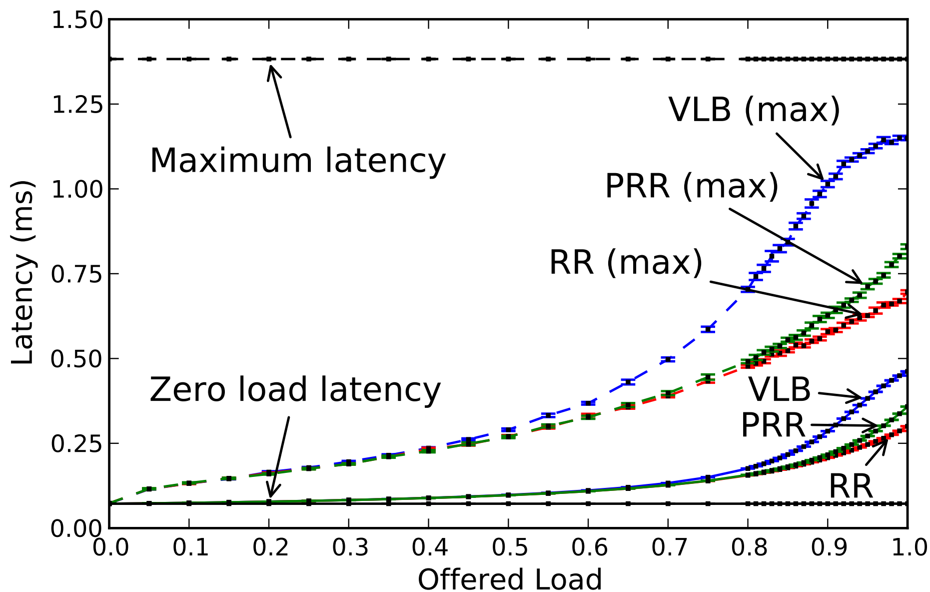


Figure 3.7: Comparison of VLB and Round Robin (RR) with maximum size packets.

Permutation Round Robin

We compared the performance of PRR to VLB by rerunning the experiment in Figure 3.6b. We found that PRR effectively achieves 100% throughput on permutation traffic as the maximum loss rate for any flow was less than a tenth of a percent. Therefore, rather than present the throughput, we show the difference in latency vs offered load. In Figure 3.7 we plot both the average latency across flows and the maximum latency experienced by a flow as a function of the offered load. When there is no congestion in the network, no queueing occurs at switches which means packets experience only the transmission delay at each hop in the network. This is known as the zero load latency and we included the theoretical value for the network in the figure. We also plotted the maximum latency, which corresponds to the delay that a packet would have if it experienced the maximum possible queueing delay at each hop. It is important to point out that this plot only shows the latency of packets that are actually delivered. For this reason, the maximum latency of a flow never reaches the theoretical maximum since packets are unlikely to experience the maximum amount of delay at every hop without being dropped.

We also simulated the performance of a simpler round robin load balancer (RR) that only performs an initial permutation on the list of paths rather than computing a new permutation after each round. Interestingly, this approach performs slightly better. The reason for this is that computing a new permutation can cause a server to use a path twice in a row if the first path in the new permutation is the same as the last path it used. Since each server may do this for a particular path, a path can receive twice as many packets in the worst case. Thus while computing new permutations can help to desynchronize servers that are sending to paths in the same order, doing so frequently reduces performance.

3.2.2 Accounting for packet size

In most other contexts where randomized routing is used on topologies similar to the FatTree, fixed size messages are used. Ethernet frames, by contrast, typically vary in size from about 80 to 1500 bytes but may even be as large as 9000 bytes when jumbo frames are allowed. This difference in size means that even if packets are spread

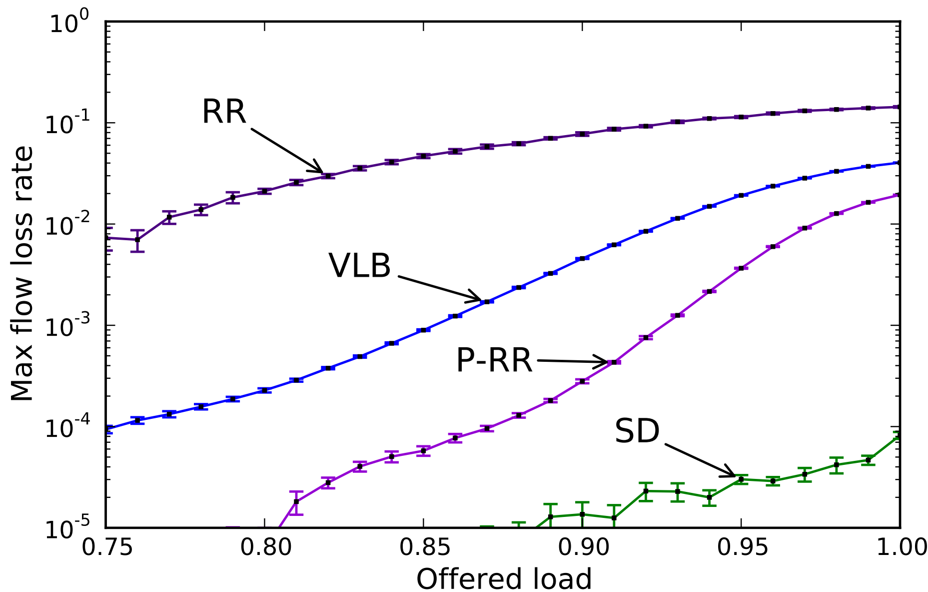


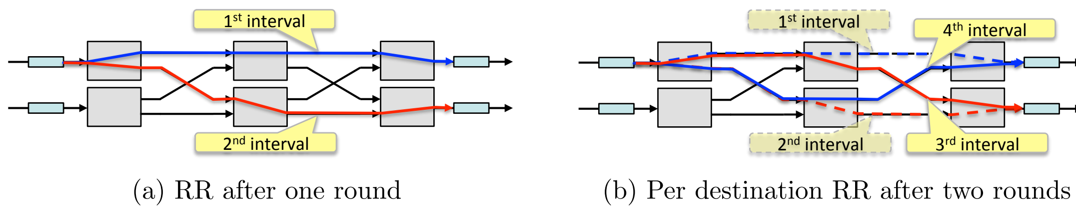
Figure 3.8: Performance with packet sizes alternating between max and min size.

evenly among the paths, it is still possible to have large imbalances in load between paths. To demonstrate this issue, we had the servers alternate between sending maximum (1500 bytes) and minimum size packets. The result is shown in Figure 3.8. Round robin performs particularly poorly here because every other path receives only large packets. Of course this traffic pattern is somewhat contrived because in practice, servers that saturate their link generally do so by sending large flows consisting mostly of maximum sized packets. Nevertheless, traffic in real networks often follows a bimodal distribution where packet sizes near the minimum and maximum are most common and has been observed in traffic studies of data center networks as well [26].

We experimented with several ways of accounting for packet sizes. The most straight forward is to include a counter with the round robin approach that keeps track of the number of bytes sent. The Surplus Round Robin (SRR) load balancer works by associating a "deficit counter" with each path that is initialized to 0. After sending a packet along a particular path, the number of bytes in the packet is subtracted from the counter associated with the path. We continue to send packets along the same path until the counter becomes negative at which point we switch to using the next path. When the counter for every path is negative, we begin a new round and all counters are incremented by a fixed quantum, e.g. 1500 bytes.

We compared this approach to round-robin and found that the best performance was always achieved with the smallest possible quantum (e.g. 1 byte). With such a small quantum, SRR may have to cycle through the list of paths one or more times before enough credits are added to find a path with non-negative credits. This suggests that there is a much simpler approach. We can sort the list of paths by the number of credits and always use the path with the most credits. When all paths have negative credit we add a new quantum to each counter and perform a permutation on the list of paths. The permutation causes ties among paths with equal credit to be broken randomly. Its use is optional and serves to periodically desynchronize servers in a similar fashion as PRR. We call this the **sorted deficit** (SD) load balancer and it is also shown in the figure above. Since it achieves the same effect as SRR with a quantum value of 1, we omitted SRR from our results and only show SD in Figure 3.8. Intuitively we want a server to minimize the imbalance that its traffic creates among the paths. Choosing the path with the most credits is the greedy choice that realizes this goal. Without making assumptions about the sizes of future packets, this is also the best that we can do.

3.2.3 Accounting for topology



(a) RR after one round

(b) Per destination RR after two rounds

Figure 3.9: Load balancing with multiple flows in a simple two port network.

So far, we have focused on a single traffic pattern, the permutation pattern, in order to demonstrate the effect of packet size and short-term imbalances on performance. While VLB provides roughly the same performance under all traffic patterns, the load balancers described above are more deterministic in nature which makes their performance more dependent on the traffic pattern. While the permutation pattern represents the worst case for flow-level routing, it represents an easy case for packet-level routing. When sending to a single destination, a server that balances its traffic

across the paths in the first half of the network automatically balances its load across the paths to the destination in the second half of the network.

When a server sends to multiple destinations, however, this assumption does not always hold. For example, consider the very simple two-port network in Figure 3.9a. Here one server has two flows to different destinations. Imagine that this server sends to both destinations evenly and at a constant rate so that it sends one packet to each destination in each round. There are two intermediate switches thus the server can choose from two paths. If we used the round-robin approach, then, in each round, the server would use the same path to send to the same destination. In other words, every round would proceed in the same fashion as shown in Figure 3.9a. This would mean that while load is balanced evenly in the first half of the network, each flow only uses one path in the second half. Of course, in practice, such a cycle might quickly be broken but over short intervals it can lead to imbalance in the second half of the network.

To mitigate this, we would have to account for the destinations of packets when choosing paths. However, routing decisions cannot simply be made independently for each destination or balancing load in the second half would come at the cost of creating an imbalance in the first half. Returning to our example, we can imagine that the server performs RR on a per-destination basis. As shown in Figure 3.9b, this ensures it uses a different path for each flow in the second round. Notice, however, that the server now uses the same path in two subsequent intervals. In general, with n destinations a server could use the same path up to n times in a row. This would be true for any approach that would do its accounting per-destination since the next path would always be dependent on the destination of the next packet. This suggests that to properly balance traffic in both halves of the network, we would need to consider the cost of creating an imbalance at every stage.

Multiphase Sorted Deficit

To examine the benefit of accounting for topology, we extended the SD load balancer to keep track of the number of bytes placed on paths in multiple stages in order to choose the least loaded path. We implemented two separate version of this load

balancer. We call the first version the two-phase (TP) load balancer. In addition to normal set of counters, the TP load balancer maintains a separate list of counters for each destination that it sends to. It simply adds the credits from both counters together and chooses the path with the most credits. We then implemented a version that maintains a separate list of counters for each subtree in the network. We call this the multi-phase (MP) load balancer. The advantage of the two phase load balancer is that it is simpler and does not require knowing the precise topology (e.g. number of ports, levels, etc). The multi-phase, by contrast, effectively keeps a counter for each link the server can send across, enabling it balance the server's load as precisely as possible.

		Maximum queue length observed at each stage in KB									
		ECMP	**VLB**	**RR**	**P-RR**	**SD**	**P-SD**	**TP**	**P-TP**	**MP**	**P-MP**
Permutation	stage 1	1024	699	18	37	19	37	19	35	9	13
	stage 2	1024	628	21	35	19	38	19	35	19	38
	stage 3	1024	586	19	35	19	37	19	35	19	37
	stage 4	1024	417	18	32	16	31	16	32	12	18
	stage 5	0	78	12	18	12	19	12	19	12	18
All-to-all	stage 1	1024	792	22	42	21	37	51	38	16	16
	stage 2	1024	120	19	31	21	28	35	28	28	31
	stage 3	1024	129	1024	132	1024	678	75	95	79	73
	stage 4	1024	625	1024	794	1024	1024	136	201	116	127
	stage 5	70	85	83	79	75	237	35	44	92	88

Table 3.1: Imbalance in queueing at various stages in the network.

Table 3.1 shows the maximum length of queues in various stages of a 3-level FatTree with $k = 12$ ports. Here we compare the permutation traffic pattern with an "all-to-all" traffic pattern in which every server sends to every other server. In this experiment, servers sent maximum sized packets at uniform rates causing their packets to be evenly spaced in time. We simulated each of the described load balancers with and without the use of periodic permutations on the list of the paths they maintain (denoted with the prefix P-). The values in the table indicate the maximum queue length observed at each stage of the network across 30 runs with the offered load at 1. To highlight the difference in queueing at various stages in the network, we increased the size of switch queues to 1 MB. This means that values of 1024 KB correspond to the maximum queue size and indicate that some packet loss is likely to occur.

Because packets are sent at fixed rates, there is a high degree of synchronization that can occur between paths and destinations, even when periodic permutations are used. The results show that all of the packet level strategies perform reasonably well

under the permutation pattern but that only the two-phase and multi-phase load balancers perform well under the all-to-all pattern. While RR and SD manage to avoid imbalances in the first half of the network, they fail to prevent the imbalances that can occur in the second half of the network. The results also show that while the multi-phase load balancer performs the best, it only performs slightly better than the simpler two-phase approach.

3.2.4 Comparison of approaches

The previous experiment was used to demonstrate the imbalances that can occur in the network and to illustrate the differences between the approaches. However, it was somewhat contrived in the sense that each flow produces traffic at a fixed rate according to a periodic process. This means the packets within each flow are spaced uniformly in time which does not represent realistic traffic. In general, we use random packet spacing by using an exponential distribution centered around a mean corresponding to the sending rate of the flow. To provide a more accurate comparison of the different approaches, we simulated three different traffic patterns under various packet sizes using both random and uniform packet inter-arrival times. We used a 3-level FatTree consisting of $k = 12$ port switches for this experiment. The three different packet sizes shown correspond to the ones described so far, i.e., maximum size, random size, and alternating minimum and maximum size packets.

		% of capacity before loss exceeeds threshold						
Inter-arrivals	Packet size	**ECMP**	**VLB**	**RR**	**SRR**	**SD**	**TP**	**MP**
Uniform	random	0.18	0.95	1.00	1.00	1.00	1.00	1.00
	maximum	0.18	0.95	1.00	1.00	1.00	1.00	1.00
	alt. max min	0.19	0.90	0.96	1.00	1.00	1.00	1.00
Random	random	0.17	0.95	1.00	1.00	1.00	1.00	1.00
	maximum	0.17	0.94	1.00	1.00	1.00	1.00	1.00
	alt. max min	0.17	0.89	0.95	1.00	1.00	1.00	1.00

Table 3.2: Throughput before maximum loss exceeds threshold - permutation traffic.

For the experiment corresponding to Table 3.2 and Table 3.3, we increased the offered load until the loss rate of some flow exceeds a certain threshold, which we set at 0.1% of its packets. This measures the throughput achievable while ensuring that no flow

Inter-arrivals	Packet size	% of capacity before loss exceeeds threshold						
		ECMP	**VLB**	**RR**	**SRR**	**SD**	**TP**	**MP**
Uniform	random	0.89	0.88	0.89	0.89	0.89	0.89	0.95
	maximum	0.13	0.10	0.06	0.08	0.08	0.07	0.27
	alt. max min	0.94	0.80	0.80	0.80	0.80	0.80	0.82
Random	random	0.81	0.86	0.87	0.86	0.86	0.86	0.87
	maximum	0.82	0.87	0.87	0.87	0.87	0.87	0.88
	alt. max min	0.75	0.77	0.78	0.78	0.78	0.77	0.77

Table 3.3: Throughput before maximum loss exceeds threshold - all-to-all traffic.

Inter-arrivals	Packet size	% of capacity before loss exceeeds threshold						
		ECMP	**VLB**	**RR**	**SRR**	**SD**	**TP**	**MP**
Uniform	random	0.93	0.98	0.98	0.98	0.98	0.98	1.00
	maximum	0.93	0.97	0.98	0.98	0.98	0.98	1.00
	alt. max min	0.92	0.94	0.95	0.95	0.95	0.95	0.97
Random	random	0.92	0.96	0.96	0.96	0.96	0.96	0.97
	maximum	0.92	0.95	0.96	0.96	0.96	0.96	0.97
	alt. max min	0.89	0.91	0.92	0.92	0.92	0.92	0.92

Table 3.4: Throughput before average loss exceeds threshold - all-to-all traffic.

experiences more than a given degree of loss. Table 3.2 shows the results under the permutation traffic pattern which shows that all of the load balancers do fairly well with only RR and VLB failing to achieve 100% throughput.

For the case of the all-to-all traffic pattern in Table 3.3, however, we see that the performance of ECMP improves but that all of the packet-level approaches perform significantly worse. The relative improvement with ECMP can be explained by the fact that with more flows, each server is effectively able to balance its traffic over more paths. One reason for the poor performance with the packet-level approaches is that with more flows per-server, each flow sends proportionally fewer packets. This means the it can tolerate less packet loss over a given window of time making the maximum loss threshold unusually sensitive to short term unfairness for cases such as the all-to-all traffic pattern since there are 432 servers. In the experiment, we measured loss over a 500 ms window. However, with 431 flows per server, the average flow sends fewer than 100 maximum sized packets over this interval even at full offered load. Thus the loss threshold will be exceeded as soon as any packet is lost. This also explains the unusually poor performance that we see with uniform inter-arrival times

and maximum sized packets. This is an artifact caused by the use of deterministic packet sizes and departure times which can lead to events becoming synchronized causing some flows to consistently experience loss. For comparison, Table 3.4 shows the results for the same experiment where the loss threshold was defined as the average loss across all flows rather than the maximum.

It is important to point out the high rate of loss that occurs with random inter-arrivals is independent of load balancing. Loss occurs not because the traffic is not evenly balanced but rather that the combined traffic to a destination can temporarily exceed the physical capacity of the network. If each server sends n flows then each flow only sends at an average rate of $\frac{1}{n}$. Thus for a large number of flows, traffic can become bursty causing some servers to receive packets from a large number of senders at the same time. In effect, the traffic generated is inadmissible when viewed over small time scales which means that loss would occur even with ideal routing. This will be demonstrated in the next section by simulating on the equivalent logical tree network

3.3 Performance in context

The goal of the previous section was to identify the routing strategy that can, in principle, provide the best performance. Our results indicated that performance depends heavily on how well behaved or bursty the traffic is. In this section, we apply a more rigorous methodology in order to build a more accurate picture of the performance that we can expect in practice.

3.3.1 Separating routing & flow-control

To separate load balancing from flow control, we need some way to understand how much queueing can result from each. This depends on how tightly each mechanism can control traffic in the data center network. For example, the job of flow control is to ensure that the traffic produced by servers represents an admissible traffic matrix. Thus one way to characterize its effectiveness is the timescale on which it can do this. At one extreme we can model the rate that a server i sends to a server j as a Poisson

	Loose: Poisson Process	**Strict:** Periodic Process
Flow Control Server i sends to j with rate r_{ij}	**Poisson send process:** $\lambda = \frac{1}{r_{ij}}$	**Periodic send process:** $T = \frac{1}{r_{ij}}$
Routing Server i selects from p paths in subtree	**Poisson path selection:** $\lambda = \frac{1}{p}$	**Periodic path selection:** $T = \frac{1}{p}$

Figure 3.10: Separating flow control from load balancing.

process with rate parameter $\lambda = \frac{1}{r_{ij}}$. We call this **"loose" flow control** because it means the flow control mechanism only ensures that the server's rate matches r_{ij} on average. At the other extreme, we can model the sending rate as a periodic function where server i sends a packet to server j on a fixed period exactly equal to $\frac{1}{r_{ij}}$. We call the periodic **"strict" flow control** because it is the finest timescale at which a server can control its sending rate.

We use a similar approach to model routing by viewing the path that a server uses as a periodic or Poisson process. Since the routing algorithm cannot control when or where the next packet will be sent, we cannot control how often a server uses a given path in a specific subtree. Here we can only model the selection of paths in each subtree as a process. For VLB, this process is Poisson since it chooses randomly without regard to the paths or destinations of previous packets. Thus it precisely represents loose routing. Strict routing by contrast, would ensure that i use a given path once for every p packets it sends through subtree s. As we discussed in 3.2.3, we cannot ensure the selection of paths is truly periodic at every subtree since different destinations share different subtrees. Strict routing is therefore an idealized representation. Given that we cannot centrally manage the route of every packet through the network, strict routing represents an upper bound on how evenly traffic can be balanced in a given subtree.

Since flow control and routing represent two different dimensions, we can use this strict/loose model to provide bounds on the space in which we are working. Given that servers cannot coordinate to choose routes or police sending rates on the time scale of individual packet transmissions, we can use strict routing and flow control to get a lower bound on the amount of buffering needed in the network needed by any approach.

Simulating the strict/loose model:
While we can simulate loose routing by using VLB, we cannot simulate strict routing precisely. However, the multiphase load balancer will achieve periodic path selection at every stage whenever possible. We can also use the logically equivalent tree representation of the FatTree (as shown in Figure 3.1b) to model ideal load balancing. With the logical tree form, we make the sizes of switch queues proportional to the number of links they represent in the FatTree. Since there is only one path in a tree, load balancing is effectively removed from consideration. This means simulating the logical tree provides a loose upper bound on the performance of routing. Optimal routing in the DCN must exist somewhere between the two points.

We can readily simulate loose and strict flow control because the sending process at servers can be configured to be periodic or Poisson. However, note that the traffic produced by servers is not truly Poisson since the sending rate of the server is constrained by the speed of its interface. Nevertheless, this lets us capture the four corners of this space with simulation. This gives us a sense of the relative importance of load balancing and flow control and provides upper and lower bounds on the performance that we can expect in practice.

3.3.2 Queueing theory model

In addition to simulation, we can model two of the four cases analytically using queueing theory. These models are described in more detail in B. The M/M/1/K FatTree model represents the FatTree topology as a network of finite capacity M/M/1 queues. Under queueing theory, the arrival rate and departure rate of packets at an M/M/1 queue is modeled as a Poisson process. As the arrival rate on every queue (on every path) follows a Poission distribution, it effectively provides a model for

loose flow control with loose routing. It also most accurately represents an all-to-all traffic pattern since the distribution of traffic rates in the network is proportional to the number of servers. We can use this model to find a conservative lower bound on the performance we might expect for the corner representing loose load balancing and loose routing. The bound is especially conservative at offered loads near 1 because real servers cannot send faster than the rate of their interface yet there is no upper bound on the number of arrivals that can occur at a queue under a random distribution. The M/M/1/K logical tree model can be used to approximate strict load balancing and loose flow control.

3.3.3 Evaluation

There are two metrics that define the performance of load balancing in this context:

- Isolation: The degree of isolation we can provide to tenants.
- Utilization: The capacity that can be used by tenants.

We define isolation as the acceptable fraction of packet loss over all flows and we call this the *loss threshold*. We then define the *usable capacity* as the offered load at which the loss threshold is exceeded. We use the usable capacity metric as a way to define utilization in this context since it represents the fraction of the capacity that can be allocated to tenants while maintaining isolation.

This evaluation focuses on investigating two factors that affect the performance of routing:

- Available buffering: The sizes of switch queues determines how much imbalance we can tolerate before losing packets.
- Flow control: Bursty traffic can create substantial loss which means the degree to which sending rates can be controlled is a key factor affecting performance.

Here we simulate the four corners of our strict/loose model to tease apart the effects of flow control from routing. This allows us to better evaluate the relative differences

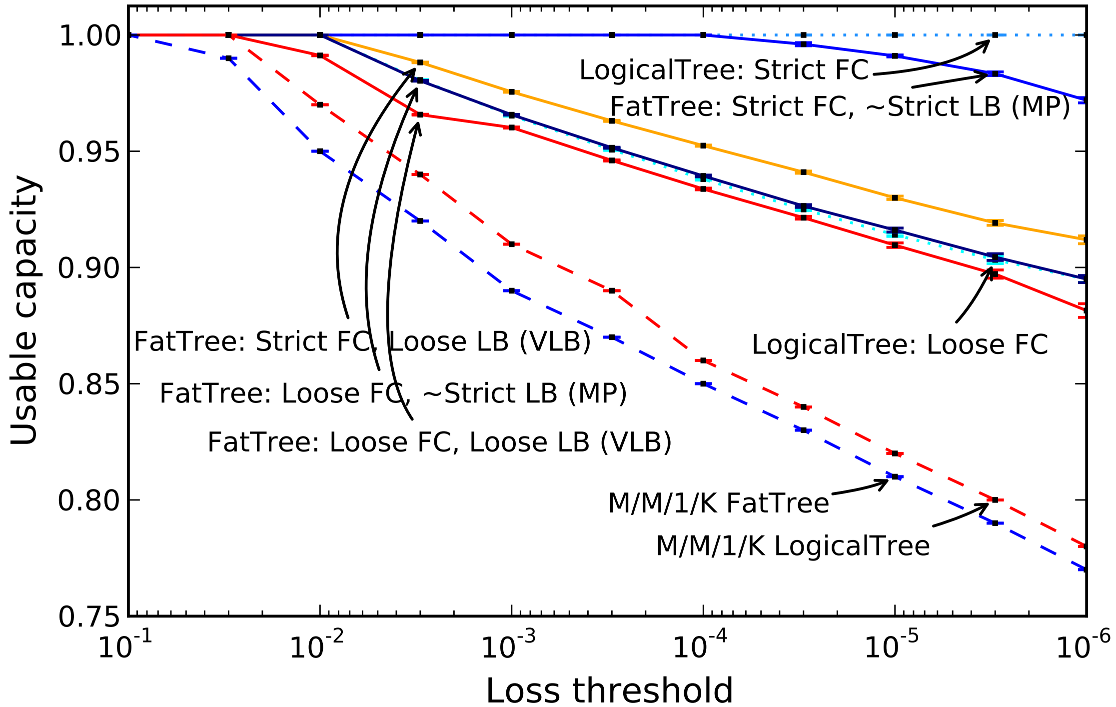


Figure 3.11: Fraction of the network's capacity achievable as a function of the acceptable loss threshold. $l = 3$ levels, $k = 12$ ports, queue size $K = 50$ packets.

between our routing approaches. By varying the switch queue size, we can determine how much buffering is needed to be able to utilize a given fraction of the network while maintaining a given degree of isolation. This also serves as a guide to help weigh some of the costs and tradeoffs of designing a network based around our approach.

Isolation vs. utilization

We begin by simulating the four corners of our loose/strict model under all-to-all traffic and compare the results with our queueing theory model. For reasons that we discuss in B.3, we made the packet size follow a Poisson distribution around the medium packet size (midway between minimum and maximum sized) so that we could provide the best comparison with our queueing theory models.

Figure 3.11 shows the maximum fraction of the network's capacity that we can safely use without exceeding a given loss threshold when the queue size is fixed at 50 packets. With M/M/1/K the departure process is Poisson so this effectively models random

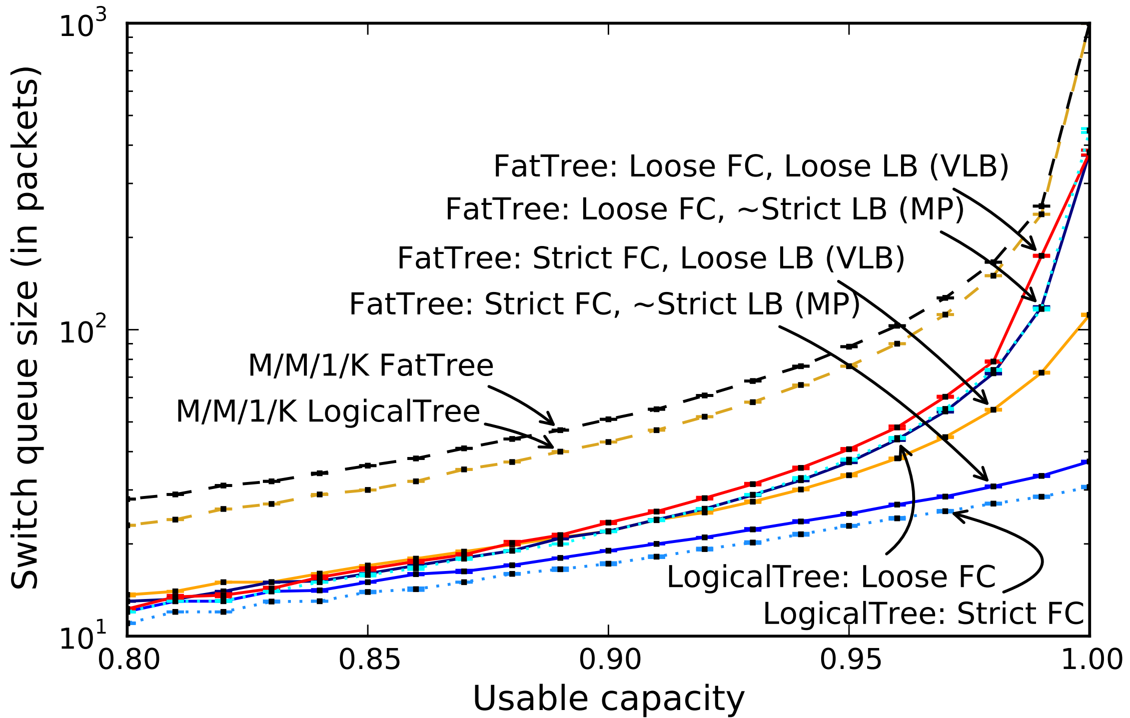


Figure 3.12: Per-port switch buffer size required to achieve given fraction of the network's capacity. $l = 3$ levels, $k = 12$ ports, loss threshold $= 10^{-3}$.

packet sizes. Since the capacity, K, is in terms of packets, we chose a value of 50 packets because it roughly corresponds to a limit of 32 KB when using average-sized packets. In addition, to simulating the 4 corners of the space described earlier, we plotted the results from our M/M/1/K FatTree and M/M/1/K LogicalTree analysis in the figure. The results show that the lower bound on performance provided by the M/M/1/K models is overly conservative. However, they show that, with a small amount of buffering, we can expect to achieve a reasonable degree of isolation (e.g. loss thresholds of 10^{-3} or 10^{-4}) while being able to effectively utilize at least 85% of the network's capacity.

In Figure 3.12 we fix the loss threshold at 10^{-3} and we plot switch queue size as a function of offered load. This is useful because it shows the per-port buffering needed at switches to avoid exceeding the loss threshold when using a given fraction of the network's capacity. Note that queue size is shown with a logarithmic scale. This is because, according to M/M/1 queueing theory, the length of the queue will grow exponentially with the traffic intensity and our FatTree and LogicalTree models

show this. Our simulations for loose flow control (Poisson send process) also shows exponential growth although at a slower rate. This makes sense because, as explained in B.3, the rate of traffic on a link is constrained by the speed of the link. Since a switch queue is fed by a finite number of links (i.e. at most 11 since $k = 12$ ports), the arrival rate at a switch queue is not truly Poisson. The figure also shows our simulated results for strict flow control (periodic send process) on both the FatTree and its equivalent logical tree. In this case, the growth rate is much smaller and, with the exception of the case using VLB routing, the required queueing stays well under 100 packets.

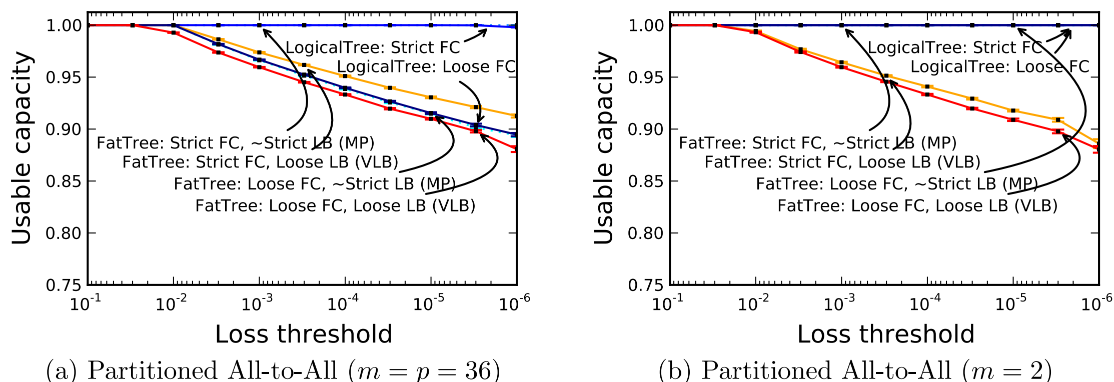
There are several key points to take away from these results.

- First, with loose flow control the arrival rates at switches appear Poisson making the benefit provided by improved load balancing largely irrelevant. This can be seen by the fact that, under loose flow control, both VLB and MP perform nearly the same as when we simulate the logical tree, where routing is not a factor. As we approach full offered load, these three curves converge and show the same scaling characteristics as the M/M/1/K models.
- Secondly, with strict flow control, the MP load balancer performs significantly better than VLB, approaching the performance of ideal routing (the Logical-Tree: Strict FC case).
- Third, even under worst-case traffic, with loose flow control we can still expect to use over 90% of the network given a reasonable amount of buffering (e.g. 100 packets worth).
- Finally, we can expect these results to scale with higher speed Ethernet links since M/M/1 queues only depend on the relative rates of arrivals and departures.

3.3.4 Partitioning the DCN into tenants

While the all-to-all traffic case is useful because it allows us to compare our results with our analytical model, it represents an extreme case. In a multi-tenant DCN, the servers will be partitioned among a number of different tenants. In this context, we

can expect that most, if not all, of a server's traffic to go to other servers belonging to the same tenant. We should not expect that it is necessary for the network to be robust to a network-wide all-to-all traffic pattern. A more reasonable worst case, therefore, would be to evaluate performance when the servers in each tenant perform an all-to-all exchange of traffic. To do this we chose to partition the network into evenly sized tenants consisting of m randomly assigned servers. Each server sends to all other servers in its partition at an average rate of $\frac{1}{m-1}$ and we call this the "all-to-all partition" pattern. Note that the permutation and all-to-all traffic patterns are effectively special cases of this pattern with $m = 2$ and $m = n$ respectively. A useful midway point between these two extremes is to chose $m = p$ where p is the number of paths. Since $n = p * k$, a network with k port switches would have k tenants.



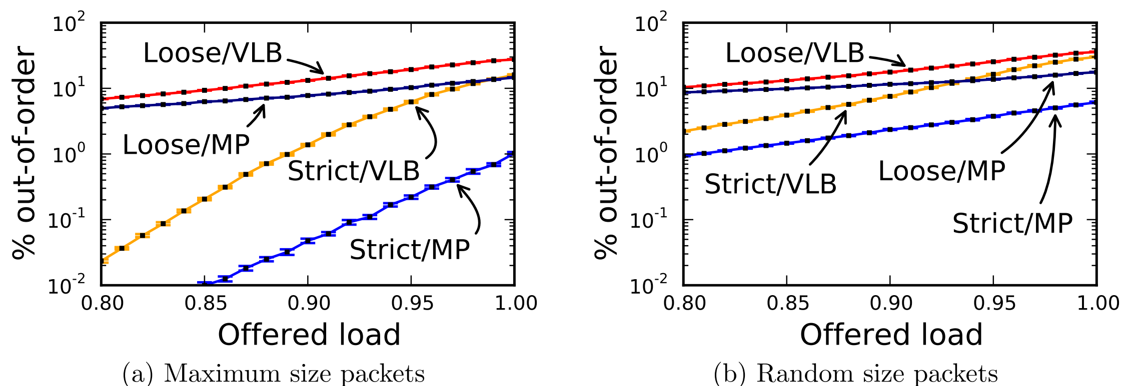
(a) Partitioned All-to-All ($m = p = 36$) (b) Partitioned All-to-All ($m = 2$)

Figure 3.13: Capacity vs loss threshold with servers partitioned into tenants of size m. $l = 3$ levels, $k = 12$ ports, loss threshold $= 10^{-3}$.

Figure 3.13 shows capacity vs threshold for the partitioned all-to-all traffic pattern with both $m = 2$ and $m = p$ below. In the $m = p$ case, there are enough flows that loose flow control drowns out the differences between loose and strict routing. When we move to strict flow control, VLB continuse to perform poorly but the MP load balancer now achieves optimal performance. With $m = 2$, a server receives traffic from only one source which means the total traffic destined for it cannot exceed the capacity of its interface even without flow control. As a result, performance under the permutation pattern depends mostly on load balancing.

3.4 Resequencing packets at end hosts

Since paths can experience uneven levels of queueing, routing packets from the same flow through different paths can easily cause them to arrive out-of-order. This is demonstrated by Figure 3.14 which shows the fraction of packets arriving out of order as a function of offered load. In this experiment the network was partitioned into tenants with each tenant's servers forming an all-to-all traffic pattern. As before, each of the lines we show represents one of the four corners of the routing/flow control space presented in section 3.3. Under strict routing and flow control, we see a minimal number of out-of-order arrivals. A significant fraction of the traffic does arrive out-of-order at the other extreme. With proper flow control, the result shows that most packets arrive in order which is consistent with the findings of [24].



(a) Maximum size packets

(b) Random size packets

Figure 3.14: Fraction of traffic arriving out-of-order.

3.4.1 Dealing with out-of-order arrivals

The effect that out-of-order arrivals have may depend on the protocol(s) used by the tenant. While an IP network provides no guarantee that packets arrive in order, it is generally treated as an implicit requirement. One of the key concerns that is typically cited is the effect out-of-order arrivals have on the performance of TCP. A TCP flow can re-order segments that arrive out-of-order but it typically interprets this as a sign of network congestion. In particular, it causes the receiver to generate duplicate ACKs which can cause the sender to react as if packets had been lost. Thus it leads to spurious retransmissions and unnecessary reductions in the sending rate. As a

result, it is generally assumed that packet-level routing cannot be used in data center networks. This assumption was recently challenged by [24]. They showed through simulation that most packets do arrive in order and that in fact the negative effect on TCP performance is out-weighed by the increase in throughput gained by packet-level routing.

In this work, we propose to address this issue head-on and resequence packets in software at the end hosts. This requires modifying the networking stack of servers to include a resequencing layer that would sit below the network layer and would ensure that all packets are reordered before delivering them to the layers above. This would shield TCP from out-of-order arrivals all together. It would also present tenants with a more robust network model upon which they can depend.

3.4.2 Design considerations

Since we are not aware of any similar work in the data center networking context, we briefly consider the high-level approaches that one might take before discussing our approach to resequencing. In order for packets to be resequenced at servers, they must be marked in some way to identify their order. This can be done in one of two ways; with sequence numbers or with timestamps. We describe the implications of each approach in the data center context below.

Sequence-based resequencing:

Using sequence numbers would mean that a server creates a sequence number counter for each server that it is sends packets to. When it sends a packet to a given destination, it would add the sequence number from the corresponding counter to the packet and then increment the counter. The receiver can then use the sequence number to determine which packet should come next and buffer any packets that arrive out-of-order. Since a missing packet may never arrive, the receiver must use a timeout to recover from loss. This is essentially the approach that TCP uses except that its sequence numbers identify the order of bytes within a flow. The key advantage of this approach over time-based resequencing is that a receiver can immediately determine

whether any packets are missing when it receives a packet. This means that packets arriving in-order can be delivered immediately and experience no added delay due to resequencing.

Per-flow state

One potential drawback of this approach is that it requires each server to maintain separate resequencing state for each server it is communicating with. At the sending side, it requires a separate sequence counter for each receiver and at the receiving side a separate queue to reorder the packets from each server sending to it. Given that data centers contain many servers which may be assigned to different tenants over time, this state would need to be managed dynamically. This may make it more difficult to implement in hardware (e.g. as a NIC feature) but can be managed in software. Since the resequencing we propose is meant to exist transparently below any network or transport layer protocol, we cannot know in advance when communication with another server begins or ends. As a consequence we would need to depend on a soft-state approach and use timeouts to remove resequencing state when servers have ceased communicating.

Sequence number agreement

This raises another practical issue which is that the sender and receiver must agree on what the initial sequence number is before they communicate. The use of timeouts means that a simple approach like assuming the initial sequence number is 0 won't work since we cant be sure whether both servers have timed out and removed their state when they resume communicating. In fact there is no way to guarantee sequence number agreement without explicit two-way communication since any flag or special packet the sender might use could be lost. For TCP, this issue is handled as part of the two-way hand shake that occurs at the start of a flow. In the resequencing layer we cannot know a priori when communication with another server begins or ends which means we cannot perform such a handshake in advance. While this is a minor issue, we would have to accept that some traffic might initially be delayed or delivered out-of-order.

No Multicast

A final drawback to this approach is that it cannot easily be extended to handle

multicast. This is because sequence numbers only have meaning between two servers. Separate state would be needed for each multicast sender in every multicast group.

Time-based resequencing:

The alternative to using sequence numbers is to mark packets with a timestamp. This approach has been used in router interconnects where the inability to support multicast and the need to maintain separate resequencing state at ports is more problematic. In the data center, this method requires that servers mark each packet with a timestamp indicating when the packet was sent. An advantage of this approach is that each server only needs to maintain one resequencing buffer since all incoming packets can be reordered based on their timestamp. The main source of difficulty is that timestamps only indicate relative order. A receiver has no way of knowing whether packets with earlier timestamps may still arrive. The only solution is to establish some **age threshold** after which packets are considered "late". To avoid out-of-order delivery, the age threshold must be large enough to accommodate the maximum delay between paths that is normally experienced.

Unnecessary delay

Thus a drawback of this approach is that the resequencer must delay every packet by the age threshold which effectively means that each packet experiences the maximum amount of network delay. To minimize this penalty, the age threshold can be made to adjust adaptively to changing network conditions [55]. However, in the data center context, servers in different subtrees have longer paths than servers in the same subtree. Since the buffer does not separate the traffic from different servers, the age threshold would need to be set to the delay over all paths leading to unnecessary delay for traffic between local servers.

Clock synchronization

The second and more practical concern is that for this approach to work, server clocks would need to be tightly synchronized. While it is not important to keep packets from two different servers in order, the packets from both servers must be buffered until they reach the age threshold and the only way the receiver can determine their age is by relying on their timestamps. This means that when they are serviced from the

same queue, the only way to keep the packets from both servers long enough is to increase the age threshold by the difference in their clocks. Since this adds directly to the overall delay, the performance can never be better than the degree to which synchronization can be achieved between servers.

3.4.3 Hybrid resequencer

We now describe the design of a new hybrid resequencer which uses both sequence numbers and time stamps. Because it uses sequence numbers, it does require that we maintain separate resequencing state per-server. However, by combining both sequence numbers and timestamps we can leverage each to overcome the limitations of the other. Resequencing state is managed dynamically at servers and we use timeouts to remove the state at both sender and receiver. We briefly describe how this hybrid resequencing scheme works below.

Receiver

A receiver creates a logically separate resequencer to manage the traffic from each server sending to it. A resequencer includes a queue and an "expected sequence number" (ESN) counter, which allows it operate based on sequence numbers and deliver packets that arrive in-order immediately. When packets arrive whose sequence numbers do not match the ESN counter, they are placed in the queue causing a timeout to be set. However, the queue orders packets based on timestamp and the timeout is set using an age threshold that adapts to the delay observed for this sender's traffic. While the queue is time based, packets are only allowed to leave the queue when their sequence number matches the ESN counter. Thus when a timeout occurs, the ESN counter is set to match the sequence number of the packet at the front of the queue. The only exception to this rule is for multicast traffic which would not use sequence numbers and would queued separately under this scheme. This policy works because ordering packets by the sender's timestamp should cause their sequence numbers to be ordered as well. Because the packet at the front of the queue contains the earliest timestamp, packets with sequence numbers between it and the current ESN value must also have exceeded the age threshold when the timeout

occurs. Whenever the ESN counter is updated, resulting from either a timeout or arriving packet, the queue is checked to see if the packet at the front has the sequence number matching the new value. If it does, the packet is released and the counter is incremented causing the check to be repeated. This ensures that any queued packets are released as soon as their sequence numbers indicate they are in-order. Likewise, when a packet moves to the front of the queue, the timeout is updated based on its timestamp and the age threshold.

Sender

The behavior at the sender is much simpler. Whenever the server has a packet to send it looks up the sequence number counter corresponding to the destination and writes the current value along with the current time to the packet and then it increments the counter. If no sequence counter exists, then one is allocated and initialized to 0. This means that when the receiver allocates a queue for this sender, it should expect the sequence numbers to start at 0 and initialize the ESN accordingly. Of course it's possible that the receiver already has a resequencing queue but that has not yet timed out. With the hybrid approach, sequence number agreement issues are resolved automatically because the receiver will begin using the correct sequence numbers once a sender's packet has reached the age threshold. The one exception to this behavior, as previously mentioned, is that multicast packets only use timestamps.

Advantages

This approach offers several benefits which can be summarized as follows.

- First, packets that arrive in-order require no buffering and can be delivered immediately without delay.
- Secondly, multicast traffic is handled automatically using the age threshold.
- Thirdly, the age threshold on each queue only needs to match the delay variation for the corresponding sender which means that packets are never buffered longer than necessary.

- Finally, clocks do not need to be synchronized since we only care about the relative difference between timestamps and arrival time for a given sender's packets.

Practical considerations:

Clock frequency

While the sender and receiver clocks do not have to be synchronized to the same time, they do need to operate at approximately the same frequency. This may be tricky if no standard clock is available. It is conceivable, however, that the receiver could be made to compensate if it can get an estimate of the difference in their clocks, e.g. by measuring the drift in the average delay it measures over some window of time.

Clock resolution

A related issue is that the clocks require enough precision to differentiate the packets. A 10 gigabit link can support close to 15 million minimum sized packets per second. This means we need access to a clock with a resolution on the order of about 50 nanoseconds or better. This should be possible on modern machines when resequencing is implemented in kernel space. Alternatively, if a standard clock with microsecond precision is available, the desired effect could also be achieved by combining the low-order bits of the sequence number with the timestamp. Since the age threshold will be much larger than one microsecond, this would allow the receiver to differentiate packets sent within the same microsecond without a significant impact on performance.

Marking packets

How packets are actually marked with timestamps and sequence numbers is another question that must be resolved. It may be possible to reuse existing protocol fields to store these values (e.g. IP options). A more general approach would be to add a shim to packets leaving the sender which is then removed by the receiver after passing through the resequencing layer.

Packet overhead

Regardless of how packets are marked, including these fields increases the packet header overhead thereby reducing the overall capacity usable by tenants. While the

hybrid resequencer does use both timestamps and sequence numbers, theses fields do not have to be very wide. To correctly resequence packets, we only need enough bits to differentiate among the maximum number of packets that can possibly be queued at once. With hybrid resequencing this depends on the age threshold which only needs to be as large as the maximum difference in delay between two servers. Consider the back-of-the-envelope calculation at 10 Gbps used above. If the age threshold were 1 ms, then the resequencer could never see more than 15,000 packets at a time which means a 14-bit sequence number would be sufficient. Similarly, for timestamps with 10 nanosecond precision, only 17 bits are needed to encode a 1 millisecond interval. Note that these numbers would be the same for a 10 millisecond age threshold at 1 Gbps Given that this is less than 32-bits combined, we could use 4 bytes to encode each field which would add negligible packet overhead and should be more than sufficient for the conditions under which scheduling would operate.

Server overhead

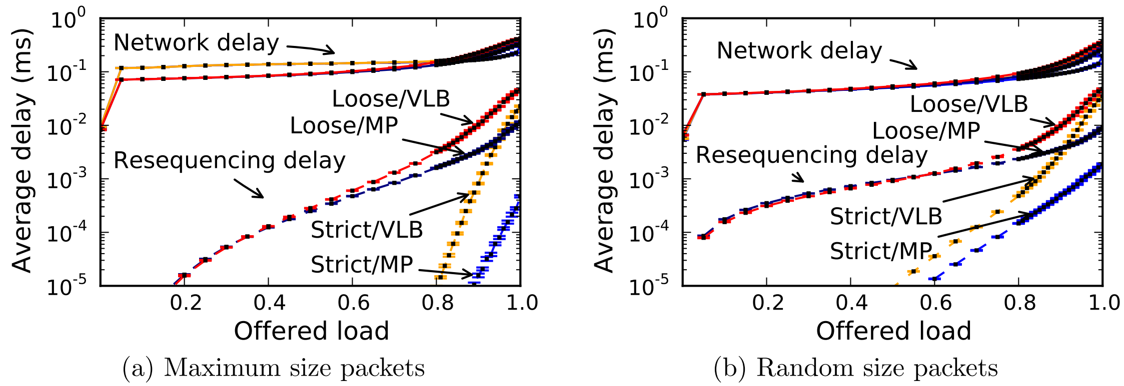
Our design was motivated by the need to minimize the overhead on a server's resources. Since incrementing and comparing sequence numbers are trivial operations, we expect the queueing of packets to be the main source of overhead at servers. Not only does buffering packets consume memory but inserting them in the correct order into queues becomes more expensive as queue sizes grow. Given the speeds supported by server NICs, buffering even a few milliseconds of traffic at line-rate could be problematic yet it is only under high-load that we should expect packets to begin arriving out-of-order. This suggest that for resequencing to be practical, the level of queueing across network paths must be fairly even.

3.4.4 Evaluation

Evaluating the overhead on servers can really only be accomplished with a full kernel-space implementation which is beyond the scope of this thesis. Since we can expect the performance to depend on the amount of queueing necessary, the evaluation is only meaningful under the traffic characteristics the resequencer would subjected to. Thus our evaluation focuses on understanding the space under which resequencing would operate. To evaluate our resequencing approach, we implemented the hybrid resequencer in the simulator and ran the switch queue size experiment shown in 3.12.

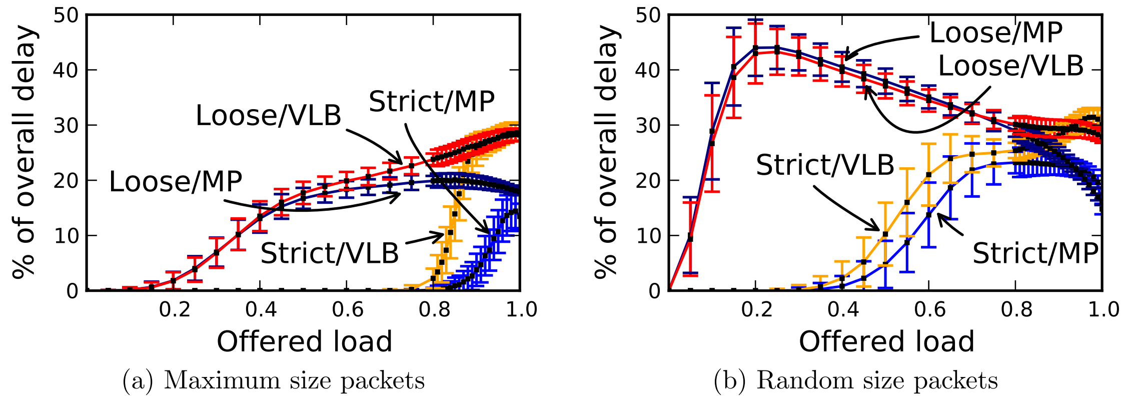
Simulator implementation

The implementation follows the description above with the caveat that we used a simple approach to set age thresholds adaptively. At each queue we record the maximum delay observed by recording the difference in the arrival time and the packet's timestamp. Whenever a packet reaches the front of the queue, we set the timeout to the packet's timestamp plus this maximum delay. In other words, the age threshold grows to the maximum delay experienced by the sender's packets. This means some packets are initially delivered out-of-order but then once the maximum delay is observed, no more packets are delivered out-of-order. During our experiments, these out-of-order deliveries mostly happen during the warmup phase so that by the time the network reaches steady-state, negligible out-of-order deliveries occur.



(a) Maximum size packets (b) Random size packets

Figure 3.15: Average delays experienced by packets.



(a) Maximum size packets (b) Random size packets

Figure 3.16: Fraction of end-to-end delay spent in resequencer (out-of-order packets).

To evaluate our hybrid resequencer we repeated the experiment shown in Figure 3.14. We measured the network delay and the delay that packets spend in the resequencer. We show the resequencing delay compared to the network delay averaged over all packets in Figure 3.15. In Figure 3.16 we show the fraction of the overall time that out-of-order packets spend in the resequencer. Since packets arriving in the proper order can be delivered immediately, we see that the average resequencing delay is at least an order of magnitude less than the average network delay. This would not be the case under a purely time-based approach since all packets would have to be delayed until the age threshold was met. The results show that of the packets that did arrive out-of-order, the time spent in the resequencer represents less than half of their overall delay. This fraction would be higher had we used only sequence numbers since timeouts would have to be set conservatively to match the maximum network delay possible. Thus when a packet is lost, any buffered packet would experience delays substantially longer than the network delay whereas under the hybrid approach, the timeout occurs as soon as the missing packet falls outside the window of delay normally observed for that particular flow.

3.5 Summary

In this chapter we explored the use of packet-level routing in multi-path data center networks, focusing specifically on the FatTree. To make efficient use of the resources in these networks, effective load balancing is critical. This is especially true when the goals include preserving agility and achieving traffic isolation among tenants. Given these objectives, tenants can only be allocated the minimum fraction of the network's capacity that is achievable under any traffic pattern where capacity is measured using the minimum throughput of any flow. We argued that flow-level routing inherently depends upon the traffic pattern used and we demonstrated that it exhibits very poor worst case performance in this context, achieving only a small fraction of the network's capacity. By contrast, we used VLB to show that packet-level routing can achieve the majority of the network's capacity.

We explored several ways to improve the performance of packet-level routing by adapting it to the data center environment. However, we found our results were very sensitive to bursty traffic. To investigate this, we presented a strict/loose model where we modeled the sending rates of servers as either periodic or Poisson and we used this to represent the range of flow control performance that we could expect in practice. We found that our multi-phase routing algorithm can achieve nearly all of the available capacity if the sending rates can be regulated to appear periodic. This result highlights the need for adequate flow control and helps motivate the scheduling framework that we introduce in Chapter 4. We also compared our simulated results against a set of queueing theory models that we developed. These were used to validate the simulations and place bounds on the worst case performance.

Finally, we investigated the issue of out-of-order arrivals that results from allowing packets within a flow to be routed separately. We found that a substantial fraction of packets could arrive out-of-order if strict flow control and routing are not achieved. To address this issue, we proposed resequencing packets in software and we introduced a novel design that leverages the benefits of using both time stamps and sequence numbers.

Chapter 4

Isolating Tenants with Distributed Scheduling

4.1 Introduction

In this chapter we introduce the distributed scheduling framework. The basic idea is similar to the distributed scheduling in router interconnects that we described in section 2.3.3 except that instead of scheduling rates between router ports, the scheduling we propose controls the rate at which servers send to one another. By explicitly coordinating the sending rate of servers, this framework offers several benefits that end-to-end approaches cannot. While it does require servers to exchange control packets, it is made scalable by the fact that the network is partitioned among tenants and that rates are assigned in a fully distributed manner. To summarize these benefits, we can view the scheduling framework as having three functions:

1. A network-wide flow control mechanism that regulates traffic and moves congestion out of the network.
2. A QoS mechanism providing isolation between tenants by enforcing the limits imposed by their virtual network abstractions.
3. A network service that can efficiently schedule the traffic between a tenant's servers.

In this chapter we will focus primarily on the first two roles and devote chapter 5 to investigate the third.

4.1.1 Objectives

The main objectives for this chapter are as follows:

- Introduce the basic concept and demonstrate that it can provide network performance isolation to tenants.
- Examine the basic tradeoffs and provide a discussion of some of the practical concerns raised.
- Show that it can be used as a flow control mechanism and evaluate the benefit it provides when combined with packet-level routing and resequencing.

4.2 Scheduling Framework

We begin by describing the basic scheduling framework.

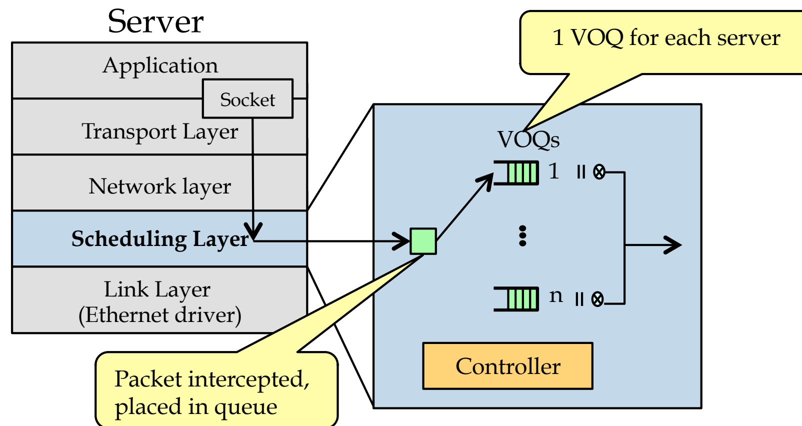


Figure 4.1: Conceptual view of scheduling as layer a implemented in the networking stack of servers.

4.2.1 Scheduling layer

As shown in figure 4.1, the networking stack of servers is augmented with a scheduling layer that exists below the network layer. This layer is assumed to exist outside of

the tenant's control and is meant to operate transparently to the layers above. Note that this is simply a conceptual view that ignores the use of virtual machines, which we discuss in section 4.5.2.

Virtual Output Queues

The scheduling layer intercepts outgoing packets produced by the server and directs them into separate *Virtual Output Queues* (VOQs) corresponding to their destination. By controlling the rate at which traffic may leave each VOQ, the scheduling layer can effectively control the rate at which servers send to one another. The rate assigned to a VOQ will depend, at least in part, on its *backlog*, that is the number of bytes that it buffers.

Scheduling Controller

The scheduling layer also includes a logical controller which is responsible for managing the VOQs and assigning them rates. VOQs require a minimum amount of state and can be allocated dynamically and removed with a timeout after a period of inactivity. In order to assign rates on VOQs, the controller runs a *distributed scheduling algorithm* which requires that it periodically exchange control messages with the scheduling controllers at the tenant's other servers. The details of this exchange and the precise manner in which rates are assigned is dependent on the scheduling algorithm used. While there are a variety of ways in which this could work, we describe the approach that we have taken in section 4.3 when we detail our distributed scheduling algorithm. Since the scheduling of rates adds overhead, it can only occur periodically and we call the interval at which rates are assigned the *scheduling interval*.

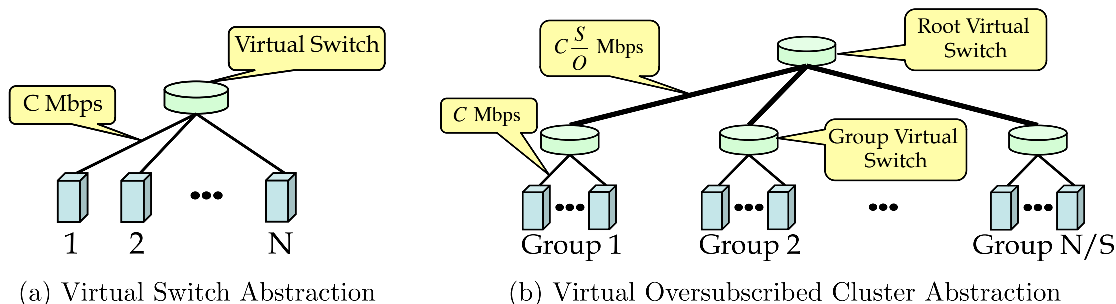
Regulating rates

An important aspect of the scheduling framework is its ability to act as a flow control mechanism that regulates the traffic produced by servers. In section 3.3.3 we showed that the performance of load balancing and resequencing is heavily dependent on the time scale at which rates are controlled. In particular we found that the best performance can only be achieved when the sending process at servers appears uniform. In interconnection networks this is referred to as a regulated flow and is necessary to control bursty traffic which is known to have a significant impact on the ability of the network to provide QoS to different classes of flows [22]. To regulate server traffic,

the scheduling framework should ensure that the departure times of packets at VOQs match the rates they are assigned as closely as possible.

4.2.2 Tenant virtual networks

As discussed in Chapter 2, we assume that the data center network has been partitioned among tenants so that tenants are assigned to different servers and have been given virtual network abstractions that provide them with guarantees on the bandwidth between their servers. It is worth reiterating that in this work we are not concerned with the specific abstractions offered to tenants or the allocation of virtual networks to tenants. Rather, we assume that the physical capacity exists to allow all tenants to use the full capacity of their virtual networks provided that each tenant stays within the limits defined by their abstraction. Here we will describe how the scheduling framework can be used as a mechanism to enforce these limits.



(a) Virtual Switch Abstraction (b) Virtual Oversubscribed Cluster Abstraction

Figure 4.2: Two different tenant virtual network abstractions.

Example 1: Virtual Switch

Figure 4.2 shows two examples of the types of virtual network abstractions that tenants may be given. The virtual switch (VS) abstraction, which has appeared in several recent papers [26] [14], represents the simplest form of virtual network. A tenant with this abstraction is given the illusion that each of its servers is connected to the same virtual switch with some capacity C. Note that while the value of C is the same across all of its servers, different tenants could be given VS's with different capacities depending on their needs and the available resources in the underlying physical network. The results from section 3.3.4 suggest that given a FatTree with full bisection bandwidth, it may be possible to give every tenant a VS with a capacity

near the full capacity of the physical interfaces of their servers as long as the traffic produced by servers can be regulated sufficiently well.

Example 2: Virtual Oversubscribed Cluster
It may not always be practical to provision the data center with full bisection bandwidth. For such cases, the Virtual Oversubscribed Cluster (VOC) abstraction provides a virtual network that more closely represents the limits of the underlying physical network. Figure 4.2b shows this abstraction as it was first presented in [14] (with the exception that we use C to represent the link capacity). A tenant with a VOC would be allocated a set of servers divided into evenly sized groups of size S. Within each group, servers would still have the illusion of sharing a virtual switch of capacity C but these "group switches" would be connected to a root virtual switch whose links are oversubscribed by an oversubscription factor O. This means that while servers within a group can communicate at a rate C, they must share a link with capacity $\frac{S*C}{O}$ when communicating with servers in other groups.

General virtual network topologies

While these are just two examples, it should be possible to support any arbitrary virtual network abstraction provided the topology forms a tree. To do so, we simply need to translate the capacities on the links in the virtual topology into limits on the aggregate sending rates of the servers that can send across them. In other words, we can express the virtual topology in terms of constraints on the rates that we assign to VOQs.

4.2.3 Constraints on rates

Before we can formulate these constraints, we need a few definitions:

Let $f_{i,j}$ represent the *flow* of traffic from server i to j.
Let $b_{i,j}$ denote the *backlog* server i has in its VOQ for server j.
Let $r_{i,j}$ represent the *rate* assigned to the VOQ.
Let F be the set of all server-to-server flows $f_{i,j} \in F$.

Let B be the set of all backlogs in VOQs, $b_{i,j} \in B$ corresponding to the flows in F.
Let R represent the set of all rates assigned, $r_{i,j} \in R$, which we call a *rate assignment*.
Let T represent the *scheduling interval*.
Note: we assume that rates (and link capacities) are expressed in terms of units of backlog per scheduling interval.
This means that $r_{i,j} = b_{i,j}$ represents the rate needed to clear the backlog of the VOQ corresponding to flow $f_{i,j}$ in one scheduling interval.

The network as a directed graph

When we assign a rate $r_{i,j}$, the corresponding flow $f_{i,j}$ consumes bandwidth on all of the links along the path from server i to j. However, notice that links in the virtual network topologies support full capacity in both directions (i.e. they are full duplex). Since the rates that we assign to VOQs consume bandwidth in one direction, we need to treat each of these bidirectional links as two logically separate links when assigning rates. Therefore we represent the virtual network with a directed graph $D = (V_D, E_D)$ where each link in the topology is represented by two edges $uv \in E_D$ and $vu \in E_D$. Despite the potential for confusion, we will continue to refer to these edges as a *links* and use the notation l rather than uv. Keep in mind, however, that flows represent traffic in one direction which means that if a flow $f_{i,j}$ exists on $l = uv$, it cannot exist on $l^{-1} = vu$.

We use the following notation to describe a link:
Let $l = uv$ be a link in the graph $D = (V_D, E_D)$, where $uv \in E_D$.
Let c_l represent the link's capacity.
Let $P_{i,j}$ represent the directed path from server i to server j.
Let F_l be the set of flows on link l, i.e., $F_l \subseteq F$ where $F_l = \{f_{i,j} \in F \mid l \in P_{i,j}\}$.
Let B_l represent the set of backlogs corresponding to the flows in F_l.

Feasible rate assignment:

We say a rate assignment is *feasible* if the rates do not violate any of the constraints on any of the links.
Thus given a graph $D = (V_D, E_D)$, a rate assignment is feasible if:

$$\sum_{f_{i,j} \in F_l} r_{i,j} \leq c_l, \forall l \in E_D \tag{4.1}$$

4.2.4 Assigning rates on VOQs

Given the constraints that must be enforced, we can now consider how rates should be assigned to VOQs.

Regulating tenant traffic

We begin by considering what happens when a server first begins sending to a destination. As its packets arrive in the scheduling layer, a new VOQ is allocated for the destination and the packets are placed in the VOQ until a rate is assigned. Assuming there is sufficient capacity available on all of the links on the flow's path, the VOQ will simply receive the rate needed to clear its backlog over the next scheduling interval (i.e. $r_{i,j} = b_{i,j}$). This means that at the end of the scheduling interval, all of the packets that were in the VOQ when the interval began will have been sent and those packets that remain in the VOQ correspond to those packets that arrived after the interval began. Thus a growing buffer corresponds to an increase in the rate of arriving traffic and would result in the VOQ receiving a larger rate, provided that the capacity constraints in 4.1 are not violated. In this way, the rates assigned to VOQs naturally reflect the rate of traffic produced at servers as long as these rates form a feasible rate assignment.

Moving congestion out of the network

We can say that the traffic pattern is inadmissible when the total backlog that must be transferred across one or more links exceeds its capacity, that is $\sum_{\forall b_{i,j} \in B_l} > c_l$ for some $l \in E_D$. This happens when the rate of traffic being produced by the tenant exceeds the capacity on some link in its virtual topology. This will cause the VOQs corresponding to the flows on the overloaded link to grow since their is not enough capacity to assign them the rates needed to clear their backlogs. If the pattern is sustained, the VOQs will overflow and packets will be lost. Without the scheduling layer, this buffering and packet loss would occur at the switch queue feeding the link. By enforcing a feasible rate assignment, however, the scheduling framework ensures

this congestion occurs in the VOQs instead. In effect, the scheduling framework moves the congestion that would normally occur in the network to the edge where it only affects the flows that are responsible for creating it.

4.2.5 Assigning rates on bottleneck links

We now consider how scheduling should assign rates in the face of inadmissible traffic. For now, we will assume that the rate of traffic arriving in VOQs does not depend on the rates assigned to VOQs on the time scale at which scheduling occurs. Of course, the traffic produced by protocols such as TCP, which exist above the scheduling layer, will depend on the rate of traffic actually delivered but we will defer this discussion until section 4.5.1. With inadmissible traffic, we must decide how to divide the limited capacity on one or more bottleneck links. Given that the tenant has no way to express the priority among its flows, how should rates be assigned? Without making assumptions about the behavior of the application or protocols above the scheduling layer, there is no clear way to answer this question. Here we will propose two different ways in which rates could be assigned and in chapter 5 we will explore whether there may be advantages to one approach over the other.

Max-min fair share

The first approach is to assign rates on the link in max-min fair fashion. The simplest way to describe max-min fairness is that all flows not bottlenecked elsewhere in the network receive an equal share on a link. If a flow cannot use its full share on a link, the unused bandwidth will be distributed evenly among those flows that can use more. Under ideal conditions, TCP flows converge to their max-min fair share of bottleneck links and given that many applications use TCP, it is not unreasonable to assume that a tenant will expect max-min fairness among its flows.

Before we can show how to calculate a max-min fair rate assignment, we first show how to divide the capacity of a single link in max-min fair fashion. Because flows may be bottlenecked by the bandwidth they receive on other links, we use the term “request” to describe the rate that a flow can use on a link.

- Let $q_{i,j,l}$ denote the requested rate for flow $f_{i,j}$ on link l.
- Let Q_l be the set of all requested rates on link l.
- Let $s_{i,j,l}$ be the share given to flow $f_{i,j}$ on link l.
- Let S_l be the set of all shares assigned on link l.

Since a flow only needs the rate needed to clear its backlog, if it is not bottlenecked elsewhere in the network then we can assume that its request simply matches its backlog, i.e., $q_{i,j,l} = b_{i,j}$.

Max-min rate assignment

Algorithm 1 Assign max-min fair share on link

```
 1: procedure MaxMin(Q_l, c_l)
 2:     S_l ← {}                                  ▷ The set of assigned shares
 3:     u_l ← c_l                                 ▷ Initialize the unused capacity
 4:     while Q_l ≠ {} do                         ▷ Terminate when no requests remain
 5:         q_{i,j,l} ← min(q_{i,j,l} ∈ Q_l)      ▷ Find the minimum request
 6:         s_{i,j,l} ← min(q_{i,j,l}, (1/|Q_l|) u_l)
 7:         u_l ← u_l − s_{i,j,l}
 8:         Q_l ← Q_l − {q_{i,j,l}}               ▷ Remove the request from consideration
 9:         S_l ← S_l ∪ {s_{i,j,l}}
10:     return S_l
```

The procedure that we present above for computing max-min shares on a link is based on one presented in [22]. It guarantees that at every iteration, any flow that remains will receive at least $\min(q_{i,j,l}, \frac{1}{|Q_l|}u_l)$. That is a flow either receives the rate it requests or it receives an equal share of the the remaining capacity.

We follow the same principle to compute the max-min share assigned for all flows in the network, i.e., the max-min rate assignment. Just as the procedure above assigns to the smallest request first, the algorithm below assigns the rates in order from smallest to largest share. At every iteration, a flow is guaranteed to receive either its requested rate or the minimum share along its path. Note that while this algorithm is our own, we do not claim that it is original.

Algorithm 2 Assign max-min rates

```
1: procedure MaxMinRates(D = (V_D, E_D), B)
2:     R ← {}
3:     Q ← {q_{i,j} = b_{i,j} : b_{i,j} ∈ B}
4:     for all l ∈ E_D do                          ▷ Initialize the state of all links
5:         u_l ← c_l
6:         Q_l ← {q_{i,j,l} = b_{i,j} : f_{i,j} ∈ F_l}
7:     while Q ≠ {} do
8:         S ← {}
9:         for all q_{i,j} ∈ Q do
10:            s_{i,j} ← min(q_{i,j}, (1/|Q_l|) u_l : l ∈ P_{i,j})    ▷ Minimum share along the path
11:            S ← S ∪ {s_{i,j}}
12:        s_{i,j} = min(s_{i,j} ∈ S)                ▷ Find the smallest share assigned
13:        for all l ∈ P_{i,j} do             ▷ Remove the flow from consideration on links
14:            u_l ← u_l − s_{i,j}
15:            Q_l ← Q_l − {q_{i,j,l}}
16:        r_{i,j} ← s_{i,j}
17:        R ← R ∪ {r_{i,j}}
18:        Q ← Q − {q_{i,j}}
19:    return R
```

Backlog-proportional share

While max-min is desirable in many contexts because it provides fairness among flows, the traffic that we schedule belongs to a single tenant. For this reason it may be sensible to assign rates to flows in proportion to their backlog instead. Conceptually, the process of assigning *backlog-proportional* rates is similar to that of max-min. At every iteration, any remaining flow $f_{i,j}$ will be receive at least $min(q_{i,j,l}, \frac{b_{i,j}}{b_l}u_l)$ where b_l represents the sum of the backlogs of the remaining flows. The procedure we present for computing backlog-proportional shares on a link is slightly different, however.

This procedure first finds those flows whose backlog-proportional share exceeds their request. These flows are added to the set of "unbottlenecked" flows U, and are simply assigned the share they request. The while loop exits when no such flows remain at which point all remaining flows belong to the set L. The set L represents the flows that are "bottlenecked" by the backlog-proportional share they receive at this link. Since the flows in U cannot use their share of the link, they must be removed from

1: **procedure** BKLGPROP(Q_l, B_l, c_l)
2: $S_l \leftarrow \{\}$
3: $u_l \leftarrow c_l$
4: $b_l \leftarrow \sum_{\forall b_{i,j} \in B_l} b_{i,j}$ ▷ Compute the backlog of all flows on l
5: $L \leftarrow \{\}$ ▷ Locally bottlenecked flows
6: $U \leftarrow \{\}$ ▷ Unbottlenecked flows (bottlenecked upstream)
7: **while** $L \neq F_l - U$ **do**
8: $L = F_l - U$
9: **for all** $f_{i,j} \in L$ **do**
10: $s_{i,j,l} \leftarrow \frac{b_{i,j}}{b_l} u_l$
11: **if** $q_{i,j,l} < s_{i,j,l}$ **then**
12: $s_{i,j,l} \leftarrow q_{i,j,l}$
13: $S_l \leftarrow S_l \cup \{s_{i,j,l}\}$
14: $U \leftarrow U \cup \{f_{i,j}\}$
15: $b_l \leftarrow b_l - s_{i,j,l}$
16: $u_l \leftarrow u_l - s_{i,j,l}$
17: **for all** $f_{i,j} \in L$ **do**
18: $S_l \leftarrow S_l \cup \{s_{i,j,l}\}$

consideration before we can determine the final shares that we assign to the flows in L. Note that if we set all backlogs equal to 1, i.e., $b_{i,j} \leftarrow 1, \forall b_{i,j} \in B_l$ then the procedure above will produce the max-min shares on the link since the flows in L always have equal backlogs and therefore an equal share of the remaining capacity. This observation will come in handy in section 4.3 when we present our distributed algorithm for solving max-min and backlog-proportional rates.

Backlog-proportional rate assignment

The centralized algorithm we present for computing backlog-proportional rates differs slightly from the algorithm used for max-min. At each iteration, the algorithm below finds the link with the smallest ratio of remaining capacity to backlog and assigns rates using the shares that it computes. The intuition behind this choice is that the share computed by the link for any of its remaining flows must be the smallest share along the flow's path.

Algorithm 3 Assign backlog-proportional rates

```
1: procedure BACKLOG-PROPORTIONAL(D = (V_D, E_D), B)
2:     R ← {}
3:     for all l ∈ E_D do                          ▷ Initialize the state of all links
4:         u_l ← c_l
5:         Q_l ← {q_{i,j,l} = b_{i,j} : f_{i,j} ∈ F_l}
6:         B_l ← {b_{i,j} : f_{i,j} ∈ F_l}
7:         b_l ← Σ_{∀b_{i,j} ∈ B_l} b_{i,j}
8:     while E_D ≠ {} do
9:         l ← l ∈ E_D, where u_l / b_l is minimum
10:        S_l ← BklgProp(B)                       ▷ Link is the bottleneck for all its flows
11:        for all s_{i,j,l} ∈ S_l do
12:            r_{i,j} ← s_{i,j,l}                 ▷ Assign rates using the shares it computes
13:            R ← R ∪ {r_{i,j}}
14:            for all o ∈ P_{i,j}, o ≠ l do       ▷ Remove flow from other links on path
15:                Q_o ← Q_o − {q_{i,j,o}}
16:                B_o ← B_o − {b_{i,j,o}}
17:                b_o ← b_o − b_{i,j}
18:                u_o ← u_o − r_{i,j}
19:        E_D ← E_D − {l}                         ▷ Remove link from consideration
20:    return R
```

Since a link is removed from consideration after each iteration, the algorithm must terminate after no than $|E_D|$ iterations. To prove the correctness of the algorithm, we claim that at every iteration, the rate assigned to a flow represents the minimum of its backlog proportional share on all links. This claim can be proven by contradiction. Suppose that in some iteration link l was chosen but a rate $r_{i,j}$ was assigned which is greater than the flow's backlog proportional share on some other link u. That is, $s_{i,j,l} > s_{i,j,u}$, which by their definitions is equivalent to $min(q_{i,j,l}, \frac{b_{i,j}}{b_l}u_l) > min(q_{i,j,u}, \frac{b_{i,j}}{b_u}u_u)$. Because the algorithm does not modify the requests once they are initialized, we know $q_{i,j,u} = q_{i,j,l} = b_{i,j}$ and since $b_{i,j}$ is constant, we are left with $\frac{u_l}{b_l} > \frac{u_u}{b_l}$ which contradicts the selection made by the algorithm since l was the link with minimum ratio $\frac{u_l}{b_l}$.

4.3 Distributed Algorithm

We now present our method for computing max-min and backlog-proportional rates in a distributed asynchronous fashion. While the distributed algorithm that we present uses the procedure for computing backlog-proportional shares, we will show that it can be used to produce both backlog-proportional and max-min rate assignments.

4.3.1 Link proxies

The basic idea is to delegate servers to act as proxies for managing the rates on each edge in the graph. Note that with the virtual switch abstraction, each server can naturally act as the proxy for its own link. Keep in mind, however, that we use two logically separate proxies to manage each of the bidirectional links in the tenant's topology. Before a server can send to another server under this scheme, it must be assigned a rate by the proxy for each link along the path to the destination. Each proxy computes the share assigned to a flow under the assumption that it is only constrained by its request. While these shares may be inconsistent initially, we can ensure that all proxies arrive at the same share for a flow by reducing its request to match the minimum share it receives along the path. This is accomplished by the exchange of control packets which is described below.

Control packets

As long as a server has backlog in one of its VOQs, it must periodically generate a *request packet* containing the backlog and requested rate for the VOQ. This packet is routed to the proxy corresponding to each link along the path to the destination. If a proxy computes a share that is less than the requested rate, it assumes that it is the bottleneck link for the flow and reduces the request field to the computed share before sending the packet on to the proxy for the next link on the path. When the packet arrives at the destination, the request field contains the flow's minimum share as currently reported by the proxies along the path. The destination server then writes this value to the "share" field of a "response packet" that is then processed by the same set of proxies in reverse order as it propagates back to the sender. The response packet also includes a separate "rate" field which contains the actual rate the sending server may assign to its VOQ. As will see shortly, separating the rates from the shares computed by proxies helps ensure that the rates assigned to VOQs always represent a feasible rate assignment.

Algorithm 4 Server

1: **function** GENERATE_REQUEST(flow $f_{i,j}$)
2:     requestPacket reqPkt($f_{i,j}$)
3:     **if** MAX_MIN **then**
4:         reqPkt.$backlog = 1$
5:     **else if** BKLG_PROP **then**
6:         reqPkt.$backlog = b_{i,j}$
7:     reqPkt.$request = b_{i,j}$
8:     **return** *reqPkt*
9: **function** HANDLE_REQUEST(flow $f_{i,j}$, requestPacket reqPkt)
10:     responsePacket rspPkt($f_{i,j}$)
11:     rspPkt.$share =$ reqPkt.$request$
12:     rspPkt.$rate =$ reqPkt.$request$
13:     **delete** reqPkt
14:     **return** rspPkt
15: **function** HANDLE_RESPONSE(flow $f_{i,j}$, responsePacket *rspPkt*)
16:     $r_{i,j} = rspPkt.rate$
17:     **delete** *reqPkt*

Server behavior

While this entire scheme is implemented at the controller in the scheduling layer, the algorithm separates the role of the servers from that of the proxies. The role of the servers is summarized in Figure 4. The *generate_request*() method is called once every scheduling interval per VOQ. Note that the algorithm is fully asynchronous and does not require VOQs to operate on the same clock. When the request packet is generated, the request field corresponds to the backlog in the VOQ so that it only receives the rate it needs to clear the backlog. To use max-min rates, the backlog field is set to 1 so that all flows have the same backlog on all links. Note that the value of the backlog does *not* include the size of control packets because these are sent as soon as they are generated.

Proxy behavior

The behavior of the proxy is shown in Figure 5. If a request arrives for an unknown flow, the proxy creates a new entry for the flow which consists of its backlog, requested rate, assigned share, and the actual rate it is allowed to send at. When the proxy processes the request packet, it records the flow's backlog and and uses the request field to compute a new share for the flow. Since the share it computes must be less than or equal to the packet's request, it overwrites the request field with its share. Notice, however, that the proxy does not change the value of the request or share that it records for the flow until it receives the flow's response packet. There are two reasons for this approach. First it allows us to mimic the central algorithm by determining the minimum share a flow receives on the path before the state is changed on any of the links. Secondly, it makes the processing of a flow's request and response at a proxy appear as an atomic operation which helps ensure that the algorithm converges regardless of the order in which control packets arrive at proxies.

Decoupling the sending rates

While the minimum share ultimately represents the rate that we assign to a flow, the rate field allows us to effectively decouple the rates assigned to VOQs from the

Algorithm 5 Proxy

1: **procedure** ADD_NEW_FLOW(flow $f_{i,j}$)
2: $\quad F_l \leftarrow F_l \cup \{f_{i,j}\}$
3: $\quad B_l \leftarrow B_l \cup \{b_{i,j} = 0\}$
4: $\quad Q_l \leftarrow Q_l \cup \{q_{i,j_l} = 0\}$
5: $\quad S_l \leftarrow S_l \cup \{s_{i,j,l} = 0\}$
6: $\quad R_l \leftarrow R_l \cup \{r_{i,j,l} = 0\}$
7: **procedure** HANDLE_REQUEST(flow $f_{i,j}$, requestPacket reqPkt)
8: $\quad$ **if** $f_{i,j} \notin F$ **then**
9: $\quad\quad$ $add_new_flow(f_{i,j})$
10: $\quad b_{i,j} \leftarrow$ reqPkt.$backlog$
11: $\quad \hat{Q}_l \leftarrow \{q_{i,j,l} : \forall q_{i,j,l} \in Q_l\}$ ▷ Create a copy of the recorded requests
12: $\quad \hat{q_{i,j,l}} \leftarrow$ reqPkt.$request$
13: $\quad \hat{S}_l \leftarrow$ BKLGPROP$(\hat{Q}_l,\ B_l)$ ▷ Compute the shares the proxy wants to assign
14: $\quad$ reqPkt.$request \leftarrow \hat{s_{i,j,l}}$
15: $\quad$ **return** pkt
16: **procedure** HANDLE_RESPONSE(flow $f_{i,j}$, responsePacket rspPkt)
17: $\quad q_{i,j,l} \leftarrow$ rspPkt.$share$
18: $\quad s_{i,j,l} \leftarrow$ rspPkt.$share$ ▷ Record the new request/share here
19: $\quad r_{i,j,l} \leftarrow assign_rate(f_{i,j},$ rspPkt$)$
20: $\quad$ rspPkt.$rate \leftarrow r_{i,j,l}$
21: $\quad$ **return** rspPkt
22: **function** ASSIGN_RATE(flow $f_{i,j}$, responsePacket rspPkt)
23: $\quad in_use \leftarrow \sum_{\forall r_{u,v,l} \in R} r_{u,v,l}$
24: $\quad available \leftarrow c_l - (in_use - r_{i,j,l})$
25: $\quad$ **return** $\min(available,$ rspPkt.$rate)$

shares being computed at proxies. This is necessary because the sum of the shares assigned at a proxy can temporarily exceed its capacity. For instance, imagine that the proxy has already assigned all of its capacity to flows when the request from a new flow arrives. The new flow will still receive a share of the link but the proxy will not be able to notify the other flows that their shares have been reduced until all of their subsequent control packets have been processed. To avoid creating congestion until this happens, a flow may need to be assigned a rate that is temporarily less than its share. The $assign_rate()$ procedure (line 22) ensures that the total rate assigned never exceeds the link's capacity regardless of what shares are assigned. By summing up the recorded rates, it calculates the rate currently "in use" by the other flows. If there is not enough capacity "available" to allow the flow's rate to match its share, the flow may have to wait until the next scheduling interval at which point the rate recorded for other flows will have been reduced.

4.3.2 Convergence to centralized rates:

The distributed algorithm will converge to the correct rates in steady state. By this we mean that if the backlogs are static over an extended period of time, the algorithm must converge to the rates produced by the centralized algorithm in both the max-min and backlog-proportional cases. The intuition is that if we set the request of every flow to match its rate, as computed by either central algorithm, then the share computed by every proxy for any given flow must match the flow's request. This is true because for both backlog-proportional and max-min rate assignments, the rate assigned to a flow is always at least as large as its share across all of the links on its path.

While we will not present a formal proof, we provide a sketch of the argument that shows why the algorithm will converge. Here we focus on the shares recorded by proxies and subsequently show that the rates must converge to the shares that they assign. We will define a *round* to be the time at which a request and its corresponding response packet have been processed by every proxy for every flow. Since we have assumed the backlogs are static, the backlog recorded by all proxies will not change after the first round. Now consider the link l chosen in the first iteration of the backlog proportional algorithm. The proxy corresponding to this link must compute

the smallest shares for all of its flows in the second round since for any flow, this proxy has the minimum ratio $\frac{c_l}{b_l}$. Since the share it computes for any flow must be the minimum share computed along the path, the request recorded for this flow at all proxies must match the share it has computed after the second round. This means that these requests can never increase after the second round since the request for a flow is only set after every proxy has computed its share. As a result, these flows have been effectively removed from consideration by all proxies along their paths. Now consider the rate assigned in the first iteration of the max-min algorithm. If this rate does not match the flow's request, then it must be bottlenecked by the share it received on some link in the centralized algorithm. The proxy that corresponds to this link in the distributed algorithm must also be the proxy that computes the smallest share when the backlogs of all flows at proxies are the same. This must be true since the number of flows at proxies must match the number of flows on links in the central algorithm in the first iteration. Using the same argument as before, this means the flow is effectively removed from consideration after the second iteration.

Rates converge to shares:
Assuming that the shares computed by the proxies converge to the rates produced by the central algorithm, it is easy to show that the rates assigned by the proxies also converge. Notice that a proxy can never assign a rate to a flow that is larger than its share. Thus, after the round in which shares converge to their final values, the rates recorded at all proxies must be less than or equal to the shares they have recorded. Since the shares can no longer change, it cannot be the case that in the following round a proxy assigns a rate to a flow that is less than its share.

4.3.3 Accounting for control-overhead

Since control packets are not placed in VOQs, this control traffic is not captured by the algorithm described above. This was done deliberately so that the rates assigned by proxies represent the actual traffic produced by the tenant and to allow control packets to be sent immediately. Of course, we must account for the extra bandwidth consumed by control packets if we are to avoid congestion. A simple way to accomplish this is to have the proxy subtract the control overhead from the capacity it can assign to flows. One difficulty that arises, however, is that the traffic in each direction on a

link is managed by a separate proxy. As we described earlier, a flow only exists at one of the proxies since it can only send in one direction which means that proxies can only account for the bandwidth consumed by request packets. Even though it may make sense for both proxies to be managed by the same server, it would be unfortunate to require coupling their state simply to account for the bandwidth of response packets. To solve this problem, we assume that as long as two servers communicate, each server maintains a VOQ for the other even when traffic is sent primarily in one direction. This is reasonable because in practice communication is rarely one way as protocols like TCP require that receivers send back some form of acknowledgement. This allows us to sidestep the issue because it means that control traffic is always the same in both directions.

Thus if C represents the link's capacity, we account for the control traffic at proxies by modifying the $add_new_flow()$ method as follows:

$$c_l = C - 2\,|F_l|\,\frac{M}{T} \tag{4.2}$$

Since the number of flows will be constant in steady state, c_l will be static which means the correctness of the algorithm is preserved. By effectively reducing the links capacity, the proxy reacts immediately to the presence of a new flow. Of course, it may require a full round before the shares that it has recorded for flows reflect this reduced capacity. It is important point out, however, the total rate assigned will respond much more quickly since the $assign_rate()$ method will effectively "steal" the bandwidth needed to support the control traffic for the new flow from the first flow that already has a rate assigned on the link.

4.3.4 Related work

Our algorithm turns out to be similar in many respects to a distributed max-min algorithm by Charny et al. [18]. They also require servers to receive explicitly assigned rates on links by having them periodically send a control packet for each of their flows containing the desired sending rate in a field that they call the "stamped rate". As with our approach, rates are computed separately on each link and the stamped rate is reduced so that when the packet arrives at the destination it contains the minimum

share computed along the path. However, their approach relies on the switches to actually process the packets and calculate rates on links which is less practical in our context. In addition, our approach differs from their work in a number of other key ways. First, they focus on max-min and use a different method to assign rates that requires less state to be maintained at switches. Our algorithm, by contrast, also uses the backlog in VOQs and can compute either backlog-proportional or max-min fair rates. Secondly, they use a single control packet that contains a bit that indicates whether or not the flow was bottlenecked along the path. If the bit is set, the sending server must set the stamped rate of its subsequent control packet to the bottleneck rate in the packet. Under our approach this is accomplished by the use of a separate response packet which contains both the minimum share computed along the path as well as the rate at which it is currently safe to send. This leads to the final difference between our approaches which is the manner in which a feasible rate assignment is maintained. They also note that to avoid congestion while the algorithm converges, servers cannot immediately begin sending at the rates they are assigned. As we described earlier, we use a separate share and rate field to explicitly decouple the sending rates from the rates computed by the algorithm. The approach they chose calls for servers to comply immediately when their sending rates are reduced but requires them to wait several round trip times before adjusting to an increased rate.

4.4 Evaluation

To evaluate our approach, we implemented the scheduling framework and the distributed algorithm as described in this chapter into our simulator.

The main goals for our evaluation are as follows:

- Show that scheduling can provide isolation, even in the face of malicious tenants.
- Understanding the basic tradeoffs with our distributed approach to assigning rates.
- Evaluate how well scheduling can be used in combination with packet-level load balancing.

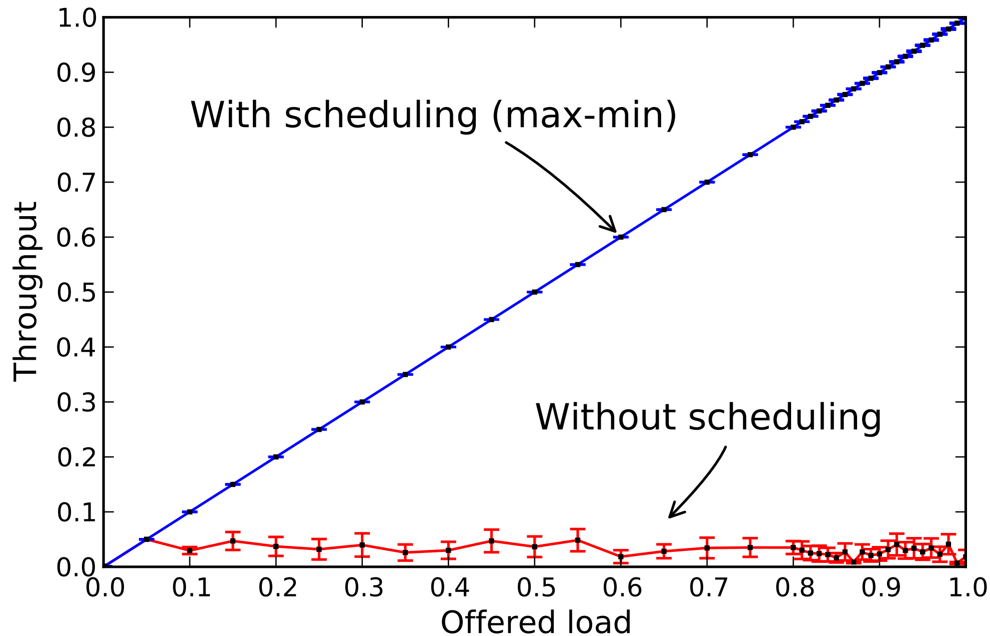


Figure 4.3: Effect of malicious traffic on the throughput of a "victim" tenant's flow.

Given these goals we focus on the virtual switch abstraction because it is the simplest to understand and evaluate.

4.4.1 Isolation

To demonstrate that scheduling can provide isolation, we need to show that a tenant cannot affect the bandwidth guarantees provided to another tenant. In order to do this, we reproduced the tree saturation scenario described in section 2.1.1. For this experiment we used a 3-level FatTree consisting of 8-port switches and 128 servers and partitioned the network into two tenants, each consisting of 64 servers assigned at random. We designated one tenant to act as a malicious tenant and the other tenant to be the "victim" and we measured the throughput of the victim tenant's traffic. The malicious tenant creates an all-to-one traffic pattern where it picks one of its servers at random and directs the traffic from all of its other servers to overload this "target" server. By exceeding the capacity of the target server, the malicious tenant saturates the links on the path to the target. Since the victim may have servers that share these links, it will see its bandwidth to these servers reduced.

The graph 4.3 shows the throughput of the victim as we increase the sending rate (offered load) of both tenants. Here we had the victim produce an all-to-all traffic pattern and we show the minimum throughput measured across all of its flows with and without the scheduling layer. Since the victim's traffic is admissible, it does not matter whether we use max-min or backlog proportional rates. We set the algorithm to produce max-min rates but we ran the experiment with backlog-proportional rates to verify that the results were the same. We see that without scheduling the malicious tenant can begin to affect the victim's throughput at around 5% offered load and can effectively starve one or more of the victim flows at higher load. As expected, the victim experiences virtually no disturbance when scheduling is used because the scheduling layer in the malicious tenant's servers prevents the total rate assigned to the target server from exceeding its capacity. In this experiment we did not distinguish between control and data packets when reporting the throughput of the tenant's traffic.

4.4.2 Distributed approach

Control overhead

In section 4.3.3 we described how the proxies account for control overhead by assuming that request and response packets are sent at a fixed interval. Each server can be the proxy for its own link which means that with the virtual switch abstraction control packets can be sent directly between the source and destination of a flow and do not have to be routed through proxies at intermediate servers. As a result, the control overhead at a server scales linearly with the number of servers that it is actively communicating with. This overhead can be quantified by $2\,|F_l|\,\frac{M}{T}$ where M is the size of a control packet, T is the scheduling interval, and $|F_l|$ is the number of flows on the link. With a scheduling interval of $T = 1$ ms, each flow generates around 1.34 Mbps which is a little over a tenth of a percent of a gigabit link.

Scalability

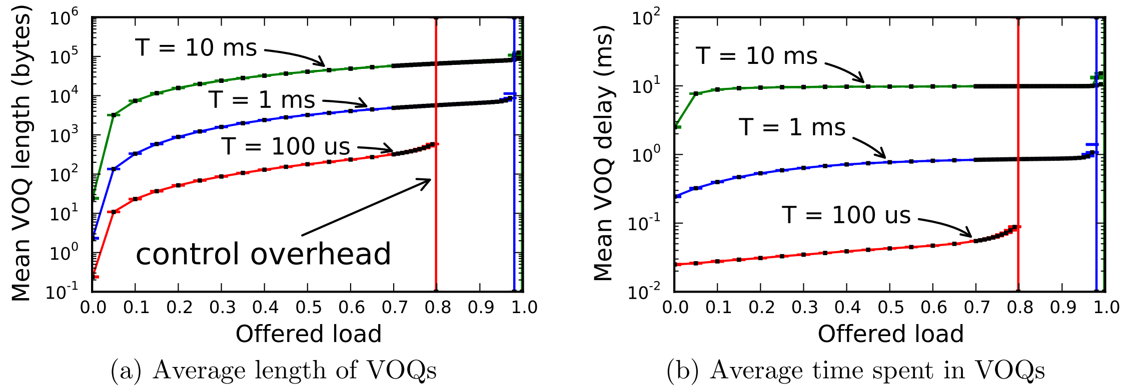
For scheduling to remain feasible, the control overhead cannot consume more than a few percent of the server's capacity. With $T = 1$ ms a server could communicate with on the order of a few tens of servers. Of course, there are many optimizations that could be used to reduce the control overhead substantially. For example, control information could be piggy backed on data packets and the scheduling interval could be made dynamic. A fixed scheduling interval, however, has the advantage of making the control overhead deterministic which is useful for evaluating our approach.

Scheduling interval

The scheduling interval represents key parameter that can be varied. If we reduced the scheduling interval by a factor of 2 we would double the overhead. Increasing the scheduling interval increases the amount of buffering required and also the amount of time packets spend waiting in VOQs. On the other hand, reducing the scheduling interval increases the control overhead which effectively reduces the bandwidth available for the tenant's traffic.

To examine this tradeoff we ran an experiment using scheduling intervals of 100 microseconds, 1 millisecond, and 10 milliseconds. In this experiment, a tenant with 16 servers creates an all-to-all traffic pattern between its servers. While all flows use the same average sending rate, the packet sizes and inter-arrival times follow a Poisson distribution leading to short but significant variations in their sending rates. We compared the control overhead, the average and maximum VOQ size, and the average time packets spent in VOQs. In this experiment, values represent the average measurements across 10 iterations where each run consisted of 100 ms warmup followed by a 1 second measurement period.

Figure 4.4 shows the average amount of time that packets are buffered in VOQs as well as the average length of VOQs. In section 4.2.4 we described how under admissible traffic patterns, VOQs receive the rate needed to clear their backlog over the next scheduling interval. This has two consequences. First, it means we should expect all packets to be delayed by the scheduling interval. Second, it means that the



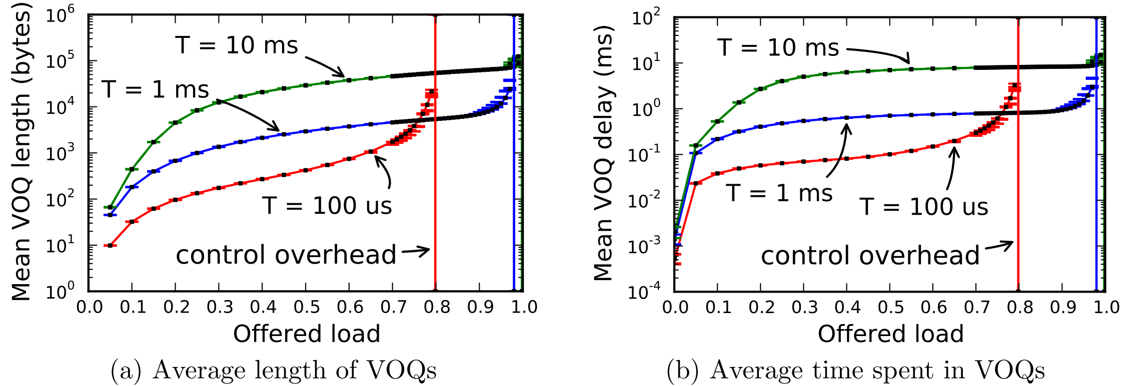
(a) Average length of VOQs

(b) Average time spent in VOQs

Figure 4.4: Effect of different scheduling intervals on VOQs

amount of buffering required to sustain a given rate depends directly on the scheduling interval. A scheduling interval of $T = 1$ ms, for example, corresponds to 122 KB at 1 Gbps. Since sending rates are Poisson, however, the traffic pattern is inadmissible on time scales. As a result, we see that the average delay is somewhat larger than the scheduling interval.

The network effectively reaches saturation at the point when the average VOQ no longer receives the rate needed to clear its backlog. Because the control traffic consumes some of the available capacity, this happens before the offered load reaches 1. Note that for $T = 100\mu s$ this happens just before 80% since the control overhead consumes $15 * 1.34$ Mbps which is more than 20% of each server's capacity. At this point the VOQs continue to grow in size until packets are dropped. It is important to understand that reducing the scheduling interval does not increase the throughput, provided VOQs are sized appropriately. In fact, the opposite is true since the increase in control traffic effectively reduces the capacity that can be used to support the tenants actual traffic. The results demonstrate that reducing the scheduling interval below a millisecond provides diminishing returns. While a slightly larger scheduling interval would mean that packets experience a few milliseconds of delay in VOQs, this may be quite reasonable for many applications given that without scheduling, packets can easily experience more than a millisecond of queueing delay at switches.



(a) Average length of VOQs

(b) Average time spent in VOQs

Figure 4.5: Effect of assigning VOQs a minimum rate of 10 Mbps.

Addressing VOQ delay with minimum rates

To address the issue of delay, we can assign minimum rates to VOQs. This may benefit latency sensitive traffic since such traffic typically consists of short flows. The minimum rate can be handled in the same way that we account for control traffic. This means that until a VOQ times out, it receives the minimum rate which is effectively subtracted from the allocatable capacity for the links along the path. We implemented this approach and repeated the same experiment with a minimum VOQ rate of 10 Mbps Figure 4.5. The results show that at low offered load this significantly reduces VOQ delay but that as the rate of the average flow approaches the minimum rate, latency begins to approach the scheduling interval.

4.4.3 Flow control

Finally, we evaluate the use of scheduling as a flow control mechanism that regulates server traffic. To do this, we revisit the strict/loose model for separating flow control and routing that we described in section 3.3. While our simulation represents an idealized implementation, it is still useful to evaluate where it falls within the space outlined by the model. The experimental setup is the same as that described in results shown in 3.3 except that we used an 8-port FatTree consisting of 128 servers. The network was partitioned into 8 tenants with 16 servers that each produced an all-to-all traffic pattern. The results are shown in Figure 4.6 and Figure 4.7. We

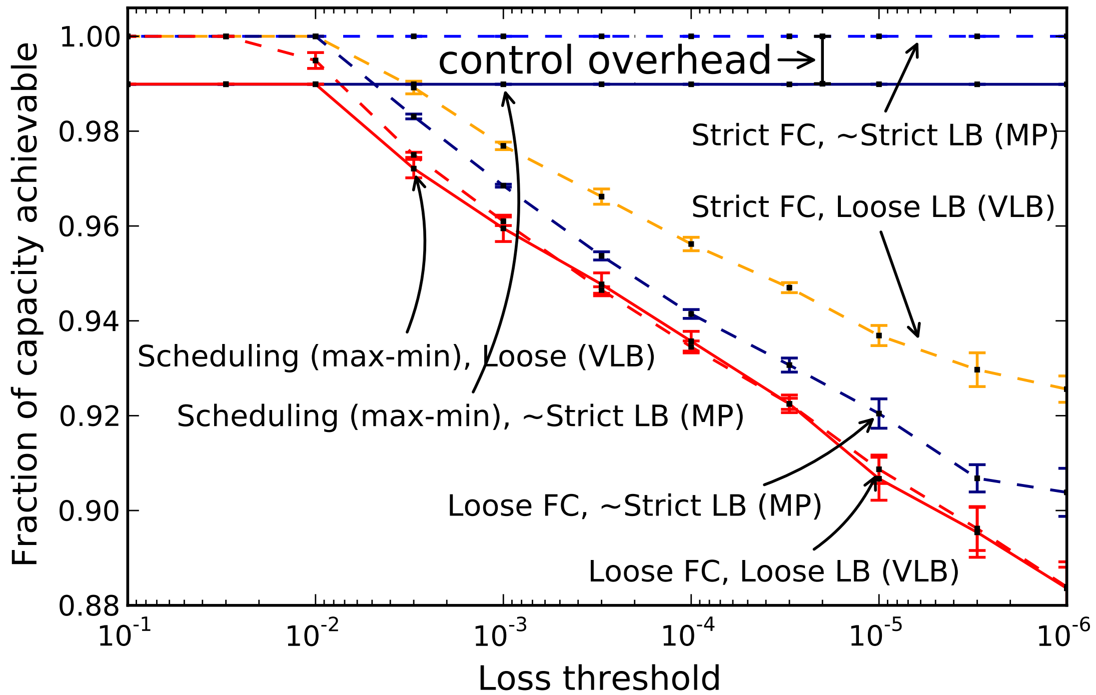


Figure 4.6: Capacity vs threshold with scheduling

subtracted the control overhead from the measured capacity so that the point labeled "control overhead" represents the capacity that can actually be used by the tenant.

These results demonstrate that with scheduling, we can achieve nearly the same performance as with strict flow control. The caveat, is that we effectively have to sacrifice part of the network's capacity to support scheduling's control traffic. However, a key point is that even if the performance in practice were to lie somewhere between strict and loose flow control, it would still provide a substantial improvement in the network capacity that routing achieves. In fact, for loss thresholds that are under 1%, the results show that scheduling effectively pays for itself since the capacity gained significantly exceeds the control overhead.

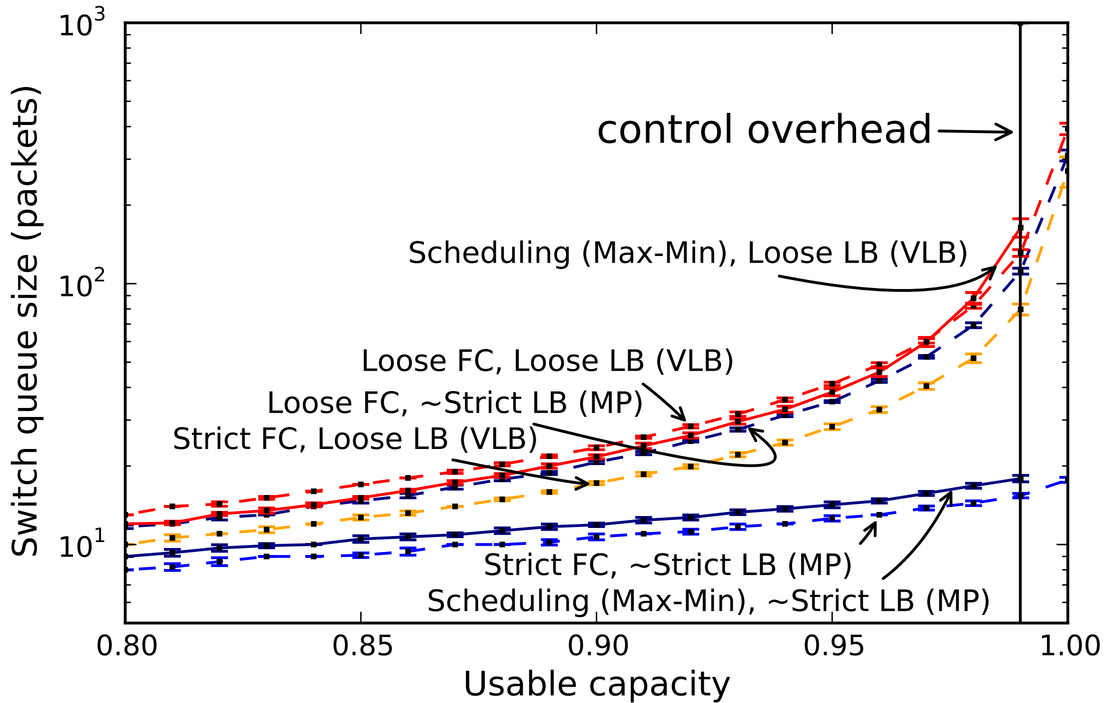


Figure 4.7: Queue size vs capacity with scheduling

4.5 Discussion & future work

4.5.1 Interactions with other protocols:

In this chapter, we presented a simple model where scheduling exists transparently to the layers above. We assumed that on the time scale at which rates are assigned, the rate of traffic arriving in VOQs is independent of the rates that we assign in the scheduling layer. Since other protocols, like TCP, may manage rates it's worth considering what effect scheduling has. Studying the interactions between TCP and the scheduling layer is beyond the scope of this dissertation. For scheduling to be practical it should not make the performance of protocols like TCP significantly worse. While it's important that scheduling not make the performance of protocols like TCP significantly worse, it is important to note that scheduling has the potential to have a positive impact on its performance as well. For example, by explicitly coordinating with other servers, scheduling could allow TCP flows converge to their max-min fair share more quickly and without first creating congestion in the process.

An interesting direction for future work would be to investigate coupling the scheduling layer with protocols like TCP to improve performance or to expose an API to the scheduling layer so that protocols, middelware, and applications could influence the way rates are assigned. While we have presented one approach, the scheduling framework could provide different network models to accommodate different applications. These models may have different objectives such as minimizing latency, maximizing the traffic transferred between servers, or simply preventing congestion while presenting a consistent view of the network to the layers above. In the next chapter we consider one such direction with the backlog scheduling problem.

4.5.2 Virtual machines

Data centers often use virtualization and may assign different tenants to the same server and some even allow users to provide their own guest operating systems. In such an environment, scheduling would have to be implemented in the hypervisor, where it exists outside of the control of tenants. This work was presented in the context in which tenants are provided with virtual network abstractions which means that if virtualization is used, tenants would receive a static slice of their server's interface. Since scheduling occurs separately within each tenant, it would not matter whether rates are scheduled between physical servers or virtual servers.

4.5.3 Practical considerations

Granularity of rates:

In our discussion of how rates are assigned, we have assumed that rates represent continuous values. In reality the scheduling layer can only control the times at which discrete packets are sent. Thus the packet size limits the granularity at which it is meaningful to control rates. Consider a 1 Gbps link and a scheduling interval of 1 ms, for example. This translates to a little over 80 maximum-sized (1500 byte) packets per interval. So when viewed over the space of one scheduling interval, rates can only

be controlled at a granularity of just over 1% of the link rate, which is more than 10 Mbps.

Controlling rates with high-precision:

A related issue is how effectively the scheduling layer can control server rates in software. In our simulation model, we can schedule an event to trigger the transmission of every packet at each VOQ. By taking into account the length of the packet being transmitted and the transmission rate, we can calculate the next transmission time thus are only limited in our precision by the granularity of packets. While it might be possible to schedule similar events in the kernel given high-precision timers, this may impose significant overhead due to context switching. Other protocols that control rates do not face this issue. TCP, for example, uses the arrival of acknowledgements to trigger the release of new packets and only uses timers to handle anomalies like packet loss. To be practical, it may be necessary to balance precision with overhead but such a tradeoff is difficult to evaluate without a complete and optimized implementation.

Chapter 5

Backlog scheduling

5.1 Introduction

In chapter 4 we introduced the distributed scheduling framework which controls the rate of traffic between a tenant's servers. We demonstrated that performance isolation can be achieved by any feasible assignment of rates and we provided a distributed algorithm that assigns rates in max-min and backlog proportional fashion. In this chapter we examine what impact different scheduling strategies may have on the performance experienced by a tenant. While there is no general way to define the optimal assignment of rates between a tenant's servers, it is worth asking whether there may be reasons for a tenant to favor one approach over another and what the impact may be on the performance of its application. We attempt to explore this question by introducing an additional scheduling objective; that of minimizing the overall time to transfer the backlog that exists between a tenant's servers. This provides one way of relating the rates assigned in the scheduling layer to the performance of the tenant application that may apply to an important class of data center applications. For example, it may benefit applications like MapReduce [23], where performance is dependent on the makespan of a set of data intensive tasks [59, 12]. To evaluate this potential benefit, we introduce the backlog scheduling problem and consider three variations of the basic problem that allow us to model scheduling within this new context.

5.2 Backlog scheduling problem

The goal of the backlog scheduling problem is to minimize the overall time required to transfer all of the backlogs that exist among a tenant's servers. To understand the nature of this problem, we start with a simplified view of scheduling where a central algorithm with global knowledge produces a set of rate assignments for all servers on fixed, periodic intervals. We also assume that all servers can send and receive at the same rate and that a tenant is provided the abstraction of having all of its servers connected to the same switch. Focusing on the virtual-switch abstraction allows us to view the server interfaces as the only bottleneck around which rates must be scheduled.

There are three variations of the basic problem that we consider:

1. **Initial-backlog:** All traffic is present at time 0. This means we are given some initial set of backlogs and we simply need to minimize the number of scheduling intervals required to clear the backlog.

2. **Deterministic backlog-schedule:** New traffic can arrive at the start of each scheduling interval and this amount is known in advance. This is a generalization of the problem where backlog increases according to a fixed schedule that is available to the algorithm.

3. **Online backlog-schedule:** New traffic can arrive at the start of each scheduling interval but the amounts are not known in advance. This online version of the problem more closely represents what any real scheduling algorithm must confront in practice.

5.2.1 Preliminary definitions:

Let $N = \{1, 2, ..., n\}$ be the set of n tenant servers whose traffic is being scheduled.

Backlog: Let the backlog $b_{i,j}^t \geq 0$ represents the data that server $i \in N$ has to send to server $j \in N$ at the start of interval t.

Let B^t be the set of all server backlogs in interval t.
It is often useful to express B^t as an $n \times n$ matrix where row i corresponds to the backlog that server i must send to each server and column j represents the backlog that each server has for server j.

$$B^t = \begin{bmatrix} b_{1,1}^t & b_{1,2}^t & \cdots & b_{1,n}^t \\ b_{2,1}^t & b_{2,2}^t & \cdots & b_{2,n}^t \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1}^t & b_{n,2}^t & \cdots & b_{n,n} \end{bmatrix}$$

Send backlog: We refer to the total amount of backlog that server i has to send to all other servers as its send backlog. We denote the send backlog of server i at interval t with $b_{i,+}^t$ which is defined as:

$$b_{i,+}^t = \sum_{j=1}^{n} b_{i,j}^t \tag{5.1}$$

Receive backlog: Similarly, the total amount of backlog destined for server j is called its receive backlog. We denote the receive backlog as $b_{+,j}^t$:

$$b_{+,j}^t = \sum_{i=1}^{n} b_{i,j}^t \tag{5.2}$$

Total backlog: We refer to the sum of all server backlogs as the total backlog and its notation is $b_{+,+}^t$.

$$b_{+,+}^t = \sum_{i=1}^{n} \sum_{j=1}^{n} b_{i,j}^t \tag{5.3}$$

Backlog degree: The maximum of a server's send and receive backlog is known as its backlog degree. We use the notation $\beta_i(B^t)$ for the backlog degree of i:

$$\beta_i(B^t) = \max(b^t_{i,+}, b^t_{+,i}) \tag{5.4}$$

Maximum backlog degree: This term refers to the maximum backlog degree of any server and is denoted simply with β.

$$\beta(B^t) = \max(\beta_i(B^t)) \quad \forall i \in N \tag{5.5}$$

Note that the notation $\beta_i(B^t)$ can be interpreted as the maximum of the values corresponding to the sum of row i and sum of column i from the matrix B^t. The notation $\beta(B^t)$, therefore, is simply the maximum of all column sums and row sums of matrix B^t.

Rates: Let $r^t_{i,j}$ represent the rate server i is assigned (by the algorithm) to server j in interval t.
We will assume that all servers can send and receive at the same rate. Furthermore, we assume that backlogs have been normalized to the rate value so that each server can send and receive 1 unit of backlog per interval.

Rate assignment: We use the notation R^t to represent the set of all rates assigned in an interval t. We refer to this as a rate assignment. As with backlogs, R^t may be expressed as an $n \times n$ matrix.

Feasible rate assignment: A rate assignment R^t is feasible with respect to B^t if it satisfies the following conditions:

$$r^t_{i,+} \leq 1, \text{ for all } i \in N \tag{5.6}$$

$$r^t_{+,j} \leq 1 \text{ for all } j \in N \tag{5.7}$$

$$0 \leq r^t_{i,j} \leq b^t_{i,j} \text{ for all } i, j \in N \tag{5.8}$$

Conditions 5.6 and 5.7 are the send and receive constraints which collectively ensure that the rates assigned do not exceed the capacity of any server's interface. Note that an equivalent way to express these constraints would be $\beta(R^t) \leq 1$. Condition 5.8 means that rates must be non-negative and they should not exceed the backlog that is present.

Residual backlog: Because rates are expressed in terms of backlog per interval, we can subtract a rate assignment from the backlog present to find the backlog that remains at the end of an interval t.
That is we let $B_r^t = B^t - R^t$ and we call B_r^t the residual backlog for interval t.

$$B_r^t = \begin{bmatrix} b_{1,1}^t - r_{1,1}^t & b_{1,2}^t - r_{1,2}^t & \cdots & b_{1,n}^t - r_{1,n}^t \\ b_{2,1}^t - r_{2,1}^t & b_{2,2}^t - r_{2,2}^t & \cdots & b_{2,n}^t - r_{2,n}^t \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1}^t - r_{n,1}^t & b_{n,2}^t - r_{n,2}^t & \cdots & b_{n,n} - r_{n,n}^t \end{bmatrix}$$

Rate schedule: The solution that the scheduling algorithm produces is called a rate schedule. A rate schedule is a set of rate-assignments representing the rates assigned during consecutive scheduling intervals and is denoted simply as R. For example, $R = \{R^0, R^1, ..., R^{l-1}\}$ is a rate schedule consisting of rates for the scheduling intervals 0 through $l-1$. The length of the schedule is equal to the cardinality of the set $|R| = l$.

Schedule-sum notation: At times it is necessary to use one additional bit of notation in order to express the sum of a set (or subset) of the matrices in a schedule. We use the notation R^+ to indicate the sum of all rate assignment matrices in the schedule r. That is:

$$R^+ = \sum_{i=0}^{l} R^i \tag{5.9}$$

The sum of the matrices from interval s to t is written as $R^{s,t}$:

$$R^{s,t} = \sum_{i=s}^{t} R^i \tag{5.10}$$

Feasible rate schedule: For a rate schedule R to be feasible, all rate assignments must be feasible and all backlog must be cleared. One way to express the second condition is that there must be no residual backlog at the end of the schedule.
So given schedule R of length $|R| = l$, R is feasible if:

$$R^t \text{ is feasible with respect to } B^t, \quad \forall t \in [0, l) \tag{5.11}$$

$$B_r^{l-1} = 0_{n\times n} \tag{5.12}$$

5.3 Initial-backlog problem:

Assume that at interval $t = 0$, there is some initial set of backlogs B^0 and that no new backlog arrives in subsequent intervals.
Since no new backlog can arrive, the backlog in interval $t+1$ is simply $B^{t+1} = B_r^t$.
The residual backlog is defined as $B_r^t = B^t - R^t$, which implies that $B_r^t = B^0 - R^{0,t}$ and that schedule R clears the backlog if $R^+ = B^0$.
Therefore a rate schedule R is feasible with respect to B^0 if:

$$\forall R^t \in R,\ R^t \text{ is feasible with respect to } B^0 \tag{5.13}$$

$$R^+ = B^0 \tag{5.14}$$

5.3.1 Problem definition:

The goal is to find a feasible rate schedule that clears the backlog using the minimum number of scheduling intervals. Therefore the problem can be stated as follows:

Given an initial set of backlogs B^0, find a minimum cardinality feasible rate schedule with respect to B^0.

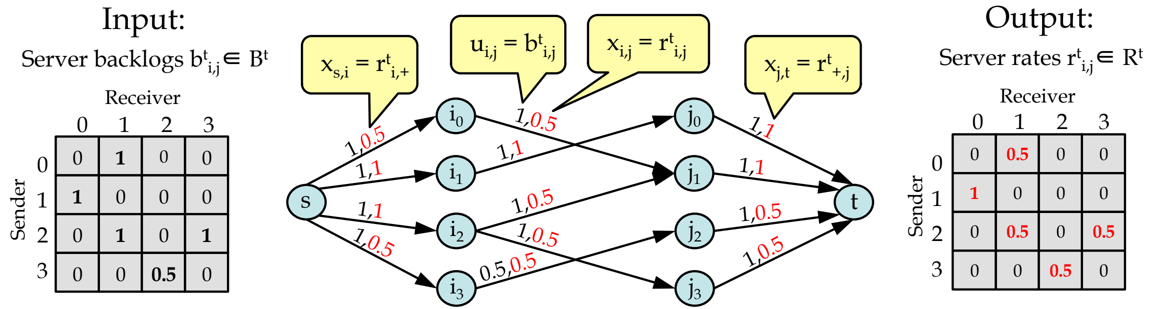


Figure 5.1: Feasible rate assignment as a feasible network flow. Flow values for an example solution are shown in red.

5.3.2 Rate-assignment as a network flow:

We can represent the problem of finding a feasible rate assignment R^t with respect to B^t as finding a feasible flow in a flow network.
The flow network graph $G = (V, E)$ can be constructed as follows.
Let each server be represented by two separate vertices; a **send node** $i \in V_S$ and a **receive node** $j \in V_R$.
For every backlog $b^t_{i,j} \in B^t$, add an edge (i, j) from send node $i \in V_S$ to receive node $j \in V_R$ with capacity $u_{i,j} = b^t_{i,j}$.
We now add two additional nodes; a source node s and a sink node t so that $V = V_S \cup V_R \cup \{s, t\}$.
For each send node $i \in V_S$ add an edge $(s, i) \in E$ with capacity $u_{s,i} = 1$.
This capacity represents the send capacity of server i which we assume is equal to 1.
Similarly, for each receiving node $j \in V_R$ add an edge (j, t) with capacity $u_{j,t} = 1$ to represent its receive capacity.
We can now assign a flow $x_{i,j} \in x$ to each edge in $(i, j) \in E$.

A flow on a network flow graph is **feasible** if for every node (except the source and sink), the sum of the flows entering the node equals the sum of the flows leaving the node and no edge is assigned a flow which exceeds its capacity. Figure 5.1, shows an example of this construction. The input B^t is shown on the left and the rate assignment R^t corresponding to the network flow x is shown on the right. It is easy to verify that a feasible flow x on the graph G represents a feasible rate assignment R^t. First notice condition 5.8 is satisfied since the flow between any send node i and

receive node j can't exceed the capacity $u_{i,j} = b^t_{i,j}$. Since the total flow leaving any send node i must match the flow entering node i, the total flow cannot exceed the capacity on edge (s, i) which ensures that 5.6 is satisfied. Likewise, the capacity on edge (j, t) ensures the last condition 5.7 is satisfied.

5.3.3 Max-min is not optimal

It is interesting to note that an algorithm that assigns rates according to max-min fairness is not optimal for the initial backlog problem. We described such an algorithm in section 4.2.5. A simple counter example is provided below. To keep this example compact, we allow non-zero diagonal values in the matrix. While this implies that servers can send to themselves, we can always reformulate the problem with additional servers so that this is not the case.

Counter example:

$$B^0 = \begin{bmatrix} 1 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

In this example, all three servers have backlog for the same two destinations which means that the capacity at each destination must be divided among three competing flows. Assigning flows in max-min fair fashion would result in a rate schedule requiring 3 intervals whereas a feasible rate schedule can be constructed with only 2 intervals if we give priority to the flows originating from server 0.

The solution produced by max-min requires $|R| = 3$ intervals:

$$R^0 = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 \end{bmatrix} R^1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{6} & \frac{1}{6} & 0 \\ \frac{1}{6} & \frac{1}{6} & 0 \end{bmatrix} R^2 = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

An optimal solution R^*, uses $|R^*| = 2$ intervals:

$$R^0 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 \end{bmatrix} R^1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 \end{bmatrix}$$

5.3.4 Optimal algorithm:

Theorem: An optimal solution to the initial backlog problem B^0 consists of $\lceil \beta(B^0) \rceil$ intervals.

Proof:
First notice that no feasible solution can consist of less than $\lceil \beta(B^0) \rceil$ scheduling intervals since $\beta(B^0)$ is the maximum amount of backlog that some server has to send or receive and no server can send or receive more than one unit of backlog per interval. To prove the forward direction, that some feasible schedule R always exists with $|R| = \lceil \beta(B^0) \rceil$ intervals, we present the following optimal algorithm and prove its correctness.

Backlog-proportional algorithm:
Note that the algorithm presented here is a simplified graph version of the backlog proportional algorithm presented in section 4.2.5.
Rates are assigned in proportion to backlog according to:

$$r_{i,j}^t = \min \left(b_{i,j}^t, \frac{b_{i,j}^t}{b_{i,+}^t}, \frac{b_{i,j}^t}{b_{+,j}^t} \right) \tag{5.15}$$

The first term ensures that the rate assigned is not more than is necessary to clear the backlog between i and j. The second and third term are the backlog proportional share of the send backlog at server i and the receive backlog at server j respectively.

The backlog-proportional algorithm uses the network flow graph construction for a rate assignment as described above. It is a greedy algorithm that assigns flow to each edge in proportion to the minimum of the edge's share of the backlog at both the send and receive node. The flow that is present at the end of each iteration

corresponds to a feasible rate assignment and the algorithm terminates after exactly $\lceil \beta(B^0) \rceil$ iterations.

Begin at interval $t = 0$ with $B^t = B^0$:
algorithm Assign backlog-proportional rates:

1. Construct the network flow graph $G = (V, E)$ to represent a rate assignment corresponding to B^t.
2. For each edge $(i, j) \in E$ s.t. $i \in V_S, j \in V_R$, assign $x_{i,j} = \min\left(b^t_{i,j}, \frac{b^t_{i,j}}{b^t_{i,+}}, \frac{b^t_{i,j}}{b^t_{+,j}}\right)$
3. Add the rate assignment R^t corresponding to flow x to schedule $R = R \cup \{R^t\}$
4. Compute the residual backlog $B^t_r = B^t - R^t$
5. Repeat from step 1 with $B^t = B^t_r$ until $B^t_r = 0$

Proof of correctness:
We first prove that the algorithm always produces a feasible schedule. We showed earlier that the construction of G ensures that any feasible flow represents a feasible rate assignment. Thus to satisfy condition 5.13, we must verify that the flow assigned to any edge does not exceed its capacity.
Clearly for any edge (i, j) where $i \in V_S, j \in V_R$, $x_{i,j} \leq u_{i,j}$ since $u_{i,j} = b^t_{i,j}$ and $x_{i,j} = \min\left(b^t_{i,j}, \frac{b^t_{i,j}}{b^t_{i,+}}, \frac{b^t_{i,j}}{b^t_{+,j}}\right) \leq b^t_{i,j}$.
For any source edge (s, i), the flow entering $i \in V_S$ must match the flow leaving so $x_{s,i} = \sum_{j \in V_R} x_{i,j}$.
Since $x_{i,j} \leq \frac{b^t_{i,j}}{b^t_{i,+}}$ $\forall i \in V_S, j \in V_R$, we know $x_{s,i} \leq \sum_{j \in V_R} \frac{b^t_{i,j}}{b^t_{i,+}}$ and $u_{s,i} = 1$.
Simplifying $x_{s,i}$ yields $\frac{1}{b^t_{i,+}} \sum_{j \in V_R} b^t_{i,j} = \frac{b^t_{i,+}}{b^t_{i,+}} = 1$ so $x_{s,i} \leq u_{s,i}$.
Similarly, for any sink edge (j, t), $x_{j,t} \leq \sum_{i \in V_S} \frac{b^t_{i,j}}{b^t_{+,j}} = 1$ and $u_{j,t} = 1$ so $x_{j,t} \leq u_{j,t}$.
Therefore, each iteration produces a feasible and the resulting set of rate assignments forms a feasible schedule since the algorithm does not terminate until $B^t_r = 0$, when the backlog is cleared.

To prove that the algorithm is optimal we need to show that it requires at most $\lceil \beta(B^t) \rceil$ iterations.
To do this, it suffices to show that $r^t_{i,+} \geq b^t_{i,+} - (\lceil \beta(B^t) \rceil - 1)$ and $r^t_{+,j} \geq b^t_{+,j} - (\lceil \beta(B^t) \rceil - 1)$ for all i and j.

Case $\beta(B^t) \leq 1$:
Since $b^t_{i,+} \leq \beta(B^0) \leq 1$ and $b^t_{+,j} \leq \beta(B^0) \leq 1$ we know $b^t_{i,j} \leq \frac{b^t_{i,j}}{b^t_{i,+}}$ and $b^t_{i,j} \leq \frac{b^t_{i,j}}{b^t_{+,j}}$ so $r^t_{i,j} = b^t_{i,j}$ for all i, j.
This implies $r^t_{i,+} = b^t_{i,+}$ and $r^t_{+,j} = b^t_{+,j}$ satisfying the conditions.

Case $\beta(B^t) > 1$:
Note that the condition $r^t_{i,+} \geq b^t_{i,j} - (\lceil \beta(B^t) \rceil - 1)$ is trivially satisfied if $b^t_{i,+} \leq (\lceil \beta(B^t) \rceil - 1)$.
Thus assume $b^t_{i,+} > (\lceil \beta(B^t) \rceil - 1)$.
Since $\lceil b^t_{i,+} \rceil = \lceil \beta(B^t) \rceil$ we can say that $b^t_{+,j} \leq \lceil b^t_{i,+} \rceil$.
$r^t_{i,j} \geq \min\left(\frac{b^t_{i,j}}{b^t_{i,+}}, \frac{b^t_{i,j}}{b^t_{+,j}}\right) = \min\left(\frac{b^t_{i,j}}{\max(b^t_{i,+}, b^t_{+,j})}\right) \geq \frac{b^t_{i,j}}{\lceil b^t_{i,+} \rceil}$.
Thus $r^t_{i,+} \geq \frac{b^t_{i,+}}{\lceil b^t_{i,+} \rceil}$.
Let $b^t_{i,+} = k + \epsilon$ where $k = (\lceil \beta(B^t) \rceil - 1) \geq 1$ and $\epsilon \leq 1$.
We can write $r^t_{i,+} \geq \frac{k+\epsilon}{k+1}$.
Since $\epsilon \leq 1$ it must be that $\frac{k+\epsilon}{k+1} \geq \epsilon$ so $r^t_{i,+} \geq \epsilon$.
Using k and ϵ, condition $r^t_{i,+} \geq b^t_{i,j} - (\lceil \beta(B^t) \rceil - 1)$ can be restated as $r^t_{i,+} \geq (k+\epsilon) - k$.
Since we have shown $r^t_{i,+} \geq \epsilon$, the condition is satisfied.
The same argument shows $r^t_{+,j} \geq b^t_{+,j} - (\lceil \beta(B^t) \rceil - 1)$.

5.3.5 Bounds on optimal

Given the optimal value, we can ask how much better is the optimal algorithm compared to other algorithms like max-min? Here we only need to consider algorithms that produce a blocking flow [3] since it does not make sense to leave spare bandwidth unassigned. As it turns out, any such algorithm produces a solution that is within twice the optimal value.

[3] A blocking flow on a network flow graph implies that the flow on any edge cannot be increased without first decreasing the flow on some other edge.

Theorem: The solution produced by any blocking algorithm is at most $2 * \lceil \beta(B^0) \rceil$.
Proof:
Let R be the rate schedule produced by the algorithm for the input B^0.
Now suppose that at some time T, some backlog $b_{i,j}^T > 0$ still remains.
We have assumed the rate assignment at each time step represents a blocking flow.
Thus for any interval $0 \leq t < T$, either $r_{i,+}^t = 1$ or $r_{+,j}^t = 1$ or both must be true.
In other words, either the backlog at i or the backlog to j is reduced by 1 or both.
Consequently, $T < b_{i,+}^0 + b_{+,j}^0$.
The definition of β ensures $b_{i,+}^0 \leq \beta(B^0)$ and $b_{+,j}^0 \leq \beta(B^0)$ and since $b_{i,j}^T > 0$, we can write that $T < 2\lceil \beta(B^0) \rceil$.
For the last interval in the schedule $T = |R| - 1$, which means that $|R| \leq 2 * \lceil \beta(B^0) \rceil$.

This result does not show that this bound is tight for an algorithm like max-min. However, in section 5.6.2, we provide an example that demonstrates that the backlog proportional algorithm can, in fact, outperform max-min by a factor of 2.

5.4 Deterministic backlog-schedule problem:

We now consider the deterministic backlog-schedule problem. In this version, the backlog in an interval may increase according to a fixed schedule which we are given. As before, the objective is to minimize the overall time it takes to transfer all of the backlog. However, for this problem we need to add a few new definitions to represent the input.

Backlog increment: In interval t, a backlog $b_{i,j}^t$ will increase by some value $f_{i,j}^t$ which we call a backlog increment.

Backlog increase: Let the backlog increase F^t be the set of all backlog increments $\forall f_{ij}^t \in F^t$ for interval t.
So the backlog at the start of each interval t is:

$$B^t = B_r^{t-1} + F^t \tag{5.16}$$

Backlog schedule: Let F be the set of all backlog increases $\forall F^t \in F$ which we will refer to as the backlog schedule.

A rate schedule R is feasible with respect to a backlog schedule F if the following two conditions are satisfied:

$$\forall R^t \in R, R^t \text{ is feasible with respect to } B^t \tag{5.17}$$

$$R^+ = F^+ \tag{5.18}$$

5.4.1 Problem definition:

Given a backlog schedule F, find a minimum cardinality rate schedule R with respect to F.

5.4.2 Optimal bounds

Given the backlog-schedule F, let R represent some optimal solution with $|R| = l$.
We can assume that the number of intervals in the rate schedule must be at least as long as the backlog schedule itself.
Thus a lower bound is:

$$|R| \geq |F| \tag{5.19}$$

If all of the backlog in the schedule were present at time 0, then this would be equivalent to the initial-backlog problem which we know cannot be solved in fewer intervals than the maximum backlog degree.
This gives a lower bound of:

$$|R| \geq \beta(F^+) \tag{5.20}$$

Likewise, waiting until the interval at which the schedule ends means an optimal initial backlog algorithm could clear the backlog in $\beta(F^+)$ intervals.

This gives an upper bound of:

$$|R| \leq |F| + \beta(F^{+}) \tag{5.21}$$

Additionally, we know that an optimal solution cannot require more intervals than would be necessary to solve each increase $F^t \in F$ as a separate initial-backlog problem. Thus if $|F| = k$:

$$|R| \leq \beta(F^0) + \beta(F^1) + ... + \beta(F^{k-1}) \tag{5.22}$$

Partitionable schedule: We say that a schedule is *partitionable* if it is possible to clear the backlog present in some interval before the end of the schedule. That is, a backlog schedule F is partitionable if there exists an interval $t < |F|$ for which the sub-schedule $F' = \{F^0, F^1, ..., F^{t-1}\}$ has a solution of length t. To solve a partitionable schedule, we could simply solve each sub-schedule independently and concatenate their solutions. Therefore we assume that a backlog schedule is *not* partitionable.

5.4.3 Linear programming formulation

We can formulate the deterministic backlog schedule problem as a linear program (LP). The basic idea is that the variables in the LP represent the rates in an optimal solution. Of course, we do not necessarily know the length of an optimal rate schedule a priori. Since we know it must lie within the bounds derived above, we can begin by thinking of the LP as a decision problem that checks whether a solution of length l exists. We then provide an objective function and show that solving the LP as a minimization problem also solves the optimization problem.

LP variables:
Assume an optimal solution R has $|R| = l$.
Let x be the set offset of variables in the LP.
For each $R^t \in R$, we add a variable $x_{i,j,t} \in x$ to represent $r^t_{i,j} \in R^t$.
Since there are n servers, there are at most n^2 rates to assign in each rate assignment $R^t \in R$ and since $|R| = l$, there are at most $n^2 l$ variables in x.

LP constraints:

The variables are subject to 4 types of constraints which follow naturally from the definition of a feasible rate assignment and a feasible schedule.

Send constraints:

Each server $i \in N$ can only send one unit of backlog in each interval $t \in [0, l)$:

$$\sum_{j=1}^{n} x_{i,j,t} \leq 1 \qquad \forall i \in N, \forall t \in [0, l) \tag{5.23}$$

Receive constraints:

Each server $j \in N$ can only receive one unit of backlog in each interval $t \in [0, l)$:

$$\sum_{i=1}^{n} x_{i,j,t} \leq 1 \qquad \forall j \in N, \forall t \in [0, l) \tag{5.24}$$

Backlog-present constraints:

The rate assigned through interval t cannot exceed the backlog present in the schedule through interval t:

$$\sum_{s=0}^{t} x_{i,j,s} \leq f_{i,j}^{0,t} \qquad \forall i, j \in N \tag{5.25}$$

Backlog-cleared constraints:

The total rate assigned between $i \in N$ and $j \in N$ must match the total backlog in the schedule:

$$\sum_{t=0}^{l} x_{i,j,s} = f_{i,j}^{+} \qquad \forall i, j \in N \tag{5.26}$$

We claim that any assignment to the variables that satisfies the constraints above must correspond to a feasible rate schedule. The send and receive constraints correspond directly to conditions 5.6 and 5.7 from the definition of a feasible rate assignment. The backlog-present constraints indirectly capture the final requirement for a feasible rate assignment which says that $r_{i,j}^t \leq b_{i,j}^t$. This condition cannot be expressed directly since $b_{i,j}^t$ is not constant as it depends on the rates assigned in previous intervals. The value of $f_{i,j}^{0,t}$ for a given interval t is constant, however, so we can write

$b_{i,j}^t = f_{i,j}^{0,t} - r_{i,j}^{0,t-1}$. Thus $r_{i,j}^t \leq f_{i,j}^{0,t} - r_{i,j}^{0,t-1}$ is an equivalent condition and moving all rates to one side of the expression gives $r_{i,j}^t \leq f_{i,j}^{0,t}$ which matches the backlog-present constraint. While the first three sets of constraints capture the requirements for a feasible rate assignment, the last set clearly matches the condition that the backlog must be cleared.

Objective function:

The formulation described thus far models a feasible rate schedule and so will provide a solution of length l if one exists. To solve the optimization problem, the LP should return a solution using the minimum number of intervals. This means if we use an upper bound on the optimal length (e.g. $l = \beta(F^+) + k$), the LP should leave all variables representing unneeded intervals equal to 0. This can be achieved if we assign the appropriate set of cost coefficients to the variables and let the objective be to minimize the total cost. In essence, the cost function should ensure that it is always more expensive to use a variable in some interval t if a solution can be found using only variables in intervals less than t.

An easy way to accomplish this is to assign costs according to:

$$c_{i,j,t} = (f_{+,+}^+)^t \tag{5.27}$$

Here $f_{+,+}^+$ is the total backlog in the schedule. The intuition behind this choice is simple. Since the cost grows exponentially with time, it can never be beneficial to transfer more backlog in an earlier interval at the expense of delaying the transfer of some backlog to a later interval. Therefore, a solution using t_1 intervals will always be cheaper than one using $t_2 > t_1$.

An example LP:

Consider the simple backlog-schedule $F = \{F^0, F^1\}$

$$F^0 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} F^1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

We need to find an optimal solution R but we do not know $|R|$.
However, we know that an upper bound is $|R| \leq \beta(F^0) + \beta(F^1) = 2 + 1 = 3$.

We can formulate the problem assuming $R = \{R^0, R^1, R^2\}$ and if the solution consists of less than 3 intervals, the extra rate assignments at the end of the schedule will consist of all zeros and can be pruned from the schedule.

Each rate $r^t_{i,j}$ is modeled by a variable $x_{i,j,t}$ so enumerating all possible rates provides the list of variables in the LP:

$$x = \begin{bmatrix} x_{1,1,0} & x_{1,2,0} & x_{2,1,0} & x_{2,2,0} & x_{1,1,1} & x_{1,2,1} & x_{2,1,1} & x_{2,2,1} & x_{1,1,2} & x_{1,2,2} & x_{2,1,2} & x_{2,2,2} \end{bmatrix}$$

A linear program in standard form looks like:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \leq b \\ \text{and} & x > 0 \end{array}$$

A and b are the matrix of coefficients and the right hand sides of the inequality constraints respectively. We show the 4 types of constraints in our LP below which collectively form the rows of A and b.

Send constraints:

$$\begin{array}{c} \\ t=0 \\ \\ t=1 \\ \\ t=2 \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccccccc} x_{1,1,0} & x_{1,2,0} & x_{2,1,0} & x_{2,2,0} & x_{1,1,1} & x_{1,2,1} & x_{2,1,1} & x_{2,2,1} & x_{1,1,2} & x_{1,2,2} & x_{2,1,2} & x_{2,2,2} \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array} \leq \begin{array}{c} TX \\ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{array}$$

Receive constraints:

$$
\begin{array}{r}
\\ t=0 \\ \\ t=1 \\ \\ t=2 \\ \\
\end{array}
\begin{array}{c}
\begin{matrix} x_{1,1,0} & x_{1,2,0} & x_{2,1,0} & x_{2,2,0} & x_{1,1,1} & x_{1,2,1} & x_{2,1,1} & x_{2,2,1} & x_{1,1,2} & x_{1,2,2} & x_{2,1,2} & x_{2,2,2} \end{matrix} \\
\begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{bmatrix}
\end{array}
\leq
\begin{array}{c}
RX \\
\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\end{array}
$$

Backlog-present constraints:

$$
\begin{array}{r}
\\ t=0 \\ \\ \\ \\ t=1 \\ \\ \\ \\ t=2 \\ \\ \\ \\
\end{array}
\begin{array}{c}
\begin{matrix} x_{1,1,0} & x_{1,2,0} & x_{2,1,0} & x_{2,2,0} & x_{1,1,1} & x_{1,2,1} & x_{2,1,1} & x_{2,2,1} & x_{1,1,2} & x_{1,2,2} & x_{2,1,2} & x_{2,2,2} \end{matrix} \\
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}
\end{array}
\leq
\begin{array}{c}
f_{i,j}^{0,t} \\
\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\end{array}
$$

Backlog-cleared constraints:

$$
\begin{array}{c}
\begin{matrix} x_{1,1,0} & x_{1,2,0} & x_{2,1,0} & x_{2,2,0} & x_{1,1,1} & x_{1,2,1} & x_{2,1,1} & x_{2,2,1} & x_{1,1,2} & x_{1,2,2} & x_{2,1,2} & x_{2,2,2} \end{matrix} \\
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}
\end{array}
=
\begin{array}{c}
f_{i,j}^{+} \\
\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\end{array}
$$

The last set of constraints are expressed as an equality but can be converted into standard form by adding a slack variable for each row. Also notice that we could have omitted variable $x_{1,1,0}$ since we can see from the schedule that $r_{1,1}^{0}$ must be 0 given that no backlog is present. In general, if $f_{i,j}^{0,t} = 0$ then we can omit variable $x_{i,j,t}$

since it means that no backlog can exist between i and j at interval t and so $r^t_{i,j} = 0$.

The costs are as follows:

$$c = \begin{bmatrix} 1 & 1 & 1 & 1 & 4 & 4 & 4 & 4 & 16 & 16 & 16 & 16 \end{bmatrix}$$

Solution:
In this example, there is only one optimal solution which would consist of the following values for the variables in the LP.

$$x = \begin{bmatrix} x_{1,1,0} & x_{1,2,0} & x_{2,1,0} & x_{2,2,0} & x_{1,1,1} & x_{1,2,1} & x_{2,1,1} & x_{2,2,1} & x_{1,1,2} & x_{1,2,2} & x_{2,1,2} & x_{2,2,2} \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since $R^2 = 0_{n,n}$, it can be pruned from the schedule leaving an optimal solution R of length 2:

$$R^0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} R^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

5.4.4 Proof of correctness

We have already shown that the formulation of the LP ensures that any feasible rate schedule consisting of at most l intervals represents a solution to the LP and vice versa. We now prove that the cost function guarantees that any minimum cost solution to the LP corresponds to an optimal rate schedule.

Proof:
Let s be the length of an optimal solution S to the backlog schedule F.
Let x' be a minimum cost solution to the LP when formulated with $|S|$ intervals.
let x be a minimum cost solution to the LP when formulated with any number of additional intervals.
Let c and c' represent the cost of x and x' respectively.
Suppose x represents some non-optimal schedule R.
This implies $c < c'$ and some $x_{i,j,s} > 0$.

Note that since each variable $x_{i,j,t} = r^t_{i,j}$, we can view their values in terms of the flow on the network flow graph for R^t.
The total backlog is $f^+_{+,+}$ which means that the total flow in both x and x' must match.
Clearly x' can have at most $f^+_{+,+}$ units of flow in interval $s-1$.
Note that x must also have at least 2 units of flow in interval $s-1$.
If this were not true, it would imply that x does not represent a blocking flow in interval $s-1$ which means we could decrease the cost by moving flow from some $x_{i,j,s} > 0$ to $x_{i,j,s-1}$.
For any interval t, the cost coefficient is $(f^+_{+,+})^t$.
The cost of x' is thus at most $f^+_{+,+} * (f^+_{+,+})^{s-1}$.
Suppose that x' has 1 unit of flow in some interval later than $s-1$ then its cost must be at least $2 * (f^+_{+,+})^{s-1} + 1 * (f^+_{+,+})^s = (f^+_{+,+} + 1) * (f^+_{+,+})^{s-1}$.
This yields a contradiction since the cost of x' is clearly greater than x.
Now suppose that the total flow x' has in interval s or later is $y < 1$.
It must be possible to move y units to an earlier interval without shifting more than a factor $y * \frac{f^+_{+,+}}{2}$ of flow to later intervals.
This is because there are at least 2 of the $f^+_{+,+}$ units of flow in interval $s-1$. The cost increased incurred for these flows will be at most $y * \frac{f^+_{+,+}}{2} * (f^+_{+,+})^{s-1}$. However the cost reduction will be at least $y * ((f^+_{+,+})^s - (f^+_{+,+})^{s-1})^{s-1}) = y * (f^+_{+,+})^s * (f^+_{+,+})^{s-1} - (f^+_{+,+})^{s-1})$

To complete the proof we show that the LP formulation is valid which also proves that the problem must be in P. To do this we show that the formulation uses a polynomial number of variables and constraints and the representation of cost is polynomial with respect to the size of the input/output. Notice that each server requires one send constraint and one receive constraint per interval. Thus there are at most $n * l$ send constraints and $n * l$ receive constraints since there are n servers and l intervals. There are at most $n^2 l$ "backlog present" constraints as we have one such constraint corresponding to each variable. Finally there are at most n^2 backlog-cleared constraints. The LP requires $O(n^2 l)$ variables and $O(n^2 l)$ constraints. Clearly n is polynomial in size with respect to the input F. Earlier we showed earlier that l was bounded by $l \leq \beta(F^+) + |F|$ which is an upper bound on the length of the output. Therefore the number of variables and constraints are also polynomial in size they are polynomial with respect to n and l. Finally, the cost of a variable in interval t is

$(f^+_{+,+})^t$ can be represented with $\log_2(f^+_{+,+}) * t$ bits which is polynomial with respect to the input.

5.5 Online backlog-scheduling:

The deterministic backlog problem allows us to model how we would assign rates given advanced knowledge of the backlog arriving at servers. In practice, we cannot know what data a server has to send until it arrives in the scheduling layer. The question that we seek to answer is how much better could we do given advanced knowledge of the arriving backlog and what is the best approach when we do not have the schedule available to us. In this section we consider the online version of the problem to explore these questions. In this version, the backlog still arrives according to a fixed schedule only this schedule is not provided to us. This allows us to compare the initial backlog algorithms, which can only consider the backlog currently present, with the optimal offline algorithm.

5.5.1 No online algorithm is optimal:

It is easy to see that no online algorithm can be optimal. Consider the two simple backlog schedules F and $\hat{F}$ that differ only in the second interval:
$F = \{F^0, F^1\}$:

$$F^0 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} F^1 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$\hat{F} = \{\hat{F}^0, \hat{F}^1\}$:

$$\hat{F}^0 = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \hat{F}^1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Clearly, each problem has only one optimal solution and these differ in their first interval:

$$R^0 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \hat{R}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Since an online algorithm is presented with the same input in interval 0, it cannot distinguish between the two problems until interval 1 which means it cannot make a deterministic decision that would be optimal in both cases.

5.5.2 Any optimal initial-backlog algorithm is 2-competitive:

We claim that any algorithm that optimally solves the initial backlog problem is 2-competitive with an optimal backlog scheduling algorithm. Given backlog schedule F, let R^* be the optimal offline solution.
Let R be the solution produced by the initial backlog algorithm.
We know that $|F| \leq |R^*| \leq |F| + \beta(F^+)$ from the bounds derived in section 5.4.2.
These bounds apply to R as well since no solution can finish before the end of the schedule and the backlog that remains when the schedule ends at $|F|$ is clearly no greater than $\beta(F^+)$.
Thus if $\beta(F^+) \leq |F|$ then since $|R^*| \leq |F| + \beta(F^+)$ and $|R| \leq |F| + \beta(F^+)$ we can write $|F| \leq |R^*| \leq |R| \leq 2 * |F|$.
Likewise, if $\beta(F^+) > |F|$ then $\beta(F^+) \leq |R^*| \leq |R| \leq 2 * \beta(F^+)$.
In either case we have $\frac{|R|}{|R^*|} \leq 2$.

5.5.3 Any blocking algorithm is 2-competitive:

This can be proven by induction on the length of F, i.e., $n = |F|$.
Basis: $n = 1$
This case has already been proven since it is equivalent to the initial backlog problem.
Inductive step: Assume n holds, show $n + 1$ also holds
First observe that if we partition a schedule $F = Fa \cup Fb$ then any blocking algorithm that solves Fa in a intervals and Fb in b intervals must solve F in at most $a + b$ intervals.
If this were not true it would imply that some interval in the solution to F is not be a blocking flow since the total rate and flow must always match.
Any schedule of length $|F'| = n+1$ can be expressed as $F' = F \cup \{F^n\}$ where $|F| = n$.
Let s and s' represent the length of an optimal solution to F and F' respectively.
Note that $s' \geq n + 1$, and $s' \geq s$ must be true.

Also note that $s' \geq s + \beta(F^n)$ otherwise the schedule F' would be partitionable.
Since $|\{F^n\}| = 1$, any blocking algorithm can solve F^n in $2\beta(F^n)$ intervals.
By the induction hypothesis we know that F requires at most $2s$ intervals.
This means $F' = F \cup \{F^n\}$ requires at most $2s + 2\beta(F^n)$ intervals.
Thus any blocking algorithm is at most $\frac{2(s+\beta(F^n))}{s+\beta(F^n)} = 2$ of optimal.

5.6 Evaluation

The purpose of backlog scheduling is to model how different scheduling algorithms may impact the performance of one type of application, whose performance hinges on the overall completion time of all of its flows. In the previous sections we examined three different versions of the problem which allowed us to place limits on the difference in performance that we might expect. However, the backlog scheduling problem provides no way to effectively evaluate how different algorithms will impact performance in practice. This is because the backlog that arrives depends on the pattern of arriving traffic which will naturally depend on the particular application. The problem also assumes that the rates assigned by the algorithm do not affect the schedule of arriving backlog. This assumption was necessary to provide some type model for scheduling but in reality the pattern of arriving traffic may depend on the pattern of traffic delivered in some arbitrarily complex manner depending on the behavior of the protocols and applications that lie above.

The objective of this evaluation is to gain a better understanding of the results that we have derived. Rather than make assumptions about what constitutes normal traffic, we will attempt to identify several traffic patterns that illustrate that the performance of the algorithms do, in fact differ, as we have predicted. We also simulate the traffic patterns that we find to confirm that the theoretical results can be reproduced in simulation. In the process, we hope to develop some intuition about why the performance of different algorithms vary and the types of traffic patterns that stress these differences.

5.6.1 Experimental setup

Determining theoretical values

In order to find the solutions to the patterns described above, we needed the ability to find the max-min and backlog-proportional solutions to a given backlog schedule and compare them with the length of an optimal schedule. To do this, we implemented both the max-min and backlog-proportional algorithms in python. We took advantage of the numeric python package NumPy [6] and represented backlog schedules and their solutions as lists of NumPy matrices. We wrote a short function that accepts a backlog schedule and used the desired algorithm to produce a rate schedule. Note that we also use this function for the initial backlog problem since it simply represents a special case backlog schedule of length 1. To compute an optimal solution, we leveraged the python convex optimization package CVXOPT [3] and developed code to produce the LP formulation for a backlog schedule and convert the solution produced by the LP solver into a rate schedule. In our evaluations we used backlog schedules consisting of up to 50 intervals with backlog for as many 50 servers. For the largest inputs, the LP formulation can require over a hundred thousand total variables and constraints and to solve these efficiently, we added additional code to support the IBM CPLEX solver [2]. All of the theoretical results presented below were collected using this code base.

Simulating a backlog schedule

We also extended our python code base to translate a backlog schedule into a script that could be read by the server nodes in our simulator. The script consists of a set of messages that is read by the message application described in A.3.3. Essentially, each backlog increase $f_{i,j,t}$ is defined as a message that will be sent to server j by the message application at server i at time t. The message application partitions the message into fixed size packets. These packets are sent immediately thus causing the full backlog corresponding to $f_{i,j,t}$ to arrive in the VOQ at t. The application also keeps track of when messages are sent and received and we record the time at which the last message is received before ending the simulation.

In order to translate the backlog schedule into a message schedule, we have to define the length of time that a scheduling interval represents and we have to express units of backlog in terms of actual bytes. Note that the intervals represented by the schedule do not have to correspond to the scheduling interval used by our distributed algorithm. The backlog scheduling problem models scheduling as operating on discrete intervals where rates are computed globally and rate and backlog quantities change instantly. While this makes it easier to reason about, the actual scheduling that we described in chapter 4 occurs in a distributed asynchronous fashion and only converges to the behavior of the ideal model in steady state. Therefore for the simulations that we present, we chose to have each interval in a schedule represent 100 ms and to avoid confusion, we refer to this as an *epoch*. Using this interval, we can express backlog in terms of bytes since a server can send one unit of backlog per interval. Thus to do this, we had to use the effective sending rate (i.e. account for protocol overheads) so that the backlog would cleared at the correct time if rates were set according to their theoretical schedules.

An example message script for the following backlog schedule is shown below:
$F = \{F^0, F^1\}$

$$F^0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} F^1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

```
# Backlog schedule F written at 2013-01-02 23:55:48.991388
# |F|=2, n=2 servers, Beta(F+)=2.000000
# The message script format is:
# <source>;<destination>;<messageNum>;<startTime>;<messageSize>;<messageRate>
0;               0;             0;            0.0 ms;          0 B;       0 bps
0;               1;             0;            0.0 ms;   11898569 B;       0 bps
1;               0;             0;            0.0 ms;   11898569 B;       0 bps
1;               1;             0;            0.0 ms;          0 B;       0 bps
0;               0;             0;          100.0 ms;          0 B;       0 bps
0;               1;             0;          100.0 ms;          0 B;       0 bps
1;               0;             0;          100.0 ms;          0 B;       0 bps
1;               1;             0;          100.0 ms;   11898569 B;       0 bps
```

5.6.2 Initial-backlog

Earlier we demonstrated that assigning rates in max-min fair fashion can lead to suboptimal solutions even in the initial backlog case. We can use the example we showed in section 5.3.3 as a template for a pattern that we can use to probe the gap in performance between the backlog-proportional and max-min algorithms. In the example, the maximum backlog degree is $\beta(B) = 2$. Send node 0 and receive nodes 0 and 1 all have backlogs matching the maximum backlog degree. Thus these nodes must reduce their backlog degree by 1 in each interval to finish in the optimal amount of time (2 intervals). For send node 0 to do this, however, it must receive more than its max-min fair share which is $\frac{1}{3}$.

The two "two-phase" pattern:

$$B^0 = \begin{bmatrix} n-1 & n-1 & 0 & 0 & ... & 0 \\ 1 & 1 & 0 & 0 & ... & 0 \\ ... & & & & & \\ 1 & 1 & 0 & 0 & ... & 0 \end{bmatrix}$$

By introducing more servers, we can extend this scenario to create a wider gap between the max-min rate and the rate it must send to finish in the optimal amount of time. Since the length of the solution depends on the last server to finish we only need to exploit this gap at node 0. With n total servers, row 0 has $n-1$ units of backlog in each column while the remaining $n-1$ rows have just 1. The maximum backlog degree is $\beta(B^0) = 2 * (n-1)$ and since $b_{0,+} = \beta(B^0)$, an optimal solution must clear 1 unit from row 0 in each interval. The max-min algorithm will fail to do this causing it to clear the backlog in two phases.

Phase 0:

$$R^t = \begin{bmatrix} \frac{1}{n} & \frac{1}{n} & 0 & 0 & ... & 0 \\ \frac{1}{n} & \frac{1}{n} & 0 & 0 & ... & 0 \\ \frac{1}{n} & \frac{1}{n} & 0 & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ \frac{1}{n} & \frac{1}{n} & 0 & 0 & ... & 0 \end{bmatrix}$$

Phase 1:

$$R^t = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & ... & 0 \end{bmatrix}$$

In the first phase, every edge will be assigned $\frac{1}{n}$ and it will last for n intervals at which point all rows except row 0 are clear. At the start of the second phase, row 0 has $n-2$ units remaining in each column. Since it will assign $\frac{1}{2}$ to each, it will take $\frac{n-2}{2}$ additional phases for a total of $n+\frac{n-2}{2}$ intervals. Since the optimal requires n intervals, with large n the performance approaches $\frac{3}{2}$.

By generalizing the two phase pattern to use f flows instead of just two, we were able to approach the worst case bound of 2. With n servers, the worst-case is found at $f=\sqrt{n}$.

$$B^0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ ... \\ n-1 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & & f-1 \end{array} \\ \begin{bmatrix} f+1 & f+1 & ... & f+1 \\ 1 & 1 & ... & 1 \\ 1 & 1 & ... & 1 \\ ... & ... & ... & ... \\ 1 & 1 & ... & 1 \end{bmatrix} \end{array}$$

Here the first and second phase both last n intervals causing max-min to have a length of $2*f^2$ while the optimal is $\beta(B^0)=(f+1)*f$. This yields a ratio of $\frac{2f^2}{f(f+1)}$ which approaches the upper bound of 2 for large n.

We collected the theoretical results for this pattern for $n=3$ through $n=50$ which we compared through simulation. The results are shown in Figure 5.2. We let each interval in the backlog schedule represent 100 ms. We fixed the scheduling interval in simulation to $T=1$ ms. We measured the length of time it took for the last server to finish sending and divided it by 100 ms epoch to compare with the length of the theoretical schedules. Since the scheduling interval is fixed, the control overhead grows as n increases causing the distributed algorithms to require slightly more than their ideal theoretical values.

In the experiment above, each backlog increase $f_{i,j,t}$ represents a message that starts at time t. The application at server i sends the message out in its entirety as soon as time t arrives which means that all of the backlog corresponding to $f_{i,j,t}$ arrives instantly in the VOQ destined for server j. While this is how it is modeled by the scheduling problem, it assumes large amounts of data can arrive as soon as it is generated by an application. On the one hand this is reasonable given that protocols like TCP often accept large messages from user space. Rather than have TCP buffer this data, it should be possible to have this data (or knowledge of its arrival) be

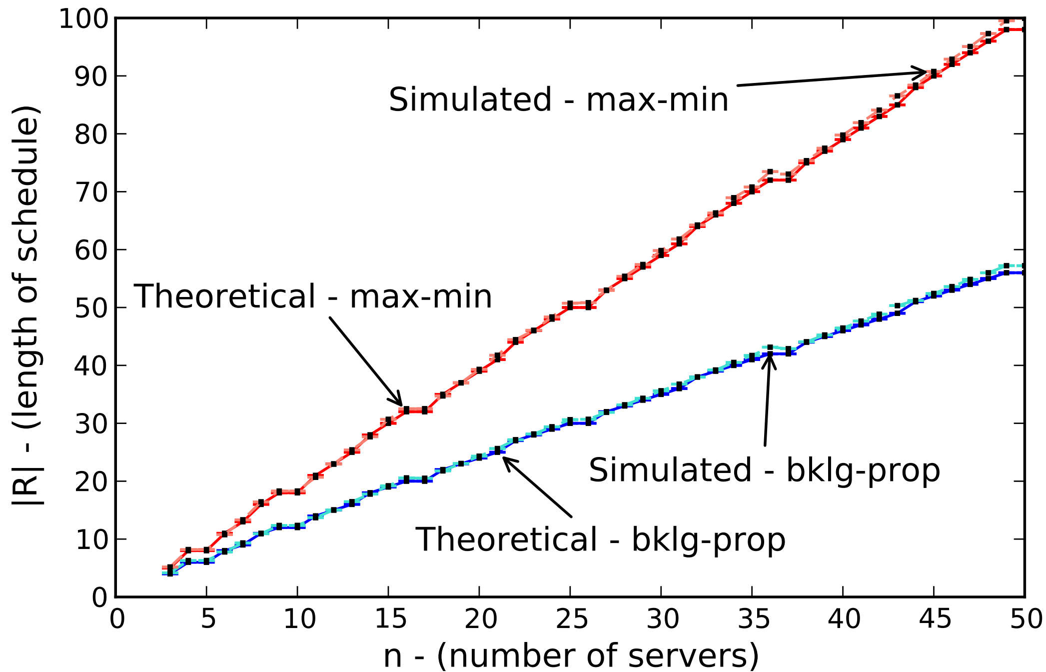


Figure 5.2: Initial backlog “two-phase” stress test

propagated immediately to the scheduling layer. For example, by manipulating the receive window, we could force TCP to deliver a large window of segments as they arrive.

We also ran the same simulation with a limit on the total rate at which messages could leave the application and enter the scheduling layer. Here we set that limit to match the sending rate of the server, i.e., 1 Gbps. The result is shown in Figure 5.3. The limit did not affect the performance of max-min, as we should expect, since it assigns all flows on a bottleneck link the same rate regardless of backlog. However, the backlog proportional algorithm is clearly affected since the backlog that exists according to the schedule arrives gradually. The limit effectively halves the gap between the algorithms causing the backlog proportional algorithm to finish in 1.5 times the optimal length. This result shows that even with this tight restriction, the backlog-proportional algorithm provides an improvement over max-min but that its benefit hinges on exploiting full knowledge of the available backlog.

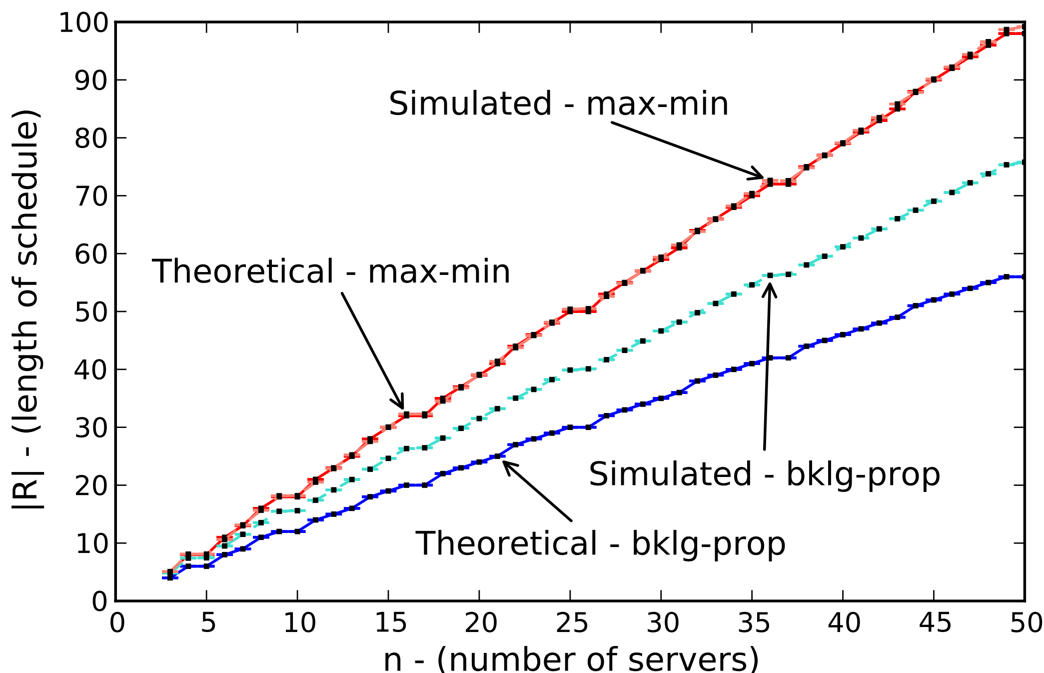


Figure 5.3: Limit on the rate of backlog entering the scheduling layer.

5.6.3 Backlog scheduling stress test

In section 5.4.2 we presented an example that showed the backlog proportional algorithm has a lower of 1.5 bound on the optimal offline solution. While we could find individual cases where the backlog proportional solution was 1.5 times the optimal length, we searched for a pattern that could be generalized to n servers. While we were unable to discern the precise pattern, this process did yield some insight into what the worse case looks like. For example, the more difficult cases arise when it is possible to finish in $|F| = \beta(F^+)$ intervals but only if certain critical backlogs are cleared in each interval. Based on our experience, we developed a "stress test" which represents a difficult problem for the online algorithms. The stress test pattern for the case of $n = 4$ is shown below:
$F = \{F^0, F^1, ..., F^{n-2}\}$

$$F^0 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} F^1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} F^2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

With n servers, the backlog schedule for the stress test has length $|F| = n - 1$ and max backlog degree $\beta(F^+) = n - 1$. Note that the diagonals are all non-zero in order to avoid having servers send to themselves. A similar stress test can be constructed to include the diagonal values which results in $|F| = \beta(F^+) = n$. We found this to be a slightly more difficult case resulting in worse online performance. However, with either version, the optimal solution S requires more than $|S| > \beta(F^+)$ intervals and the ratio $\frac{|S|}{\beta(F^+)}$ converges to just under 1.3 times as n scales. We suspect if we could preserve $|S| = \beta(F^+)$ by using a variation of this pattern, we may be able to force the max-min and the backlog-proportional algorithms to exhibit their worst case bounds.

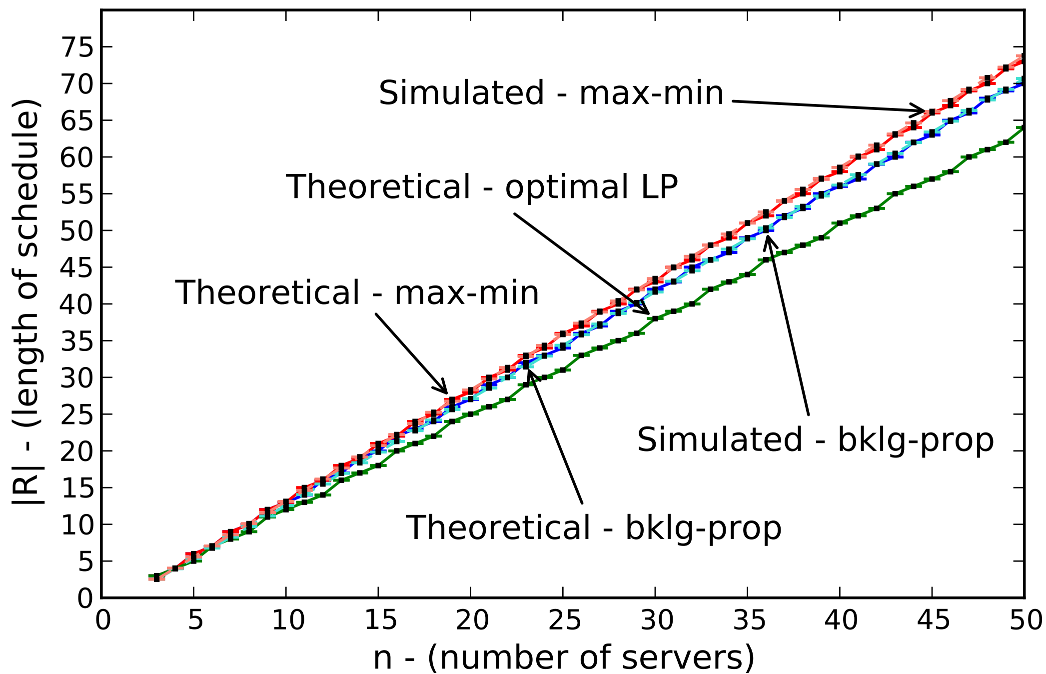


Figure 5.4: Performance with “stress test” backlog schedule.

We collected the theoretical values for $n = 3$ through $n = 50$ as before and the results are shown in Figure 5.4. The length of the solution produced by the LP is also shown. Here, we did not put a limit on the arrival of backlog into the scheduling layer for the simulated results. While this pattern did not quite push the theoretical bound of 1.5, the results do illustrate performance in the range that we predicted.

5.7 Extending the results to oversubscribed trees

To model the backlog scheduling problem, we focused specifically on the virtual switch abstraction. We have strong reason to believe, however, that many of these results can be extended to any oversubscribed tree-structured network, such as the virtual oversubscribed cluster. For example, it is easy to see that the LP formulation can be extended to accommodate any network. We simply replace the server send and receive constraints with the constraints matching condition 4.1, which are the equivalent constraints on the individual links as described in section 4.1. It can also be shown that the general backlog proportional algorithm described in 4.2.5 is still optimal for the initial backlog problem. While we do not provide a proof here, if the maximum backlog degree is redefined as the maximum ratio of backlog to capacity $\frac{\beta(B_l)}{c_l}$ over any link l, then following a similar approach to that used in section 5.3.4, it can be shown that the backlog proportional algorithm requires at most $\left\lceil \frac{\beta(B_l)}{c_l} \right\rceil$ intervals which is the minimum possible. We also wrote the python scripts described in section 5.6.1 to support the oversubscribed cluster and accept the oversubscription factor O, cluster size S, and link capacity C as parameters. The default values for these parameters are chosen to allow the virtual switch to be treated as a special case. We tested many different example schedules with several oversubscribed cluster parameters and did not see results that violate the competitive bounds that we proved for the virtual switch abstraction.

5.8 Summary

In this chapter we explored what impact different scheduling strategies may have on the performance of an application in a data center network. In particular, we focused on the metric of minimizing the total time to transfer all the backlog that must be sent between servers which we modeled as an optimization problem. We showed that to clear the backlog that is currently present, our backlog-proportional algorithm is optimal and that max-min can require up to twice as long. By introducing the backlog schedule to model the arrival of future backlog, we then reasoned about how the scheduling algorithms would perform more generally. We provided a linear programming formulation that optimally solves a backlog schedule offline and we

set bounds on the performance of online algorithms. We provided several examples of backlog schedules that demonstrate these results which we also verified through simulation with our distributed algorithm from chapter 4.

Chapter 6

Conclusion

6.1 Summary

This thesis makes several contributions. First, we explored the use of packet-level routing in FatTree data center networks constructed from commodity Ethernet switches. We showed that, in the context of providing bandwidth guarantees to tenants, packet-level routing provides a fundamental advantage over flow-level routing, allowing a significantly higher fraction of the network to be utilized. We proposed several new routing algorithms and demonstrated that they can outperform purely random algorithms like VLB if server traffic can be regulated to a sufficient degree. We also proposed a novel resequencing method that uses a combination of timestamps and sequence numbers to deal with the out-of-order arrivals that result from routing at the level of packets.

As a second contribution, this thesis introduced a distributed scheduling framework to control the rate of traffic between a tenant's servers. It has two primary functions. By tightly controlling the sending rates of servers, it acts as a flow control mechanism that can prevent congestion and minimize queueing in the network. This provides a benefit to packet-level routing because it limits the queueing that normally occurs. Our results show that it enables our multi-phase routing algorithm to achieve near optimal performance and that this improvement more than compensates for the control overhead produced. Secondly, it complements previous work by providing a distributed mechanism to enforce the bandwidth limits imposed on tenants by their virtual network abstractions. We provided a novel distributed algorithm that can set

rates in both max-min and backlog-proportional fashion and demonstrated that it provides the desired performance isolation with the virtual switch abstraction.

Finally, we proposed backlog scheduling as a means to investigate whether certain scheduling strategies may be preferable for different classes of applications. We focused on the objective of minimizing the overall time required to transfer backlog between servers and we used a formal approach, modeling the problem in three incremental steps. We showed that to clear the backlog already present, the backlog proportional algorithm was optimal and completes up to twice as quickly as max-min. We then model the arrival of new backlog by introducing the backlog schedule and we provided a linear program that optimally solves a backlog schedule offline. We showed that given advanced knowledge of the arriving backlog, it is possible to perform better but that the backlog-proportional algorithm is likely within 1.5 of the optimal.

6.2 Future Directions

Our investigation of packet-level routing in Chapter 3 focused exclusively on the FatTree topology. A logical next step would be to determine whether our results can be applied to other mutli-path topologies. We expect that some of the results can be extended to apply to other forms of the Clos network such as VL2.

One of the key unknowns that we confronted in our study is the precision with which sending rates can be controlled. This presented a significant obstacle in our evaluation of load balancing since we found that the results were very sensitive to bursty traffic. To overcome this, we attempted to model the possible range of performance that we could expect from flow control mechanisms in real data center networks. We did this by modeling the sending rates of servers as a process that was either Poisson or periodic. The results showed that our multi-phase load balancing algorithm only provides a significant improvement when the sending process is periodic. The scheduling framework can, in principle, simulate the periodic process by strictly controlling the departure times of packets at each VOQ. However, it remains to be seen whether it is practical to achieve the performance of this strict model with a real implementation.

Therefore, an important step to further this work would be to determine where a real implementation may lie on the strict/loose spectrum that we have examined.

Distributed scheduling opens up a rich problem space and the work that we have done represents one approach. The goal was to explore the concept and understand some of the trade-offs. There are many possible optimizations that were not pursued in an effort to simplify the design and evaluation. A logical next step would be to explore some of these optimizations and their tradeoffs. Additionally, our evaluation focused solely on the virtual switch abstraction. Our proxy based algorithm supports any network topology. However, more complex abstractions, such as the Virtual Oversubscribed Cluster (VOC), raise additional scaling challenges since proxies for oversubscribed links need to manage rates for many more servers. While support for such topologies needs to be further developed, one property that could be exploited is that with more servers, the relative differences between their rates decreases, particularly with max-min. This means rates on these links can be scheduled over a longer scheduling interval without significantly impacting performance. Another approach might be to distribute the proxy state for such links among "local proxies". For example, to manage rates on an oversubscribed link in the VOC abstraction, a local proxy could be assigned at each group switch to manage the rates for the servers at that switch. For local proxies to converge to the rates assigned by a single proxy, they would need to maintain complete replicas of the state managed by the other proxies. Alternatively, we could accept an approximate solution in exchange for reduced control overhead by allowing local proxies to exchange just the aggregate backlog and bandwidth requested by their servers.

The use of VOQs with changing rates means the scheduling layer presents an inconsistent view of the network to the layers above. How precisely the scheduling layer may interact with the protocols and applications above is a question worthy of further exploration. One direction might be to examine how the scheduling layer interacts with common protocols, such as TCP, and determine whether performance can be improved by making scheduling protocol aware.

Another key question is how scheduling impacts the performance of different types of applications. Given the unknown dependency between rates assigned and the behavior of the application, the backlog scheduling problem provides one model to

help answer this question. However, it assumes the arriving backlog is effectively independent of the backlog delivered on the timescale at which scheduling occurs. At a high level, the role of the network in any distributed system is to deliver messages between the various nodes. Therefore, the performance of any data center application can only depend on the network as far as it is bottlenecked on the exchange of some message between its servers. Without making assumptions about the application, we cannot know which messages represent a bottleneck to the application at any given time. Moreover, the scheduling layer cannot even determine where the boundaries of the application layer messages are.

In-spite of these issues, it may still be possible to provide some general way to characterize the performance of the application in terms of the rates assigned to VOQs. We briefly describe one possible approach, which we call message scheduling, that attempts to capture these unknowns. The essential idea is that messages can be defined implicitly based on the amount of backlog that arrives over some period, e.g. one scheduling interval. Given that the scheduling layer has no way to know what messages matter most to the application, the goal would be to bound the time it takes to deliver any message. In order to do this, lets suppose that in the absence of competing traffic, it takes x units of time to deliver a given message. If it takes X units to deliver the message with a given rate schedule, the *stretch* for that message would be $\frac{X}{x}$. The precise objective would then be to produce a schedule that minimizes the maximum stretch over all messages. It is likely that if some algorithm could bound the maximum stretch to some value S, then the application would complete in at most S times the completion time it would have if every message were delivered in the minimum time possible. This problem could be modeled incrementally in a similar fashion to the backlog scheduling problem and it may provide an interesting alternative to complement the approach that we have taken in this work.

Appendix A

FatTree DCN Simulator

A.1 Introduction

This appendix describes our FatTree data center network simulator. This simulator was developed specifically to conduct the work presented in this thesis. However, because it was built on top of the publicly available OMNeT++ [7] and INET [4] frameworks, some of its components may potentially benefit other researchers who are familiar with these tools. This appendix provides an overview of the simulator and also serves as the primary documentation for others that wish to incorporate parts of our simulation model into their own projects.

The simulator can be found at:
`http://www.arl.wustl.edu/~mah5/dc_sim.html`

A.1.1 Motivation

The simulator was developed to satisfy the following objectives:

- Scalable: Capable of simulating data centers of a reasonable scale, e.g. hundreds or thousands of servers.

- Adaptable: The networking stack of end hosts can be easily modified to incorporate new mechanisms, such as the distributed scheduling and packet-level routing techniques described in this thesis.
- Reuse: Should leverage existing tools when possible and be reusable by others.

There are many existing tools for network simulation, such as ns-2/ns-3 [5]. These tools are already capable of simulating Ethernet networks and with server nodes that emulate the full TCP/IP stack. Often, these simulation models are highly detailed and focus on providing the most complete and accurate representation of real world hardware and software. While this makes them powerful simulation tools, it also makes them less suited for our needs for several reasons. First, we do not need to simulate every detail of each individual networking protocol. Simulating unnecessary detail adds complexity that can obscure the essential behavior we are trying to study and also adds overhead that can limit the speed and scale of our simulations. Second, we will need to make substantial changes to the server nodes and switch nodes to support the type of networks and mechanisms that we describe in this work. For example, Ethernet does not natively support topologies with multiple paths, such as the FatTree. Since there are different proposals to construct FatTree DCNs from existing switches, we would have to implement one of these approaches in the simulator in order to simulate FatTree Ethernet networks. Making use of the existing protocol implementations at server nodes would also be difficult since we would likely have to address many of the same implementation-level issues that we would face implementing our distributed scheduling framework and packet-level routing mechanisms in a real operating system. Given these considerations, we decided to develop our own light-weight packet-level simulator based on the OMNeT++ framework.

A.1.2 OMNeT++

OMNeT++ is a C++ framework for discrete event simulation. It consists of an open-source simulation kernel, a library of extensible C++ components, and a suite of tools to facilitate statistics collection, results analysis, graphical debugging, etc. It is technically not a simulator in and of itself, but rather a simulation framework that provides all of the general components necessary to build a simulator. Vargas

et al. [57] provide a good overview of OMNeT++'s design objectives. OMNeT++ is also well documented and the reader is encouraged to consult the publicly available user manual for more complete and up-to-date information. Here we provide a brief overview of the relevant features needed to understand our simulator.

The OMNeT++ architecture consists of several components. First, there is a simulation kernel that contains all of the machinery necessary to support simulation, such as the event queue, random number generators, statistics collection tools, etc. Second, there is a model component library that includes both a set of standard OMNeT++ components as well as any user-defined components. The simulation kernel interfaces with one of two environments; a graphical user environment, called TKENV, or a command line environment, called CMDENV. TKENV is useful for graphical debugging and visual demonstrations whereas CMDENV is more suited to running large batches of automated simulations. Finally, the last component is the simulation model which is executed by the simulation kernel and displayed by the environment.

Simulation Model

The simulation model is the primary component supplied by the user. OMNeT++ is built around the notion of modules, which are simple reusable components that can be extended by the user. Modules can be connected together and interact via message passing and they form the basis of the simulation model.

Modules

There are two types of modules; simple modules and compound modules. Simple modules are the basic building blocks of a simulation model and can be connected together to create compound modules. Each simple module corresponds to a C++ class that derives from OMNeT++'s cSimpleModule base class. While the C++ class determines how the module behaves, the modules themselves are defined using OMNeT++'s "Network Description Language" (NED). The NED description for a simple module contains two sections. First, is the parameters section which defines any configurable properties and settings that the module may have. Second, is the gates section which defines the "ports" that allow modules to exchange messages with other modules. Compound modules include several additional sections to define the submodules that they contain and the internal connections between submodules. Note that submodules may be either simple modules or other compound modules.

Finally, a compound module can be defined as a "network" which is the top level module that contains all of the other modules in the simulation model.

Channels

The connections between modules are called channels. The primary purpose of channels is to deliver messages between modules. While OMNeT++ is not specifically designed to simulate communication networks, it does provide several types of channels such as "DelayChannel" and "DatarateChannel" which are intended to represent physical links in a communication network.

Messages

Every event in simulation is represented by a message and all messages are instances of objects derived from the cMessage class that OMNeT++ provides. When a message arrives at a module, it uses the message properties to determine the type of event that occurs. Thus to simulate any event, a module must first send a message which causes the message to be placed on the event queue. To schedule an event to occur at a specific time, a module can send a message to itself and specify the time at which the message should arrive. A module can also trigger an event to occur in another module by sending a message out one of its ports. In this case, the module that receives the message is determined by what is on the other end of the channel connected to the module's outgoing port. Similarly, the time of message arrival depends on the type and properties of the channel and message. For example, OMNeT++ provides a cPacket class that extends cMessage. When it is sent over a DatarateChannel, the time of arrival is determined by the packet size and the channel's data rate property. New message types can also be created using OMNeT++'s message definition language. OMNeT++ automatically reads the resulting .msg files and produces the corresponding C++ class which the user may optionally extend.

Experiments

Since a user will often want to conduct many different experiments using the same simulation model, OMNeT++ enforces a strict separation of the simulation model from specific simulation experiments. The simulation model consists of all the NED files, message files, and C++ code for the modules and messages in the simulation. An experiment, by contrast, is defined by a special INI configuration file and is considered separate from the model. The configuration file specifies which network module to

use, how the various module parameters should be set, which statistics to collect, as well as all of the general simulation parameters, such as the simulation run time, the warmup interval, the number of repetitions to perform, the seeds to use for the random number streams, and so forth.

A.1.3 INET framework

The INET framework [4] is a popular third-party network simulation package developed for the OMNeT++ environment. It provides models for many common protocols such as IPv4, IPv6, UDP, various TCP flavors, and many of the commonly used Ethernet standards. To minimize complexity and avoid simulating unnecessary details, we did not build our simulation model directly on top of the INET models. Instead, we primarily use the INET code as a reference for our own implementation and to leverage some of the constants and properties that it already defines (e.g., sizes of fields in protocol headers). However, we did write an interface to support INET's implementation of TCP in our model but we did not use it for any of the simulations that we performed in this thesis.

A.2 Simulator Overview

Our data center network simulator consists of the C++ source, NED and message files, and a set of support utilities to create a light-weight OMNeT++ simulation model for FatTree DCNs. Here we provide a brief overview of the simulator and its components and provide a high-level view of how we modeled FatTree DCNs.

Dependencies

Our project uses OMNeT++ version 4.3 which is free for non-commercial use and runs on Windows, OS X, and most Linux/Unix systems.

Some of the modules in our code also depend on the following:

- INET 2.0.0 [4]. Note that to create an OMNeT++ project using our code, INET must be included in the project's workspace and listed as a project reference.
- Boost C++ libraries [1]. We used boost version 1.52 and we link our code against the boost_system and boost_filesystem libraries.

Directory structure

The root directory consists of three folders:

- **src:** the root folder for all of the simulation modeling files (i.e., NED files and their corresponding C++ code)
- **BuildFatTree:** contains a simple command line application to generate NED files for FatTree topologies.
- **simulations:** contains ini files for several example experiments as well as some python scripts for results processing.

The structure of the src folder containing the model files is as follows:

- **common:** includes commonly used headers and also houses the base classes that are common to multiple parts of the project.
- **networks:** this folder includes the NED files defining the data center topologies.
- **node:** all of the server code is kept here and its subfolders correspond to the layers of the networking stack.
- **packets:** contains the OMNeT++ message definitions and C++ source that define all the packets and control messages used.
- **switch:** contains the components that make up the network switches.

A.2.1 Modeling FatTree DCNs

Here we briefly describe our representations for the various elements that make up our FatTree DCN simulation model.

Packets

Since we did not use the INET models for protocols like Ethernet and IP, we model these protocols by creating simple message types that capture their essential behavior. We created the following message definitions to represent the different packet types for each protocol:

- DCN_EthPacket: We extended OMNeT++'s cPacket type to model Ethernet packets. We customized the automatically generated C++ class produced by the message definition so that we could account for the minimum payload and the various header fields in the Ethernet frame when setting the packet size. We assumed the standard MTU of 1500 bytes to determine the maximum size of Ethernet frames.
- DCN_IPPacket: We extended the DCN_EthPacket type to create a message type for IP packets. It adds 20 bytes to account for the size of the IP header fields. In addition, we assumed that we could overload existing IP options fields in order to account for the sequence numbers and timestamps used by our resequencing approach. As a result, we added an additional 8 bytes to the IP overhead.
- DCN_UDPPacket: To represent UDP datagrams, we extended the DCN_IPPacket type. It simply extends the packet size accounting logic to include the overhead of the UDP header.

We did not create a separate TCP packet type since we rely on INET's TCP model to support TCP which already defines its own TCPSegment type. To support this message type in our framework, we use the cMessage class's encapsulate/decapsulate feature to transport TCPSegments inside DCN_IPPackets.

Links

To model network links, we created a DCLink class which is an extension of the

standard OMNeT++ cDatarateChannel class. This class automatically simulates transmission delay by using the data rate of the link and the packet's size field. The links can support any bit rate but we primarily used a value of 1 Gbps for all of the links in the network. The links can also simulate propagation delay as well as channel noise by introducing random bit errors. However, we did not use these features in any of our simulations.

Switches

We assume switches are store and forward switches (i.e., no cut-through switching). We model the switches as ideal output-buffered switches by placing a queue in front of each port and by directing packets into the queues corresponding to their destinations as soon as they arrive. The routing logic at switches is hardcoded specifically to route packets across the FatTree topology. While there are a variety of different approaches to construct FatTree DCNs from existing switches, we do not tie our model to any particular approach and only assume that some mechanism exists to allow servers to choose paths for their packets. We model this by including a "path" field in the DCN_EthPacket class that can be set by a server enabling it to indicate to the switches which path the packet should take. Dropping at switches occurs when the output queue grows beyond the defined limit, which can be set in terms of bytes, packets, or both.

Servers

Since this thesis focused on software-level mechanisms, the majority of the complexity of our simulation model resides in the server nodes. We structured the server nodes according to the layers of an operating system's networking stack. We devote section A.3 to describing each of the key modules that make up our server model.

Networks

In the network folder, we provide a set of NED files (SubFatTree.ned, FatTree.ned, and ServerNode.ned) to recursively construct FatTree networks by generating a compound module to represent each subtree. However, we found that this approach was not very efficient because each time a packet traverses the network, it must be placed on the event queue every time it passes between the compound modules corresponding to different subtrees. We found that it was far more efficient to define one network compound module that directly contains all of the switches and servers as submodules.

As a result, we wrote a stand alone application called *BuildFatTree* to automatically produce NED files for different FatTree networks. While the link speed is configurable, the program only produces fully provisioned FatTree networks (i.e., full bisection bandwidth) that are constructed from links of the same speed. The program also supports the ability to produce the logically equivalent tree topology for a given FatTree network, as described in Chapter 3.

A.2.2 BuildFatTree

BuildFatTree application:
The BuildFatTree application is written in C++ and can be built by entering the BuildFatTree folder and typing "make". Running the application without any arguments will cause its usage to be displayed. Once a NED file is produced, it should be placed in the *src/networks* folder so that OMNeT++ can find it. Note that the OMNeT IDE attempts to index all C++ and NED source files which can cause it to slow down dramatically when processing large files. For this reason, the generated NED files for large networks should not be placed in the working directory or the IDE should be instructed not to look in the path containing these files.

A.3 Server components

In this section we provide some more information about the various modules that make up the server nodes. All of these modules are found in the *src/node* directory. With the exception of the server folder (described below), the structure of this directory reflects the various layers of the network stack of servers.

A.3.1 Server

Sever nodes are compound modules that implement the Server interface defined by Server.ned. The Server interface is simple. Only one parameter must be specified,

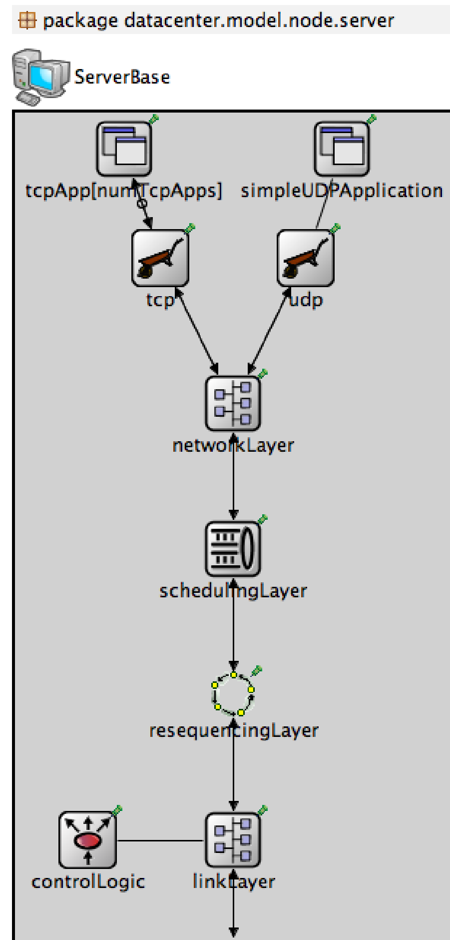


Figure A.1: The compound module representing a server.

the address, it interfaces with the network through the port gate. The ServerBase module defines the structure of the server node and is shown in Figure A.1.

A.3.2 Control module

Allows exchanging messages between modules without affecting the simulation. As opposed to keeping a memory reference or using the direct connection, the motivation behind the use of a separate control module was two fold. First, messages can be addressed to specific modules in other servers using their names. Delivery details handled automatically making it easier to exchange information. Second, it preserves the ability of the model to be distributed.

A.3.3 Application Layer

While there were several application level modules that we developed, we primarily used one which we describe here.

Message Application

The message application is designed to send fake data representing generic application-layer messages. We used the message application to generate many of the traffic patterns that we used in this dissertation. These messages are read in from a script file. The message application at each of the server nodes in the simulation read through the script to determine the messages that they have to send to other servers.

An example of the message script format is shown below:

```
# Example script file 2012-04-13 16:53:12
# The message script format is:
# <source>;<destination>;<messageNum>;<startTime>;<messageSize>;<messageRate>
      0   ;      1      ;     1      ;    0 s    ;    512 KiB  ;  100 Mbps
      0   ;      1      ;     2      ;  500 ms   ;    3.2 MiB  ; 3 Mbps + 100 kbps
      1   ;      2      ;     1      ;   1.5 s   ;    1 GiB    ;    0 bps
```

Each message has a source and a destination identifying the sending and receiving server respectively. The messageSize indicates the number of application-level bytes in the message. The startTime indicates when the sending server should begin sending the message. Once the startTime is reached, the sending server will begin sending the message by generating UDP packets whose size is determined by the MessageApplication's payloadSize parameter. The actual packets produced by the application are therefore larger than the payloadSize. If the messageRate parameter has a non-zero value, then the packets will be produced at intervals to match the specified rate. Note that this rate is in terms of the raw Ethernet sending rate and does account for the protocol overheads. If no messageRate is specified (i.e., the value is 0 bps) then the maxSendRate parameter determines when the next packet will be sent. The messsageNum field is optional and is simply intended to make it easier to

differentiate between different messages between the same pair of servers. The MessageApplication automatically handles unit conversion and mathematical expressions by leveraging the parser that OMNeT++ uses to process ini files.

The MessageApplication can also instantiate a "ScriptGenerator" class. The ScriptGenerator class is used to generate script files for different traffic patterns according to the "scriptGenType" parameter. When this feature is used, only the first server (with address 0) generates the script file during the first stage of module initialization. The rest of the servers simply read the script file during the second stage of module initialization.

A.3.4 Transport Layer

This folder contains modules to support a simplified UDP protocol and several modules to interface with INET's TCP models. The UDP module is called SimpleUDP and simply handles encapsulation and decapsulation of DCN_UDPPackets.

A.3.5 Network Layer

The purpose of the network layer is simply to multiplex and demultiplex traffic for the transport layer modules. Currently, there is only one type of network layer which is called VirtualIP. While fairly simple, it does have to interact with the InetTCPWrapper class to decapuslate and encpauslate TCP packets when TCP is used.

A.3.6 Scheduling Layer

The scheduling layer is one of the more complex layers. As shown in Figure A.2, the scheduling layer is a compound modules that contains several simple modules. The demultiplexer module ("demux") accepts traffic arriving from the network layer. The multiplexer module ("mux") forwards traffic down to the next lower layer in the stack. The multiplexer is also responsible for notifying the controller when a packet arrives for a destination for which there is currently no VOQ. Traffic can also enter into the

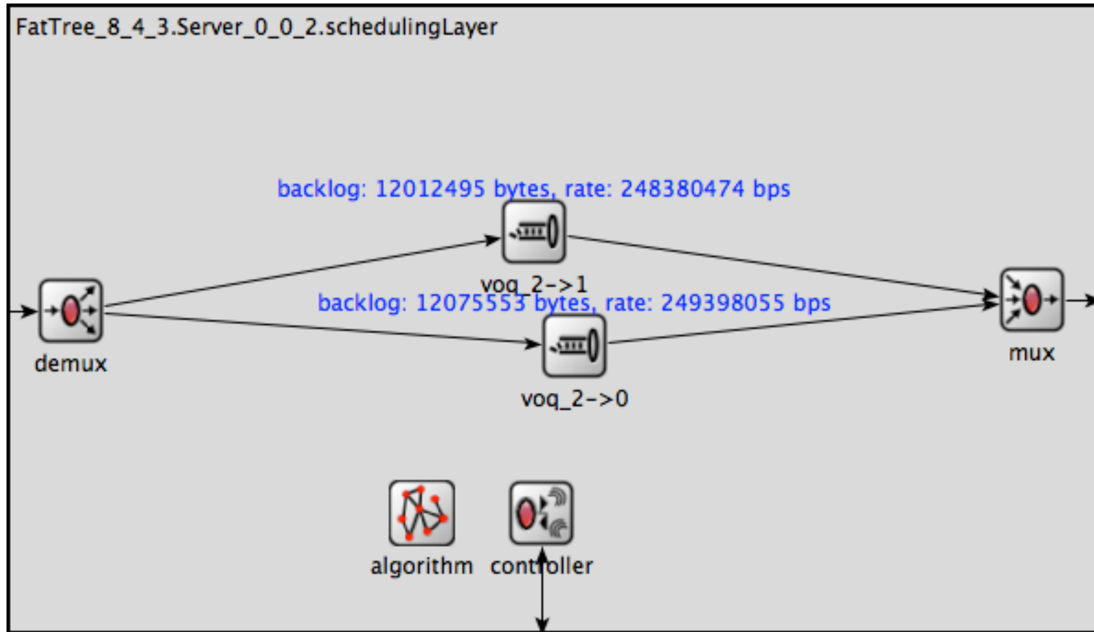


Figure A.2: The compound module representing the scheduling layer.

scheduling layer via the controller module which filters out scheduling packets and delivers them to the SchedulingAlgorithm module (shown simply as "algorithm").

We provide two different scheduling algorithm modules. The first can be found in the "proxy" subfolder and corresponds to the proxy algorithm described in section 4.3. The other module is called "schedFromeFile" which is found in the subfolder of the same name. This module allows rates to be read in and set from a rate schedule script. The format of the rate schedule script mirrors the format of the backlog scheduling script described in section 5.6.1. The primary purpose of this module is to provide a way to test the scheduling framework in the simulator.

A.3.7 Resequencing Layer

The resequencingLayer folder contains the implementation of our hybrid resequencing approach described in section 3.4. While we implemented only one approach to resequencing, the Resequencer.ned file defines an interface for reseqeuencers that other all resequencers must implement. The HybridResequencer module is a compound module that implements the Resequencer interface and represents our hybrid resequencer

approach. The HybridResequencer dynamically allocates ResequencerBuffer modules when a packet arrives from a sender for which there is currently no buffer. The modules are removed according to a timeout. The HybridResequencer module also adds sequence numbers to outgoing packets and demultiplexes incoming packets into the appropriate ReseqeuncerBuffer instance. The ResequencerBuffer modules actually handle all of the complexity of queueing and releasing packets based on sequence numbers and timestamps.

A.3.8 Link Layer

The LinkLayer compound module contains several types of submodules. First, it contains the ServerPortQueue module which is an extension of OMNeT++'s Queue container and it buffers packets as they are sent into the network. The rxMeter and txMeter are submodules that can optionally be enabled to measure and produce statistics for the incoming and outgoing server bandwidth respectively. The LoadBalancer.ned file contains all of the various approaches to load balancing described in chapter 3. Finally, path modules are dynamically allocated to represent each of the paths that the server can send to. The path modules are simply used to collect statistics about the number of bytes each server places on each path.

List of load balancers

- ECMP: Equal Cost Multi-Path load balancer randomly hashes flows to paths based on their source and destination address.
- VLB: Valiant Load Balancing randomly assigns each packet to a path.
- RR/P-RR: Round Robin & permutation round robin load balancers described in section 3.2.2.
- SRR/P-SRR: Surplus Round Robin load balancer described in section 3.2.2.
- SD/D-SD: Sorted-Deficit load balancer described in section 3.2.2.
- TP/P-TP: Two-Phase load balancer described in section 3.2.3.

- MP/P-MP: Multi-Phase load balancer described in section 3.2.3.

Appendix B

DCN Queueing Models

B.1 Overview

This appendix describes the analytical models that we developed to analyze the performance of load balancing in FatTree DCNs. Our basic approach depends on modeling the network as a series of M/M/1 queues with finite capacity. Before describing our approach in more detail, we briefly review some of the necessary queueing theory concepts below.

B.2 Basic queueing theory concepts

Queueing theory has often been applied to model packet-switched communication networks. Kendall [33] introduced some notation that has been widely adopted to describe the basic model of a queue. This notation is most often seen in the form A/B/C where A describes the arrival process, B the departure process, and C the number of "servers" that serve the queue. We will not use the term "servers" or their "service times" in relation to queues in order to avoid confusion with the word referring to the physical server machines in the data center network. In our context, C is always 1 because a queue models the buffering of packets being transmitted on a link. In this work, we fill focus on the M/M/1 queue, which is the classic model of a queue.

B.2.1 M/M/1 Queue

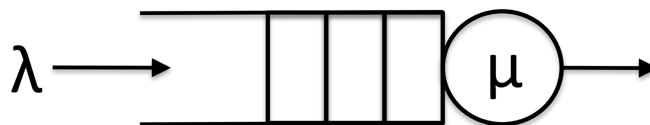


Figure B.1: A simple M/M/1 queue.

Figure B.1 provides a simple representation of the M/M/1 queue. The arrival of packets at the queue are assumed to follow a Poisson process, specified with the parameter λ. This means that if packets arrive at an average rate λ, their arrival times are exponentially distributed around $\frac{1}{\lambda}$. Similarly, the transmission time, that is the time it takes to transmit a packet, is also assumed to be exponentially distributed so the transmission process is also Poisson and specified with μ. Allowing both processes be Poisson enables us to model the number of packets in a queue as a simple birth-death process. This means we can construct a Markov chain to compute their steady state probabilities, which is the reason M is used in the notation.

To summarize the relevant properties of an M/M/1 queue:
λ is the arrival rate
μ is the transmission rate
$\rho = \frac{\lambda}{\mu}$ is the **traffic intensity** (also called duty factor)
p_k is the steady state probability of their being k packets in the system

$$p_k = (1 - \rho)\rho^k \tag{B.1}$$

B.2.2 M/M/1/K Queue

The M/M/1 queue is assumed to have infinite capacity and, as a result, the throughput, T, is simply λ if $\rho < 1$. To accurately model throughput in a packet switched network, we need to capture the fact that queues have finite capacity. The model that does this is the M/M/1/K queue, where K is used to denote capacity. An M/M/1/K queue has space for K packets, and this value includes any packet that is being transmitted. To find the throughput we need to calculate the probability of the queue being full when a packet arrives. In other words, we need the steady state probability

p_K. As before, the steady state probabilities can be computed from the Markov chain. The only difference with the M/M/1 case is that the chain is finite [34]. Since the probability of a packet being dropped is p_K, the probability that it gets through is $1 - p_K$. So the throughput T, in terms of packets is $T = \lambda(1 - p_K)$.

To summarize the relevant properties of an M/M/1/K queue:
K is the number of packets that can be stored in the queue plus the packet being transmitted.
p_k is the steady state probability of their being k packets in the system.
p_K is the probability of a packet being dropped.

$$p_K = \frac{(1-\rho)\rho^K}{1-\rho^{K+1}} \tag{B.2}$$

$T = \lambda(1 - p_K)$ is the expected throughput in terms of packets.

B.2.3 Modeling a network of queues

As it turns out, the memoryless property of the Poisson process makes it easy to analyze a network of M/M/1 or M/M/1/K queues. Part of Burke's Theorem [17] states that the output process of an M/M/1 queue with input parameter λ and output parameter μ generates a Poisson output process at rate λ. This means that if we were to place two M/M/1 queues in series, they would behave identically to two separate M/M/1 queues with input parameter λ. Similarly, with two M/M/1/K queues in series, the input parameter λ at the downstream queue must match the throughput T of the upstream queue. This mutual independence allows us to analyze a network of queues by considering each queue separately and to analyze a given queue we only need to determine its input rate λ. Figure B.2 shows how we can calculate these rates in a network of queues by considering the splits and joins in the topology. Since the sum of two Poisson processes with parameter λ_A and λ_B is another Poisson process with parameter $\lambda_A + \lambda_B$, we can divide any fraction of an upstream queues throughput among down queues. Thus we can split traffic from a queue or merge traffic from multiple queues as long as the net rate between a set of queues is preserved as indicated in the figure.

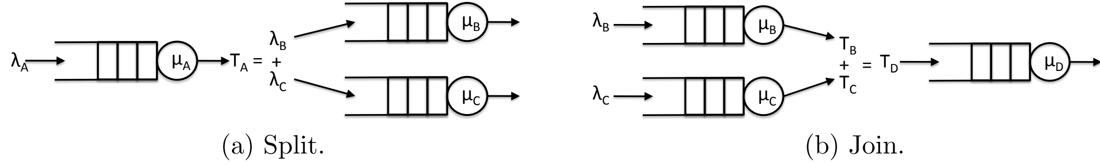


Figure B.2: Splitting and joining traffic at M/M/1 queues.

B.3 M/M/1/K FatTree

Using the basic approach described above, we now explain our M/M/1/K model for the data center network. The basic idea is that we can replace each switch with a set of M/M/1/K queues. By placing one queue in front of each link, the queues essentially capture the behavior of an ideal output queued switch. We can then view each server as a traffic source that emits packets as a Poisson process with a rate λ, which corresponds to the offered load. If we assume that the destination and path of each packet is random, then we have a model that represents an all-to-all traffic pattern with random load balancing (i.e., VLB). Since the traffic is symmetric, the behavior of all down queues or all up queues at a given level will be identical. Thus if we analyze the queue at a downward facing port at level 1 (a port connected to a server), then we can characterize the throughput or loss that the average flow will experience. Because the queues have a finite capacity K, we can also use this model to understand the relationship between switch queue size, loss, and offered load.

Note that this model is only approximate. For example, traffic on a link in real network can never exceed the speed of the link so the arrival of packets at queues will not truly be Poisson, particularly at rates close to the speed of the link. Additionally, the departure times at M/M/1/K queues are independent of the arrival times but in a real network, this is not the case. This means that this queueing theory model most accurately represents the case where packets are exponentially distributed around some mean size.

To determine the throughput/loss of the average flow, we need to find the rate of arriving traffic at a downward facing queue at level 1. To do this, we need to determine how traffic arriving at a given port is split among the other ports at the switch. We can use the fact that since each server sends to every other server, the fraction of a

server's traffic destined for a given subtree is equivalent to the fraction of the total number of servers contained in that subtree. We can then leverage the recursive structure of the FatTree to express these rates in terms of the traffic at queues at other levels of the tree. We detail all of these expressions that we derived below.

Let $\lambda_D(i)$ be the rate of arriving traffic at a down port at level i
Let $\lambda_U(i)$ be the rate of arriving traffic at a down port at level i
Let $T_U(i)$ be the throughput at an up port at level i.
Let $T_D(i)$ be the throughput at a down port at level i.

Let $S(i)$ be the number of servers in the subtree at level i.

$$S(i) = \begin{cases} (\frac{k}{2})^i & \text{if } i < l \\ k(\frac{k}{2})^{l-1} & \text{if } i = l \end{cases} \tag{B.3}$$

Let $n = S(l)$ be the number of servers in the network.

Let $D(i)$ be the number of down ports at a switch at stage i.

$$D(i) = \begin{cases} \frac{k}{2} & \text{if } 0 < i < l \\ k & \text{if } i = l \end{cases} \tag{B.4}$$

Let $U(i)$ be the number of up ports at a switch at stage i.

$$U(i) = \begin{cases} \frac{k}{2} & \text{if } 0 < i < l \\ 0 & \text{if } i = l \end{cases} \tag{B.5}$$

Let $S_D(i)$ be the number of servers reachable on a down port at level i.
Let $S_U(i)$ be the number of servers reachable on up ports at level i.

$$S_D(i) = \frac{S(i)}{D(i)} \tag{B.6}$$

$$S_U(i) = n - S(d) \tag{B.7}$$

Let $f_{D \to D}(i)$ be the fraction of traffic arriving at a down port that is routed to another down port at level i.

Let $f_{D \to U}(i)$ be the fraction of traffic arriving at a down port that is routed to an up port at level i.

$$f_{D \to D}(i) = \frac{S(i-1)}{n - S(i-1)} \tag{B.8}$$

$$f_{D \to U}(i) = \frac{S_U(i)}{n - S(i-1)} \tag{B.9}$$

Let $\lambda_{D \to D}(d)$ be the rate from one down port to another.
Let $\lambda_{D \to U}(d)$ be the rate from a down port to an up port.
Let $\lambda_{U \to D}(d)$ be the rate from an up port to a down port.

$$\lambda_{U \to D}(i) = \frac{T_U(i+1)}{D(i)} \tag{B.10}$$

$$\lambda_{D \to D}(i) = T_U(i-1) F_{D \to D}(i) \tag{B.11}$$

$$\lambda_{D \to U}(i) = T_U(i-1) \frac{F_{D \to U}}{U(i)} \tag{B.12}$$

We can now express λ_D and λ_U in terms of the traffic coming from up ports and down ports.

$$\lambda_U(i) = \lambda_{D \to U}(i) D(i) \tag{B.13}$$

$$\lambda_D(i) = (D(i) - 1)\lambda_{D \to U}(i) + U(i)(\lambda_{U \to D}(i)) \tag{B.14}$$

$$T_U(i) = \lambda_U(i) * \text{mm1k_throughput}(\lambda_U(i), K) \tag{B.15}$$

$$T_D(i) = \lambda_D(i) * \text{mm1k_throughput}(\lambda_D(i), K) \tag{B.16}$$

mm1k_throughput(ρ, K) is computed as follows:

$$\text{mm1k_throughput}(\rho, K) \begin{cases} \frac{(1-\rho)\rho^K}{1-\rho^{K+1}} & \text{if } \rho < 1 \\ \frac{1}{K+1} & \text{if } \rho = 1 \end{cases} \tag{B.17}$$

Now that we have an expression for the throughput for any port at a given level, we can compute $T_D(1)$, the throughput of a downward facing port at level 1. This is equivalent to the throughput of the average flow and $1 - T_D(1)$ is equivalent to the expected loss of the average flow. Since these values depend on the queue size K, the offered load λ, and the dimensions of the FatTree (k and l), we can use these expressions to determine the offered load at which the loss exceeds a given threshold with a fixed queue size or the queue size needed to remain under a fixed loss threshold for a given offered load. We wrote a python script to carry out these calculations which we used to produce the results shown in section 3.3.3.

B.4 M/M/1/K LogicalTree

To model the logically equivalent tree for a FatTree network, we can use the exact same idea. However, for the links in the logical tree, we scale the sizes of their queues in proportion to the number of links that they represent in the FatTree. The number of FatTree links that a link at level i represents turns out to be equivalent to the number of servers in a subtree at level i divided by the number of down ports at a switch at level i.

This means that the only expressions that are different in the logical tree representation are:

$$T_U(i) = \lambda_U(i) * \text{mm1k_throughput}(\lambda_U(i), K\frac{S(i)}{D(i)}) \tag{B.18}$$

$$T_D(i) = \lambda_D(i) * \text{mm1k_throughput}(\lambda_D(i), K\frac{S(i+1)}{D(i+1)}) \tag{B.19}$$

References

[1] boost. `http://www.boost.org`, March 2013.

[2] Cplex. `http://ibm.com/`, March 2013.

[3] Cvxopt. `http://abel.ee.ucla.edu/cvxopt/`, March 2013.

[4] Inet framework. `http://inet.omnetpp.org`, February 2013.

[5] ns-3. `http://http://www.nsnam.org/overview/what-is-ns-3/`, March 2013.

[6] Numpy. `http://www.numpy.org`, March 2013.

[7] Omnet++. `http://www.omnetpp.org`, March 2013.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *7th Symposium on Networked Systems Design and Implementation (NSDI 2010)*, 2010.

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008.

[10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *SIGCOMM '10: Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, pages 63–74, New York, NY, USA, 2010. ACM.

[11] Amazon. Amazon elastic compute cloud (amazon ec2), 2011. `http://aws.amazon.com/ec2/`.

[12] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–16. USENIX Association, 2010.

[13] H. Ballani, D. Gunawardena, and T. Karagiannis. Network sharing in multi-tenant datacenters.

[14] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM '11: Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, 2011.

[15] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, 2010.

[16] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[17] Paul J. Burke. The output of a queuing system. *Operations Research*, 4(6):pp. 699–704, 1956.

[18] Anna Charny and Raj Jain. Congestion control with explicit rate indication. In *PROC. ICC'95*, pages 1954–1963, 1995.

[19] Y. Chen, R. Griffith, J. Liu, R.H. Katz, and A.D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82. ACM, 2009.

[20] C. Clos. A Study of Non-blocking switching networks. *Bell Syst. Tech. J.*, 32:406–424, 1953.

[21] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM '11: Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, 2011.

[22] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[23] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] A. Dixit, P. Prakash, and R.R. Kompella. On the efficacy of fine-grained traffic splitting protocolsin data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 430–431. ACM, 2011.

[25] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[26] 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 *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, New York, NY, USA, 2009. ACM.

[27] Albert Greenberg, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 57–62, New York, NY, USA, 2008. ACM.

[28] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 39(4):63–74, 2009.

[29] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4):75–86, 2008.

[30] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. Case study for running hpc applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 395–401, New York, NY, USA, 2010. ACM.

[31] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[32] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *CLUSTER*, pages 116–125, 2008.

[33] David G. Kendall. Some problems in the theory of queues. *Journal of the Royal Statistical Society. Series B (Methodological)*, 13(2):pp. 151–185, 1951.

[34] Leonard Kleinrock. *Queueing Systems. Volume 1: Theory*. Wiley-Interscience, 1975.

[35] E. Krevat, V. Vasudevan, A. Phanishayee, D.G. Andersen, G.R. Ganger, G.A. Gibson, and S. Seshan. On application-level approaches to avoiding tcp throughput collapse in cluster-based storage systems. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 1–4. ACM, 2007.

[36] Terry Lam and George Varghese. Netshare: Virtualizing bandwidth within the cloud. Not published?, February 2009.

[37] Myungjin Lee, Sharon Goldberg, Ramana Rao Kompella, and George Varghese. Fine-grained latency and loss measurements in the presence of reordering. In *SIGMETRICS 2011, San Jose, CA*, June 2011.

[38] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, 100(10):892–901, 1985.

[39] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th annual conference on Internet measurement*, pages 1–14. ACM, 2010.

[40] Dan Li, Chuanxiong Guo, Haitao Wu, Kun Tan, Yongguang Zhang, and Songwu Lu. Ficonn: Using backup port for server interconnection in data centers. In *Proceedings of Infocom 2009*, April 2009.

[41] Santosh Mahapatra and Xin Yuan. Load balancing mechanisms in data center networks. In *the 7th International Conference and Expo on Emerging Technologies for a Smarter World (CEWIT)*, September 2010.

[42] J.C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review*, 42(5):44–48, 2012.

[43] D. Nagle, D. Serenyi, and A. Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53. IEEE Computer Society, 2004.

[44] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. *SIGCOMM Comput. Commun. Rev.*, 39(4):39–50, 2009.

[45] Rong Pan, Balaji Prabhakar, and Ashvin Laxmikantha. Qcn: Quantized congestion notification. *IEEE802*, 1, 2007.

[46] Prashanth Pappu, Jyoti Parwatikar, Jonathan Turner, and Ken Wong. Distributed queueing in scalable high performance routers. In *in Proceedings of IEEE Infocom*, 2003.

[47] Prashanth Pappu, Jonathan Turner, and Ken Wong. Work-conserving distributed schedulers for terabit routers. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 257–268, New York, NY, USA, 2004. ACM.

[48] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. Technical report, Carnegie Mellon University, 2007.

[49] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM '11: Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, 2011.

[50] V.S. Rajanna, S. Shah, A. Jahagirdar, and K. Gopalan. Xco: Explicit coordination for preventing congestion in data center ethernet. In *Proc. of International Workshop on Storage Network Architecture and Parallel I/Os*, 2010.

[51] A. Scicchitano, A. Bianco, P. Giaccone, E. Leonardi, and E. Schiattarella. Distributed scheduling in input queued switches. In *IEEE ICC*, 2007.

[52] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: performance isolation for cloud datacenter networks. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[53] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the data center network. In *To appear in Proc. of NSDI, April 2011.*, 2011.

[54] Ankit Singla and Chi-Yao Hong. Distributed traffic engineering in data center networks. Technical report, University of Illinois at Urbana Champaign, 2011.

[55] Jonathan Turner. Resilient cell resequencing in terabit routers. In *Proceedings of the Allerton Conference on Communication, Control and Computing*, 2003.

[56] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 263–277, New York, NY, USA, 1981. ACM.

[57] A. Varga et al. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, volume 9. sn, 2001.

[58] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D.G. Andersen, G.R. Ganger, G.A. Gibson, and B. Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. *ACM SIGCOMM Computer Communication Review*, 39(4):303–314, 2009.

[59] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM '11: Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, 2011.

[60] T. Wood, A. Gerber, KK Ramakrishnan, P. Shenoy, and J. Van der Merwe. The case for enterprise-ready virtual private clouds. *Usenix HotCloud*, 2009.

[61] X. Yuan. On nonblocking folded-clos networks in the computer communication environment. In *25th IEEE International Parallel & Distriburted Processing Symposium (IPDPS)*, 2011.

Improving Data Center Network Performance, Haitjema, Ph.D. 2013