Achieving Consistent Latencies in Datacenter Networks

David Zats

December 19, 2014

Achieving Consistent Latencies in Datacenter Networks

by

David Zats

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

in

Engineering — Electrical Engineering and Computer Sciences

in the

Graduate Division
of the
University of California, Berkeley

Fall 2014

Achieving Consistent Latencies in Datacenter Networks

Abstract

Achieving Consistent Latencies in Datacenter Networks

by

David Zats

Doctor of Philosophy in Engineering — Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Randy Katz, Chair

In this thesis, we begin by analyzing the increasing trend of running large-scale services on Warehouse Scale Computers to provide user functionality. We see that user demands for interactivity place stringent requirements on these services, compelling them to consistently meet tight deadlines. To provide users a rich experience while meeting these deadlines, services typically parallelize requests, dividing tasks among many servers to speed-up computation. Consequently, services depend on the network to provide consistent, low-latency communication so that servers can coordinate and deliver their results in a timely manner.

We explain why current networks are poorly suited for delivering consistent, low-latency performance. These networks typically run the TCP/IP stack, which explicitly trades consistent performance for generality and interoperability. This decision was appropriate for the Internet, which was designed to interconnect heterogeneous networks. However, given the new stringent requirements that these networks must support, we hypothesize that they can no longer accept this design decision. Instead, they must employ a series of tightly-integrated layers to deliver the consistent latencies services depend on.

To understand the benefits of foregoing generality, we design and implement two approaches *DeTail* and *FastLane*. DeTail is a network stack that employs a series of tightly integrated layers. Each layer in the *DeTail* stack depends on, and overcomes the limitations of, others, reducing the latency spikes that often plague networks. *FastLane* employs an alternate approach, where network components directly communicate with transports, helping them respond more agilely to events that drive up latencies.

To evaluate these approaches, we employ both simulation and implementation platforms. Based on these platforms, we see that both *DeTail* and *FastLane* are able to dramatically reduce latencies by foregoing generality and having the layers of the network stack directly depend on the functionality of others. We argue that this approach is a fruitful one and will become increasingly important as network requirements become increasingly stringent.

To my wife and family

Contents

List of Figures

List of Tables

Acknowledgements

I would like to acknowledge my advisor, Randy Katz, for all his help and support in finding and solving exciting and challenging research problems. His backing made exploring both unconventional problems and unique solutions possible. In addition, I would like to thank my committee for their guidance throughout this process. Both Ion Stoica and John Chuang provided useful feedback that dramatically enchriched the quality of the research.

I am also grateful for the contributions of my collaborators and colleagues to various chapters of this work. This research benefitted tremendously from help and enthusiasm of Tathagata Das, Prashanth Mohan, Ganesh Ananthanarayanan, Rachit Agarwal, and Anand Iyer. I am also indebted to both Dhruba Borthakur and Amin Vahdat for providing their deep industry insight to help make this research more practical and impactful.

My graduate school experience has also benefitted immensely from the valuable feedback I received on my research ideas and projects. Aurojit Panda, Shivaram Venkataraman, Andrew Krioukov, Sameer Agarwal, Mosharaf Chowdhury, Justine Sherry, Charles Reiss, Ariel Rabkin, and many others provided many important suggestions that not only strengthened my work, but also helped me to become a better researcher.

Finally, I would like to thank my wife and parents. This thesis would not have been possible without all of for their love, support, and encouragement.

Chapter 1

Introduction

Today, more applications are being provided to users through *Warehouse-Scale Computing* [40]. *Warehouse-Scale Computing* employs large-scale datacenters, consisting of hundreds of thousands of machines to provide Internet-Scale services. Users access these services remotely (e.g., through a web browser), obtaining access to the functionality they provide. While users typically expect the response times experienced with locally-running applications, these services may be tasked with performing operations far exceeding the computational and / or storage capacity of individual servers. To provide high-quality results while meeting tight user-expected response times, services parallelize the computation of every request, spreading processing across thousands of servers.

In this environment, predictably meeting these tight response times, termed *user-interactivity deadlines*, is critically important. Measurement studies have demonstrated the high financial cost of missing user-interactivity deadlines, indicating that users both perceive and react negatively to services with delayed response-times [44]. At the same time, the high degree of parallelism employed by services to speed up computation makes it challenging to consistently meet these deadlines as just one slow or faulty component can stall the whole computation. Fortunately, unpredictability in server computation time has been largely mitigated through backup tasks that are launched on different servers when the the original ones stall [21, 57].

However, achieving predictable network latencies continues to be a challenge. As networks represent a shared environment, congestion from too many servers transmitting at the same time quickly, and unpredictability, drives up latencies. Launching backup tasks is not an appropriate solution as doing so only creates additional resource demands, further aggravating congestion events. At the same time, measurement studies indicate the unpredictable network latencies, alone, can lead to missed deadlines [18] and hence represent a limiting factor in the drive towards richer services. In this work, our goal is to achieve tighter, more predictable network latencies so fewer deadlines are missed, improving the user experience.

We begin by describing *Warehouse-Scale Computing* and the reasons for this trend, the structure and types of Internet Services, and why meeting user-interactivity deadlines is critically important. We then describe the challenges involved in achieving predictable network latencies and the unique opportunities for doing so in datacenter networks. In this context, we present our research statement and our hypothesis, discussing how we intend to use tighter integration across network layers to achieve more uniform network latencies. We conclude with the roadmap for the rest of the thesis.

1.1 Warehouse Scale Computing

As described in [40], *Warehouse Scale Computing* is the practice of co-locating tens or hundreds of thousands of servers in one location, or warehouse. While co-location is not a recent phenomenon, what is new is the scale at which it is being performed. All of these servers are typically owned by *one entity* and they operate *together* to provide services to a large number of users, subject to interactive latency bounds. Currently, more functionality is moving from client-based applications to Internet-based services.

Many reasons have been cited for driving this trend [40]. Some services, such as web search require an ever-increasing number of servers to index the expanding Internet. In other cases, ease of use and manageability are key drivers. Users do not need to worry about managing their local platform, while companies can reduce the costs associated with providing legacy support and ensuring compatibility across a wide range of hardware. Furthermore, companies can employ statistical multiplexing, reducing per-user costs, simply by leveraging the fact that only a fraction of their customer base will use their services at any one point in time. These many powerful incentives suggest that this trend will continue.

1.2 Internet Services

The resources made available by *Warehouse Scale Computing*, in turn, have driven the creation of massive-scale Internet Services, spanning these large computation domains. Two common examples of web services are (i) customized page creation and (ii) web search. Both these services depend on the many servers of a datacenter collaborating together to provide a rich result to the user in a timely manner.

In the case of customized page creation, the contents of a page are customized for the individual user accessing the web site. For example, a Facebook page contains a News Feed, Search, Advertisements, and even a Chat application. The contents of each of these components depend on the user accessing the web site. The News Feed contains events recently reported by the user's friends while the Chat application reports which of the user's friends are currently online. This level of per-user customization limits the amount of precomputation that can be performed. Instead, when a user signs in, the contents of each component are processed in parallel by different servers in the datacenter. These results are then combined into a coherent whole.

On the other hand, in web search, a query is typically divided up into subqueries that are sent to individual servers [18, 56]. The individual servers process their portion of the query in parallel, reducing the total time necessary to perform the computation. Upon completion, they send their results to the aggregator which, as in the case of for customized page creation, combines them into a coherent whole. In both these cases, we see that thousands of servers must work together to address a single request.

1.3 Meeting user-interactivity deadlines

Meeting user-interactivity deadlines is both very challenging and critically important for Internet Services. With the large number of servers communicating and processing in parallel, there is a significant chance at least one will stall when processing every query. When this happens, all of the computation that depends on the delayed response will stall as well. By driving up the time taken to generate the complete result, these delays cause user-interactivity deadlines to be violated.

Violating user interactivity deadlines is costly. Prior studies have shown that users do not tolerate inflated web page response times. Amazon's experiments show that increasing page completion times by just 100ms leads to a 1% loss in sales while Google sees that increasing completion times by 500ms leads to a 20% loss in revenue [44]. As a result, web sites expend much effort on reducing service response times [50]. Some reductions are achieved by reducing the number of sequential dependencies required to perform the computation. This allows more of the computation to be performed in parallel, reducing the time taken by the service. However, the cost of missing these deadlines is so high that ultimately functionality and hence the richness of the response is sacrificed to ensure that deadlines are met.

To make matters more challenging, these deadlines must be met consistently. Even meeting deadlines 99.9% of the time means that one in a thousand users will experience poor performance [18]. This is a large number of customers for large-scale websites. Additionally, at least at one major online retailer, it is often the case that the more important customers (i.e., those with longer histories) will experience longer processing times and hence have a greater likelihood of missing their deadlines [32]. Thus the inability to consistently deliver a high-quality response in a timely manner to a small fraction of the user base could disproportionately affect the web site's revenue. These challenging demands drive modern-day websites to expend considerable effort on ensuring that user-interactivity deadlines are missed as rarely as possible.

One common way to reduce the impact of unpredictable server processing delays is to launch backup tasks, speculatively when it becomes evident that the original one has stalled [21, 57]. While this approach is well-suited for servers whose resources are relatively easy to isolate, it is poorly suited for the network. As the network represents a shared environment, sending extra traffic between the source and destination is likely to aggravate congestion events, forcing more delays, and perhaps even congestion collapse.

Isolating the traffic generated between different entities in the datacenter is extremely challenging. Not only may services compete with each other for network resources, but the servers participating in the same service may also contend, causing unnecessary delays [55]. Mechanisms that explicitly allocate resources between servers are wasteful. With static allocations, the servers needing extra resources are unable to obtain them even when others are not using their full allocation. In the absence of a simple, efficient isolation strategy, we must create a new set of techniques aimed at achieving more predictable network latencies in the presence of congestion.

1.4 Achieving predictable network latencies

Currently, servers competing for network resources can dramatically increase network latencies. Measurement studies from a production Microsoft datacenter show that the queuing delays caused by congestion can cause RTTs to increase by 40x [18]. When this happens, flows violate important transfer deadlines and stall the completion time of the query as a whole.

The problem is far greater than that indicated by the published measurements. These measurements do not include the time taken to retransmit dropped packets. In addition queuing delays, these packets have to wait for the sender to perform the retransmission, before arriving at the destination. As the sender may take a long time to discover that the drop has occurred, we expect these packets to experience far higher latencies.

Fortunately, in datacenter networks there are many opportunities for reducing network latencies. Modern datacenter networks have scaled out, leveraging many low-cost switches to increase aggregate bandwidth in a cost-effective manner. Many paths typically exist between a source and destination [15, 37]. Often, when a link is congested, packets can take alternate paths to reach the destination [24]. Additionally, the single administrative domain enables us to develop new mechanisms and employ coherent policies to address congestion-related issues.

However, the TCP/IP stack, which is typically run on today's datacenters [18], makes fundamental design decisions that limit its ability to effectively handle congestion and the resultant increase in network latency. Each layer minimizes the assumptions made about the others, focusing instead on mechanisms that attempt to ascertain the state of the network *indirectly.* For example, instead of assuming the network will inform it when congestion occurs, TCP relies on packet drops. While these decisions are well-suited for the Internet, which connects heterogeneous networks, they are poorly suited for the datacenter. By minimizing the assumptions employed by each layer, the TCP/IP stack foregoes many opportunities for responding more quickly and accurately to congestion. Since a quick, accurate congestion response can often dramatically reduce network latencies, we must take a different approach for datacenter networks.

1.5 Research statement and hypothesis

Based on this observation, we make the following hypothesis:

Foregoing generality, by having each layer of the network stack make explicit assumptions about, and pass information to, the others, allows us to respond to congestion more quickly and accurately, reducing high-percentile network latencies.

We explore two design alternatives that strive to achieve this goal. The first, *DeTail*, presents a tightly integrated network stack where the layers work together to overcome each others limitations. The second, *FastLane*, enlists switches to explicitly signal senders when packet drops occur, thereby avoiding the long time taken by traditional drop detection mechanisms. We see that both of these approaches are effective at reducing high-percentile flow completion times, supporting our hypothesis.

1.6 Thesis roadmap

In the following chapter, we begin by describing modern datacenter network topologies. We then discuss the basic assumptions employed by TCP/IP stack, explaining why they lead to poor high-percentile performance in these environments. Next we describe the tools available to us that we can leverage to improve network latencies. We conclude this section by highlighting prior proposals for datacenter networks and discussing where they fall short.

Chapter 3 describes the platforms we use to evaluate different datacenter networking proposals. Performing a thorough evaluation is challenging primarily because of the difficulty in reproducing effects only experienced at large scale. We describe the combination of testbed-based implementations and scaled up simulations we employ to address this issue.

Chapter 4 analyzes the impact of inflated high-percentile latencies on page creation. To perform this analysis, we compare a measured latency distribution from a production datacenter with a family of distributions having tighter latency bounds. Our analysis shows that high-percentile latencies have a far greater impact on page creation than the median. Based on these results, the remainder of our thesis focuses on proposing and evaluating mechanisms for achieving tighter latency bounds.

In Chapter 5, we present our first proposal, *DeTail* for reducing high-percentile latencies. We begin by describing three important causes of high latencies: (i) packet drops and retransmissions, (ii) uneven load balancing, and (iii) ineffective prioritization. We then propose a tightly integrated networking stack, which focuses on addressing all three problems. We demonstrate how these layers work together, overcoming each others limitations, and sharing information, to dramatically tighten latency bounds.

While *DeTail* is effective at reducing latencies for 1Gbps networks, we see that it requires more buffering than is typically available for 10Gbps networks. Additionally, *DeTail* can experience severe performance degradation when network components misbehave, with just one misconfigured switch or server being able to stall the whole network. Given the need to address these concerns for always-on Internet Services, in Chapter 6, we propose an alternate approach. We enlist the switches to inform the senders which packet was dropped whenever drops occur. We see that this approach has many of the same benefits as *DeTail* while avoiding its key limitations.

Chapter 7 concludes the thesis. We recap how our two proposals support our hypothesis that by having the layers of the network make more explicit assumptions about each other and passing information between them, we are able to achieve tighter latency bounds. We propose a series of next steps that can be taken to enrich this work and to further verify the viability of this approach.

Chapter 2

Related Work

To understand the challenges involved in achieving predictable network latencies, we first describe the current datacenter topologies where our network protocols are run. In this context, it becomes clear how the design decisions employed by the TCP/IP stack lead to inflated high-percentile latencies. Fortunately, unlike the Internet as a whole, the layers of a datacenter network stack are not restricted to making minimal assumptions about each other and inferring behavior indirectly. We describe the unique design opportunities available in datacenter networks that we can leverage to more effectively handle congestion. We conclude this chapter by listing current attempts to do so and describing where they fall short.

2.1 Datacenter Network Topologies

Originally, datacenter network topologies employed a tree structure with servers at the leaves. As depicted in Figure 2.1, switches interconnect these servers allowing them to communicate. As every node in the tree is responsible for supporting all of the the traffic between its children, it becomes increasingly difficult to support worst-case traffic demands higher in the tree.

Consider the case where all servers in the left half of the tree are sending traffic to the servers on the right half of the tree. We see that the links between the first level *Tors* (Top of Rack Switches) and the second level *Aggs* (Aggregate switches) will carry twice the load of the links connected to the servers. Furthermore, the links between the Aggs and the Core switch will carry four times the load of the links connected to the servers. The challenge is actually far greater than that depicted by this simple example. A single Tor will be connected to many more servers, leading to a much larger multiplicative factor in the load between the Tors and the Aggs. Similarly, many more Tors typically connect to a single Agg, resulting in an even larger multiplicative load factor higher in the tree.

Ideally, nodes higher in the tree would leverage links with increasing capacities to match these demands. In practice switches with high capacity links are far more expensive per

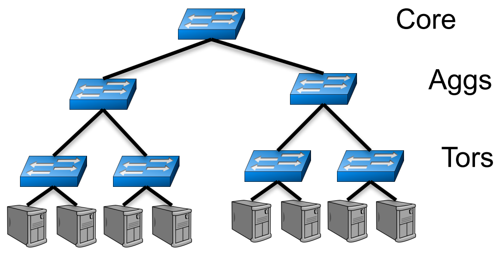


Figure 2.1: A traditional datacenter tree tropology.

Gbps than lower capacity ones [15]. To mitigate costs, datacenter networks resorted to high oversubscription rates, with switches at each hop having far more downstream bandwidth towards the servers than upstream bandwidth towards the Core. The resulting scarcity of cross datacenter bandwidth was a performance bottleneck, limiting the throughput of datacenter applications.

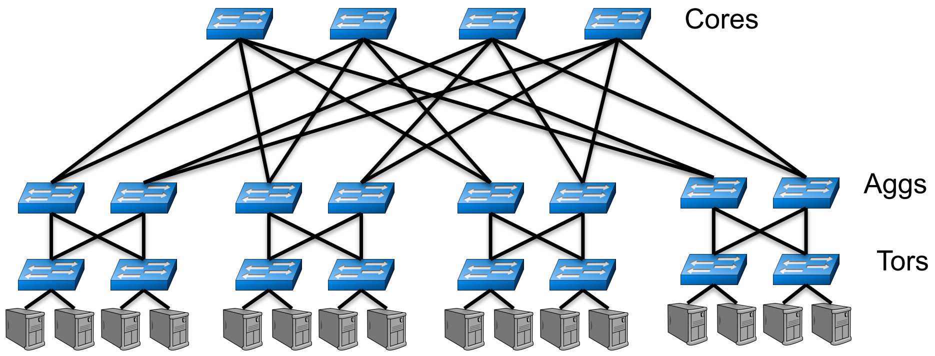


Figure 2.2: A scaled-out topology employing many lower-speed, lower-cost links to increase aggregate bandwidth.

To address this limitation, many new topologies scale out, employing many lower-capacity (and hence lower-cost) links between the source and destination in order to increase aggregate bandwidth. Of these topologies, the one that has gained the most traction is the folded-Clos approach advocated for by FatTrees and VL2 [15, 37]. We depict the specific topology proposed by FatTree in Figure 2.2. Unlike in a traditional, oversubscribed, tree topology, we see many more low-speed links as we go up towards the Cores forming the root of the tree. By taking this approach, these topologies enable datacenters to achieve far greater aggregate bandwidth in a cost-effective manner.

Many other datacenter topologies have been proposed, including DCell and Jellyfish [39, 54]. DCell employs a recursive structure to scale out the bandwidth available, while avoiding the need to use expensive high-speed switches. Jellyfish explores the bandwidth and latency advantages of connecting top of rack switches randomly. Regardless of the approach, all of these proposals seek to alleviate the bandwidth bottleneck that plagued traditional datacenters.

2.2 TCP/IP Design Decisions

Currently, datacenter networks typically run the TCP/IP stack [18]. As the TCP/IP networking stack was created for the Internet, many of its design decisions are poorly suited for datacenters. Here, we describe the TCP/IP networking stack and explain how its design decisions lead to suboptimal performance for datacenter networks.

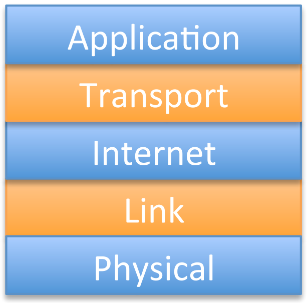


Figure 2.3: The TCP/IP networking stack.

The TCP/IP stack consists of the series of layers depicted in Figure 2.3. Going down the stack, we see that each layer depends on lower ones to provide smaller pieces of the functionality required to transmit a message from one application to another. The application layer contains the applications wishing to communicate. Each application generates messages and passes them to the transport layer, which is responsible for delivering them to the correct application running on the correct host. Transports can also provide reliability services such as ensuring that transfers arrive reliably and in-order. The transport layer breaks up the messages it receives into packets and relies upon the Internet layer to deliver them to the correct host. The Internet layer achieves this goal, in part by relying on the Data Link Layer, which is responsible for sending packets at every hop between the source and the destination. To provide this functionality, the Data Link Layer leverages the Physical Layer, which is responsible for transmitting and receiving the actual bits of the packet.

From the discussion of each layer's functionality, two aspects of the TCP/IP networking stack become clear. First, each layer only communicates with the ones right above or below it. For example, Transport does not interact with the Data Link Layer and cannot obtain information about it to make more informed decisions. Prior work has shown how this lack of information can lead to extreme performance degradation in the presence of unreliable links [23]. Second, we see that the interface between layers is minimal. For example, transport cannot request that the Internet Layer deliver a packet reliably.

These examples highlight how the TCP/IP stack traded performance for generality to enable communication even when different technologies were employed at different layers. While well-suited for the Internet, which is designed to interconnect heterogeneous networks [28], this approach is poorly suited for datacenters. As datacenters are typically controlled by one administrative entity, it is simple to ensure that certain mechanisms be deployed at various layers of the stack, thereby allowing other layers to depend on them to achieve better performance.

Consider how the current limitations affect TCP's performance. As mentioned earlier, the Internet Layer does not provide any reliability guarantees. To provide a reliable service to the application, TCP must address this problem within its own layer. TCP achieves this by requiring that the destination send acknowledgments to the source whenever packets are received. If the source does not obtain an acknowledgment within a certain period of time, it times out and retransmits the packet.

The absence of detailed network state forces TCP to contend with many performance limitations. For example, it is unclear how the timeout should be set. If it is set too low, then too many packets will be retransmitted simply because the acknowledgment did not arrive on time. These spurious retransmissions would lead to a significant, and unnecessary, increase in network load. If the timeout is set too high, then the source will sit and wait far too long to retransmit the packet, causing unnecessary delays. While TCP mitigates this problem by using previous round-trip-times as an indicator of when a timeout should occur, queuing delays can make round-trip-times highly variable [18, 42]. As a result, TCP must still wait a long time to ensure that the packet drop has occurred, often unnecessarily delaying its retransmission.

In an attempt to further reduce retransmission delays, enhancements were adopted to leverage out-of-order delivery as an early indicator of loss [34]. That is, packets not received before those transmitted later were presumed to be lost. As soon as this loss is detected, TCP retransmits them, avoiding the need to wait for a timeout. Unfortunately, this approach is problematic for datacenters for multiple reasons. First, many paths typically exist between a source and destination. Ideally, we would spread packets across these paths to evenly balance the load. However, doing so will increase the likelihood that they be delivered to the destination out of order, causing TCP to believe that a loss had occurred. Second, messages in the datacenter are small [24]. If a message only consists of one packet, then there will not be any later transmissions indicating that the previous one had been dropped. Instead, TCP will have to fall back, relying on timeouts.

In the absence of other signals in the network, TCP uses the same packet drop detection mechanisms as an indication of congestion, dropping its transmission rate. Unfortunately, at the point that TCP detects a packet drop, the network has been congested for a long time as the buffers of the switch will first fill before a packet is dropped. To address this problem, a mechanism called ECN (Explict Congestion Information) [33], was added to inform TCP earlier that congestion was occurring. ECN provides TCP with 1 bit of information specifying that a queue has built up. TCP then reduces its rate to drain the queue, thereby reducing the latency experienced by delay-sensitive flows. This approach is used in combination with

Random Early Drop (RED) [35], which performs probabilistic marking based on average queue lengths and CoDel [48], which marks packets if the minimum queue occupancy over a certain duration exceeded a threshold. While ECN provides more information to transport, it is still quite minimal, opting for generality and interoperability over performance. For example, as ECN does not inform TCP how much it should reduce its rate by, TCP typically cuts its rate by half and then searches for the correct rate by increasing it periodically and checking to see if congestion occurs. Furthermore, as support for ECN is optional, TCP cannot depend on its existence and must be robust to environments where only a subset of the switches over which a flow traverses employ / enable ECN marking.

In addition to making TCP take a long time to react to packet drops and to find its appropriate sending rate, the absence of direct signals from other layers make TCP prone to pathological conditions that can cause dramatic throughput degradation and / or latency spikes. One such problem is Incast [55]. Incast is a phenomenon where many sources simultaneously send to the same destination, experience packet drops, and timeout, causing link underutilization. This phenomenon is common in datacenters as they must typically aggregate results across many servers that were enlisted to parallelize processing. These problems demonstrate the perils of relying on a stack consisting of independent layers which make minimal assumptions about each other when trying to achieve predictable network latencies.

2.3 Datacenter Design Opportunities

Fortunately, the datacenter environment is unique. The single administrative domain provides the opportunity to tightly integrate the layers of the stack, achieving greatly improved network performance. Furthermore, single administrative control makes it possible to modify all of the servers and / or switches in a datacenter, creating a homogenous environment where all nodes leverage the same stack, further simplifying design and deployment.

Additionally, modern commodity switches provide a suite of protocols that can be used to improve datacenter performance [3]. Here we focus on two, which are important for our goal of achieving reduced network latencies. The first, Pause, and its extension Priority Flow Control (PFC) performs link-layer flow control [12]. The second, Quantized Congestion Notification (QCN) performs fine-grained rate limiting at the link layer [5]. We now discuss each in turn.

Pause leverages inter-switch communication to prevent packet drops. When the buffers of a switch start to fill, it sends Pause messages to all of its neighbors contributing to congestion, asking them to postpone transmission. As the buffers drain, the switch will inform its neighbors that it is safe to resume transmission. With the tight timing requirements that exist with Pause [13], it can be configured in such a way as to prevent congestion-related packet drops. Networks that do so are typically known as *lossless.*

A key advantage of Pause is that it operates at a much finer time granularity than end-to-end approaches employed by host-based protocols (e.g. TCP). Consider how long it takes for TCP to react to congestion. First the switch must mark that congestion is being experienced by setting the ECN flag. Then the packet must travel to the destination, where a specially marked acknowledgment is generated to inform the source that congestion is occurring. This acknowledgment must then travel back to the source. Finally, the source receives the acknowledgment, reducing its transmission rate. In contrast, a switch experiencing congestion can generate a Pause message in hardware, have it sent to the appropriate neighbors, which react to it in hardware, postponing transmission over the link. Clearly, Pause is far more effective at preventing packet drops than a host-based approach.

However, Pause also suffers from a significant disadvantage known as head-of-line blocking. As an example, lets look at the case where two flows A and B, are sharing a link. The switch receiving packets from this link starts to experience congestion, with too many flows simultaneously contending for the same output port. In our example, only flow A is contributing to this contended output port, while flow B is traversing through another one. If the switch decides to send a Pause message, it will inform the previous hop to postpone transmission over the link, in effect unnecessarily delaying the transmission of flow B.

PFC was introduced to mitigate this problem by allowing the transmission of eight different priorities, or traffic classes, to be postponed individually. By mapping different traffic types into different priorities, PFC removes the head-of-line blocking that would occur between them. Unfortunately, head-of-line blocking can still occur within a priority, and there are relatively few priorities, making the division of traffic coarse-grained. Therefore, while helpful, PFC must still be used carefully to avoid a significant drop in throughput.

Another protocol at our disposal is Quantized Congestion Notification (QCN). With QCN, switches monitor their output queues. As queues start to build, switches send notifications back to the sources, requesting that they reduce their rate by a certain amount. Sources respond, dropping their rate. To recapture bandwidth that is no longer used, after a short duration, sources will begin probing the network by slowly increasing their transmission rate. By performing fine-grained rate adjustments, we have seen that QCN is capable of achieving high throughputs while maintaing low queue occupancies and hence low latencies.

Unfortunately, QCN also has some important limitations. First, QCN relies on hardware rate limiters at sources. The number of such rate limiters is fixed, typically to a small number. This means that a large fraction of the source's traffic may have to share the same rate limiter, causing head-of-line blocking where multiple flows are capped to the rate of the flow contributing most to congestion and hence having the slowest rate. Second, QCN works independently of, and is unaware of TCP. This means that QCN and TCP may interfere, making conflicting decisions. Or, QCN may hide congestion information from TCP, leading to suboptimal decisions. Therefore QCN, like PFC, must be used judiciously to achieve high networking performance.

2.4 Recent Datacenter Network Proposals

The need to create new approaches to achieve predictable latencies as well as the opportunity to quickly make significant modifications in datacenter environments have generated many proposals for addressing various aspects of the problem. These approaches primarily focus on three areas: (i) reducing the likelihood or cost of packet drops, (ii) evenly balancing network load, and (iii) effectively prioritizing latency-sensitive flows.

As described earlier, packet drops pose a problem for TCP because they can lead to resource-wasting timeouts. One way to address this problem is by reducing the likelihood of packet drops. Two prior solutions that advocate for this approach are DCTCP and HULL [18,19]. Both of these approaches strive to reduce TCP's buffer occupancy while maintaining high throughputs. Leaving switch buffers free enables them to absorb unpredictable traffic, reducing the likelihood of packet drops and retransmissions.

DCTCP achieves its goal by modifying TCP's response to ECN marks. In response to both ECN marks and packet drops, TCP traditionally cuts its rate by half. It then periodically increases its rate by a fixed increment. This coarse-grained rate reduction forces TCP to require high ECN thresholds (and correspondingly large buffers) to achieve high throughputs. If ECN thresholds are not set sufficiently high such that TCP overshoots its ideal rate by a large fraction, then when TCP cuts its rate in half, it will spend a long time transmitting at less than its ideal rate. This will result in a large drop in throughput.

To address this problem, DCTCP proposes employing fine-grained rate adjustments based on the frequency with which ECN marks occur. For example, if only a few ECN marks are received within a short duration, TCP will only reduce its rate by 20%. This fine-grained rate adjustments enable ECN thresholds to be set much lower without a significant drop in throughput. By setting the ECN thresholds lower, we experience a corresponding drop in buffer utilization, freeing switch buffers to be used for absorbing unpredictable packet bursts.

One problem that DCTCP fails to address is that servers typically send traffic in large bursts. A key reason why these bursts occur is that modern network interface cards enable TCP to offload large amounts of data to them. These network interface cards then divide up the data into packets for TCP, and transmit them. Having the network interface card packetize the data, allows servers to reduce CPU loads and is necessary for efficiently achieving the high network speeds required in datacenter networks.

At the same time, these bursts pose significant challenges for the network. Bursts can cause packet drops even if sources are all transmitting at their ideal average rates. When two bursts arrive at a switch at the same time, they fill up buffers, potentially overflowing them. Furthermore, they make it more challenging to determine when to set the ECN threshold. Setting it too low causes it to react to a burst, unnecessarily decreasing throughput. Setting it too high leads to greater buffer occupancies and hence a reduced ability for switches to absorb unpredictable traffic.

To address this problem, HULL proposes new, simple hardware for pacing out packets. The hardware is placed after the network interface card and is designed to match the *aver-*

age incoming and outgoing transmission rates. By spreading out bursts, HULL avoids the problems associated with handling them, enabling DCTCP to operate much more effectively.

An alternative approach to reducing the problem associated with packet drops is to mitigate the cost of them. This is the approach advocated for by [55] and by pFabric [20]. Both of these approaches advocate for reducing the minimum retransmission timeout far below the 200ms typically used in today's Internet. They both showcase greatly reduced high-percentile latencies with sub 1ms timeouts. While effective, this approach has a limit at which employing tighter timeouts will lead to spurious retransmissions and the resultant increase in network load.

The second problem, evenly balancing load across the network, while highly desirable, is particularly challenging given TCP's sensitivity to out-of-order delivery. Recently multiple efforts have been proposed to address this problem. Hedera, employs a centralized approach where a controller periodically remaps flows to alleviate hot-spots [16]. By performing these operations infrequently, Hedera reduces the likelihood of out-of-order delivery. Another approach, MPTCP [52], leverages a host-based solution where each host opens multiple subflows to the same destination. Subflows take different paths and each one behaves like a traditional TCP flow. In response to congestion, MPTCP balances load across these subflows, moving traffic away from congested paths.

The third problem of effectively prioritizing latency-sensitive flows came from the fact that all packets transmitted in a datacenter network used to have the same priority. When they arrived at a switch, they were processed in FIFO order, waiting their turn to be transmitted. This approach supported TCPs focus on cross-flow fairness, where every flow should obtain its fair share of a congested link's bandwidth.

This decision is suboptimal for datacenter environments where some flows are much more latency-sensitive than others [18, 56]. Clearly, the packets of such flows should be given a higher priority than the packets of latency-insensitve ones. To create a prioritization mechanism, two questions arise: (i) how to determine a flow's priority and (ii) which switch mechanisms to employ to enforce it. D^3 and PDQ determine priorities based on the flow's deadline and then traverse through the path, reserving bandwidth. They do this frequently (once every round-trip-time) to support newly-arriving flows with short deadlines. pFabric takes a different approach, determining priorities based on the number of bytes remaining in the flow and employing switch-based modifications that pick and transmit packets based on this value.

While these approaches make strides in addressing the problems they focus on, often they are unable to completely resolve them. DCTCP and HULL are unable to completely avoid packet drops while Hedera and MPTCP both operate a timescales that are far too coarse-grained to assist short flows. Other approaches, such as D^3 and PDQ have the significant drawback of requiring that no transmissions occur during the first round-trip-time (RTT) because they must first make a reservation. This is a significant limitation for datacenters where most flows could complete in just one RTT [24]. Finally, some approaches, such as pFabric, make strong assumptions about the ability to predictably achieve tight timeouts, which may be difficult to do in practice.

Perhaps more importantly, it is often unclear how these mechanisms can be combined into a coherent solution. By making smaller rate adaptations in response to ECN marks, DCTCP's and HULL's fine-grained mechanisms take longer to converge to the ideal rate, making them difficult to combine with an adaptive load balancing scheme. D^3 and PDQ on the other hand, make it challenging to evenly balance load without either over-reserving or under-reserving resources since they do not know ahead of time which of their requests will be granted.

In a departure from prior work, in this thesis, we strive to simultaneously address multiple causes of unpredictable flow completion times, or to at a minimum develop mechanisms that could be readily combined into a complete solution. But before doing so, we must develop a platform for evaluating different proposals and analyze the impact of inflated high-percentile flow completion times on page creation. The next two chapters address each of these issues in turn.

Chapter 3

Platform

As mentioned in the previous chapter, to evaluate various datacenter proposals, we must create one or more platforms in which to compare them. This represents a significant challenge. Different platforms have varying limitations which make them incapable of evaluating differing aspects of the datacenter proposals we are comparing. Testbeds typically are not sufficiently large, lacking the hundreds of servers necessary to observe datacenter phenomena. Those that are typically do not support rich custom topologies (e.g. FatTrees) with configurable switch behavior. Simulations on the other hand are unable to verify that their results accurately model real-world phenomena.

In this chapter, we present our approach to addressing this problem. We employ two platforms: a simulator and a test-bed based implementation. By using the test-bed based implementation to verify our basic assumptions and the simulator to evaluate scaled-up performance, we are able to perform a much more thorough evaluation. We now describe each of our platforms in turn.

3.1 Simulation

Many options exist for simulating datacenter network proposals. Broadly, there are two types of simulators: packet-level simulators and flow-level simulators. Common examples of packet-level simulators include NS-2 [10] and NS-3 [11]. These simulators calculate the transmission and reception times of each packet as it traverses every hop from the source to the destination. They explicitly account for queuing by having packets ”wait” at a node until its transmission time. In certain cases, these simulators can even process the code of the TCP Linux stack to determine exactly when events such as retransmission should take place [9].

Unfortunately, all of these benefits come with a significant limitation: packet-level simulators are unable to scale to the tens or hundreds of thousands of nodes typically employed in datacenters. The two primary reasons for this limitation are that (i) the large amount of per-packet processing required to simulate transmission through the network and (ii) the difficulty of parallelizing event-based simulators. As a result, all of the packet-level simulators we are aware of are single-threaded.

An alternative is to employ flow-level simulators. Prior work has demonstrated that these simulators achieve far greater scalability than packet-level simulators [52]. Unfortunately, they do so at a cost: as they no longer simulate the dynamics of every packet traversing through the network, these simulators do not enable fine-grained analysis. Since the focus of this thesis is on high-percentile latencies, we depend on this fine-grained analysis to evaluate the viability of different protocols for datacenter networks. Consequently, we focus on achieving the best performance possible with packet-level simulators.

To create a simulation environment, we chose with NS-3 as a starting point [11]. In addition to the challenges typically facing packet-level simulators, NS-3 has two additional problems: (i) it is not designed to handle a large number of small connections entering and leaving throughout the duration of the simulation and (ii) it lacks implementations of many of the components typically available in modern datacenter switches.

Addressing the first problem required two steps. First, we traced through NS-3 operations for connection establishment and teardown, ensuring all resources were released once a connection finished, reducing the memory leaks in the system. Second, we had to ensure that the connections, themselves were deallocated as soon as the transfer was completed. For example, we needed to remove TCP's TIMED-WAIT state, which is designed to have a socket persist after transmission to process any spuriously retransmitted packets. To avoid this problem, we deallocate the socket as soon as the transfer completes and send Reset (RST) packets, avoiding the long durations associated with TCP's traditional connection teardown.

The second problem represents a different type of challenge. Not only do we need to implement the required functionality, we need to ensure that the reaction times to different operations are correct. For example, when employing link-level flow control, we would see very different behavior based on the time it takes for pause messages to be transmitted and responded to. To address this problem, we analyze IEEE standards specifying required reaction times [13], product documentation [4, 12], and prior studies of server processing times [43]. Where appropriate, we include these delays in our simulation to improve its accuracy.

3.2 Implementation

Creating the implementation environment is also challenging. To properly analyze the behavior of different networking proposals, we need knowledge and control of the underlying network topology. Additionally, as many proposals depend on switch modifications, we require the capability of modifying switch behavior (both hardware and software) as well. These requirements make IaaS-based solutions infeasible as they do not provide knowledge or control over the underlying network [2], which is unfortunate as they would have enabled us to evaluate proposals at larger scales.

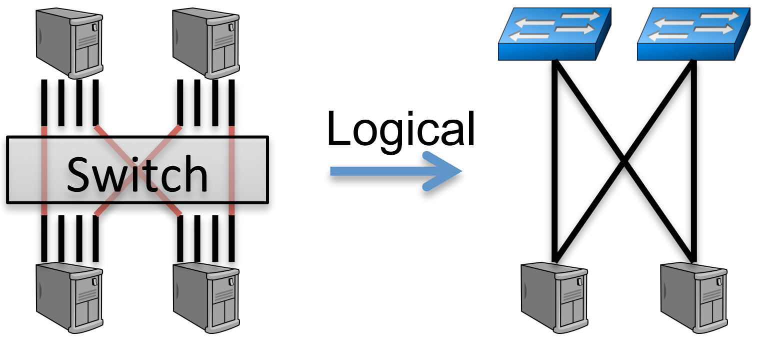


Figure 3.1: Emulab enables us to create custom topologies. Enabling / disabling certain links (left) allows us to use the switch as a patch-pannel to achieve the desired topology (right).

Instead, we rely on a combination of Emulab [7] and Click [45]. Emulab is a testbed environment that allows us to create a custom topology between a set of servers, while Click is a software router platform. In combination, these two technologies enable us to create a custom topologies and evaluate different networking proposals.

Emulab testbeds typically have servers that are connected to the same switch multiple times [25]. In this environment, we can specify both which links are enabled as well as how they are connected together. Figure 3.1 demonstrates how this powerful abstraction allows us to use the switch as a patch-pannel, creating a custom topology between the servers in the testbed. On the left, we see the physical links that are connected together, while on the right we see the logical topology this creates. Once the topology is created, all that remains is having the appropriate servers in the topology emulate switches (e.g., the top two servers in this Figure).

To emulate switch behavior, we chose Click. Click is a modular software router platform designed to make adding new functionality simpler. In Click, every operation is performed by a unique module. Different modules are used for polling a port to see if packets have arrived, looking up the destination IP address to determine the next hop, decrementing IP's time to live field, recomputing the checksum, sending the packet to the output port, and many of the other operations performed by a router. A configuration file is used to specify the flow of packets through these components.

A key benefit of this design is that it allows new components with the needed functionality to be placed anywhere between existing ones. Similarly, existing components can be swapped out with new ones performing different functionality. This extreme modularity dramatically increases prototyping speed by enabling us to quickly program new functionality.

Unfortunately, while Click's modularity makes adding new features simple, this platform does not focus on achieving predictable performance.We require that our software routers

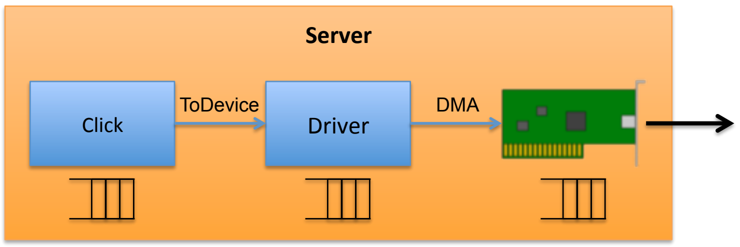


Figure 3.2: Path packets take after being "sent" by Click. The are (i) stored in the device driver's ring buffer, (ii) DMAed to the NIC, and (iii) stored in the NICs buffer before being placed on the wire.

and end hosts perform predictably to obtain accurate measurements, especially at high percentiles. To address these limitations, we make the following key modifications to the Click platform.

3.2.1 Controlling Switch Buffers

One problem we faced with Click is that it does not control the complete forwarding path between packet reception and transmission. When Click "sends" a packet, it is actually enqueued on the driver's ring buffer. Later, the packet is DMAed to the NIC, where it sits in another buffer until finally being placed on the wire. We depict this in Figure 3.2.

In total, the NIC and driver buffers can store hundreds of KB worth of packets. This is significant, especially compared to the limited buffers commonly available on modern ethernet switches [3]. Minimizing the occupancy of these buffers (while maintaining high throughput) is critical for many reasons. First, it allows the buffering employed by Click to more accurately reflect the per-port buffering capacity of real switches. Second, it allows the decisions taken by Click to take effect in the appropriate amount of time, instead of waiting for a large number of packets to be flushed out out the driver and NIC. Finally, it enables Click to have a more accurate understanding of the congestion being experienced.

We explored two alternatives to addressing this problem. The first was to reduce the size of the ring buffer at the driver, causing it to fill sooner and keeping more packets under Click's control. The second was to cap Click's sending rate to one that is slightly smaller (e.g. 2%) than that of the link. We chose the latter approach because (i) it would control the buffers used by both the driver and the NIC (the first approach only helps the driver) and (ii) it is a general approach that works across different drivers.

There was one difficulty with this approach. We needed to strike a balance between the desire to keep buffers as lightly utilized as possible and keeping the driver and NIC buffers sufficiently full to ensure high throughput. Fortunately, our experiments show that for the 1Gbps links used in our implementation, allowing up to 8KB to be forwarded to the driver and NIC before the expected transmission time is sufficient for maintaining high throughput.

3.2.2 Scheduling Modules Predictably

Another challenge is ensuring that Click's modules are scheduled predictably to minimize the variance in packet processing times. There are two aspects of this problem: (i) Click's data flow abstractions do not enable us to specify when certain functionality should be executed and (ii) Linux treats the processes and kernel threads running Click just like any other, often choosing to swap them out in favor of performing other computation. Both of these problems make it challenging to achieve predictable packet processing times.

Addressing the first aspect of the problem primarily involves understanding how Click maps modules into threads of execution and the order in which it executes them. Fortunately, in multithreaded environments, Click provides the functionality to allow us to specify which modules will be executed by which threads. However, it does not allow us to specify in which order they will be executed. Furthermore, as a default, it employs a stride scheduler which strives to achieve greater efficiency by running modules that do useful work more frequently than those that do not.

While effective, Click's stride scheduler has an undesirable side effect. Consider the case where a module is tasked with performing an important operation that occurs relatively infrequently. With Click's stride scheduler, when that event occurs, it will take the stride scheduler a long time to run that module, delaying its operation. To address this problem, we swap the stride scheduler with a round-robin one. With the round-robin scheduler, every Click module runs at a predictable frequency, ensuring that important events will not be unnecessarily delayed.

The second problem represents a greater challenge. Ideally, we want the processes and kernel threads running Click to never be swapped out, so as to prevent any blips in performance. One possibility is to port Click to another operating system with more explicit control of how threads and processes are scheduled. Given the high development costs associated to with such an approach, we choose to work with the Linux scheduler instead, leveraging all the knobs it provides.

The Linux scheduler allows a subset of available cores to be isolated at boot time. When this flag is specified, the Linux scheduler will avoid running tasks on these cores. This means that any threads or processes we choose to run on these cores will experience a minimally contended environment. By pinning Click's processes and threads to these cores, we minimize how frequently they are interrupted, achieving more predictable packet processing times.

3.2.3 Minimizing Control Traffic

The last problem we face with Click is reducing the amount and impact of control traffic. Control traffic transmission and processing can cause network unpredictability. ARP, for example, which dynamically creates a mapping between IP and MAC addresses can delay packet transmission until the mapping for a new IP address is obtained. Similarly, distributed routing protocols can take a long time to converge, causing packets to take suboptimal paths, or worse yet travel in circles.

As the focus of this work is primarily on the achieving predictability over static topologies, we sidestep this problem by hard-coding all the necessary IP to MAC mappings as well as the next-hop decisions that are made as packets traverse from the source to the destination. This ensures that packet transfer times are determined based on queuing, propagation, and transmission delays instead of delays due to the transmission and processing of control traffic.

Having created both our simulation and implementation environments, in the next chapter, we turn our attention to analyzing the impact of unpredictable flow completion times on page creation workflows. In Chapters 5 and 6, we propose and evaluate two alternatives to addressing this problem.

Chapter 4

Impact of Highly Variable Latencies

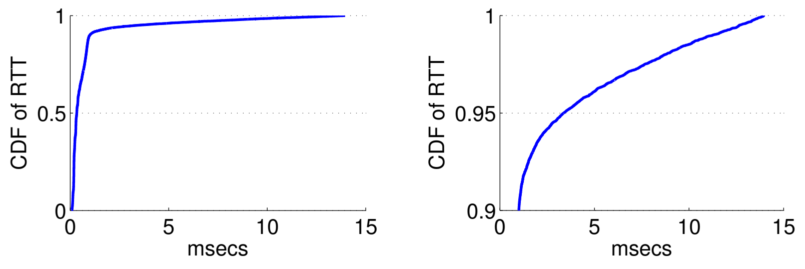
In this chapter, we evaluate the impact of highly variable latencies on page creation. To perform this evaluation, we must understand both the distribution of latencies typically encountered as well as the page creation workflows commonly employed. We analyze each in turn, leveraging traffic measurements from production datacenters for the former and previously reported workflow patterns for the latter.

From this analysis, we see that page creation workflows are typically both complex and application-dependent. So we model a set of simple workflows, stressing the properties typically found in the more complex, real-world ones. We compare how these simple workflows perform under the measured distribution to synthetic ones having tighter latencies, showing that highly variable latencies significantly impact page creation times.

Given the large impact on page creation, we propose solutions for addressing this problem in the following chapter.

4.1 Traffic Measurements

Microsoft researchers [18] have published measurements from a production datacenter network. Servers connect to this network via 1 Gbps links and run services like web search. Figure 4.1 (reproduced from [18]) depicts the measured intra-rack round-trip-times (RTTs). While RTTs are typically low, congestion causes them to vary by two orders of magnitude, forming *a long-tail distribution.* In this particular environment, intra-rack RTTs take as little as $61\mu s$ and have a median duration of $334\mu s$. But, in 10% of the cases, RTTs take over $1ms$. In fact, RTTs can be as high as $14ms$. These RTTs are the measured time between the transmission of a TCP packet and the receipt of its acknowledgement. Since switch queue size distributions match this behavior [17], *the variation in RTTs is caused primarily by congestion.*



(a) Complete Distribution (b) 90th-100th Percentile

Figure 4.1: Measured intra-rack RTTs, reproduced from [18].

4.2 Page Creation Workflows

The most commonly described page creation workflow is called *Partition-Aggregate* [18, 56]. This workflow is typically used to parallelize operations such as web search, reducing completion times. We depict this workflow in Figure 4.2. Top level aggregators (TLAs) receive search requests. They divide (partition) the query across multiple mid-level aggregators (MLAs), who further partition the query across worker nodes. Worker nodes perform the computation in parallel and send their results back to their MLA. Each MLA aggregates the results it receives and forwards them on to the TLA. The TLA aggregates results obtained from all MLAs, providing a complete result.

The interactions to create a page may be much more complicated than those depicted by the figure. A partition-aggregate tree may have more levels and multiple partition-aggregate trees may executed to obtain the result to a single query [18]. As a result, conservative timeouts must typically be set at each level so that higher levels have sufficient time to aggregate the results. Typically 10ms is given to each worker node and 50ms is given to each aggregator. When a node takes too long to respond, the web site has two poor options. It can either (i) return early, degrading the quality of the result or (ii) delay the response, violating user-interactivity deadlines.

In 2013, Facebook reported that they employ even more complex workflows to construct a single page [49]. In these workflows, over 1000 objects may be retrieved to construct a single page. To make matters more complicated, these retrievals may have sequential dependencies. That is, the server may have to wait for the results of previous retrievals to determine the next object to request. This severely limits the achievable parallelism and hence the number of operations that can be reliably performed within user-interactivity deadlines.

From these two examples, we see that workflows are both complex and application-dependent. In the next section, we evaluate how the tail of latencies impacts page creation workflows.

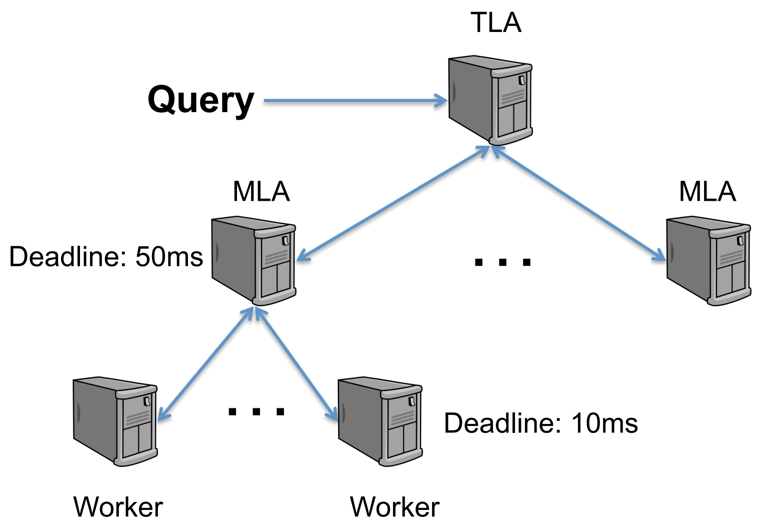


Figure 4.2: Partition-aggregate workflow. To satisfy top-level user-interactivity deadlines of 200ms, workers are typically given 10ms deadlines and MLAs are given 50ms deadlines.

4.3 Impact on Workflows

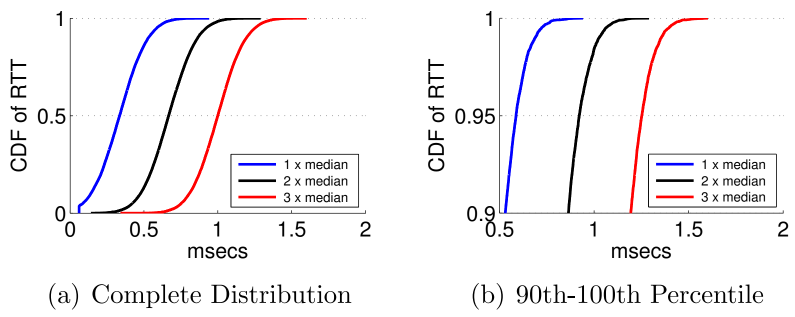
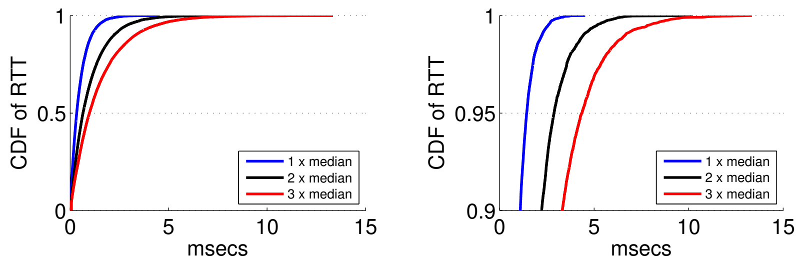


Figure 4.3: Intra-rack RTTs following a normal distribution (synthetic). The distributions vary in their median, which is set to be 1x, 2x, or 3x of that of the measured distribution.

As the workflows employed for page creation have complex interactions that depend greatly on the functionality being provided, evaluating them is challenging. Our approach is to employ models that stress the key aspects of these workflows. Looking back at the previous section, we see that regardless of the functionality being provided, workflows have sequential and parallel components. So, we evaluate the impact of long tails by creating



(a) Complete Distribution (b) 90th-100th Percentile

Figure 4.4: Intra-rack RTTs following a exponential distribution (synthetic). The distributions vary in their median, which is set to be 1x, 2x, or 3x of that of the measured distribution.

models of sequential and parallel workflows and quantifying how they would be affected by highly variable latencies.

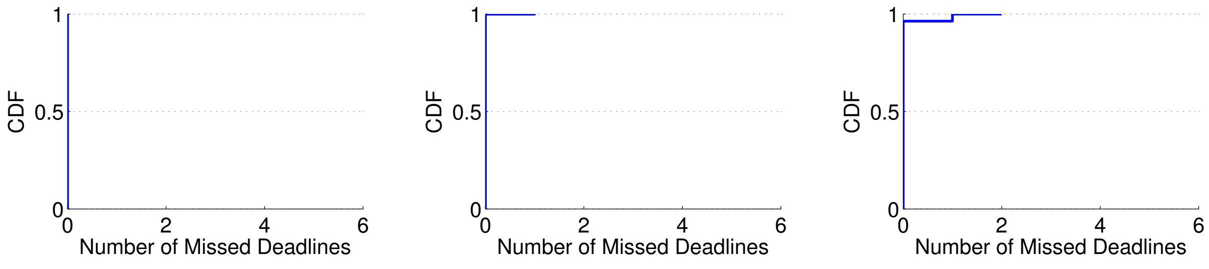
Our models take as input (i) the amount of time it takes the server to obtain a response for each request and (ii) the number of parallel/sequential requests in each workflow. We obtain values for the former by drawing them from the measured distribution in Figure 4.1. This approach assumes that each request/response takes one RTT, which is appropriate since we must send requests to nodes before we can obtain responses from them. We note that this is a conservative assumption because we assume no server processing delays and no packet drops. Packet drops cause retransmissions, and perhaps even timeouts, inflating latencies.

As comparison points, we investigate how these workflows perform under normal and exponential distributions. For both distribution types, we report the results for a family of curves having medians that are 1x, 2x, and 3x that of the measured distribution. To ensure a fair comparison, we require that the minimum value drawn from these distributions be larger than or equal to the minimum in the measured one. Figures 4.3 and 4.4 depict the normal and exponential distributions, respectively.

A key parameter when specifying the normal distribution is the standard deviation (σ). For the purpose of understanding the impact of tails on workflows, we set sigma to be the difference between the 50th and 16th percentile of the measured distribution. We chose to set sigma based on the lower percentile (16th) instead of the higher one (84th percentile) because we wanted to highlight the benefits of a smaller tail. This provides a better contrast to the measured distribution.

We now analyze how these different distributions affect our parallel and sequential workflows, discussing each in turn. Since high-percentile completion times are considered to be a key metric, we focus on 99.9th percentile workflow completion times. This is the highest percentile evaluated by prior work [18, 31].

Parallel



(a) 1x measured (b) 2x measured (c) 3x measured

Figure 4.5: Probability that a number of workers in a 40-worker workflows miss their 10ms deadline under the exponential distribution.

In parallel workflows, an aggregator requests results from many workers at the same time. As shown in Figure 4.2, workers currently have just 10ms to perform their computation and deliver their result. Results from worker nodes who do not meet this deadline are typically discarded. To make matters worse deadlines are becoming tighter, as workflows become more complex to provide richer content, while meeting the same user-interactivity requirements.

To assess the impact of different distributions, we report how they affect workflows with varying numbers of workers. We vary the number of workers from 40, the number typically responding to the same MLA, to 4000. We compute the number of workers that have missed their deadlines by drawing the round trip times from the specified distributions and counting how many have exceeded the 10ms threshold.

A challenge with our evaluation is determining how many times to run the model. Running the model more times increases the number of data points, enabling analysis at finer-grained percentiles. At the same time, increasing the number of runs increases computation time. To balance these two competing concerns, we chose to run each model 10000 times. This ensures that even at the 99.9th percentile, there will be 10 runs having a larger number of workers miss their deadlines.

40-worker

We first evaluate the likelihood that a 40-worker workflow has a certain number of workers miss their deadlines. We depict the probabilities under the exponential distribution in Figure 4.5 and the measured distribution in Figure 4.6. Under the normal distributions (not depicted), our model does not see a single worker miss their deadline. Looking at the exponential distribution with the longest tail (where the median is 3x of the measured one),

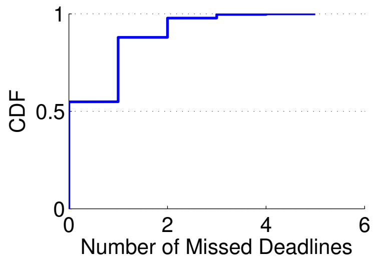
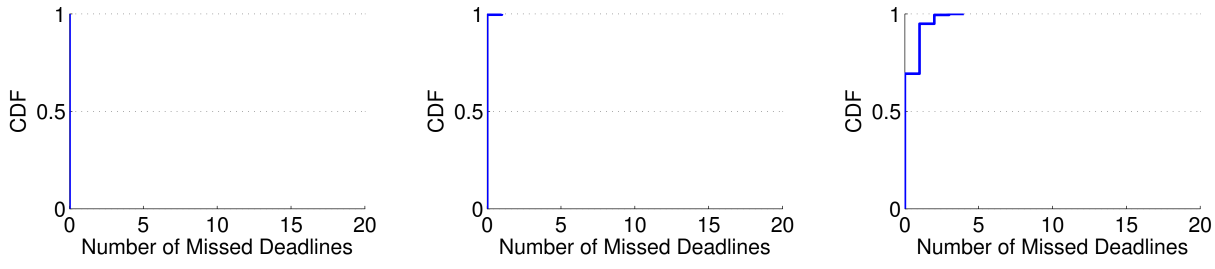


Figure 4.6: Probability that a number of workers in a 40-worker workflow miss their 10ms deadline under the measured distribution.

we see that at the 99.9th percentile a 40-worker workflow will see 1 (2.5%) worker miss its deadline. On the other hand, a 40-worker workflow under the measured distribution will see 4 (10%) workers miss their deadline. As both the worst-case normal and exponential distributions can have medians 3x higher than that of the measured one, this experiment demonstrates that tail performance impacts workflows much more than the median.



(a) 1x measured (b) 2x measured (c) 3x measured

Figure 4.7: Probability that a number of workers in a 400-worker workflows miss their 10ms deadline under the exponential distribution.

400-worker

By scaling the workflow up to a larger number of workers (400), we uncover new properties. In Figures 4.7 and 4.8 we report the probability that a certain number of workers will miss their deadlines under the exponential and measured distributions, respectively. Once again, our model sees no missed deadlines under the normal distribution. Under the measured distribution, at the 99.9th percentile, a workflow will have 15 (3.75%) workers miss their deadlines. Compare this to to the exponential distribution with the largest tail, where just 3 (0.75%) of workers miss their deadlines.

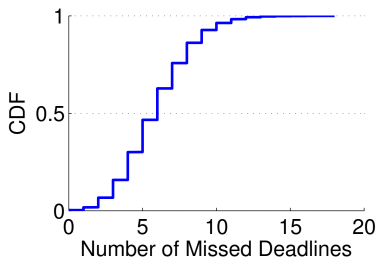


Figure 4.8: Probability that a number of workers in a 400-worker workflow miss their $10ms$ deadline under the measured distribution.

Perhaps most important, is the performance of the median workflow. We see that under all distributions other than the measured one, the median workflow has zero missed deadlines. Under the measured distribution, the median workflow has 6 (1.5%) of its workers miss their deadlines. Thus for sufficiently large workflows, it becomes likely that some workers will miss their deadlines during *most* queries.

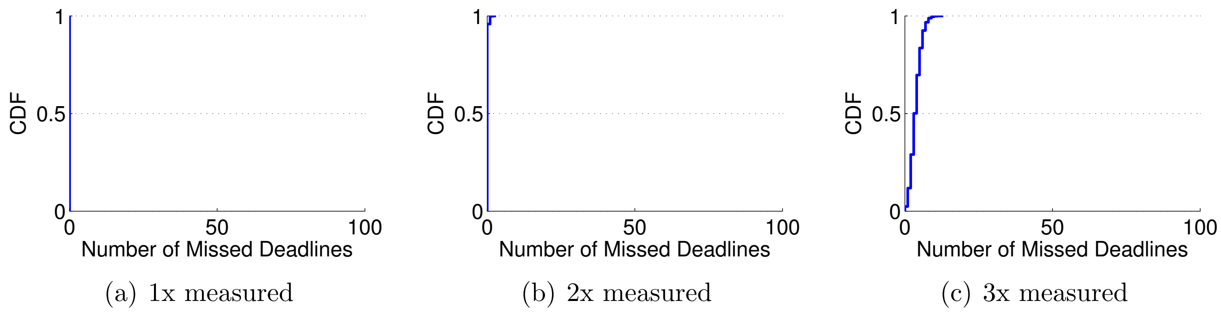


Figure 4.9: Probability that a number of workers in a 4000-worker workflows miss their $10ms$ deadline under the exponential distribution.

4000-worker

We scale up the number of workers once more to 4000. The impact of the exponential and measured distributions on the number of workers that miss their deadlines is depicted in Figures 4.9 and 4.10, respectively. As before no workers miss their deadlines under the normal distribution. Looking at the impact of the measured distribution, we see that for this large workflow, we expect workers to miss their deadlines for *every* query. And the number of workers that miss their deadlines is not small; at least 34 (0.85%) of workers miss their deadlines across all our runs with the measured distribution.

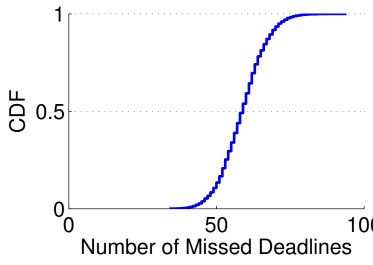


Figure 4.10: Probability that a number of workers in a 4000-worker workflow miss their $10ms$ deadline under the measured distribution.

Clearly, these results have demonstrated that tail performance has greater impact on workflow completion times than the median. But there are two attributes to the tail that must be considered – its length and the probability of hitting it, or *heaviness*. In our experiments, the max value drawn from our exponential distributions was 13.35, while the maximum value drawn from our measured distribution was 13.95. At the same time, our results show a dramatic difference between the number of missed deadlines between the worst-case exponential distribution and the measured one.

For the purposes of this evaluation, we quantify the heaviness of the tail as the probability of drawing values larger than 10ms. Since our focus is on the response times that lead to missed deadlines, this quantification is appropriate for our purposes. We see that under the measured distribution, the probability of drawing a value larger than 10ms is over 1.4%. Compare this with 0.095% under the worst-case exponential distribution. As the median 4000-worker workflow misses 1.5% and 0.075% of its deadlines under the measured and exponential distributions, respectively, the heaviness of the tail seems to be a better indicator of the difference we see in workflow performance.

4.3.1 Sequential

As mentioned earlier, many workflows have serial dependencies where the next request cannot be performed until the results of the previous one have been obtained. Thus a request that hits the tail in this environment may cause the whole workflow to stall, delaying the time it takes to complete. When this happens, the same sequential dependencies make it much harder to abandon the request and return early. Today, web-sites address these problems by working to reduce the number of sequential requests required for each page creation [50].

To obtain some intuition about the impact of the tail in this environment, we model a sequential workflow. In this workflow, a single server performs a series of requests. Each request is only performed once the results of the previous one have arrived. The workflow

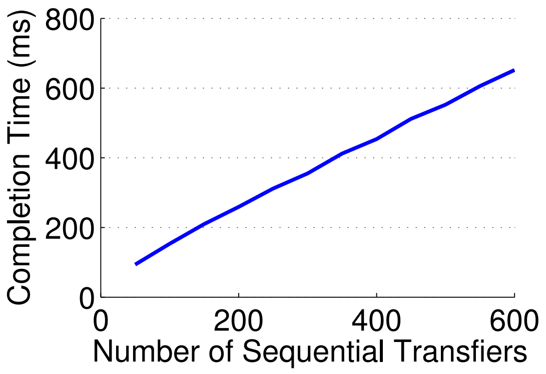


Figure 4.11: 99.9th percentile completion times of sequential workflows under the measured distribution.

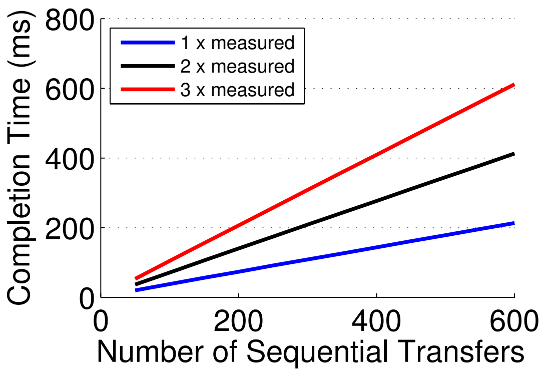


Figure 4.12: 99.9th percentile completion times of sequential workflows under the normal distribution.

completes when the last request has arrived. As before, we draw request completion times from the normal, exponential, and measured distributions. We also run our model 10000 times for the same reasons described for parallel workflows.

In Figures 4.11, 4.12, and 4.13, we report the 99.9th percentile completion times under the measured, normal and exponential distributions. Datacenter workflows are expected to meet user-interactivity deadlines of 200ms, 99.9% of the time. So, these figures demonstrate how many requests the workflow can reliably support under each distribution, for a given deadline.

These figures indicate that increases in median completion time lead to significant increases in workflow completion time. Under the normal distribution the number of retrievals we can support within 200ms drops from 500 to a little over 150 as the median increases by 3x. In the exponential distribution, the number of sequential queries we can support within

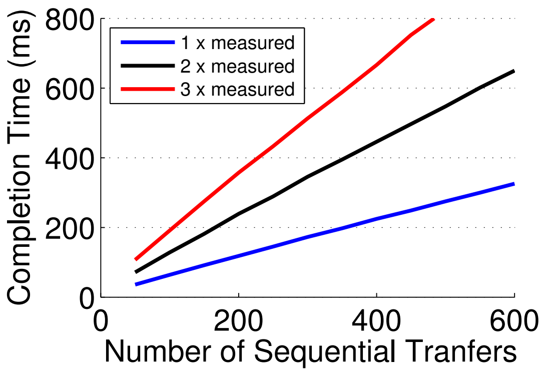


Figure 4.13: 99.9th percentile completion times of sequential workflows under the exponential distribution.

200ms drops from 350 to 100 as we increase the median by 3x.

We note however, that the tail still significantly impacts performance. All of the normal distributions we evaluated outperform the measured one, which must have fewer than 150 sequential data retrievals to complete within 200ms. Both the exponential distributions having 1x and 2x the median also outperform the measured one.

Unlike the parallel workflows, the performance of sequential ones depends most on the mean. Under the normal distribution, increasing the median by 3x increases the mean by the same amount. This corresponds closely to the 3x drop we experience in the number of data retrievals. We experience the same behavior for the exponential distribution. Finally, we see that the exponential distribution with a median of 3x has a higher mean than the measured one and hence performs worse. As the mean of the distribution depends greatly on both the length and the heaviness of the tail, improving high-percentile performance is very important for these workflows as well.

4.3.2 Takeaways

Long-tailed latency distributions make it challenging for workflows used for page creation to meet interactivity deadlines. *While events at the long tail occur rarely, workflows have so many retrievals, that it is likely that several will experience long delays for every page creation.* Hitting the long tail is so significant that workflows actually perform better under distributions that have higher medians but shorter tails.

At the same time, tail performance is likely to be far worse than that depicted in Figure 4.1. Packet drops may cause retransmissions, or worse yet trigger timeouts, dramatically inflating tail latencies. Servers may experience unpredictable delays, hurting response times as well.

Facebook engineers tell us that leaving highly variable latencies unaddressed forces their applications to choose between two poor options [26]. They can set tight data retrieval timeouts for retrying requests. While this increases the likelihood that they will render complete pages, long tails generate non-productive requests that increase server load. Alternatively, they can use conservative timeouts that avoid unnecessary requests, but limit complete web page rendering by waiting too long for retrievals that never arrive. *A network that reduces tail latency allows such applications to use tighter timeouts to render more complete pages without increasing server load.*

Throughout the remainder of this thesis, we focus on high-percentile latencies as the are a much better indicator of workflow performance than the median. In the next chapter, we discuss the key causes of long latencies and present a solution, DeTail that addresses these problems.

Chapter 5

DeTail

In the previous chapter, we evaluated how tail latencies impact page creation workflows. We showed that the tail significantly delays workflow completion times, reducing the quality of the page that may be constructed within user-interactivity deadlines.

Given these problems, in this chapter, we begin by focusing on the causes of high latencies. We see that there are three key causes: (i) packet losses and retransmissions, (ii) uneven load balancing, and (iii) the absence of prioritization. We describe each in turn, explaining how they contribute to increased latencies.

As discussed in Chapter 2, while each of these causes has been studied by prior work, and new mechanisms have been proposed, none of them represents a complete solution. The previously proposed mechanisms typically only focus on a subset of the causes and its is unclear how to combine them into a coherent whole that addresses all aspects of the problem. To provide a complete solution, we depart from prior work by proposing a new stack, *DeTail*, consisting of a series of tightly integrated layers that work together to overcome all three causes of long-tails.

DeTail leverages link-layer flow control to prevent packet drops, network-layer per-packet adaptive load balancing to evenly balance load, and in-network prioritization to provide resources to latency-sensitive flows. Finally, *DeTail* employs end-host rate throttling, mitigating the head-of-line blocking that can occur due to link-layer flow control. We evaluate *DeTail* on a network consisting of 1Gbps links, demonstrating its effectiveness at reducing the tail.

While effective, *DeTail's* reliance on link-layer flow control to create a lossless interconnect presents two sets of challenges. First, as link speeds increase, the buffer requirements for creating a lossless interconnect do as well. Second, approaches based on link-layer flow control can experience extreme performance degradation when switches and/or end-hosts misbehave. In the next chapter, we propose a solution that addresses both of these problems.

5.1 Causes of Long Tails

As described in Chapter 2, prior work focused on three causes of long tails: (i) packet losses and retransmissions, (ii) uneven load balancing, and (iii) the absence of prioritization. Here we delve deeper, providing a thorough explanation of how each of these causes increases tail latency. We then briefly revisit competing proposals, highlighting where they fall short. In the next section, we present *DeTail* and demonstrate how it address all three causes.

5.1.1 Packet Losses and Retransmissions

Packet losses can drive up latencies, especially when they lead to resource-consuming timeouts. In datacenters, these timeouts are typically set to $1 - 10ms$ [18, 55]. Since datacenter round-trip-times can be in the 10's to 100's of μs [20], just one timeout guarantees that the short flow will hit the tail. Using shorter timeouts may mitigate this problem, but it increases the likelihood of spurious retransmissions that increase network and server load.

Additionally, partition-aggregate workflows increase the likelihood of Incast [18, 56]. Workers performing computation typically respond simultaneously to the same aggregator, sending it short flows. This sometimes leads to correlated losses that cause many flows to timeout and hit the tail.

5.1.2 Uneven Load Balancing

Uneven load balancing drives up latencies when traffic is unnecessarily forwarded on a more congested path, despite the availability of less congested ones. Recall that modern datacenter topologies leverage multiple slower paths between a source and destination to increase aggregate bandwidth [15, 38, 39]. At the same time, TCP's single-path assumption typically limits datacenter networks to performing flow-based hashing to spread load across available paths. Since hashing is random and does not take into account flow size, this approach can lead to congestion hotspots, driving up latencies for all flows traversing through them.

Topological asymmetries make handling this problem even more important. Datacenter network failures are common [30] and can affect the network in a variety of ways. For example, a common type of failure reduces the speed of a $1Gbps$ link to $100Mbps$ [52]. Even when failures do not occur, asymmetries can occur due to incremental deployments or network reconfigurations. From prior work [52], we have seen that *an adaptive approach that spreads a flow's packets across multiple paths is an effective starting point for addressing such asymmetries.*

5.1.3 Absence of Prioritization

Datacenter networks represent a shared environment where many flows have different sizes and timeliness requirements. Traces from production datacenters show us that they must support both latency-sensitive and latency-insensitive flows, with sizes ranging from $2KB$ to $100MB$ [18].

To achieve high throughputs, large throughput-sensitive background flows will try to consume all available bandwidth and buffers. When the packets of short, latency-sensitive flows arrive, they may become enqueued behind those of the large background flows. This increases the likelihood that latency-sensitive flows will hit the long tail and miss their deadlines. We must consider different flow requirements to avoid harming latency-sensitive flows.

5.1.4 Current Solutions Insufficient

	Packet Losses	Prioritization	Load Balancing
DCTCP	$\surd$	$\times$	$\times$
HULL	$\surd$	$\times$	$\times$
pFabric	$\surd$	$\surd$	$\times$
D^3	$\surd$	$\surd$	$\times$
Hedera	$\times$	$\times$	$\surd$
MPTCP	$\times$	$\times$	$\surd$
DeTail	$\surd$	$\surd$	$\surd$

Table 5.1: Effectiveness of prior work at addressing causes of long tails in networks.

Prior approaches have been proposed to address each of these problems. We described these approaches in Chapter 2. Here, we highlight the causes that each of these approaches fails to address. As shown in Table 5.1, DCTCP, HULL, pFabric, and D^3 do not perform dynamic load balancing. On the other hand, Hedera and MPTCP neither handle prioritization nor mitigate the effects of packet loss.

Unfortunately, it is unclear how these approaches can be combined into a coherent solution that addresses all three causes of high latency. Furthermore, certain solutions such as pFabric rely on mechanisms such as extremely tight timeouts that may be difficult to achieve in practice. Given these limitations, in the next section we propose a new stack that employs a series of tightly integrated layers to simultaneously address all of the causes of tail latency.

5.2 The DeTail Stack

DeTail is a new stack that leverages a set of tightly integrated layers to address the causes of long tails. In this section, we first provide an overview of DeTail's functionality and discuss how it reduces tail latency. We then delve into the stack internals, describing

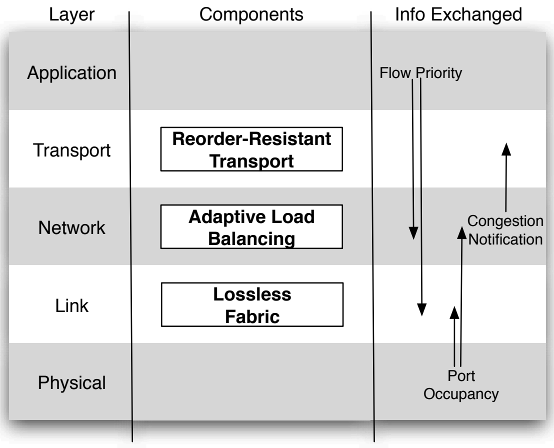


Figure 5.1: The DeTail network stack uses cross-layer information to address sources of long tails in flow completion times.

the specific mechanisms performed at every layer. We conclude this section by providing intuition about how to set the parameters of the stack.

5.2.1 Overview

In Figure 5.1, we depict the *DeTail* stack. We see that each of the tightly integrated layers makes explicit assumptions about and obtains information from the others. Here we discuss how *DeTail* addresses all three causes of high latency.

Our first goal when creating *DeTail* is to reduce packet drops and retransmissions while avoiding making single-path assumptions such as those employed by TCP. Furthermore, as high-percentile performance is extremely important, we focus on minimizing drops or ideally preventing them altogether.

Given these requirements, we choose to leverage link-layer flow control to construct a *lossless fabric* [5]. Switches in lossless fabrics use link-layer flow control to prevent their neighbors from transmitting to them when their buffers are full, thus ensuring that packets are not dropped due to congestion. Since the network is now responsible for preventing packet drops, this obviates the need for transport-level mechanisms that leverage the single-path assumption to respond to them more quickly (e.g., TCP's fast-recovery and fast retransmit). By simply disabling these mechanisms, we create our reorder resistant transport.

Next, we focus on creating an approach to balancing load across available network paths. The decision to employ link-layer flow control allows us to be ambitious - we can have every switch make load balancing decisions on a per-packet basis. However, a key question is how we can inform switches of downstream congestion so they can make more informed decisions.

Fortunately, link-layer flow control makes addressing this problem much more tractable. When network congestion increases beyond a certain point, flow control prevents upstream

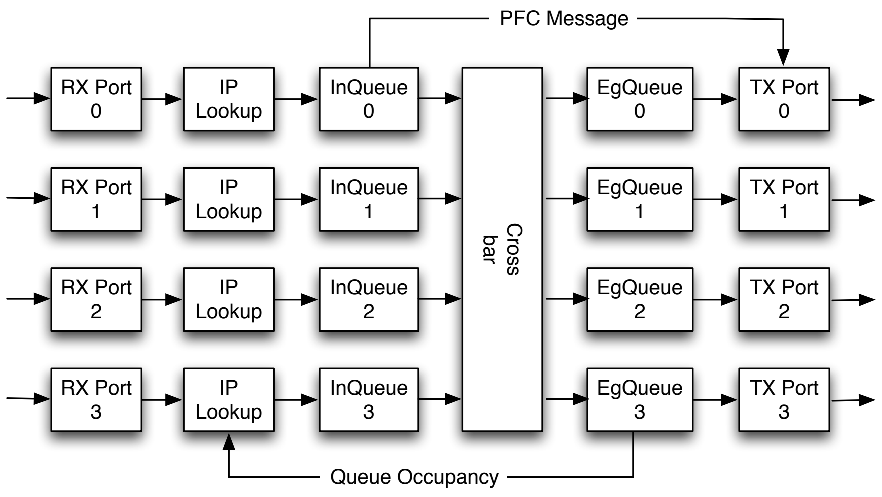


Figure 5.2: Assumed CIOQ Switch Architecture

switches from transmitting. If congestion continues to increase, the buffers at these upstream switches will fill, eventually preventing their upstream switches from transmitting as well. As link-layer flow control causes congestion information to propagate upstream, we simply use local buffer occupancies at every switch as an indicator of downstream congestion.

Lastly, we need to consistently prioritize the latency-sensitive short flows in the network. To do so, we identify the locations in the fabric where priorities should be set and considered. We made the applications responsible for setting priorities and ensure that the queuing, transmission, load balancing, and flow-control decisions our network employs all consider priority.

5.2.2 Stack Internals

Now we discuss the internal mechanisms in the *DeTail* stack that achieve the functionality presented earlier. We begin by describing our assumed switch architecture. Then we go up the stack, discussing what DeTail does at every layer. We conclude by discussing the benefits of our stack.

Assumed Switch Architecture

In Figure 5.2, we depict a four-port representation of a Combined Input/Output Queue (CIOQ) Switch. The CIOQ architecture is commonly used in today's switches [4, 47]. This architecture employs both ingress and egress queues, which we denote as InQueue and

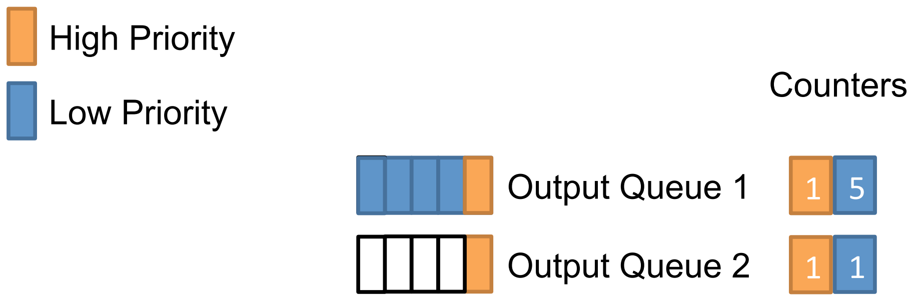


Figure 5.3: We maintain per-priority counters for each queue. Each counter representing the *drain bytes* for that priority. For clarity, in this figure, we count packets instead of bytes.

EgQueue, respectively. A crossbar moves packets between these queues.

When a packet arrives at an input port (e.g., RX Port 0), it is passed to the forwarding engine (IP Lookup). The forwarding engine determines on which output port (e.g., TX Port 2) the packet should be sent. Once the output port has been determined, the packet is stored in the ingress queue (i.e., InQueue 0) until the crossbar becomes available. When this happens, the packet is passed from the ingress queue to the egress queue corresponding to the desired output port (i.e., InQueue 0 to EgQueue 2). Finally, when the packet reaches the head of the egress queue, it is transmitted on the corresponding output port (i.e., TX Port 2).

To ensure that high-priority packets do not wait behind those with low-priority, the switch's ingress and egress queues perform strict priority queueing. Switches are typically capable of performing strict priority queueing between eight different priorities [8]. We use strict prioritization at both ingress and egress queues.

We employ strict priority queuing and higher layers of the stack make decisions based on ingress and egress queue occupancies. Thus, we cannot simply provide one counter per queue. Doing so will make higher layers unable to differentiate between queues that are full of high-priority packets and queues that only have low priority ones. Clearly, the inability to differentiate between these situations may cause suboptimal decisions to be made, particularly when handling high-priority packets.

We address this problem by having the switch provide per-priority ingress and egress queue occupancies to higher layers in the stack. Each queue maintains a *drain bytes* counter per priority. As shown in Figure 5.3, this is the number of bytes of equal or higher priority in front of a newly arriving packet. We chose this approach for two reasons. First, as our switches use strict priority queuing, it provides an accurate quantification of the amount of time the newly arriving packet will spend waiting in the queue. Second, we can maintain these values by simply incrementing/decrementing the counters for each arriving/departing packet in parallel. As a result, this approach works well, even at fast link speeds.

Having higher layers continuously poll the counter values of each queue may be prohibitively expensive. To address this issue, the switch associates a signal with each counter.

Whenever the value of the counter is below a pre-defined threshold, the switch asserts the associated signal. These signals enable higher layers to quickly select queues without having to obtain the counter values from each. When multiple thresholds are used, a signal per threshold is associated with each counter. We describe how these thresholds are set in Section 5.2.3.

Link Layer

At the link layer, DeTail employs flow control to create a lossless fabric. While many variants of flow control exist [14], we use the one that recently became part of the Ethernet standard: Priority Flow Control (PFC) [12]. PFC has already been adopted by vendors and is available on newer Ethernet switches [8].

The switch monitors ingress queue occupancy to detect congestion. When the drain byte counters of an ingress queue pass a threshold, the switch reacts by sending a Pause message informing the previous hop that it should stop transmitting packets with the specified priorities. When the drain byte counters reduce, it sends an Unpause message to the previous hop asking it to resume transmission of packets with the selected priorities[1].

We generate Pause/Unpause messages based on ingress queue occupancies because packets stored in these queues are attributed to the port on which they arrived. By sending Pause messages to the corresponding port when an ingress queue fills, DeTail ensures that the correct source postpones transmission.

Our choice of using PFC is based on the fact that packets in lossless fabrics can experience head-of-line blocking. With traditional flow control mechanisms, when the previous hop receives a Pause message, it must stop transmitting all packets on the link, not just those contributing to congestion. As a result, packets at the previous hop that are not contributing to congestion may be unnecessarily delayed. By allowing eight different priorities to be paused individually, PFC reduces the likelihood that low-priority packets will delay high priority ones.

Network Layer

At the network layer, DeTail makes congestion-based load balancing decisions. Since datacenter networks have many paths between the source and destination, multiple shortest path options exist. When a packet arrives at a switch, it is forwarded on to the shortest path that is least congested.

As mentioned earlier, DeTail monitors egress queue occupancies to make load balancing decisions. Ideally, DeTail would pick an acceptable port with the smallest drain byte counter at its egress queue for every forwarding decision. However, with the large number of ports in today's switches, the computational cost of doing so is prohibitively high. We leverage

[1]PFC messages specify the duration for which packet transmissions should be delayed. We use them here in an on/off fashion.

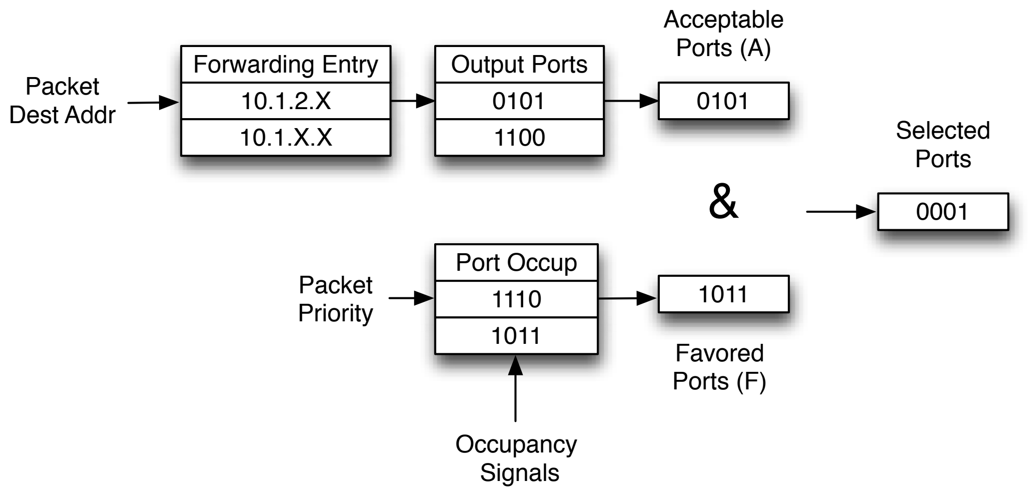


Figure 5.4: Performing Adaptive Load Balancing - A packet's destination IP address is used to determine the bitmap of *acceptable ports* (A). The packet's priority and port buffer occupancy signals are used to find the bitmap of the lightly loaded *favored ports* (F). A bitwise AND (&) of these two bitmaps gives the set of *selected ports* from which one is chosen.

the threshold-based signals described earlier. By concatenating all the signals for a given priority, we obtain a bitmap of the favored ports, which are lightly loaded.

DeTail relies on forwarding engines to obtain the set of available shortest paths to a destination. We assume that associated with each forwarding entry is a bitmap of acceptable ports that lead to shortest paths for matching packets[2].

As shown in Figure 5.4, when a packet arrives, DeTail sends its destination IP address to the forwarding engine to determine which entry it belongs to and obtains the associated bitmap of acceptable ports (A). It then performs a bitwise *AND* (&) of this bitmap and the bitmap of favored ports (F) matching the packet's priority, to obtain the set of lightly loaded ports that the packet can use. DeTail randomly chooses from one of these ports and forwards the packet[3].

During periods of high congestion, the set of favored ports may be empty. In this case, DeTail performs the same operation with a second, larger threshold. If that does not yield results either, DeTail randomly picks a port from the bitmap. We describe how to set these thresholds in Section 5.2.3.

[2]Bitmaps can be obtained with the TCAM and RAM approach as described in [15].
[3]Round-robin selection can be used if random selection is costly

Transport Layer

A transport-layer protocol must address two issues to run on our load-balanced, lossless fabric. It must be resistant to packet reordering and it cannot depend on packet loss for congestion notification.

Our lossless fabric simplifies developing a transport protocol that is robust to out-of-order packet delivery. The lossless fabric ensures that packets will only be lost due to relatively infrequent hardware errors/failures. As packet drops are now much less frequent, it is not necessary that the transport protocol respond agilely to them. We simply need to disable the monitoring and reaction to out-of-order packet delivery. For TCP NewReno, we do this by disabling fast recovery and fast retransmit. While this leads to increased buffering at the end host, this is an acceptable tradeoff given the large amount of memory available on modern servers.

Obtaining congestion information from a lossless fabric is more difficult. Traditionally, transport protocols monitor packet drops to determine congestion information. As packet drops no longer happen due to congestion, we need another approach. To enable TCP NewReno to operate effectively with DeTail, we monitor the drain byte counters at all output queues. Low priority packets enqueued when the appropriate counter is above a threshold have their ECN flag set. This forces the low priority, deadline-insensitive TCP flow contributing to congestion to reduce its rate.

These types of modifications often raise concerns about performance and fairness across different transports. As the vast majority of datacenter flows are TCP [18] and operators can specify the transports used, we do not perform a cross-transport study here.

Application Layer

DeTail depends upon applications to properly specify flow priorities based on how latency-sensitive they are. Applications express these priorities to DeTail through the sockets interface. They set each flow (and hence the packets belonging to it) to have one of eight different priorities. As the priorities are relative, applications need not use all of them. In our evaluation, we only use two.

Applications must also react to extreme congestion events where the source has been quenched for a long time. They need to determine how to reduce network load while minimally impacting the user. As the correct mechanisms depend heavily on the application running and the functionality it performs, this is outside the scope of our work.

Benefits of the Stack

DeTail's layers are designed to complement each other, overcoming limitations while preserving their advantages.

As mentioned in Chapter 2, link-layer flow control can cause head-of-line blocking. In addition to using priority, we mitigate this by employing adaptive load balancing and ECN.

Adaptive load balancing allows alternate paths to be used when one is blocked and ECN handles the persistent congestion that aggravates head-of-line blocking.

DeTail's per-packet adaptive load balancing greatly benefits from the decisions made at the link and transport layers. Recall that using flow control at the link layer provides the adaptive load balancer with global congestion information, allowing it to make better decisions. And the transport layer's ability to handle out-of-order packet delivery allows the adaptive load balancer more flexibility in making decisions.

5.2.3 Parameter Settings

Now that we have described the mechanisms employed by DeTail, we discuss how to choose their parameters. We also assess how end-host parameters should be chosen when running DeTail.

Link Layer Flow Control

A key parameter is the threshold for triggering PFC messages. Pausing a link early allows congestion information to be propagated more quickly, making DeTail's adaptive load balancing more agile. At the same time, it increases the number of control messages. As PFC messages take time to be sent and responded to, setting the Unpause threshold too low can lead to buffer underflow, reducing link utilization.

To strike a balance between these competing concerns, we must first calculate the time to generate PFC messages. We use the same approach described in [12] to obtain this value.

For $1GigE$, it may take up to $36.456\mu s$ for a PFC message to take effect[4]. $4557B$ (bytes) may arrive after a switch generates a PFC message. As we pause every priority individually, this can happen for all eight priorities. We must leave $4557B \times 8 = 36456B$ of buffer space for receiving packets after PFC generation. Assuming $128KB$ ingress buffers, this implies a maximum Pause threshold of $(131072B - 36456B)/8 = 11827$ *Drain Bytes* per priority. Setting the threshold any higher leads to potential packet loss.

Calculating the Unpause threshold is challenging because the specifics of congestion cause queues to drain at different rates. Our calculations simply assume a drain rate of 1*Gbps*, requiring an Unpause threshold of at least $4557B$ to ensure the ingress queues do not underflow. However, ingress queues may drain faster or slower than 1*Gbps*. If they drain slower, additional control messages may have to be sent, re-pausing the priority. If they drain faster, our egress queues reduce the likelihood of link underutilization.

These calculations establish the minimum and maximum threshold values to prevent packet loss and buffer underflow. Between the desire for agility and reduced control message

[4]We do not consider jumbo frames. Also, PFC is only defined for 10*GigE*. We use 1*GigE* for manageable simulation times. We base PFC response times on the time specified for Pause Frames. This is appropriate since 10*GigE* links are given the same amount of time to respond to PFC messages are they are to Pause Frames.

overhead, we set the Unpause threshold to the minimum value of 4557 *Drain Bytes* and the Pause threshold to 8192 *Drain Bytes* (halfway between the minimum and the maximum).

Clearly, the optimal threshold settings, and hence the buffer requirements differ based on both the number of priorities as well as link speeds. We discuss the implications for environments with higher link speeds and more priorities in the following chapter.

Adaptive Load Balancing

When performing threshold-based adaptive-load balancing, we must determine how many thresholds to have for a given priority (i.e., most favored, favored, and least favored ports) as well as what these thresholds should be. Clearly, increasing the number of thresholds increases complexity, so the benefits of each additional threshold must outweigh the complexity cost.

Through a simulation-based exploration of the design space with the other parameters as described above, we determined that having two thresholds of $16KB$ and $64KB$ yields favorable results.

Explicit Congestion Notification

The threshold for setting ECN flags represents a tradeoff. Setting it too low reduces the likelihood of head-of-line blocking but increases the chance that flows will back off too much, underutilizing the link. Setting it too high has the opposite effect. Through experiments, we determined that a threshold of $64KB$ drain bytes appropriately makes this tradeoff.

End-Host Timers

Setting the timeout duration (i.e., RTO_{min} in TCP) of end host timers too low may lead to spurious retransmissions that waste network resources. Setting them too high leads to long response times when packets are dropped.

Traditionally, transport-layer protocols recover from packet drops caused by congestion and hardware failures. Congestion occurs frequently, so responding quickly to packet drops is important for achieving high throughput. However, DeTail ensures that packet drops only occur due to relatively infrequent hardware errors/failures. Therefore, it is more important for the timeout duration to be larger to avoid spurious retransmissions.

To determine a robust timeout duration for DeTail, we simulated all-to-all incast 25 times with varying numbers of servers (connected to a single switch) and different values of RTO_{min}. During every incast event, one server receives a total of $1MB$ from the remaining servers. We saw that values of $10ms$ and higher effectively avoid spurious retransmissions.

Unlike this simulation, datacenter topologies typically have multiple hops. Hence, we use $200ms$ as RTO_{min} for DeTail in our evaluations to accommodate larger topologies.

5.3 Experimental Setup

As described in the previous section, DeTail depends heavily on the type of switch used and the timing of various events (e.g. Pause/Unpause reaction times). Before evaluating how DeTail performs, we must simulate and emulate the switch model using the platforms described in Chapter 3. Here we describe the modifications to our simulator and implementation in turn. In the following section, we evaluate DeTail.

5.3.1 Simulator

Our NS-3 based simulator closely follows the switch design depicted in Figure 5.2. Datacenter switches typically have 128-256KB buffers per port [18]. To meet this constraint, we chose per-port ingress and egress queues of 128KB.

Network simulators typically assume that nodes are infinitely fast at processing packets, this is inadequate for evaluating DeTail. We extended NS-3 to include real-world processing delays. Based on prior work, we employ 1Gbps links with 25μs switching delays [18]. We rely upon published specifications to break-down this delay as follows, providing explanations where possible:

- 12.24μs transmission delay of a full-size 1530B Ethernet frame on a 1$GigE$ link.
- 3.06μs crossbar delay when using a speedup of 4. Crossbar speedups of 4 are commonly used to reduce head of line blocking [47].
- 0.476μs propagation delay on a copper link [12].
- 5μs transceiver delay (both ends of the link) [12].
- 4.224μs forwarding engine delay (the remainder of the 25μs budget).

We incorporate the transceiver delay into the propagation delay. The other delays are implemented individually, including the response time to PFC messages.

One remaining limitation of the simulator is its lack of support for ECN. Consequently, our simulations do not evaluate explicit congestion notification (as discussed in Section 5.2.2). As we will show, even without ECN-based throttling of low priority flows our simulations demonstrate impressive results.

5.3.2 Implementation

By default, our Click implementation does not use CIOQ. Instead, the forwarding engine places packets directly into the output queue. This output-queued approach is poorly suited to DeTail because we rely on ingress queues to determine when to send PFC messages.

To address this difference, we modify Click to have both ingress and egress queues. When packets arrive, the forwarding engine simply annotates them with the desired output port and places them in the ingress queue corresponding to the port on which they arrived.

Crossbar elements then pull packets from the ingress queue to the appropriate egress queue. Finally, when the output port becomes free, it pulls packets from its egress queue.

Despite all of our attempts to optimize Click, Pause / Unpause messages still take far longer to be created and acted upon in software than in dedicated hardware. As a result, we had to either decrease the number of supported priorities or increase the buffers used within Click to prevent congestion-related packet drops. In the following section, we will show that our evaluation workloads consist of two traffic classes. As a result, we opted to decrease the number of priorities to two, thereby allowing us to provide a better assessment of the advantages of DeTail given the typical buffer constraints experienced in datacenter networks.

5.4 Experimental Results

In this section, we evaluate DeTail through extensive simulation and implementation, demonstrating its ability to reduce tail latencies for a wide range of workloads. We begin with an overview describing our traffic workloads and touch on key results. Next, we compare simulation and implementation results, validating our simulator. Later, we subject DeTail to a wide range of workloads under a larger topology than permitted by the implementation and investigate its scaled-up performance.

5.4.1 Overview

To evaluate DeTail's ability to reduce the flow completion time tail, we compare the following approaches:

Flow Hashing (*FH*): Switches employ flow-level hashing. This is the status quo and is our baseline for comparing the performance of DeTail.

Lossless Packet Scatter (*LPS*): Switches employ packet scatter, sending each packet on a randomly chosen shortest path, along with Priority Flow Control (PFC). While not industry standard, *LPS* is a naive multipath approach that can be deployed in current datacenters. The performance difference between *LPS* and DeTail highlights the advantages of Adaptive Load Balancing (ALB).

DeTail: As already explained in previous sections, switches employ PFC and ALB.

All three cases use strict priority queueing and use TCP NewReno as the transport layer protocol. For *FH*, we use a TCP RTO_{min} of $10ms$, as suggested by prior work [18, 55]. Since *LPS* and DeTail use PFC to avoid packet losses, we use the standard value of $200ms$ (as discussed in Section 5.2.3). Also, we use reorder buffers at the end-hosts to deal with out-of-order packet delivery.

We evaluate DeTail against *LPS* only in Section 5.4.4. For all other workloads, *LPS* shows similar improvements as DeTail and has been omitted for space constraints.

Traffic Model: Our traffic model consists primarily of high-priority data retrievals. For each retrieval, a server sends a 10-byte request to another server and obtains a variable sized response (i.e., data) from it. The size of the data (henceforth referred to as *retrieval data size*) is randomly chosen to be $2KB$, $8KB$, or $32KB$, with equal probability. We chose discrete data sizes for more effective analysis of 99^{th} and 99.9^{th} percentile performance. The rate of generation of these data retrievals (henceforth called *retrieval rate*) and the selection of servers for the retrievals are defined by the traffic workload. We assumed the inter-arrival times of retrievals to be exponentially distributed (that is, a Poisson process). Where specified, we also run low-priority, long background data transfers.

Key results: Throughout our evaluation, we focus on 99^{th} and 99.9^{th} percentile data retrieval latencies, or completion times, to assess DeTail's effectiveness. We use the percentage reduction in the completion times provided by DeTail over Flow Hashing as the metric of improvement. Our key results are:

- DeTail completely avoids congestion-related losses, reducing 99.9^{th} percentile completion times of data retrievals in all-to-all workloads by up to 71% over Flow Hashing.
- DeTail effectively moves packets away from congestion hotspots that may arise due to disconnected links, reducing 99.9^{th} percentile completion times by up to 89% over Flow Hashing. *LPS* does not do as well and actually performs worse than *FH* for degraded links.
- Reductions in individual data retrievals translate into improvements for sequential and partition-aggregate workflows, reducing their 99.9^{th} percentile completion times by 54% and 78%, respectively.

5.4.2 Simulator Verification

We use our implementation platform to validate our simulator. We construct a 36-node, 16-server FatTree topology. Over-subscription is common in datacenter networks [6]. To model the effect of a moderate over-subscription factor of four, we rate-limited the ToR-to-aggregate links to $500Mbps$ and the aggregate-to-core links to $250Mbps$.

We designated half of the servers to be front-end (web-facing) servers and half to be back-end servers. Each front-end server continuously selects a back-end server and issues a high-priority data retrieval to it. The data retrievals are according to a Poisson process and their rate is varied from 100 to 1500 retrievals/second.

For validation, our simulator used the same workload and topology, with parameters matched with that of the implementation. Figure 5.5 compares the simulation results with the implementation measurements. For rates ranging from 500 to 1500 retrievals/sec, the

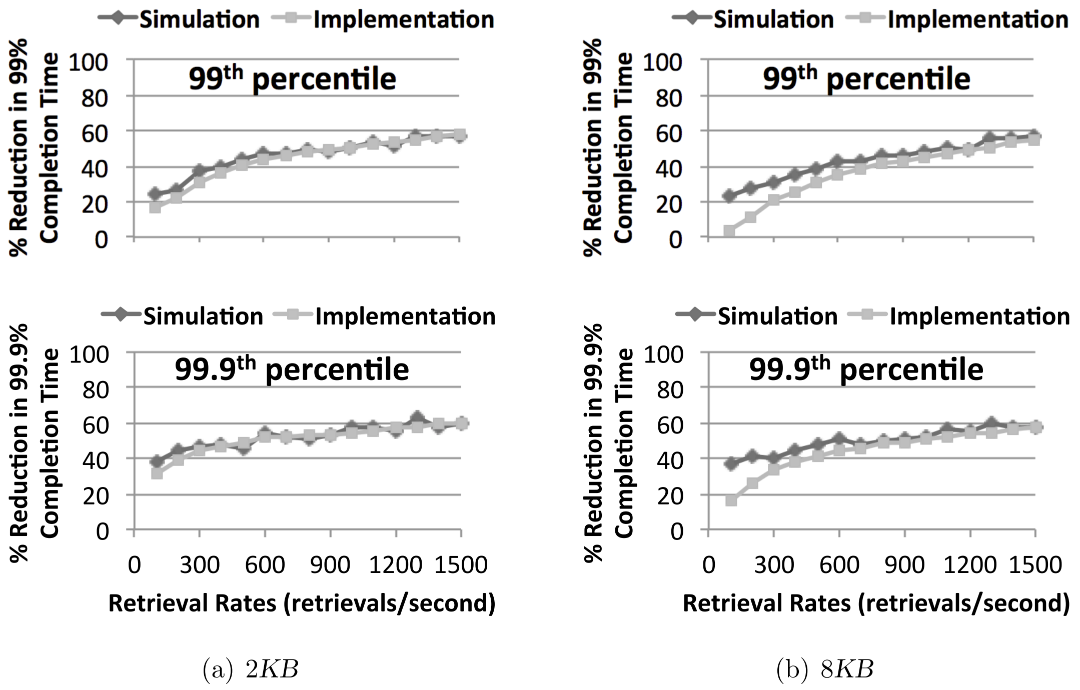


Figure 5.5: Comparison of simulation and implementation results - Reduction by DeTail over FH in 99^{th} and 99.9^{th} percentile completion times of $2KB$ and $8KB$ data retrievals

percentage reduction in completion time predicted by the simulator closely matches implementation measurements (results for $32KB$ data retrievals and LPS are similar). Note that the difference increases for lower rates. We hypothesize that this is due to end-host processing delays that are present only in the implementation (i.e., not captured by simulation) dominating completion times during light traffic loads.

Our results demonstrate that our simulator is a good predictor of performance that one may expect in a real implementation. Next, we use this simulator to evaluate larger topologies and a wider range of workloads.

5.4.3 Microbenchmarks

We evaluate the performance of DeTail on a larger FatTree topology with 128 servers. The servers are distributed into four pods having four ToR switches and four aggregate switches each. The four pods are connected to eight core switches. This gives an oversubscription factor of four in the network (two from top-of-rack to aggregate switches and two from aggregate to core switches). We evaluate two traffic patterns:

- **All-to-all**: Each server randomly selects another server and retrieves data from it. All 128 servers engage in issuing and serving data retrievals.

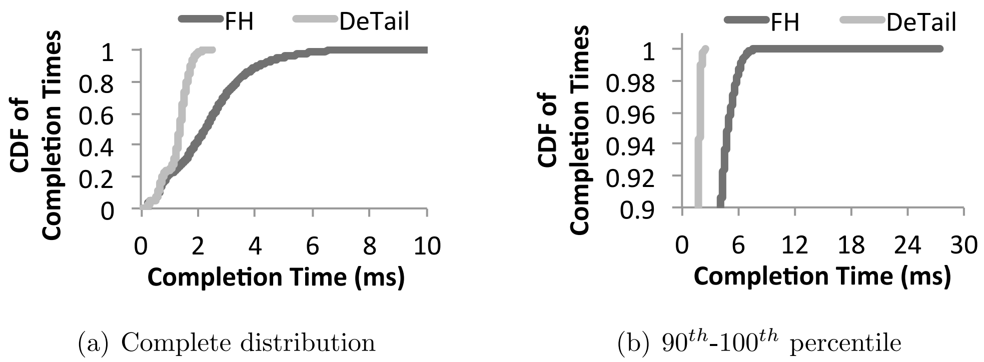


Figure 5.6: CDF of completion times of $8KB$ data retrievals under all-to-all workload of 2000 retrievals/second

- **Front-end / Back-end**: Each server in first three pods (i.e, front-end server) retrieves data from a randomly selected server in the fourth pod (i.e., back-end server).

The data retrievals follow a Poisson process. In addition, each server is engaged in, on average, one $1MB$ low-priority background flow. Using a wide range of workloads, we illustrate how ALB and PFC employed in DeTail reduce the tail of completion times as compared to *FH*.

All-to-all Workload: Each server generates retrievals at rates ranging from 500 to 2000 retrievals/second, which corresponds to load factors[5] of approximately 0.17 to 0.67. Figure 5.6 illustrates the effectiveness of DeTail in reducing the tail, by presenting the cumulative distribution of completion times of $8KB$ data retrievals under a rate of 2000 retrievals/second. While the 99^{th} and 99.9^{th} percentile completion times under *FH* were $6.3ms$ and $7.3ms$, respectively, DeTail reduced them to $2.1ms$ and $2.3ms$; a reduction of about 67% in both cases. Even the median completion time improved by about 40%, from $2.2ms$ to $1.3ms$. Furthermore, the worst case completion time was $28ms$ under *FH* compared to $2.6ms$ Flow completion times can increase by an order of magnitude due to congestion and mechanisms employed by DeTail are essential for ensuring tighter bounds on network performance.

Figure 5.7 presents the reductions in completion times for three data sizes at three retrieval rates. DeTail provided up to 70% reduction at the 99^{th} percentile (71% at 99.9^{th} percentile) completion times. Specifically, the 99.9^{th} percentile completion times for all sizes were within $3.6ms$ compared to $11.9ms$ under *FH*. Within each data size, higher rates have greater improvement. The higher traffic load at these rates exacerbates the uneven load balancing caused by *FH*, which ALB addresses.

[5]load factor is the approximate utilization of the aggregate-to-core links by high-priority traffic only

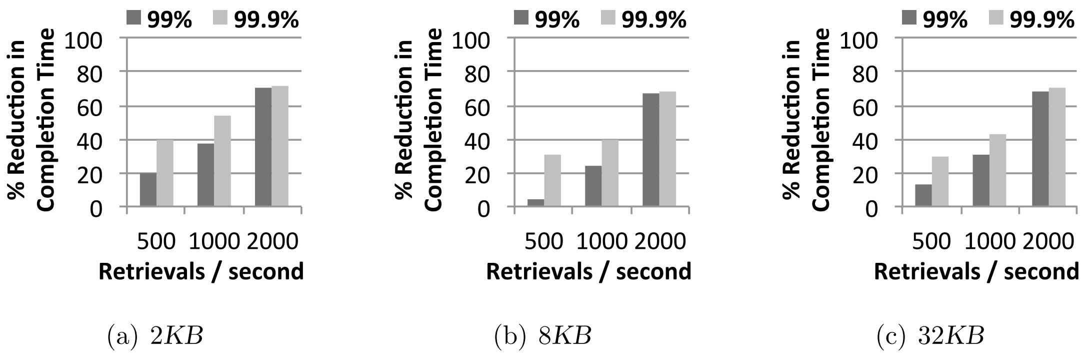


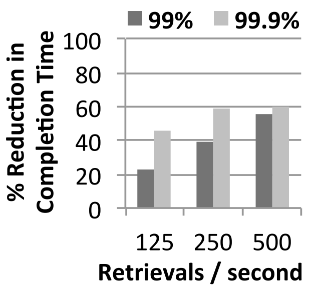
Figure 5.7: All-to-all Workload - Reduction by DeTail over *FH* in 99th and 99.9th percentile completion times of 2*KB*, 8*KB* and 32*KB* retrievals

Front-end / Back-end Workload: Each front-end server (i.e., servers in the first three pods) retrieves data from randomly selected back-end servers (i.e., servers in the fourth pod) at rates ranging from 125 to 500 retrievals/second, which correspond to load factors of approximately 0.17 to 0.67 on the aggregate-to-core links of the fourth pod. Figure 5.8 shows that DeTail achieves 30% to 65% reduction in the completion times of data retrievals at the 99.9th percentile. This illustrates that DeTail can perform well even under the persistent hotspot caused by this workload.

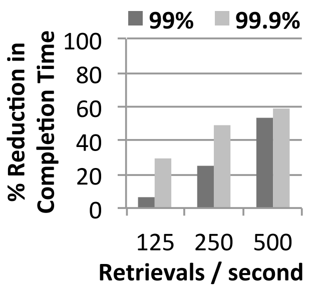
Long Background Flows: DeTail's approach to improving data retrievals (i.e., high-priority, short flows) does not sacrifice background flow performance. Due to NS-3's lack of ECN support, we evaluate the performance of background flows using the 16-server implementation presented earlier. We use the same half front-end servers and half-backend servers setup, and apply a retrieval rate 300 retrievals/second. Additionally, front-end servers are also continuously engaged in low-priority background flows with randomly selected back-end servers. The background flows are long; each flow is randomly chosen to be one of 1*MB*, 16*MB* or 64*MB* with equal probability. Figure 5.9 shows that DeTail provides a 38% to 60% reduction over *FH* in the average completion time and a 58% to 71% reduction in the 99th percentile. Thus, DeTail significantly improves the performance of long flows. A more extensive evaluation of DeTail's impact on long flows is left for future work.

5.4.4 Topological Asymmetries

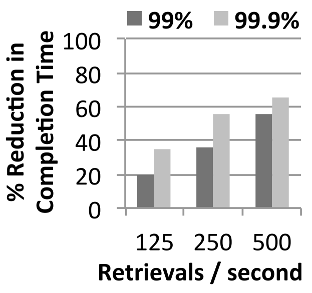
As discussed in Section 5.1.2, a multipath approach must be robust enough to handle topological asymmetries due to network component failures or reconfigurations. We consider



(a) 2*KB*



(b) 8*KB*



(c) 32*KB*

Figure 5.8: Front-end / Back-end Workload - Reduction by DeTail over *FH* in 99th and 99.9th percentile completion times of 2*KB*, 8*KB* and 32*KB* data retrievals

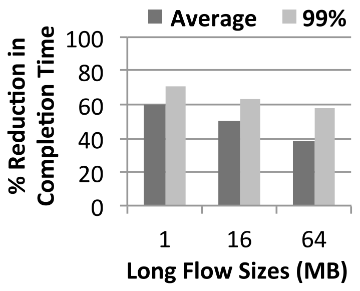
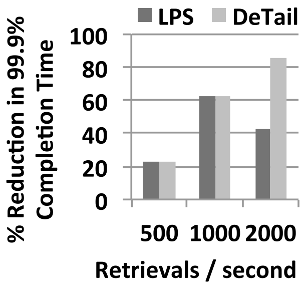


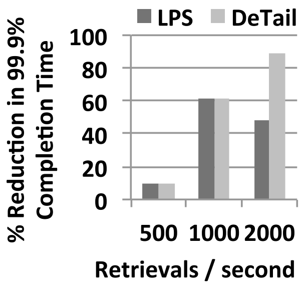
Figure 5.9: Long Flows - Reduction by DeTail in completion times of long, low-priority flows

two types of asymmetries: disconnected links and degraded links. These asymmetries lead to load imbalance, even with packet scatter. In this section, we show how ALB can adapt to the varying traffic demands and overcome the limitations of packet-level scattering. Besides *FH*, we evaluate DeTail against *LPS* to highlight the strength of ALB over packet scatter (used in *LPS*). We assume that the routing protocol used in the network has detected the asymmetry and converged to provide stable multiple routes.

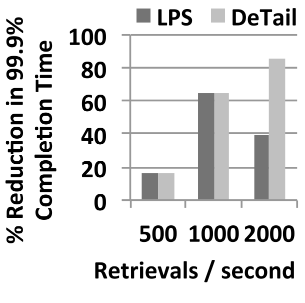
Disconnected Link: We evaluate an all-to-all workload with Poisson data retrievals on the same topology described in the previous subsection, but with the assumption of one disconnected aggregate-to-core link. Figure 5.10 presents the reduction in 99.9th percentile completion times for both *LPS* and DeTail (we do not present 99th percentile for space constraints). DeTail provided 10% to 89% reduction, almost an order of magnitude improvement (18ms under DeTail compared to 159ms under *FH* for 8*KB* retrievals at 2000



(a) $2KB$



(b) $8KB$



(c) $32KB$

Figure 5.10: Disconnected Link - Reduction by *LPS* and DeTail over *FH* in 99.9^{th} percentile completion times of $2KB$, $8KB$ and $32KB$ retrievals

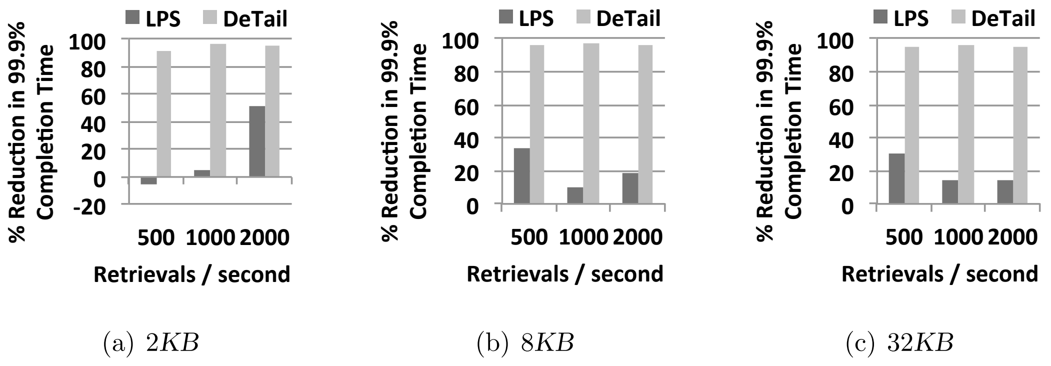
retrievals/second). *LPS's* inability to match DeTail's improvement at higher retrieval rates highlights the effectiveness of ALB at evenly distributing load despite asymmetries in available paths.

Degraded Link: Instead of disconnecting, links can occasionally be downgraded from $1Gbps$ to $100Mbps$. Figure 5.11 presents the results for the same workload with a degraded core-to-agg link. DeTail provided more than 91% reduction compared to *FH*. This dramatic improvement is due to ALB's inherent capability to route around congestion hotspots (i.e., switches connected to the degraded link) by redirecting traffic to alternate paths. While the 99.9^{th} percentile completion time for $8KB$ at 2000 retrievals/second (refer to Figure 5.11(b)) under *FH* and *LPS* was more than $755ms$, it was $37ms$ under DeTail. In certain cases, *LPS* actually performs worse than *FH* (i.e., for $2KB$, 500 retrievals/second).

In both fault types, the improvement in the tail comes at the cost of increased median completion times. As indicated in Chapter 4, reducing the tail is fair more important than reducing the median when it comes to meeting deadlines.

5.4.5 Web Workloads

Next, we evaluate how the improvements in individual data retrievals translate to improvements in the sequential and partition-aggregate workflows used in page creation. Here we randomly assign half the servers to be front-end servers and half to be back-end servers. The front-end servers initiate the workflows to retrieve data from randomly chosen back-end servers. We present the reduction in the 99.9^{th} percentile completion times of these workflows.



(a) $2KB$ (b) $8KB$ (c) $32KB$

Figure 5.11: Degraded Link - Reduction by LPS and DeTail over FH in 99.9^{th} percentile completion times of $2KB$, $8KB$ and $32KB$ data retrievals

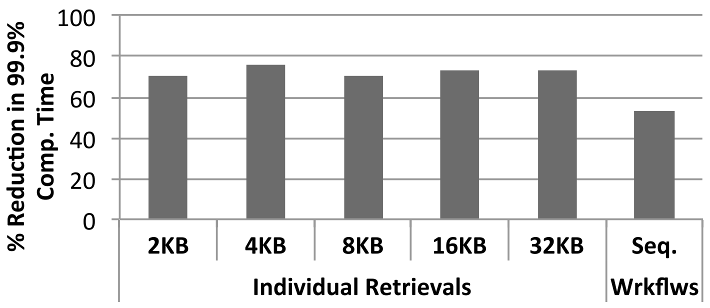


Figure 5.12: Sequential Workflows - Reduction by DeTail over FH in 99.9^{th} percentile completion times of sequential workflows and their individual data retrievals

Sequential Workflows: Each sequential workflow initiated by a front-end server consists of 10 data retrievals of size $2KB$, $4KB$, $8KB$, $16KB$, and $32KB$ (randomly chosen with equal probability). As described in the Chapter 4, these retrievals are performed one after another. Workflows arrive according to a Poisson process at an average rate of 350 workflows/second. Figure 5.12 shows that DeTail provides 71% to 76% reduction in the 99.9^{th} percentile completion times of individual data retrievals. In total, there is a 54% improvement in the 99.9^{th} percentile completion time of the sequential workflows – from $38ms$ to $18ms$.

Partition-Aggregate Workflows: In each partition-aggregate workflow, a front-end server retrieves data in parallel from 10, 20, or 40 (randomly chosen with equal probability) back-end servers. As characterized in [18], the size of individual data retrievals is set to $2KB$. These workflows arrive according to a Poisson process at an average rate of 600 workflows/second.

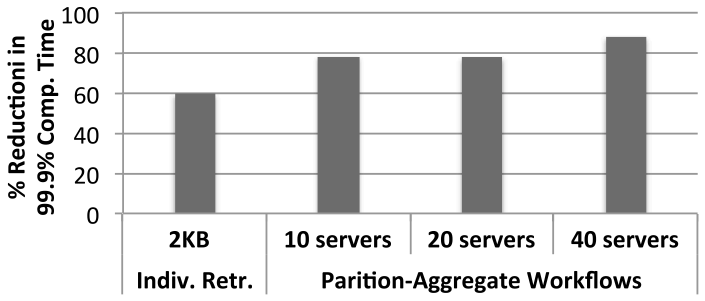


Figure 5.13: Partition-Aggregate Workflows - Reduction by DeTail over *FH* in 99.9th percentile completion times of partition-aggregate workflows and their individual retrievals

Figure 5.13 shows that Detail provides 78% to 88% reduction in 99.9th percentile completion times of the workflows. Specifically, the 99.9th percentile completion time of workflows with 40 servers was 17ms under DeTail, compared to 143ms under *FH*. This dramatic improvement is achieved by preventing the timeouts that were experienced by over 3% of the individual data retrievals under *FH*.

These results demonstrate that DeTail effectively manages network congestion, providing significant improvements in the performance of distributed page creation workflows.

5.5 Takeaways

Our evaluation demonstrates that DeTail achieves impressive performance results, reducing the tail of flow completion times by over 70%. Furthermore, DeTail is robust to link degradation and failure, which can lead to topological asymmetries. In the presence of these, DeTail performs even better than flow hashing, reducing tail completion times by up to 91%. From these results, we see the impressive ability of a tightly integrated set of layers to reduce data retrieval latencies, achieving a much tighter distribution in completion times.

However, DeTail's reliance on lossless interconnects represents to additional challenges. First, although DeTail is effective at mitigating head-of-line blocking it can still occur and be particularly harmful when hosts or servers misbehave. Consider the case where a switch or host sends PFC messages nonstop, preventing the previous hop from sending to it. Messages begin to queue at the previous hop, until it must in turn send PFC messages, stopping all traffic destined to it. In this way, large portions of the network fabric may become blocked by a single misbehaving entity. Facebook engineers tell us that that this limitation is a big concern because of the high availability requirements of web-facing datacenters.

Second, DeTail requires a large amount of buffering to ensure lossless operation. Recall that we must set aside capacity to ensure that packets are not dropped before PFC messages take effect. As link speeds and the number of available priorities increase, buffer requirements will as well. This is especially significant given the small buffers typically available on today's switches.

In the next chapter, we present FastLane, our approach to addressing these limitations.

Chapter 6

FastLane

In the last chapter, we proposed a solution for reducing tail latency. While effective, this approach has two limitations that need to be addressed: (i) it is not resilient to misconfigured servers and switches and (ii) it has large buffer requirements that increase with link speed. These limitations represent a significant hurdle in the presence of ever-increasing link speeds and SLA requirements.

In this chapter, we begin by describing how these limitations stem from DeTail's reliance on lossless interconnects. By disallowing packet drops, lossless interconnects reduce the range of environments they can effectively support. To address this problem, we propose a different approach - we allow packet drops but require switches to directly notify senders when they occur. As a result, direct notifications enable us to dramatically reduce tail latency while avoiding the limitations of lossless interconnects.

We evaluate prior attempts to perform direct notification showing that they are unable to achieve the anticipated tail reduction. Based on a thorough analysis, we determine which design decisions prior approaches employed that limited their effectiveness. We use the insights gained from this analysis to design FastLane, a new approach to directly notifying the source when packet drops occur.

We evaluate FastLane, demonstrating its ability to reduce the tail by up to 81% on networks with 10Gbps links and just 128KB of buffering per port. Furthermore, we demonstrate that FastLane's performance degrades gracefully with smaller buffer sizes, making it appropriate for a wide range of environments. The one limitation of FastLane is that network stacks incorporating it can no longer use local output queues as an indicator of downstream congestion. We discuss how to address this problem at the end of this chapter.

6.1 The Disadvantages of Lossless Interconnects

The disadvantages of lossless interconnects stem from their decision to prohibit congestion-related packet drops. Here we describe how this requirement leads to both susceptibility to misconfigured servers / switches as well as to larger buffer demands.

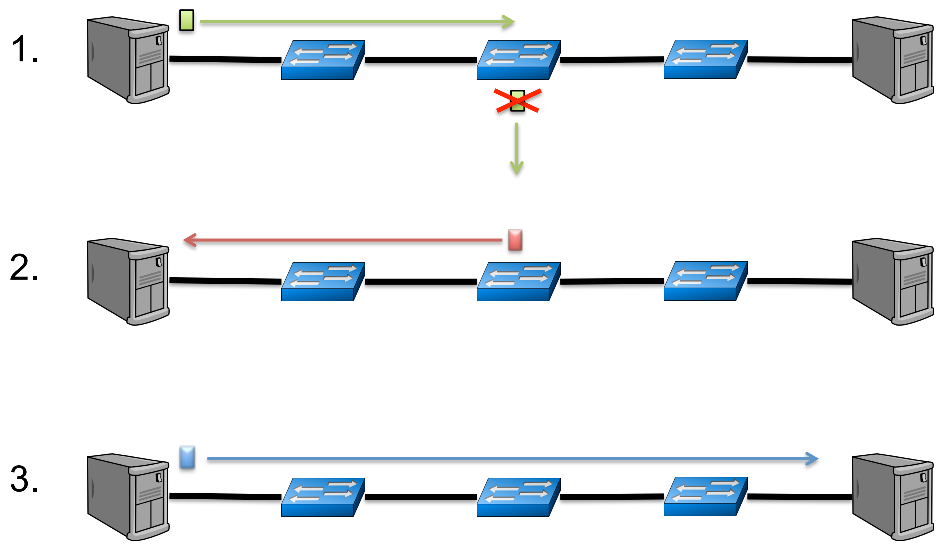


Figure 6.1: In response to a packet drop (1), the switch sends a notification directly to the source (2). Upon receiving the notification, the source resends the packet (3).

Datacenter networks employing lossless interconnects become susceptible to switch/server misconfiguration because they lose the ability to drop packets. If just one network device is able to prevent its packets from draining, buildup will occur. As buffers begin to fill, flow control messages will propagate, causing head-of-line blocking and stalling the whole network. The network will experience lockup and be unusable until the buffers are somehow cleared.

By allowing networks to drop packets, we can reduce the impact of misconfiguration. Broadly, a misconfigured switch will have two options: it will be able to drop all packets or to send them across all ports. If the misconfigured switch does the former, then only the traffic traversing through it will be affected. Even if it does the latter, it will only be able to consume a small fraction of the total resources of then network. In either case, the network will degrade gracefully, avoiding lockup.

The increasing buffer requirements of lossless interconnects are even more apparent. Because of the limited ability to reduce processing speeds and the inability to change propagation delays, as link speeds go up, the receiving end must set aside evermore capacity to accept evermore packets before Pause messages take effect. Switches that do not do this will be unable to prevent congestion-related packet drops. As buffers represent a large fraction and hence cost for a switch ASIC, these increased resource demands may become prohibitively expensive [19].

6.2 Direct Drop Notifications

Rather than preventing drops with link-layer flow control, we minimize the time it takes to recover from them. By having switches directly notify sources of packet drops as soon as they occur, we enable sources to retransmit early. Sources learn of packet drops as quickly as possible, avoiding time-consuming timeouts which drastically inflate latency.

In Figure 6.1, we depict an example of how direct notifications work. When a packet drop occurs, the switch transmits a notification back to the sender. Upon receiving the notification, the sender decides when to retransmit the packet. Importantly, it does not wait for a time-consuming timeout.

Unlike link-layer flow control, drop notifications do not have large buffer requirements and are resilient in the face of misconfigured servers and switches. Since switches can still drop packets, lockup will not occur. As we will show later in this chapter, drop notifications only consume a small fraction of network resources (1% of bandwidth, 2.5% of buffers), even under extreme traffic patterns. By simply placing limits on the bandwidth and buffers used by notifications at every node in the network, we can limit the resources consumed by misconfigured nodes, ensuring graceful degradation.

At the same time, drop notifications retain many of the same positive aspects of link-layer flow control. Just like link-layer flow control, they improve high-percentile latencies by avoiding timeouts. They also allow per-packet load balancing. With drop notifications, out of order delivery is no longer necessary in order to quickly detect and recover from loss. As in the previous chapter, we can disable the server's reaction to out of order acknowledgements and spread the packets across the paths available in the network.

6.3 Existing Direct Notification Schemes

Using direct notification for improving flow completion time was proposed by ICMP Source Quench and Quantized Congestion Notification (802.1Qau) [1, 36]. To the best of our knowledge, both have failed to gain widespread adoption, and Source Quench has since been deprecated. Here we investigate why these proposals are ineffective at reducing high percentile completion times in datacenters. We use the insights gained to propose a series of design principles that must be satisfied for direct notification to be effective.

6.3.1 ICMP Source Quench

ICMP source quench was a protocol switches used *to signal congestion* to the source. A switch experiencing congestion generates and sends ICMP messages to sources requesting them to reduce their transmission rates. The quench message contained the first 8 bytes of the offending packet's transport header so the source could determine which flow to throttle.

The advantage of this approach is that it enabled switches to generate source quench messages as frequently as their control plane supports. The specification did not have to concern itself with the generation rates of different switch hardware. However, conditions under which such messages were sent were poorly defined, and the message itself did not contain any information as to what triggered it. The latter is a main disadvantage, as it was impossible for sources to identify whether the notification was sent in response to a packet drop or building congestion. As a result, when Linux supported Source Quench, it responded to those messages in the same way as it does to ECN [53]. It reduced the congestion window but it did not retransmit the packets until out-of-order delivery or a timeout indicated a loss.

Source quench messages suffered from two other problems. As they had the same priority as the offending data packet, quench messages often took a long time to arrive at the source, thus diminishing potential gains [22]. At the same time, there were no safeguards to ensure that source quench messages did not overconsume resources in the presence of extreme congestion.

To quantify the impact of these design decisions, we evaluated Source Quench using the workload in §6.6. In this workload, we have bursts of short flows (up to 32KB in length) and long flows (1 MB in length). Figure 6.2 shows the 99.9th percentile completion times for the short flows. We see that under this workload, Source Quench does not perform significantly better than TCP. More importantly, we see that an idealized drop notification mechanism, without the limitations of Source Quench, could reduce high-percentile completion times by 81%.

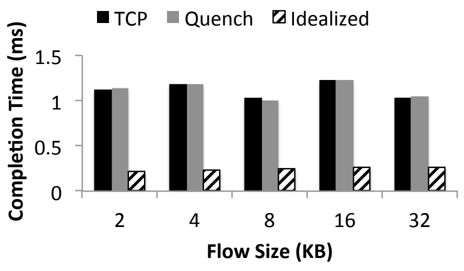


Figure 6.2: 99.9th percentile flow completion times.

6.3.2 Quantized Congestion Notification

As discussed in Chapter 2, Quantized Congestion Notification (QCN) is a direct notification scheme proposed as part of the standardized data center bridging protocols [1]. With QCN, switches send notifications directly to sources, informing them *the extent of the congestion* being experienced. Upon receiving notifications, sources reduce the rate of transmission, based on the amount of congestion reported. Sources then periodically increase their transmission rates until another notification is received.

The key limitation of QCN is that rate-limiting is performed in the NIC. This has the following problems: (i) transport is unaware of congestion being experienced and cannot make more informed decisions (e.g., MPTCP selecting another path [52]), (ii) QCN cannot discern whether acknowledgments are being received, and must instead rely on a combination of timers and bytes transmitted to determine when to raise the transmission window, and (iii) in practice NICs have an insufficient number of rate limiters, so flows may be grouped together, causing head-of-line blocking [18]. The lack of coordination between the rate limiter and transport has led to significant drops and TCP timeouts. QCN can degrade TCP performance so significantly that prior work recommends enabling QCN *only* in heterogeneous environments where it is beneficial to control unresponsive flows (e.g., UDP) [29].

6.4 Direct Notification Design Principles

Based on the lessons learned from a deeper investigation of the advantages and the disadvantages of the ICMP Source Quench and the QCN protocols, we have distilled a set of design principles for direct notifications:

1. **Notifications (and the triggers that generate them) must be well-specified:** When a notification does not make it clear which packet triggered it and whether the original packet was dropped, sources cannot determine the appropriate action to take. If sources respond conservatively, delaying transmission until an indirect indicator (e.g. a timeout) arrives, flows will suffer large delays. If sources respond aggressively, retransmitting the packet, they risk increasing network load aggravating congestion events.

2. **Notifications must be created in the data plane:** When the network is congested, switches may have to generate notifications for many flows within a short time. If notifications are created by the control plane, they may overwhelm it in meeting the generation requirements of the protocol. Ideally, a notification could be generated using simple modifications on the original packet, thus ensuring quick generation in the data plane.

3. **Notifications must be transmitted with high priority:** Queuing delays at each hop can be much larger than uncongested network RTTs. Transmitting notifications at high priority avoids these delays, informing the source as quickly as possible. Ideally, the notification will be extremely small and prioritizing them will not significantly delay the transmission of already enqueued data packets.

4. **Safeguards must ensure that notifications do not aggravate congestion events:** The transmission of high-priority notifications takes resources away from other traffic. We must ensure that notifications do not consume too many resources, aggravating congestion events. In the presence of persistent congestion, notifications should be dropped and sources should timeout, ensuring the stability of the network.

5. **Notifications must be sent to the transport layer:** Lower-layer mechanisms for regulating transmission rates do not have sufficient flow-level information to make informed decisions about the state of congestion. As a result, they must employ heuristics, which in turn may harm high-percentile flow completion times. Moreover, by hiding congestion/drop information from transport, they prevent it from making the best decision possible.

From Table 6.1, we see that ICMP Source Quench does not satisfy Design Principles 1-4 and QCN does not satisfy principles 3-5. We argue that while simple, these principles are fundamental for achieving predictable flow completion times. In the next section, we present the design of our direct notification scheme, FastLane, discussing how it achieves these goals. In §6.6, we evaluate FastLane, demonstrating that it dramatically reduces tail latency.

Principle	1	2	3	4	5
Source Quench	$\times$	$\times$	$\times$	$\times$	$\surd$
QCN	$\surd$	$\surd$	$\times$	$\times$	$\times$
FastLane	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Table 6.1: Design principles satisfied by ICMP Source Quench, QCN, and FastLane.

6.5 FastLane Protocol

In this section, we begin with an overview of FastLane. Next, we delve into the details of FastLane's notifications. We show that they provide pinpoint information to the source, consume very few network resources, and can be generated with low latency. Later, we describe the safeguards FastLane employs to ensure that notifications do not consume excessive resources during periods of extreme congestion. We conclude this section by discussing the transport modifications required to support FastLane.

6.5.1 Overview

When multiple sources share a path, the queues of a switch on it may start to fill. Initially, the switch has sufficient resources to buffer arriving packets. But eventually, it runs out of capacity and must discard some packets. This is where FastLane takes action. For every dropped packet, it sends a notification back to the source, informing it which packet was lost.

To provide the source with sufficient information to respond effectively, the notification must contain at least (i) the transport header and length of the dropped packet and (ii) a flag that differentiates it from other packets. The notification is sent to the source with the highest

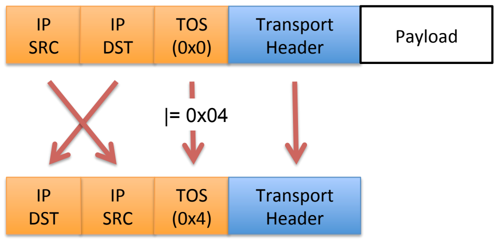


Figure 6.3: Transforming packets into notifications.

priority, informing it of the drop as quickly as possible. Upon receiving this notification, the source determines precisely what data was dropped and retransmits accordingly.

During periods of congestion, it may be best to postpone retransmitting the dropped packet. Section 6.5.4 describes how transports decide when to retransmit. To protect against extreme congestion, FastLane also employs explicit safeguards that cap the bandwidth and buffers used by notifications (Section 6.5.3).

6.5.2 Generating Notifications

Drop notifications must provide sources with sufficient information to retransmit the dropped packet (**Principle 1**). To achieve this goal, they should include (i) a flag / field differentiating them from other packets, (ii) the source and destination IP addresses and ports denoting the appropriate flow, (iii) the sequence number and packet length to denote which bytes were lost, and (iv) the acknowledgement number and control bits so the source can determine the packet type (i.e., SYN, ACK, FIN).

A naive approach to generating notifications would involve the control plane's general-purpose CPU. But the control plane could become overwhelmed when traffic bursts lead to drops, generating many notifications within a short duration.

Instead, we developed a series of simple packet transformations that can quickly be performed in the data plane (**Principle 2**). The transformations to create a FastLane notification are depicted in Figure 6.3. We start with the packet to be dropped and then (i) flip the source and destination IP address, (ii) set the IP TOS field, and (iii) truncate the packet, removing all data past the TCP header. We then forward the packet on to the input port from which it arrived. The input port assigns and transmits the packet with the highest priority (**Principle 3**). While we expect that this approach would be performed in hardware, we note that transforming a packet only takes 12 lines of Click code [45].

Our transformations need to provide one more piece of information - the length of the original packet. We have two options for accomplishing this (i) we can avoid modifying the total length field in the IP header, keeping it the same as the original packet, or (ii) we can create a TCP option that contains the length and is not truncated. Our evaluation uses the former approach.

This approach relies solely on simple packet manipulation. Prior work has demonstrated that such operations can be performed very quickly in the data plane [27]. Additionally, sending the packet back on the input port, while not strictly necessary, avoids the need to perform an additional IP lookup. Lastly, as the IP header checksum is a 16 bit one's complement checksum, flipping the source and destination IP addresses does not change its value. We can simply update it incrementally for the changes in the TOS field.

6.5.3 Controlling Resource Consumption

Notifications sent in response to drops can contribute to congestion in the reverse path. They take bandwidth and buffers away from regular packets, exacerbating congestion events. As FastLane prioritizes notifications so they arrive as quickly as possible, safeguards must be in place to ensure that they do not harm network performance.

Our safeguards take the form of bandwidth and buffer caps (**Principle 4**). To understand how to set these caps, we must analyze both average and short-term packet loss behavior and the resulting increase in notification load. A high-level goal when setting these caps is for notifications to be dropped when the network is experiencing such extreme congestion, that *the best option is for sources to timeout.*

Controlling Bandwidth

To understand how much bandwidth should be provisioned for drop notifications, we analyze the impact that average packet drop behavior has on notification load. Through this approach, we can bound worst-case bandwidth use.

Given a drop probability, p, we calculate the fraction of the load used by notifications as:

$$l_n = \frac{ps_n}{s_r + ps_n}, \tag{6.1}$$

where s_r is the average size of a regular (non-notification) packet and s_n is the size of the notification. To obtain a quantitative result, we assume that packets are 800 B long and notifications are 64 B long. We choose the packet size based on reports from production datacenters [24]. Based on these assumptions, we see that just 1% of the load would be used by notifications if 12% of the packets were being dropped. As a 12% drop rate would cause TCP's throughput to plummet, we cap the links of every switch, clocking out notifications at a rate limited to 1% of the capacity of the link. We ensure that our approach is work conserving – both FastLane's notifications and regular traffic use each other's spare capacity when available.

When FastLane's notifications are generated faster than they are clocked out, the buffers allocated to them start to fill. Once these buffers are exhausted, notifications are dropped. We argue that at this point, the network is so congested that letting the drop occur and

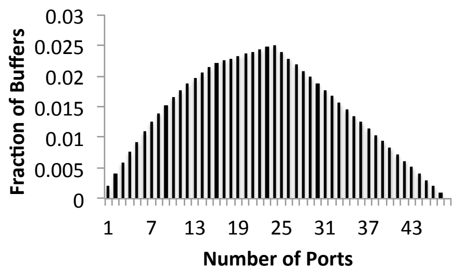


Figure 6.4: The fraction of a switch's buffers used by notifications when ports receive bursts simultaneously.

triggering a timeout is the best course of action for returning the network to a stable state. We describe how to size the buffers used by notifications next.

Controlling Buffers

Traffic bursts may lead to many packets being dropped over short timescales. As a result, many drop notifications may be created and buffered at the switch. We need to determine how much buffering to set aside for drop notifications, so we can leave as much as possible for regular transmissions. To do this, we must consider a variety of factors, including burst size and how many bursts can arrive simultaneously at a switch.

We begin by looking at a single burst. In the worst case, there may be no buffering available to absorb the packets of the burst. Then the number of bytes necessary to store the resulting notifications is approximated by the following equation:

$$b_{size} \times \frac{n_{size}}{d_{size}} \times (1 - \frac{1}{p_{in}}), \tag{6.2}$$

where b_{size} is the size of the burst, n_{size} is the size of the notification, d_{size} is the size of the average data packet and p_{in} is the number of ports simultaneously sending to the same destination. The first part of this equation calculates how many notifications (in bytes) would be created if all of the packets in the burst were dropped. The second part of the equation accounts for the fact that the port receiving the burst is simultaneously transmitting packets. This means that b_{size} / p_{in} packets will sent by the output port while receiving the burst. They will not be dropped and notifications for them will not be generated.

Multiple bursts may arrive at the same switch simultaneously. For each one we will need to store the number of bytes specified by Equation 6.2. However, the same input port cannot simultaneously contribute to multiple bursts. When combined with Equation 6.2, this means that assigning an input port to a new burst reduces the number of notifications generated by the previous one.

To provide some intuition for the implications of this property, we plot the fraction of buffers consumed when varying numbers of a switch's ports simultaneously receive bursts. For this calculation we assume (i) burst sizes of 160KB, doubling the typical burst size reported by prior work [18] and (ii) a 48-port switch with 128KB per port as seen in production TOR switches [3].

In Figure 6.4, we depict the fraction of the switch's buffers consumed when varying numbers of its ports receive simultaneous bursts. When calculating these values, we assume that all the input ports are used and that they are spread evenly across the bursts.

From this figure, we observe that increasing the number of ports that are simultaneously receiving bursts beyond a certain point *decreases* the number of drops and hence the number of notifications generated. To understand why this happens, we look at Equation 6.2. Note that as the number of simultaneous burst increases, the number of ports contributing to each goes to 1, driving the number of bytes used by notifications to zero.

Based on this analysis, we see that allocating 2.5% of switch buffers should be sufficient to support drop notifications. In our evaluation we use a cap of $2.5\% \times 128KB = 3.2KB$. However, we note that FastLane is still useful even when its buffer allocation is exhausted and some notifications are dropped. Environments with strict deadlines will see a larger fraction of flows complete on time [41, 56]. Scenarios with hundreds of sources participating in Incast will complete faster because there will be fewer rounds of timeouts and synchronized pull-backs.

6.5.4 Transport Modifications

Now that we have described how to generate notifications safely and efficiently, we turn our attention to the transport modifications required to make use of them (**Principle 5**). Here, we discuss how TCP uses notifications to improve high-percentile flow completion times. Later, we demonstrate the generality of our approach by describing how pFabric can leverage notifications as well.

TCP

TCP uses notifications to perform retransmission and rate throttling as well as to support multiple paths. We now describe the details of each in turn.

Retransmission and Rate Throttling: The goal of FastLane is to enable transport protocols to quickly identify and retransmit dropped packets. However, in certain cases, retransmitting as quickly as possible may aggravate congestion events. In the presence of persistent congestion, retransmitted packets may be dropped at the point of congestion, over and over again, creating a *ping-pong* effect. This wastes both upstream bandwidth and buffers and is hence undesirable.

Our modifications to TCP must strike a balance between retransmitting dropped packets as quickly as possible and delaying transmission to mitigate congestion events. Fortunately, addressing this problem for Control Packets (i.e., SYN, FIN, ACK) is simple. We retransmit them immediately as they are small and hence unlikely to significantly contribute to congestion[1].

The retransmission of data packets is more challenging to address. Ideally, we would wait precisely the amount of time necessary to avoid a packet drop before retransmitting. Unfortunately, given the complex dynamics of the network in addition to unpredictable server delays, determining the wait time is very difficult. Instead, we propose a simpler approach. We *measure* the ping-pong behavior to determine how much to throttle the number of simultaneous retransmissions.

Every TCP source maintains a list of entries for packets being retransmitted, sorted by their sequence number. Here, retransmitted packets are those which are unacknowledged and for which notifications have been received since the last timeout. Entries in this list are annotated with the number of retransmission attempts *entry.retx* as well as a flag indicating whether a packet is being retransmitted *entry.issent*. The source also maintains two variables *sim_retx* and *bound_sim*. *sim_retx* tracks the number of retransmissions in flight, while *bound_sim* sets the upper bound. Similarly to TCP's congestion recovery scheme, on the first drop notification triggering recovery, we set *bound_sim* to $\alpha = \frac{cwnd}{2}$, where *cwnd* is the congestion window. For every drop notification while in recovery mode, we exponentially decrease *bound_sim* according to the following equation:

$$bound\_sim \leftarrow \frac{\alpha}{max(entries.retx)}$$

We then traverse the list, in order of sequence number, retransmitting packets for which *entry.issent* is false until $sim\_retx \geq bound\_sim$. As acknowledgments for retransmitted packets arrive, reducing *sim_retx*, additional packets in the list are retransmitted. When all of the packets in the list are acknowledged, the source exits recovery, setting the congestion window to α. The algorithm for processing drop notifications is presented in 1.

For clarity, we omit the following functionality. As TCP relies on cumulative acknowledgements, we must always resend the packet with the smallest sequence number to ensure forward progress. This means that even if *sim_retx* equals *bound_sim*, we must retransmit the first packet whenever a notification arrives for it. We achieve this by allowing *sim_retx* to grow above *bound_sim* when it is necessary to satisfy this constraint.

Supporting Multiple Paths: The cumulative nature of acknowledgments makes it challenging to extend TCP to effectively use multiple paths. Cumulative acknowledgments do not specify the number of packets that have arrived out of order. This number is likely to be high in multipath environments (unless switches restrict themselves to flow hashing). Packets received out of order have left the system and are no longer contributing to congestion.

[1]Cases where control packet retransmission significantly adds to congestion are extreme. In this situation, we rely on the bandwidth and buffer caps to drop notifications, forcing timeouts and returning the network to a stable state.

Algorithm 1 Maintains a list of entries for dropped packets

```
If entry.seqno exists in list
    $entry.retx \leftarrow entry.retx + 1$
    $entry.issent \leftarrow 0$
    $sim\_retx \leftarrow sim\_retx - 1$
Else
    create new entry for seqno
    $entry.retx \leftarrow 1$
    $entry.issent \leftarrow 0$
    insert entry into list

$bound\_sim \leftarrow \alpha / max(entries.retx)$

For entry in list
    If $sim\_retx < bound\_sim$ && $entry.issent$ is 0
        retransmit packet having $entry.seqno$
        $entry.issent \leftarrow 1$
        $sim\_retx \leftarrow sim\_retx + 1$
```

Thus this information would allow TCP to safely inflate its congestion window and hence achieve faster completion times.

To address this problem, we introduce a new TCP option that contains the number of out-of-order bytes received past the cumulative acknowledgment. When a source receives an acknowledgment containing this option, it accordingly inflates the congestion window. This allows more packets to be transmitted and reduces dependence on the slowest path (i.e., the one whose data packet was received late).

How much the congestion window should be increased depends on whether the acknowledgment is a duplicate. If the acknowledgement is new, then the window should be inflated by number of out-of-order bytes stored in the TCP option. If the acknowledgment is a duplicate, then the window should be inflated by the maximum of the new out-of-order value and the current inflation value. This ensures correct operation even when acknowledgments themselves are received out-of-order.

pFabric

pFabric is a recent proposal that combines small switch buffers, fine-grained prioritization, and small RTOs to improve high percentile flow completion times [20]. To leverage the multiple paths available in the datacenter, pFabric avoids relying on in-order delivery. Instead it uses SACKs to determine when packets are lost and timeouts to determine when to retransmit them.

When a FastLane notification arrives, we have pFabric store it in a table, just like TCP. But, the response to notifications is based on the congestion control algorithm of pFabric.

Before resending any data packets, the source sends a probe to the destination. The probe packet is used as an efficient way to ensure that congestion has passed. Once the probe is acknowledged, the source begins resending up to *bound_sim* packets. In this case, *bound_sim* starts at 1 whenever a notification arrives, and increases exponentially with every successful retransmission, in effect simulating slow start.

6.6 Experimental Results

We now evaluate the performance of FastLane under a wide variety of datacenter-oriented network configuration and application workloads — we vary short flows from 2KB to 32KB, network utilization from 20% to 80%, the fraction of total load contributed to by short flows from 10% to 50%, buffer sizes from 16KB to 128KB, and the resource caps imposed on FastLane from 0.25× to 2× of those computed in §6.5.

We compare the performance of FastLane against that of TCP-NewReno with CoDel early marking [34, 48] and pFabric [20]. TCP-NewReno is a well-established, well-tested simulation model and pFabric is a multipath protocol focused on improving the performance of short flows. For both protocols, we send data in the first RTT, similar to TCP Fast Open [51]. While DCTCP [18] is the best-known TCP for data centers, we chose to compare our performance to the more recently proposed pFabric since the latter outperforms both DCTCP and PDQ [18, 41] under similar workloads.

Our key findings from the evaluation are:

- FastLane reduces 99.9th percentile short flow latency, or completion times, by 81% over TCP and 52% over pFabric. We note that pFabric outperforms DCTCP by 4x at the 99th percentile.

- FastLane achieves the above performance by using just 1% higher bandwidth and 2.5% larger buffers for notifications. Perhaps more surprisingly, FastLane actually *reduces* TCP's sensitivity to shallow buffers.

- While the actual numbers vary, FastLane consistently reduces high percentile completion times of short flows across all network configurations and application workloads used in the evaluation.

6.6.1 Methodology

In this section, we provide some details about the methodology, including the network and protocol configuration, and application workloads.

Network Configurations. Our simulation platform uses a 128-server FatTree topology, with an oversubscription factor of 4. The network has 10 Gig links with 128KB per port when running TCP and 64KB per port when running pFabric. These numbers are based on the amount of buffering typically available in TOR switches [3] and pFabric's buffer calculation [20], respectively. Based on [43], we model server processing delays as taking 5μs per packet, processing up to 16 packets in parallel.

For our implementation, we employ a 16-server, full bisection bandwidth, FatTree topology. All of the links in the topology are 1 Gbps. Given the reduced link speeds, we scale buffers to 64KB per port.

Timeouts. For our simulations, we set the timeout for TCP to be $1ms$ and for pFabric to be $250\mu s$. $1ms$ timeouts for TCP are considered aggressive based on prior work [18]; setting $250\mu s$ timeouts for pFabric balances pFabric's desire for small timeouts with the practical limitations of timeout generation and unpredictable server delays [43, 55]. However, for our implementation, we use the traditional datacenter timeout value of $10ms$ [18].

Notifications and Load balancing. We evaluate TCP both with Source Quench and with FastLane. When Source Quench assists TCP, quench message generation is triggered by CoDel's marking algorithm. When FastLane assists TCP (and pFabric), we institute bandwidth and buffer caps on notifications. Based on our analysis in §6.5, we cap the bandwidth to 1% and the buffers to 2.5% of 128KB = 3.2KB. For load balancing, we use flow hashing when in-order delivery is required (i.e., for TCP) and use packet scatter otherwise.

Application workflows, short flows and long flows. All experiments use request-response workflows. Requests are initiated by a 10 byte packet to the server. We classify requests into two categories: short and long. Short requests result in a response that can be a flow of size 2, 4, 8, 16, or 32KB, with equal probability. This spans the range of small, latency-sensitive flows typically observed in datacenters [18]. As these requests are usually encountered in partition-aggregate workflows, our sources initiate them in parallel, such that the total response size is 32 KB, 64KB, 96KB, 128KB, or 160KB with equal probability. Note that 160KB / 2KB = 80 senders, twice the number of workers typically sending to the same aggregator [18].

Long requests generate a response that is 1MB in length. Since most servers are typically engaged in just one or two long flows at a time [18], our long requests follow an all-to-all traffic pattern.

6.6.2 Simulation Results

We now present the simulation results for FastLane We first report our results across a range of utilizations for a workload where 10% of the load is caused by short request-response workflows and 90% of the load is caused by long workflows. This is the distribution typically seen in production datacenters [24]. Then we keep the utilization constant at 60% and vary the fraction of the load caused by the short request-response workflows. After establishing

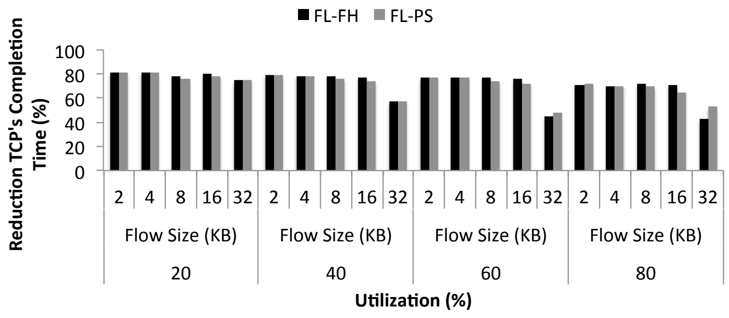


Figure 6.5: Reduction in 99.9th percentile flow completion time for varying network utilizations when TCP is assisted by FastLane with flow hashing (FL-FH) and with packet scatter (FL-PS)

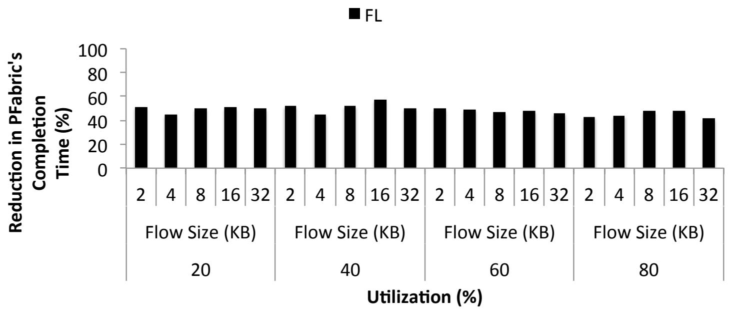


Figure 6.6: Reduction in 99.9th percentile flow completion time for varying network utilization when pFabric is assisted by FastLane (FL).

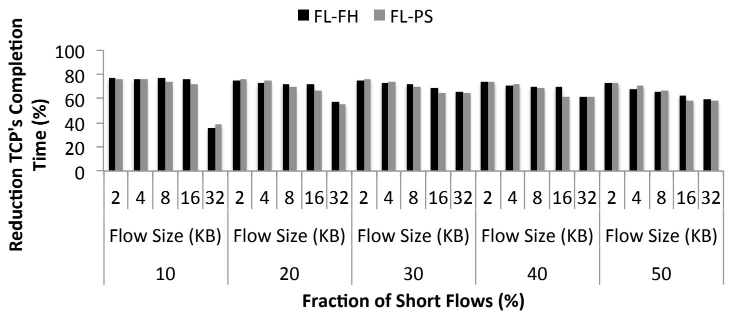


Figure 6.7: Reduction in 99.9th percentile flow completion times for varying fraction of short flows when TCP is assisted by FastLane with with flow hashing (FL-FH) and with packet scatter (FL-PS)

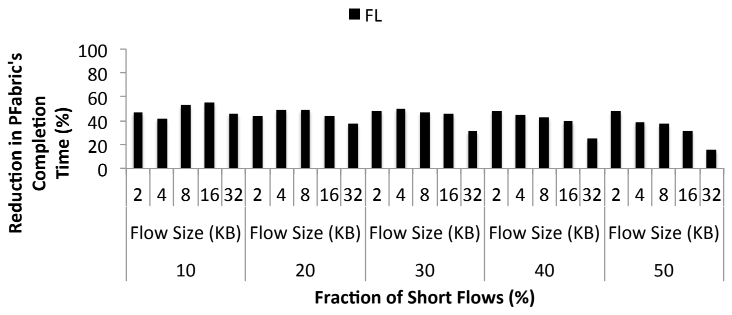


Figure 6.8: Reduction in 99.9th percentile flow completion time for varying fraction of short flows when pFabric is assisted by FastLane (FL).

the high-level benefits of FastLane, we evaluate its sensitivity to (i) bandwidth and buffer caps, (ii) smaller buffer sizes, and (iii) varying amounts of server latency. For all of these experiments, we define utilization as the average load on the core.

Varying Utilization

Figure 6.5 shows the performance of FastLane (with respect to TCP) for 99.9th percentile flow completion times as network utilization changes from 20% to 80%. In most cases, Source Quench does not benefit TCP, so we do not report its results. FastLane, on the other hand, dramatically improves performance irrespective of whether flow hashing is used. At 20% utilization, 2KB flow completion times reduce from 1.12ms to 0.21ms, an 81% reduction, when using FastLane with packet scatter (FL-PS). A priori, one would expect that as network utilization increases, the benefits of FastLane would be reduced since higher loads decrease the amount of time that can be saved by avoiding a timeout. However, even at 80% utilization, FL-PS helps 2KB flows reduce their completion times by over 70%.

We make two remarks regarding results in Figure 6.5. First, it may seem that FastLane is less beneficial for 32KB flows. However, a deeper look into our results suggests that 32KB flows suffer less timeouts as our workload has fewer of them simultaneously transmit to the same destination. As a result, FastLane has fewer opportunities to help. Second, packet scatter does not seem to provide significant benefits. This is primarily due to the short flow sizes — if we turn our attention to the long 1 MB flows, we notice that FL-PS reduces their average completion times by 62%.

As shown in Figure 6.6, FastLane also reduces the 99.9th percentile flow completion times for pFabric. At 40% network utilization, FastLane reduces 2KB completion times from 0.41ms to 0.20ms, a 52% reduction. FastLane does not affect pFabric's long flow performance, other than at 80% utilization where it reduces average completion times by up

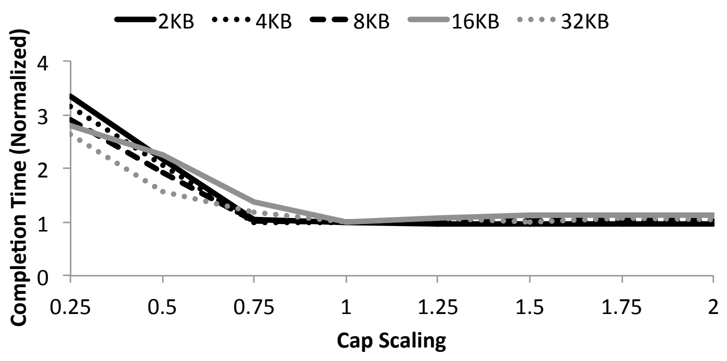


Figure 6.9: FastLane's sensitivity to the bandwidth and buffer caps when aiding TCP).

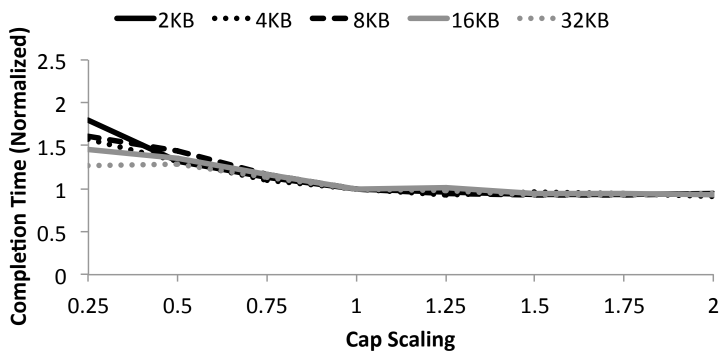


Figure 6.10: FastLane's sensitivity to the bandwidth and buffer caps when aiding pFabric.

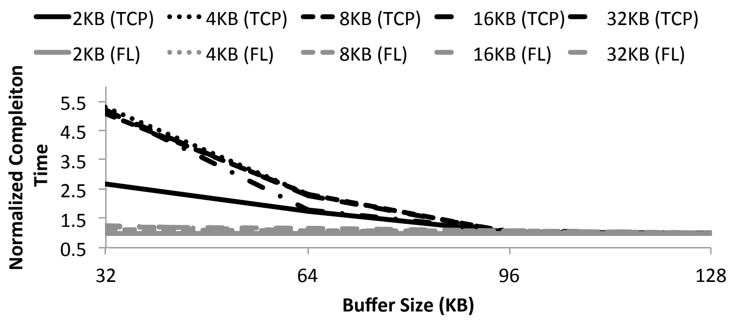


Figure 6.11: Reduction in 99.9th percentile completion time for varying buffer sizes for TCP with and without FastLane.

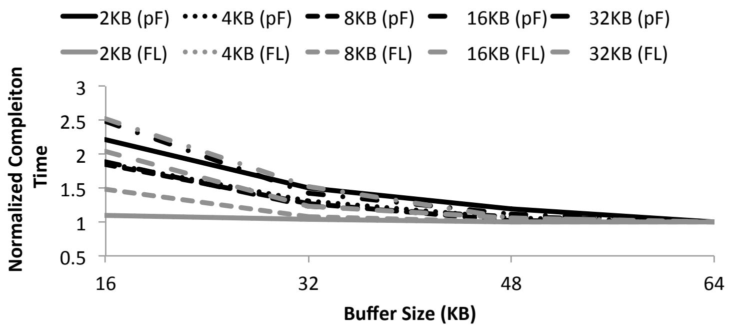


Figure 6.12: Reduction in 99.9th percentile completion time for varying buffer sizes for pFabric with and without FastLane.

to 38%.

Varying Fraction

We now evaluate the performance of FastLane with total load fixed to 60% and the short flows contributing to a larger fraction of the network load (see Figure 6.7 and Figure 6.8). Even when 50% of the load is due to short flows, FastLane provides significant benefit to TCP (e.g., FL-PS reduces the 99.9th percentile completion times of both 2 and 4KB flows by over 70%). And FastLane's benefits for 32KB flows actually increase under this traffic mix because the more bursty workload leads more flows to experience timeouts, providing FastLane more opportunities to help. As shown in Figure 6.8, FastLane continues to provide significant benefits to pFabric as well.

With respect to long flows, the results for TCP are very similar to the case of short flows contributing to 10% of the network load. For pFabric, in the extreme case of short flows contributing to 50% of the load, average long flow completion times do inflate by 23%. We argue that this is a worthwhile tradeoff to make as FastLane decreases latency-sensitive, short flow completion times by up to 47% in this scenario.

Sensitivity Analysis

We now evaluate the performance of FastLane with varying bandwidth and buffer caps for the notifications, varying buffer sizes, and varying server latency. For these experiments, we set the total network load to be 60% and consider the scenario where short flows contribute to 50% of the network load. This workload has the greatest number of bursts and should hence stress FastLane the most.

Sensitivity to Bandwidth and Buffer Caps:
Here we explore how sensitive FastLane is to the 1% bandwidth and 2.5% buffer caps that

we use throughout the evaluation. We simultaneously scale the bandwidth and buffer caps by the same factor (e.g., a scaling of 0.5 reduces the bandwidth and buffers available to notifications by half). Normally, FastLane's notifications may use extra bandwidth beyond that specified by the cap when the link is idle (i.e., they are work conserving). To more accurately understand the effect of the cap, we prohibit notifications from using extra resources in this experiment.

In Figures 6.9 and 6.10, we depict FastLane's sensitivity to the cap when it is assisting TCP and pFabric, respectively. These figures show 99.9th percentile completion time for varying flow sizes, normalized by the completion times when no scaling is used (i.e., cap scaling = 1). The characteristics of FastLane with TCP and FastLane with pFabric are quite different. Both do not see a significant performance hit until we scale the bandwidth and buffers to below 0.75. However, FastLane's performance degrades more gradually when assisting pFabric because pFabric's fine-grained timeouts reduce the performance impact of packet drops. Based on these results, we see that our current bandwidth and buffer caps balance the need to be robust to extreme congestion environments with the desire to consume fewer resources.

Small Buffer Performance:
Here we evaluate how FastLane performs with smaller buffer sizes. We start with the default TCP and pFabric buffers of 128KB and 64KB, respectively, and reduce them to see the performance impact. We keep the buffer cap constant at 3.2KB throughout this experiment.

In Figure 6.11, we report the results for FastLane when assisting TCP. The numbers for each flow are normalized by the 99.9th percentile completion time that would occur at 128KB (each protocol and flow is normalized separately). We see that with FastLane, TCP's 99.9th percentile flow completion times do not degrade as we reduce buffer sizes. Without FastLane, TCP's performance degrades rapidly and severely. However, we note that FastLane is not immune to the impact of buffer reduction. Its average flow completion times do increase as buffer sizes decrease. In particular, average long flow completion times increase by 98% from 1.89 ms to 3.76 ms as we go from 128KB to 32KB.

Figure 6.12 shows the results for the same experiment performed with pFabric. FastLane is not able to prevent the 99.9th percentile completion times of 8, 16 and 32KB flows from increasing. Average long flow completion times suffer as well, increasing by approximately 5$\times$ for both FastLane and unaided pFabric as we reduce buffers from 64KB to 16KB.

We highlight a few important points. First, pFabric already tries to use the minimum buffering possible. Second as these numbers are normalized to what each flow would achieve in Figure 6.8, FastLane outperforms pFabric even in situations where they have same normalized value. Thus, FastLane improves pFabric's short flow performance at all of these points.

These results show us that FastLane improves TCP's ability to use small buffers and does not harm pFabric's ability to do the same. The ability to degrade gracefully in the presence of small buffers is important. Buffering typically consumes 30% of the space and power of a switch ASIC, limiting the number of ports a single switch can support [19].

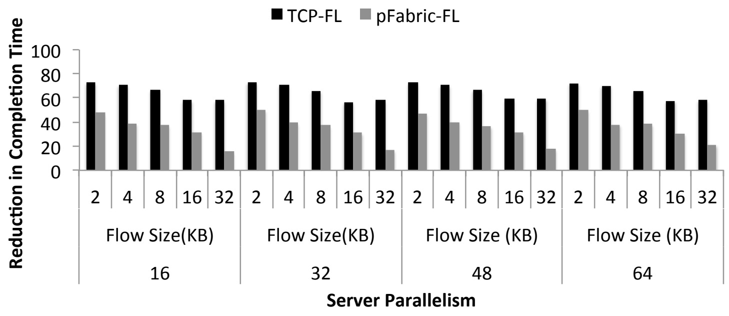


Figure 6.13: 99.9th percentile reduction in flow completion time with varying server parallelism.

Server Parallelism:
Our simulations have a server model that processes 16 packets in parallel. As server hardware varies greatly, we explore how different amounts of parallelism affect flow completion times. Figure 6.13 reports the reduction in 99.9th percentile flow completion times for TCP and pFabric as a function of server parallelism. FastLane's performance improvement does not diminish as the amount of parallelism increases.

6.6.3 Implementation Results

We now discuss the implementation results for FastLane. For ease of implementation, when developing FastLane, we disabled the more advanced features of Linux TCP (i.e., SACK, DSACK, Timestamps, FRTO, Cubic). To provide a fair comparison, we show results for FastLane versus TCP with these features disabled. But, we also report how FastLane compares to TCP with all of these features enabled. We show that FastLane still outperforms TCP, demonstrating its utility.

We begin by running the same base workload as the simulation, varying the utilization while keeping the fraction of load contributed by short flows constant at 10% (see Section 6.6.2). Then we evaluate how FastLane performs under a workload consisting of longer flow sizes. To avoid the hardware limits of our virtualized topology (Emulab), we partition the nodes into frontend and backend servers, with frontend servers requesting data from backend servers.

Varying Utilization

Figure 6.14 reports the reduction in 99.9th percentile flow completion times when FastLane assists TCP under various utilizations. We see that FastLane reduces the flow completion times of short flows by up to 68% (e.g., at 20% utilization, 8KB flows complete in 4.6 ms

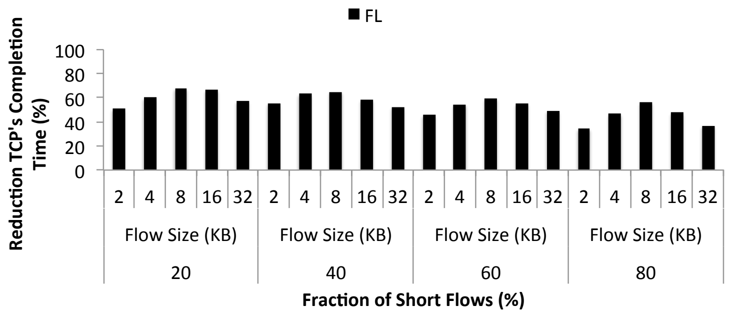


Figure 6.14: (Implementation result) Reduction in TCP's 99.9th percentile flow completion time when assisted by FastLane.

Util	$2KB$	$4KB$	$8KB$	$16KB$	$32KB$
20%	51%	61%	68%	63%	−4%
40%	55%	63%	64%	55%	46%
60%	44%	53%	58%	51%	40%
80%	32%	42%	48%	40%	22%

Table 6.2: (Implementation result) Reduction in 99.9th percentile flow completion vs TCP with advanced features.

with FastLane as compared to 14.4 ms with unaided TCP). Average long flow completion times reduce at high utilizations as well - we report a 23% reduction at 80% load. But at low utilizations, FastLane's long flow performance slightly underperforms unaided TCP's.

Table 6.2 compares FastLane's completion times to TCP with SACK, DSACK, Timestamps, FRTO, and Cubic enabled. In general, FastLane achieves a comparable reduction as that reported in Figure 6.14, demonstrating its utility. The one point where FastLane slightly underperforms TCP is for 32KB flows at 20% utilization. This occurs because the inflation in flow completion times occurs after the 99.9th percentile for this flow size, utilization, and workload.

Long Flows

Our implementation setup allows us to evaluate the flow completion times of longer flows, while maintaining manageable runtimes. Table 6.3 reports the reduction in average flow completion times when FastLane is used versus unaided TCP and TCP with the advanced features enabled (TCP-A). Flow sizes are 1, 16, or 64 MB with equal probability.

We see that FastLane reduces average completion times by as much as 31% at high uti-

	FL (TCP)			FL (TCP-A)		
Util	1MB	16MB	64MB	1MB	16MB	64MB
20%	-4%	-4%	-4%	6%	3%	2%
40%	10%	7%	8%	14%	12%	11%
60%	28%	26%	26%	21%	23%	23%
80%	29%	30%	28%	25%	29%	31%

Table 6.3: (Implementation result) Reduction in average completion time of long flows

lizations. However, when the network is under-utilized, FastLane may slightly underperform TCP for long flows. We believe that this performance impact is small and that the benefits of FastLane far outweigh its modest cost.

6.6.4 Takeaways

Our results show that FastLane achieves its goal, leveraging tight layer integration to reduce the effect of packet drops on flow completion times, without incurring the limitations of lossless interconnects. We see that FastLane not only requires very few buffer resources, just 3.2KB per port for 10Gbps links, but that its performance degrades gracefully with decreasing buffer sizes.

The one disadvantage of using FastLane as compared to a lossless interconnect is that while it enables per-packet load balancing, it does not provide the adaptive load balancer with any information about the path to select. This is because we no longer employ flow control and hence do not propagate information about downstream congestion. To address this issue, we could employ a mechanisms such as those advocated by F10 [46]. We leave determining the best way to incorporate their approach to future work.

Chapter 7

Conclusion

Through the process of designing and implementing DeTail and FastLane, we learned many lessons, which we document here. Perhaps the most important one is the realization that we have only begun the enormous effort necessary to drive down tail latencies. We touch upon some of the many avenues for future exploration. We conclude by revisiting our thesis and summarizing how DeTail and FastLane provide two compelling examples of the benefits of tightly integrating the layers of the network stack.

7.1 Lessons Learned

Our experiences with DeTail and FastLane represent many important lessons, some of which we discuss here. First, we learned the need to design with failure in mind. While failures are typically rare, our focus on high-percentile latencies required that we consider them in our designs. Second we learned the value of having a carefully designed, controlled environment when evaluating high-percentile latencies. Only in such an environment can we separate out the impact of various proposals from the noise of the system. Finally, we saw how the combination of multiple evaluation platforms allows us to have greater confidence in our conclusions. This is especially true in environments where an at-scale testbed is unavailable. Here we delve into each of these lessons, describing their significance in turn.

7.1.1 Handling Failures

The focus on high-percentile latencies coupled with the need to engage many servers to answer every query drove us to consider and design for hardware failure. We learned that failure can manifest itself in different ways. Devices could stop performing a task (e.g. a link failing) or they could start to behave erratically, and unpredictably (e.g. a server or switch constantly transmitting pauses). We saw that proposals that degrade gracefully in the presence of such failures have a much higher likelihood of adoption.

Handling network failures where certain devices stop performing a task is straightforward. By simply routing traffic around the damaged link / switch we can avoid using it. Doing so allows the network to degrade gracefully and is an effective temporary solution. Handling cases where devices start to behave erratically is much more challenging. Since we cannot predict the erratic behavior, we need to focus on reducing the harm that these devices can cause to the network as a whole. We can achieve our goal by both designing functionality that a device is unable to use to harm the whole network as well as by leveraging nearby devices to ensure that the erratic behavior of the failed one is contained. FastLane is an example where we have removed the dependence on pause to reduce the harm a single device can have on the network as a whole and have used rate limiters so that nearby nodes can contain erratic message transmission.

Our experience taught us that given the requirement to handle failures, datacenter networking proposals should be designed at the outset to degrade gracefully in their presence.

7.1.2 Carefully Controlled Evaluation Environment

Reliably measuring and comparing the high-percentile latencies of our implementations is challenging. Our implementations depend on systems consisting of many complex components that interact in unpredictable ways. This is especially true at the timescales we focus on in datacenter networks. We must understand and control these interactions to obtain meaningful results from our testbed.

Many examples of these interactions and the approaches we used to address them are described in Chapter 3. We were unable to make progress in evaluating different design alternatives until we had taken all of these steps to create a controlled environment. Furthermore, many of these issues were not uncovered until we had implemented a datacenter networking proposal. For example, we only became aware of the driver's large ring buffer when implementing priority flow control. In hindsight, we would have likely been more productive had we invested the time necessary to carefully analyze the systems operating in our testbed upfront. By first understanding and learning how to predict the interactions of these systems, we would have had a saved time later in the process when we were attempting to distill the benefits of various networking proposals.

7.1.3 Combining Multiple Evaluation Platforms

One decision which was very beneficial was to create both simulation and implementation platforms at the outset. This decision had two key advantages. First, it enabled us to understand when the simulation or implementation was not reporting expected results. Given the complexities of the environment, with many transfers constantly starting and stopping, having these two platforms helped us to better understand the high-percentile latencies we should be achieving and which platform contained the bug or limitation. Second, by using the implementation platform, we could validate the assumptions made by our simulator, and have much greater confidence in our scaled up simulation results. Given the lack of a large-scale tested with fine-grained network control, we believe the combination of these two platforms represents the only viable alternative.

7.2 Future Work

Understanding and reducing tail latencies is challenging. As discussed throughout this thesis, the TCP/IP stack was not designed to achieve predictable high-percentile performance. Advances in (ii) admission control, and (iii) cross-service priority inheritance, and (iii) datacenter-wide performance debugging, would enable us to make further strides towards our goal. We discuss each in turn, describing how they would help us to better understand and reduce the tail.

7.2.1 Admission Control

Ultimately, latency increases when a network is unable to support the load that is offered to it. Instead of being transmitted through the network, packets begin to queue, causing latencies to spike. Most of our efforts have been focused on more efficiently using network resources, such as alternate paths. It is likely that opportunities still exist to further improve network efficiency. However, regardless of how efficient our network is, services will eventually overload it. To address this case, we need to develop mechanisms that inform services when the network is unable to support their load and must deny their transmissions.

By informing services early and upfront, we could allow them to make more informed decisions about how much load to shed. Unlike the network, which only has the information contained in the packet header, services have much more knowledge of which transfers can be abandoned to minimally impact the user. Knowing that they have exceeded network capacity early (as opposed to waiting for a timeout) also provides them more time to mask this problem for the end-user. These potential benefits highlight the need to develop mechanisms that detect network overload (as opposed to a transient burst) and that interface with services, informing them when traffic cannot be admitted.

7.2.2 Priority Inheritance

Determining the priority of a transfer between two servers is nontrivial. Services are often composed of, or interact with others to answer a request. Certain services or requests may be deemed more important than others, requiring tighter latency guarantees. Or they may have different deadline requirements altogether. To address this problem, we cannot simply look at the type of transfer being performed. Instead, all of the transfers performed in satisfying a request must consider its priority.

We can propagate request priority by setting up special rules such as having each transfer inherit the priority of the arriving one which triggered it. By employing these simple mechanisms, we can ensure that the services in the datacenter work in concert to meet user latency requirements.

7.2.3 Datacenter-wide Performance Debugging

Modern datacenters consist of many components which must work together to deliver a result. When it takes too long to answer a request, it is challenging to determine the cause of the delay. Network packets may have been dropped causing a timeout. Or a server may be running too many resource-intensive tasks that are competing for resources. We need a mechanism to determine how much time a request spends at each stage of processing to allow us to debug performance problems.

With such stringent expectations that requests be answered within tight latency bounds, performing debugging only represents the first step. We must determine how to detect performance issues when they arise, as quickly as possible. Additionally, we must develop systems that automatically mitigate these issues, instead of waiting for an operator to parse through the logs to determine the problem and execute a solution. Online, datacenter-wide performance debugging represents a significant opportunity to address these problems.

7.3 Thesis Summary

In this thesis we begin by identifying a class of workloads, that while increasingly common, are poorly supported by existing datacenter networks. These workloads consist of many interactions, all of which must complete within tight latency bounds to satisfy user demands. As datacenter networks are based on the TCP/IP stack, they inherit its design decisions favoring generality and interoperability over high, predictable performance.

Our hypothesis is that by foregoing this generality and by tightly integrating the layers of the network stack, we can improve performance, dramatically reducing tail latency. Our two proposals, DeTail and FastLane, support this hypothesis. DeTail presents a network stack whose layers depend on each other and work together to overcome their respective limitations. FastLane takes a different approach, having switches provide transport explicit information so it can make better decisions. Both of these examples highlight the many opportunities for improving network efficiency that can be attained by removing the unnecessary burdens of generality and interoperability. We believe that this approach is well-suited for datacenters and that continuing on the path to tighter integration will enable them to deliver richer services, further enhancing the user experience.

Bibliography

[1] 802.1qau - congestion notification. http://www.ieee802.org/1/pages/802.1au.html.

[2] Amazon ec2. http://aws.amazon.com/ec2/.

[3] Arista 7050 switches. http://www.aristanetworks.com.

[4] Cisco nexus 5000 series architecture. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-462176.html.

[5] Data center bridging. http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns783/at_a_glance_c45-460907.pdf.

[6] Datacenter networks are in my way. http://mvdirona.com/jrh/TalksAndPapers/ JamesHamilton_CleanSlateCTO2009.pdf.

[7] Emulab. http://www.emulab.net.

[8] Fulcrum focalpoint 6000 series. http://www.fulcrummicro.com/product_library/FM6000_Product_Brief.pdf.

[9] Network simulation cradle. http://research.wand.net.nz/software/nsc.php.

[10] Ns2. http://www.isi.edu/nsnam/ns/.

[11] Ns3. http://www.nsnam.org/.

[12] Priority flow control: Build reliable layer 2 infrastructure. http://www.cisco.com /en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809.pdf.

[13] Ieee standard part 3: (csma/cd) access method and physical layer specifications - section two. *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, 26 2008.

[14] D. Abts and J. Kim. High performance datacenter networks: Architectures, algorithms, and opportunities. *Synthesis Lectures on Computer Architecture*, 6(1), 2011.

[15] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[16] M. Al-fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[17] M. Alizadeh. Personal communication, 2012.

[18] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *SIGCOMM*, 2010.

[19] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, 2012.

[20] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Decon-

structing datacenter packet transport. In *ACM HotNets*, 2012.

[21] 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*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[22] F. Baker. Requirements for ip version 4 routers, 1995.

[23] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. *SIGCOMM Comput. Commun. Rev.*, 26, August 1996.

[24] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.

[25] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Experience with deter: a testbed for security research. In *TRIDENTCOM*, 2006.

[26] D. Borthakur. Personal communication, 2012.

[27] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. A. Mujica, and M. Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn. In *SIGCOMM*, pages 99–110, 2013.

[28] D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM*, 1988.

[29] D. Crisan, A. S. Anghel, R. Birke, C. Minkenberg, and M. Gusat. Short and fat: Tcp performance in cee datacenter networks. In *Hot Interconnects*, pages 43–50. IEEE, 2011.

[30] J. Dean. Software engineering advice from building large-scale distributed systems. http://research.google.com/people/jeff/stanford-295-talk.pdf.

[31] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, 2007.

[33] S. Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[34] S. Floyd and T. Henderson. The newreno modification to tcp's fast recovery algorithm, 1999.

[35] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1, August 1993.

[36] F. Gont. Deprecation of icmp source quench messages, 2012.

[37] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[38] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[39] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant

network structure for data centers. In *SIGCOMM*, 2008.

[40] U. Hoelzle and L. A. Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 1st edition, 2009.

[41] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*, August 2012.

[42] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 25(1):157–187, Jan. 1995.

[43] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *ACM SoCC*, 2012.

[44] R. Kohavi and R. Longbotham. Online experiments: Lessons learned, September 2007. http://exp-platform.com/Documents/IEEEComputer2007 OnlineExperiments.pdf.

[45] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18, August 2000.

[46] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 399–412, Berkeley, CA, USA, 2013. USENIX Association.

[47] N. McKeown. White paper: A fast switched backplane for a gigabit switched router. http://www-2.cs.cmu.edu/ srini/15-744/readings/McK97.pdf.

[48] K. Nichols and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20:20–20:34, May 2012.

[49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[50] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. In *SIGOPS OSR*, 2009.

[51] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. Tcp fast open. In *ACM CoNext*, 2011.

[52] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM*, 2011.

[53] P. Sarolahti. Linux tcp. http://0gram.me/misc/network/linuxtcp.pdf.

[54] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *USENIX NSDI*, 2012.

[55] 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. In *SIGCOMM*, 2009.

[56] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting

deadlines in datacenter networks. In *SIGCOMM*, 2011.

[57] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.