FastLane: Agile Drop Notification for Datacenter Networks

David Zats
Anand Padmanabha Iyer
Ganesh Ananthanarayanan
Randy H. Katz
Ion Stoica
Amin Vahdat

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-173
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-173.html

October 23, 2013

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **23 OCT 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE
FastLane: Agile Drop Notification for Datacenter Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The drive towards richer and more interactive web content places increasingly stringent requirements on datacenter networks. The speed with which such networks respond to packet drops limits their ability to meet highpercentile flow completion time SLOs. Indirect notifications to packet drops (e.g., duplicates in an end-toend acknowledgment sequence) are an important limitation to the agility of response to packet drops. We propose FastLane, a new in-network drop notification mechanism. FastLane enhances switches to send highpriority per-flow drop notifications to sources, thus informing sources as quickly as possible. Consequently sources can retransmit packets sooner and throttle transmission rates earlier. Sources can also make better decisions given more precise information and the ability to differentiate between out-of-order delivery and packet loss.We demonstrate, through simulation and implementation that FastLane reduces 99.9th percentile completion times of short flows by up to 75%. These benefits come at minimal cost?safeguards ensure that FastLane consume no more that 1% of bandwidth and 2.5% of buffers.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**15**	

Acknowledgement

This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

FastLane: Agile Drop Notification for Datacenter Networks

David Zats[1], Anand Iyer[1], Ganesh Ananthanarayanan[1], Randy Katz[1], Ion Stoica[1], Amin Vahdat[2]

[1]*University of California, Berkeley*, [2] *Google / University of California, San Diego*

Abstract

The drive towards richer and more interactive web content places increasingly stringent requirements on datacenter networks. The speed with which such networks respond to packet drops limits their ability to meet high-percentile flow completion time SLOs. Indirect notifications to packet drops (e.g., duplicates in an end-to-end acknowledgment sequence) are an important limitation to the agility of response to packet drops. We propose FastLane, a new *in-network drop notification* mechanism. FastLane enhances switches to send high-priority, per-flow drop notifications to sources, thus informing sources as quickly as possible. Consequently, sources can retransmit packets sooner and throttle transmission rates earlier. Sources can also make better decisions, given more precise information and the ability to differentiate between out-of-order delivery and packet loss. We demonstrate, through simulation and implementation, that FastLane reduces 99.9th percentile completion times of short flows by up to 75%. These benefits come at minimal cost—safeguards ensure that FastLane consume no more that 1% of bandwidth and 2.5% of buffers.

1 Introduction

Creating rich web content involves aggregating outputs from many *short* network flows, each of which contains only a few KBs of data. Strict user-interactivity deadlines mean that networks are increasingly evaluated on high percentile completion times of these short flows. Achieving consistent flow completion times is increasingly challenging given the trends in modern datacenters. The burstiness of workloads continues to increase, while deep buffered switches are becoming prohibitively expensive [8, 9]. Consequently, switch buffers can quickly be overwhelmed leading to packet drops [8].

Existing approaches for reacting to packet drops are, essentially, *indirect*. They rely on on duplicate acknowledgements and timeouts which take a long, highly variable time to arrive because of unpredictable queueing and server delays. These delays can be far larger than the transmission time of short flows, greatly inflating high-percentile completion times [20]. Setting smaller timeouts only reduces completion times to a point, at which spurious retransmissions and server overheads become prohibitive. Even with tight timeouts [8, 28], our measurements show that relying on indirect indicators can increase short flow completion times four-fold.

Recent solutions to deal with packet drops typically leverage either end-to-end mechanisms with improved window management or explicit rate control [8, 9, 19, 27, 29]. Often the former techniques are unable to respond to congestion quickly enough, leading to further packet drops. The latter techniques inflate the completion times of most datacenter flows, requiring multiple RTTs for transmissions that could have completed in just one [10, 12, 17].

In this paper, we argue for enlisting switches to *directly* transmit *high-priority* notifications to sources as soon as drops occur. Such direct notifications are the *fastest* way to inform sources of drops because switches are first to know with certainty that the drop has occurred. As a result, sources no longer wait for any of the indirect mechanisms. Instead, they can respond quickly and effectively by retransmitting the packet, thus improving high-percentile completion times.

An earlier proposal to directly notify sources, (i.e., ICMP Source Quench [16]), failed to gain widespread adoption in the wide area. We argue that this was due to three fundamental limitations. First, switches would send notifications in response to both drops and building congestion, without telling the source which event occurred and which packet had triggered it [24]. Consequently, transport protocols could not retransmit quickly. Second, as notifications were not prioritized, they would suffer delays, limiting the sources ability to respond agilely. Finally, there were no safeguards in place to ensure that notifications did not contribute to congestion collapse [11].

Addressing the above limitations presents conflicting challenges. Limiting resource consumption of notifications suggests putting minimal information in the notifications and making them simple to generate at switches. On the other hand, low semantic content in the notifications limits the capabilities of sources to respond sufficiently. In the absence of detailed information about the event that occurred, sources react conservatively, waiting until the event has been verified, else they risk exacerbating congestion.

We solve these challenges by presenting *FastLane*, a lightweight drop notification mechanism. FastLane includes the transport header of dropped packets in notifications, providing sources sufficient information to respond quickly. It prioritizes the notifications, ensuring timely arrival and installs analytically-determined buffer and bandwidth caps to avoid overwhelming the network. We show that FastLane is both practical and transport-agnostic by describing how to efficiently generate notifications in the data plane and extending both TCP and the recently proposed pFabric to leverage them [10].

In addition to making sources agile to packet drops, notifications have another useful property. They enable sources to distinguish between out-of-order delivery and packet loss. This allows networks to perform per-packet load balancing, effectively avoiding hotspots and their inherent delays, without triggering retransmissions based on out-of-order delivery.

We evaluate FastLane in a number of scenarios, using both testbed experiments and simulations. Results from our evaluation demonstrate that FastLane improves the 99.9th percentile completion time of short flows by up to 75% compared to TCP. FastLane also improves short flow completion times by 50% compared to pFabric. It achieves these improvements even when we cap the bandwidth and buffers used by drop notifications to as little as 1% and 2.5%, respectively.

The rest of this paper is organized as follows. In the following section, we discuss the need for drop notifications. In Section 3, we describe the mechanisms employed by FastLane. The details of our simulation and in-kernel implementation are described in Section 4. We evaluate FastLane and report both implementation and simulation results in Section 5. We contrast our approach with prior work in Section 6 and conclude in Section 7.

2 The Case for Drop Notification

Datacenter networks are expected to meet strict performance requirements. Hundreds of intra-datacenter flows may be required to construct a single web-page [23]. The worst-case performance is critically important as workflows (e.g. partition-aggregate) must either wait for the last flow to arrive or degrade page quality [8].

Measurement studies from a variety of datacenters have shown that most flows are short [12, 17]. As seen from Microsoft's production datacenters, latency-sensitive flows are no exception, with most ranging from 2 - 20KB in length [8].

In this section, we begin by describing how directly notifying sources of packet drops helps them improve short flow completion times. Next, we investigate existing direct notification schemes, discussing why their design decisions dramatically limit their effectiveness. After this investigation, we propose a series of principles for direct notification, which we use in the next section to design FastLane.

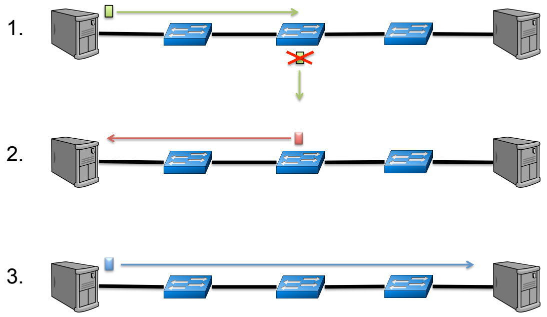

Figure 1: In response to a packet drop (1), the switch sends a notification directly to the source (2). Upon receiving the notification, the source resends the packet (3).

2.1 Notifying Drops

Datacenter networks are susceptible to packet drops. Workflows, such as partition-aggregate, routinely generate traffic bursts by requiring many workers to simultaneously respond to the same destination. As datacenter switches are shallow-buffered, these workflows can easily overwhelm them, resulting in packet drops [8]. Furthermore, this problem is likely to become worse as websites employ more workers to create richer web pages, while meeting the same strict deadlines.

Unfortunately, short flows are particularly sensitive to drops. If a flow consists of just a few packets, there is a significant chance that all of the packets in the window will be dropped, resulting in a timeout. Even if just one of the flow's packets is dropped, the flow may not be sending sufficient data to generate three duplicate acknowledgements and will have no option but to timeout.

Timeouts are purposely set to large values to increase the likelihood that the missing packet has actually been dropped. Sources must set timeouts sufficiently high such that queueing and unpredictable server delays do not trigger spurious retransmissions. While reducing buffer sizes can mitigate the former, this is often accompanied by a corresponding reduction in throughput. The latter is far more challenging to control. Recent work has shown that even highly-optimized servers can take hundreds of microseconds to respond to requests in the presence of bursts [20]. Given the low per-request processing time in this study, TCP acknowledgment generation is likely to behave similarly. These latencies are far greater than the unloaded RTTs of modern datacenters, which are typically in the 10's of μs [10].

In Figure 1, we depict the benefits of directly notifying sources of drops. When the congested switch drops a packet, it sends a *high-priority* notification back to the source, informing it precisely which packet was dropped, *as quickly as possible*. In response to this notification, the source resends the dropped packet, minimizing recovery time.

It may seem as though most of the benefits of di-

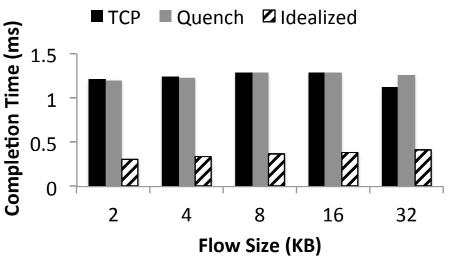

Figure 2: 99.9th percentile flow completion times.

rect notification are obtained when the packet is dropped early in the network. However, as discussed earlier, timeouts are typically set conservatively and server delays are highly unpredictable. TCP Offload Engines (TOE), which offload acknowledgement generation to the NIC do exist. However, they currently support just a handful of concurrent, long-lived connections and are hence poorly suited for the many short flows in datacenter environments [6].

2.2 Existing Alternatives

ICMP Source Quench and Quantized Congestion Notification (802.1Qau) are two proposals that rely on direct notification [1, 16]. 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 datacenter environments. We use the insights gained from our investigation to propose a series of design principles for direct notification.

2.2.1 ICMP Source Quench

ICMP source quench was a protocol switches used to signal congestion to the source. A switch experiencing congestion would generate and send ICMP messages to sources requesting that they 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 conditions under which source quench messages were sent were poorly defined and the message itself did not contain any information as to what triggered it. The advantage of this approach is that it enabled switches to generate source quench messages as frequently as their control plane could support. The specification did not have to concern itself with the generation rates of different switch hardware. The disadvantage of this approach was that sources did not know whether the notification was sent in response to a packet drop or building congestion. Nor did they know whether notifications would be sent consistently in response to each. As a result, when Linux supported Source Quench (15 years ago), it responded to those messages in the same way as it does to ECN [26]. It reduces the congestion window but it waited for 3 duplicate acknowledgements or a timeout to retransmit the packet.

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 diminishing potential gains [11]. At the same time, there were no safeguards in place to ensure that source quench messages did not consume too many resources in the presence of extreme congestion.

To quantify the impact of these design decisions, we evaluated Source Quench using the workload in Section 5. In this workload, we have bursts of short flows (up to 32KB in length) and long flows (1 MB in length). Figure 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 that does not have limitations of Source Quench could reduce high-percentile completion times by 75%.

2.2.2 Quantized Congestion Notification

Quantized Congestion Notification (QCN) is a direct notification scheme proposed as part of the datacenter 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 stems from the fact that rate-limiting is being 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 [25]), (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 [8]. The lack of coordination between the rate limiter and transport has led to significant drops and TCP timeouts. QCN can degrade TCP performance by so much that prior work recommends enabling QCN *only* in heterogeneous environments where it is beneficial to control unresponsive flows (e.g. UDP) [14].

2.3 Direct Notification Design Principles

Based on the lessons learned from evaluating Source Quench and QCN, we have distilled a set of design principles for direct notifications:

- Notifications (and the triggers that generate them) must be sufficiently specified so *transport* knows whether a packet was dropped, and if so, which one it was. This enables transports to respond effectively.

- Notifications must be created in the data plane so that many of them can be generated within a short duration without overwhelming the control plane.

- Notifications must be transmitted with high priority to ensure timely arrival, but safeguards must ensure they do not aggravate congestion events.

In the next section, we present the design of our solution, FastLane, and show how it achieves all of these goals.

3 Design of FastLane

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.

3.1 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 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 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 3.4 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 3.3).

3.2 Generating Notifications

Notifications must provide sources sufficient information to retransmit the dropped packet. 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).

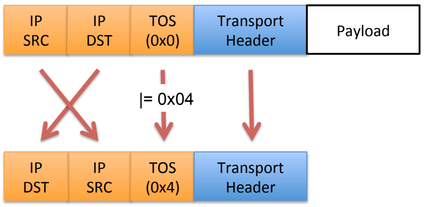

Figure 3: Transforming packets into notifications.

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. This is not an effective approach.

Instead, we developed a series of simple packet transformations that can quickly be performed in the data plane. The transformations to create a FastLane notification are depicted in Figure 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. While we expect that this approach would be performed in hardware, we note that transforming a packet only takes 12 lines of Click code [21].

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. FastLane implements the former approach in this paper.

This approach relies solely on simple packet manipulation. Prior work has demonstrated that such operations can be performed very quickly in the data plane [13]. 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.

3.3 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. 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*.

3.3.1 Controlling Bandwidth

To understand how much bandwidth should be provided to 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:

$$l_n = \frac{ps_n}{s_r + ps_n}, \tag{1}$$

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 [12]. 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 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.

3.3.2 Controlling Buffers

Traffic bursts may result in 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, which means that each packet will generate a notification. Then the number of bytes necessary to store the resulting notifications is approximated by the following equation:

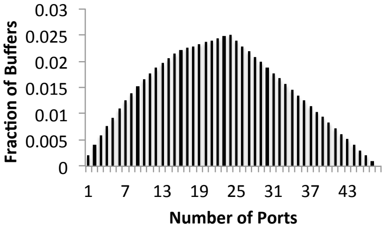

Figure 4: The fraction of a switch's buffers used by notifications when ports receive bursts simultaneously.

$$b_{size} \times \frac{n_{size}}{d_{size}} \times (1 - \frac{1}{p_{in}}), \tag{2}$$

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 2. However, the same input port cannot simultaneously contribute to multiple bursts. When combined with Equation 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 port simultaneously receive bursts. For this calculation we assume (i) burst sizes of 160KB, doubling the typical burst size reported by prior work [8] and (ii) a 48-port switch with 128KB per port as seen in production TOR switches [2].

In Figure 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 2. Note that as the number of simultaneous burst increases, the num-

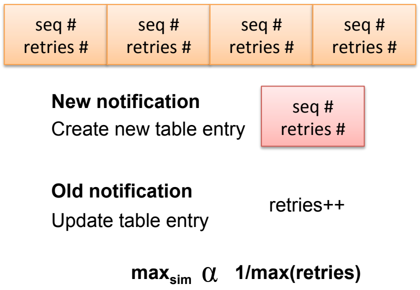

Figure 5: Modifications to TCP for handling notifications of dropped data packets.

ber 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 will complete on time [19,29]. Scenarios with hundreds of sources participating in Incast will complete faster because there will be fewer rounds of timeouts and synchronized pull-backs.

3.4 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. Here, we discuss how TCP uses notifications to improve high-percentile flow completion times. Later we will present our proposed modifications to pFabric.

3.4.1 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: Our modifications to TCP perform retransmissions when receiving notifications for both data and control (i.e., SYN, FIN, ACK) packets. Control packets that need to be resent are transmitted immediately as they are small and hence should not significantly contribute to congestion[1]. Retransmitting data packets is more challenging as we must strike a balance between the desire to retransmit as quickly as possible and the need to avoid ping-pong packet retransmissions. Ping-pong packet retransmissions occur when a notification is generated and a packet is retransmitted, only to be dropped again because of persistent congestion. This process can repeat over and over, wasting precious network resources. We must walk the fine line between retransmitting as soon as possible and avoiding these unnecessary drops.

Ideally, when the notification is transmitted to the source, the source would wait just enough time such that the retransmission would arrive at the destination. This is very difficult to achieve given that we do not know the number of flows simultaneously contributing to that switch and are unable to accurately predict server delays. Instead, we take a simpler approach; we measure the amount of ping-pong behavior the flow is experiencing and throttle the number of simultaneous retransmissions accordingly.

As shown in Figure 5, when receiving a notification for a data packet, the source stores an entry in a table specifying the data that has been lost as well as the number of times it has attempted to retransmit the packet. The source traverses this table in order of sequence number. It resends as many data packets as it can, subject to the condition that the max number of simultaneous retransmits not rise above max_{sim}. While recovering from losses, the source sends no new packets.

The goal when setting max_{sim} is to back off quickly in the presence of persistent congestion. We initially set its value to half the congestion window and have it multiplicatively decrease as the number of times the source attempts to retransmit the same packet increases. When all of the packets have been successfully retransmitted, we clear max_{sim} and set the congestion window to be half the value it was upon entering recovery.

Two edge cases arise with this approach. First, our algorithm may stall if it retransmits packets with higher sequence numbers first because TCP's acknowledgements are cumulative. To address this issue, we resend the packet immediately when we receive a notification for the smallest sequence number. Second, in extreme periods of congestion, TCP may timeout and begin resending. At this point, notifications may be received that are no longer relevant. We address this problem by including a TCP option in all packets that specifies the number of times the flow has experienced a timeout. We ensure that the TCP option is not truncated when creating the notification and check received notifications to see if they should be processed.

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

[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.

congestion. 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 by 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.

3.4.2 pFabric

pFabric is a recent proposal that combines small switch buffers, fine-grained prioritization, and small RTOs to improve high percentile flow completion times [10]. 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 max_{sim} packets. In this case, max_{sim} starts at 1 whenever a notification arrives, and increases exponentially with every successful retransmission, in effect simulating slow start.

From these examples, we see how different transport protocols can make use of drop notifications in different ways. In the next section, we describe how we setup our simulation and implementation environment.

4 Evaluation Setup

We evaluate FastLane through a Click-based [21] implementation to demonstrate the feasibility of our approach and through an NS3-based [5] simulation to investigate how its performance scales. In this section, we describe both our simulation and implementation. In the following section, we report our results.

4.1 Implementation

To implement FastLane, we modified both Click to generate notifications in response to drops as well as the Linux kernel to process them. Here we describe the modifications required for each.

4.1.1 Click Switches

We implemented the functionality required by FastLane by creating two Click elements. The first stores the input port of each arriving packet. The second takes as input all packets that are being dropped and performs the transformations as described in Section 3.

One difficulty with Click was supporting fine-grained rate-limiting. To ensure that notifications do not consume too much bandwidth, we needed to clock them out every $6\mu s$. Achieving this necessitated that we avoid Click's timing system and instead rely on the cycle counter. Every time Click checks to see if there is a packet available to transmit, we check if enough cycles have passed since the last time the notification was transmitted. If so, then we allow the notification to be sent. Otherwise, Click transmits a data packet.

The other difficulty was ensuring that the buffers allocated in Click represented the amount of buffering available per port. After a packet is "sent" by a Click element, it is actually enqueued in the driver's ring buffer. It is later DMAed to the NIC, where it is stored in another buffer until it is placed on the wire. In effect, this means that a port could have far more buffering than indicated within Click. To address this issue, we slightly rate-limited all of our links by 2% to underflow these buffers. In total, all of our modifications to Click consumed approximately 1700 lines of code.

4.1.2 Linux Servers

We modified Linux 3.2 to support FastLane. Our modifications for processing arriving notifications begin with *tcp_v4_rcv()*. We check to see if the arriving packet is a notification when obtaining the socket context. Recall that as the TCP header was copied from the dropped packet (Section 3), this requirers flipping the source and destination ports.

As normally done, we call *tcp_v4_do_rcv()* with the socket. But, at the beginning of this function, we once again check to see if the packet is a notification. If so, we perform the custom processing required. We check to see if the notification is for a data or control packet. If the notification is for a control packet and it needs to be retransmitted, we do so immediately. For data packets, we use the kernel's linked list API to store the entry.

After storing the entry, we check to see whether we can retransmit the packet based on the rules described in Section 3. If so, we walk through the socket's $\hat{w}$rite_queue to find the packet and call *tcp_transmit_skb()* to send it out. We chose not to use the retransmission routine *tcp_retransmit_skb()* to avoid the kernel's bookkeeping. Finally, we included functionality in *tcp_ack* to both determine when retransmissions have been successful and to attempt new ones.

In addition to processing notifications, we also modified Linux to support out-of-order delivery by no longer going into fast recovery when receiving three duplicate acknowledgements. In total, we added approximately 900 lines of code to the Linux kernel.

4.2 Simulation

Generally, our simulation of TCP has the same functionality as our implementation. Here discuss the salient differences. After, we will describe our simulation of pFabric.

4.2.1 TCP

In addition to including the FastLane processing logic described in the implementation, our simulation also models server processing delays. Modeling server processing delays is challenging as it depends on both the hardware used and the load on the system. Based on [20], our simulation assumes up to 16 packets may be processed at once and that processing each packet takes $5\mu s$. While modern servers may have more cores, lock contention, software processing requirements, and unpredictable software behavior will likely limit their ability to achieve greater parallelism.

4.2.2 pFabric

When simulating pFabric [10], we use the same model of server delay as described earlier. We also make two modifications to the proposed approach in the paper. First, to avoid priority inversion in request-response workloads, we prioritize each flow based on the total number of bytes in the response instead of the number of bytes left to transmit. As described in the paper, such a prioritization scheme has near-ideal performance. Second, both in pFabric and in FastLane's extension of it, we use special probe packets that do not contain any data. This enables the destination's NIC to quickly echo them back to the source, avoiding unpredictable server delays.

5 Evaluation

We now present our evaluation of FastLane. Our goal is to show the performance of our proposal in a wide variety of settings that encompass the common traffic characteristics present in today's datacenters.

We evaluate both the performance of TCP-NewReno with CoDel early marking [15, 22] and pFabric [10]. TCP-NewReno is a well-established, well-tested simulation model and pFabric is a recently-proposed multipath protocol focused on improving short-flow performance. pFabric has been shown to outperform both DCTCP and PDQ [8, 19]. When Source Quench assists TCP, quench message generation is triggered by CoDel's marking algorithm. When FastLane assists these protocols, we institute bandwidth and buffer caps on notifications. Bandwidth is capped to 1% while buffers are capped to 2.5% of 128KB = 3.2KB, based on our analysis in Section 3.

When evaluating TCP and pFabric in simulation, we set the timeouts to $1ms$ and $250\mu s$, respectively. $1ms$ timeouts for TCP are considered aggressive based on prior work [8]. 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 [20, 28]. In our implementation of TCP, we use traditional datacenter timeout values of $10ms$ [8].

We report the results of our simulated 128-server FatTree topology. Our simulated topology uses 10 Gig links with 128KB per port when running TCP and 64KB per port when running pFabric. 128KB per port is the amount of buffering typically available in TOR switches [2] and 64KB is based on pFabric's buffer calculation. To assess the practicality of our approach, we also report the results from our 16-server FatTree topology running on Emulab [4]. As described earlier, we use Click to provide the switch functionality [21]. All of the links in the topology are 1 Gig. Given the reduced link speeds, we scale buffers to 64KB per port. Both topologies are full bisection bandwidth, demonstrating the advantages of FastLane when most packets are dropped at the last hop. We use flow hashing to spread the load across these topologies when in-order delivery is required (i.e., for TCP) and use packet scatter otherwise.

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 [8]. As these requests are typically 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 [8].

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 [8], our long requests follow an all-to-all traffic pattern. Throughout our evaluation, we refer to both short and long requests based on the flow size of the response.

In this section, we begin by presenting our simulation results. Once the high-level benefits have been established, we use the simulator to dig more deeply, exploring various properties of FastLane such as how efficient it

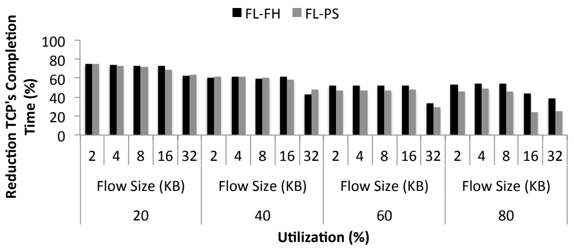

Figure 6: Reduction in TCP's 99.9th percentile flow completion time when assisted by FastLane. For FastLane, we show the results with both flow hashing (FL-FH) and packet scatter (FL-PS).

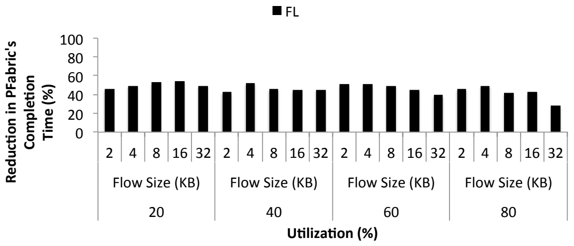

Figure 7: Reduction in pFabric's 99.9th percentile flow completion time when assisted by FastLane (FL).

is in terms of bandwidth and buffer consumption and its sensitivity to different bandwidth and buffer caps. Next, we report our implementation results, which demonstrate the feasibility of using FastLane. We conclude this section by summarizing the key takeaways.

5.1 Simulation Results

Here we evaluate how FastLane improves both TCP's and pFabric's performance. 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 [12]. Then we keep the utilization constant at 60% and vary the fraction of the load caused by the short request-response workflows. After establishing 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.

5.1.1 Varying Utilization

In Figure 6, we report the reduction in TCP's 99.9th percentile flow completion time across a range of utilizations. In most cases, Source Quench does not benefit TCP, so we do not report its results. However, with FastLane, performance improves dramatically, irrespective of whether flow hashing is used. At 20% utilization, 2KB flow completion times reduce from 1.2*ms* to 0.3*ms*, a 75% reduction, when using FastLane with packet scatter (FL-PS). As utilization increases, the percentage reduction decreases because higher loads decrease the amount of time that can be saved by avoiding a timeout. However, at 80% utilization, FL-PS helps a 2KB flow complete in 0.9*ms* as opposed to 1.6*ms*, a reduction of 44%.

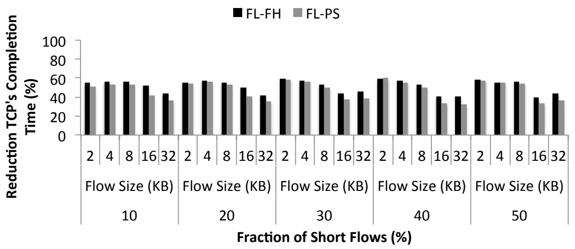

Figure 8: Reduction in TCP's 99.9th percentile flow completion time for a different fractions of short flows.

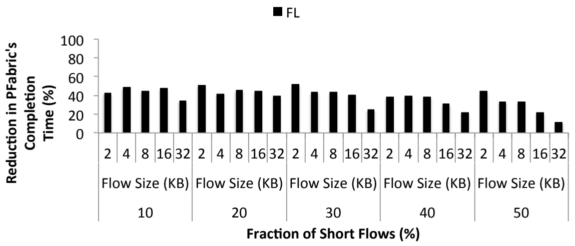

Figure 9: Reduction in pFabric's 99.9th percentile flow completion time for different fractions of short flows.

Interestingly, we often see greater improvement in short flow completion times when FastLane is used with flow hashing as compared to packet scatter. As long flows can capture more resources with packet scatter, they contend more with short flows. We see evidence of this by looking at long flow completion times. With packet scatter, average long flow completion times reduce up to 24%, whereas with flow hashing they remain within a few percent of unaided TCP.

As shown in Figure 7, FastLane also helps pFabric's 99.9th percentile flow completion times. At 40% utilization, FastLane reduces completion times from 0.6*ms* to 0.3*ms*, a 50% reduction. As both pFabric and FastLane's extension to it use packet scatter, their average long flow completion times stay within 5% of each other, with FastLane often performing better.

5.1.2 Varying Fraction

While only 10% of the load is typically caused by short flows, we wanted to evaluate FastLane in a wider range of environments. In Figure 8, we show how effective FastLane is when short flows represent different fractions of the total load. In all cases, the total load is held constant at 60%. 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 55%). As shown in Figure 9, FastLane continues to provide significant benefits to

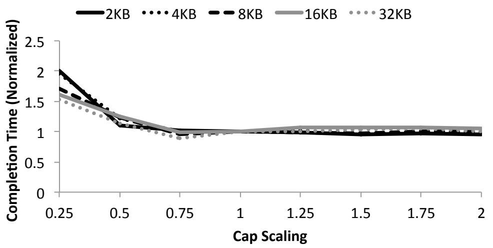

Figure 10: FastLane's sensitivity to the bandwidth and buffer caps when aiding TCP.

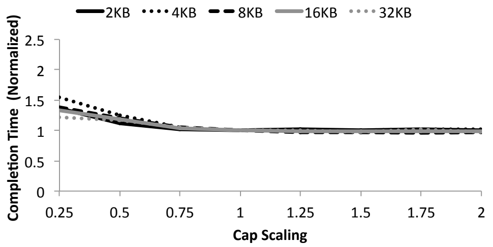

Figure 11: FastLane's sensitivity to the bandwidth and buffer caps when aiding pFabric.

pFabric as well.

With respect to long flows, the behavior of this experiment is very similar to the previous one for TCP. For pFabric, in the extreme case that 50% of the load is due to short flows, average long flow completion times do inflate by a modest 10%. We argue that this is a worthwhile tradeoff to make as FastLane decreases short flow completion times by up to 44% in this scenario.

5.1.3 Sensitivity Analysis

To perform our sensitivity analysis, we use the workload with 60% total load, where half is due to short flows. This workload has the greatest number of bursts and should hence stress our system the most. We evaluate how sensitive FastLane is to different buffer caps, smaller buffers, and varying server latency.

Sensitivity to 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 10 and 11, we depict FastLane's sensitivity to the cap when it is assisting TCP and pFabric, respectively. These figures show the 99.9th percentile

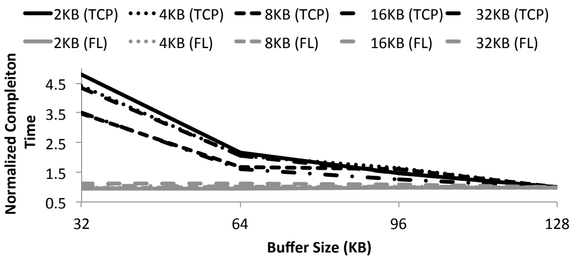

Figure 12: 99.9th percentile completion time of TCP with and without FastLane for varying buffer sizes.

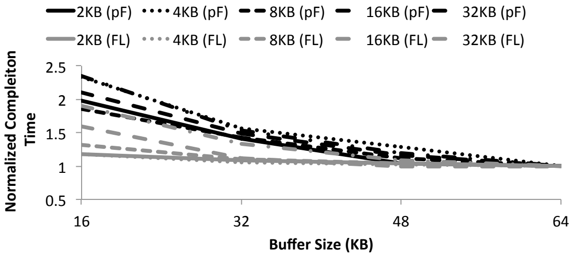

Figure 13: 99.9th percentile completion time of pFabric with and without FastLane for varying buffer sizes.

completion time for different 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 performance hit until we scale the bandwidth and buffers to 0.5. 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 12, 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 38% from 3.9ms to 5.4 ms as we

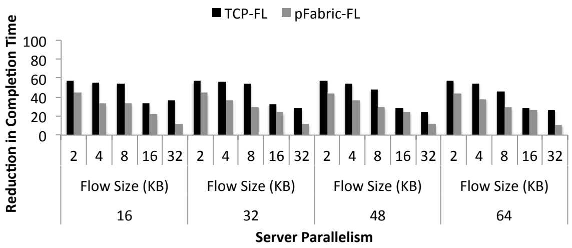

Figure 14: 99.9th percentile reduction in flow completion time with varying server parallelism.

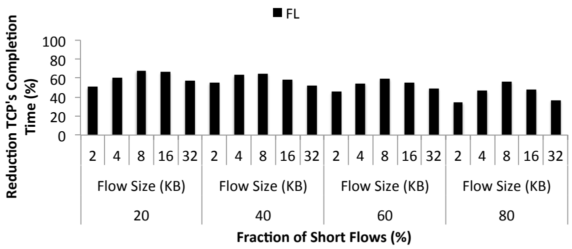

Figure 15: Reduction in TCP's 99.9th percentile flow completion time when assisted by FastLane.

go from 128KB to 32KB.

Figure 13 shows the results for the same experiment performed with pFabric. FastLane is not able to prevent the 99.9th percentile completion times of 16 and 32KB flows from increasing. Average long flow completion times suffer as well, increasing by 72% with FastLane and 88% with 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 9, 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 [9].

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 14 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.

5.2 Implementation Results

Here we discuss the implementation feasibility of our proposal. 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 an fair head-to-head comparison, we demonstrate the performance improvement of 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 5.1.1). 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 fronted and backend servers, with the frontend servers requesting data from backend servers.

5.2.1 Varying Utilization

Figure 15 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 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.

Compared to the implementation results, the simulator reports a greater reduction in flow completion times. We argue that this is primarily due to the limitations of our testbed. Our unoptimized version of Linux, coupled with the burstyness of the workload, leads to server delays that limit our potential to reduce flow completion times. Evidence of this can be found by looking at the 2KB flows, which do not see as big a reduction because they typically require that more sockets be created within a short timespan. Optimizing servers is a large research undertaking in its own right [18,20]. We anticipate that as servers continue to be optimized, the benefits of FastLane will increase, approaching those reported by the simulation.

Table 1 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 15, demonstrating its utility. The one point where FastLane slightly underper-

Util	2*KB*	4*KB*	8*KB*	16*KB*	32*KB*
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 1: Reduction in 99.9th percentile flow completion vs TCP with advanced features.

	FL (TCP)			FL (TCP-A)		
Util	1*MB*	16*MB*	64*MB*	1*MB*	16*MB*	64*MB*
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 2: Reduction in average completion time of long flows

forms 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. While not shown in the table, at this utilization, we do see a reduction of 55% in the 99.95th percentile completion time for 32KB flows.

5.2.2 Long Flows

Our implementation environment allows us to evaluate the flow completion times of longer flows, while maintaining manageable runtimes. Table 2 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. No small flows are run in this experiment.

We see that FastLane reduces average completion times by as much as 31% at high utilizations. FastLane slightly underperforms TCP for long flows at light utilizations. This performance impact is small and is observed in traffic scenarios not typically experienced in datacenters. We conclude that the benefits of FastLane far outweigh its modest cost.

5.3 Summary

Our conclusions from evaluating FastLane are:

- FastLane reduces 99.9th percentile completion times of short flows up to 75% over TCP and 50% over pFabric.
- FastLane's ability to use packet scatter results in a 24% reduction in long flow completion times versus TCP. Even in extreme scenarios, FastLane's long flow performance stays within 10% of pFabric's.
- FastLane maintains its performance benefits when TCP's advanced features are enabled, demonstrating its utility.
- With FastLane, average flow completion times for 1-64MB flows increase by 4% at light utilizations, but decrease by up to 31% at higher loads.

6 Related Work

Researchers have proposed an extensive set of transport modifications for datacenter networks. DCTCP, HULL, and D2TCP rely on end-host solutions that avoid / minimize modifications to network elements [8, 9, 27]. FastLane can improve upon these proposals by reducing the cost of drops.

Proposals such as D^3 and PDQ have opted instead to rely on extensive network modifications to support explicit reservations [19, 29]. During every RTT, these proposals request resources for the next one. Most flows in the datacenter are short and can complete within one RTT [12]. FastLane could enable these proposals to safely transmit in the first RTT, dramatically reducing flow completion times.

Industry has adopted standardized Ethernet link-layer improvements, such as Quantized Congestion Notifications (802.1Qau [1]). FastLane avoids the performance problems of this approach by directly notifying transport.

[30] proposes to orchestrate the datacenter bridging protocols [3] into a stack. DeTail, like other lossless interconnects [7], requires relatively larger per-port buffers to guarantee that packets are not dropped. Back-of-the-envelope calculations suggest that these requirements are higher than the buffers currently available for commodity 10 gigabit switches [2].

7 Conclusion

In this paper, we presented FastLane, an agile drop signaling mechanism for improving high-percentile datacenter networking performance. By having switches directly notify sources when drops occur, FastLane tries to minimize the delay incurred by senders in detecting and responding to packet drops.

We demonstrated the efficacy and generality of our work by modifying TCP and pFabric to take advantage of FastLane. The testbed experiments and simulations show that the rapid notification mechanism helps achieve significant reduction in worst-case flow completion times, up to 75%. These improvements do not come at a large cost— FastLane achieves them even when its bandwidth and buffers are capped to 1% and 2.5%, respectively.

Perhaps the greatest value of FastLane is that all of these advantages are transport agnostic and can benefit many protocols. With the increasing interest in improving worst-case performance in datacenters, we hope our efforts are well placed.

8 Acknowledgements

This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

References

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

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

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

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

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

[6] Tcp chimney. http://download.microsoft.com/download/5/E/6/5E66B27B-988B-4F50-AF3A-C2FF1E62180F/ENT-T557_WH08.pptx.

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

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

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

[10] Alizadeh, M., Yang, S., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. Deconstructing datacenter packet transport. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2012), HotNets-XI, ACM, pp. 133–138.

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

[12] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 267–280.

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

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

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

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

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

[18] Han, S., Marshall, S., Chun, B.-G., and Ratnasamy, S. Megapipe: a new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 135–148.

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

[20] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 9:1–9:14.

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

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

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

[24] Postel, J. Internet control message protocol, 1981.

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

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

[27] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 115–126.

[28] Vasudevan, V., Phanishayee, A., Shah, H., Krevat, E., Andersen, D. G., Ganger, G. R., Gibson, G. A., and Mueller, B. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM* (2009).

[29] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM* (2011).

[30] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. H. Detail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference* (New York, NY, USA, Aug 2012), SIGCOMM '12, ACM.