FastLane: Making Short Flows Shorter with Agile Drop Notification

David Zats[⋆†], Anand Padmanabha Iyer[⋆], Ganesh Ananthanarayanan[§],
Rachit Agarwal[⋆], Randy Katz[⋆], Ion Stoica[⋆], Amin Vahdat[†]

[⋆]*University of California, Berkeley* [†]*Google* [§]*Microsoft Research*

Abstract

The drive towards richer and more interactive web content places increasingly stringent requirements on datacenter network performance. Applications running atop these networks typically partition an incoming query into multiple subqueries, and generate the final result by aggregating the responses for these subqueries. As a result, a large fraction — as high as 80% — of the network flows in such workloads are *short* and *latency-sensitive*. The speed with which existing networks respond to packet drops limits their ability to meet high-percentile flow completion time SLOs. Indirect notifications indicating packet drops (e.g., duplicates in an end-to-end acknowledgement sequence) are an important limitation to the agility of response to packet drops.

This paper proposes *FastLane*, an *in-network drop notification* mechanism. *FastLane* enhances switches to send high-priority drop notifications to sources, thus informing sources as quickly as possible. Consequently, sources can retransmit packets sooner and throttle transmission rates earlier, thus reducing high-percentile flow completion times. We demonstrate, through simulation and implementation, that *FastLane* reduces 99.9^{th} percentile completion times of short flows by up to 81%. These benefits come at minimal cost — safeguards ensure that *FastLane* consume no more than 1% of bandwidth and 2.5% of buffers.

Categories and Subject Descriptors C2.2 [*Computer - Communication Networks*]: Network Protocols

Keywords Datacenter networks, Transport Protocols

SoCC '15, August 27 - 29, 2015, Kohala Coast, HI, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3651-2/15/08...$15.00.
http://dx.doi.org/10.1145/2806777.2806852

1. Introduction

Modern data centers host a wide variety of applications including web search, social networks, recommendation systems, and database storage. The workloads encountered in these applications have two common features. First, a large fraction of the network flows are *latency-sensitive short flows*; for instance, [12] reports that 80% of the flows have less than 10KB of data. Second, applications produce an output by aggregating the results from multiple short flows and must *wait for the last of these flows* to finish. As a result, these applications face an increasingly stringent latency requirement, with "efficiency" often measured in terms of 99.9 percentile of flow completion time [16, 17].

Motivated by above applications, a number of protocols [8–10, 21, 29, 31] have been proposed to reduce the flow completion time in data centers. While these protocols reduce the likelihood of packet drops and perform extremely well in reducing the flow completion time in absence of packet drops, they resort to TCP-style mechanisms (duplicate acknowledgements and timeouts) when the packets are in fact dropped, leading to high flow completion times in such scenarios. Unfortunately, short flows are particularly sensitive to packet drops. These flows may not contain enough packets to generate three duplicate acknowledgements or at the least increase the likelihood of all packets in a window being dropped, thus leading to timeouts. The problem is further exacerbated by the bursty nature of the data center workloads and frequent packet drops due to lack of deep buffered switches [8, 9].

How should the network react when these short flows experience packet drops? Clearly, approaches that rely on duplicate acknowledgements and timeouts will incur delays much larger than the lifespan of short flows. Even with tight timeouts [8, 30], our measurements (§4) show that relying on indirect indicators (e.g., duplicates in an end-to-end acknowledgement sequence) can increase short flow completion times five-fold. End-to-end approaches with improved window management [8, 9, 29] can also be far from effective, requiring multiple round trip times (RTTs) worth of delay to properly respond to congestion. During this time period, many packet drops and timeouts may occur. Finally, explicit rate control mechanisms [21, 31] do not solve the problem either — these mechanisms may in fact inflate the completion times of most datacenter flows, requiring multiple RTTs for transmissions that could have completed in just one [10, 12, 20].

This paper presents *FastLane*, a lightweight drop notification mechanism that can be integrated with existing transport layer protocols for efficient handling of packet drops. *FastLane* allows switches to directly transmit notifications to sources, quickly informing them of packet drops. By generating and transmitting notifications at the location of the packet drop, *FastLane* informs sources *as quickly as theoretically possible*. This enables sources to respond to the packet drop sooner, thus reducing high-percentile flow completion times. Explicit drop notification in *FastLane* also enables the sources to distinguish between out-of-order delivery and packet loss. This in turn enables the network to perform per-packet load balancing, exploiting existing path redundancy [7, 20] to avoid hot-spots and the associated delays.

The idea of in-network notification has already been advocated by proposals like ICMP Source Quench [19] and QCN [1]. *FastLane* differs from these proposals in two main aspects. First, while the above proposals focused primarily on *congestion control*, *FastLane* explicitly focuses on the problem of efficient *loss notification and recovery*. While the two problems are related, bursty traffic and small buffers may lead to packet drops even with congestion control mechanisms. Second, *FastLane* resolves a number of challenges that resulted in limiting the adoption of previous proposals. The most important among these are: (i) making notifications semantically rich enough to allow sources to identify the event that triggered it and which (if any) packet was lost; (ii) ensuring that notifications arrive at the source as quickly as possible; and (iii) incorporating safety mechanisms, preventing notifications from causing congestion collapse.

FastLane resolves the first challenge by including the transport header of the original packet in drop notifications, providing sources sufficient information to identify that drops had occurred and which packets were lost. To resolve the second challenge, *FastLane* makes the design decision of generating notifications in the data plane and giving them the highest priority. We show that this can be performed efficiently with minimal changes in the underlying switch hardware. Finally, *FastLane* prevents notifications from consuming excessive network resources by installing (tunable) analytically-determined buffer and bandwidth caps. *FastLane* has the additional benefit of being *transport-agnostic*. We demonstrate that *FastLane* can be easily integrated with existing transport layer protocols by extending both TCP and pFabric [10] to leverage *FastLane*'s functionality.

We perform extensive evaluation of *FastLane*, integrated with two transport layer protocols — TCP and pFabric [10]. Our evaluation, which includes both simulation and implementation, shows that *FastLane* can improve the 99.9th percentile completion time of short flows by up to 81% for TCP and by up to 52% for pFabric. *FastLane* achieves these improvements even when we cap the bandwidth and buffers used by notifications to as little as 1% and 2.5%, respectively. Since increasing the network bandwidth and switch buffers by 2.5% is significantly easier than reducing the 99.9th percentile completion time by 52%, we believe that this is an appropriate trade-off to make.

2. The Case for Drop Notification

Measurement studies from a variety of datacenters have shown that workloads are dominated by short flows that are often latency-sensitive. For instance, a measurement study across ten datacenters [12] shows that more than 80% of the flows have size less than 10KB. A study from Microsoft's production datacenter also shows that latency-sensitive flows typically range from 2–20KB in size [8].

Another peculiar property of data center workloads is the partition-aggregate workflow, where applications produce an output by aggregating results from multiple short flows. On the one hand, such workflows induce extreme traffic bursts on the network, resulting in frequent packet drops. On the other hand, their completion time is bound by the time it takes for the last flow to arrive. When flows stall, applications such as web search must typically make the difficult decision between delaying the response and violating stringent user-perceived deadlines [23] or returning early and degrading the quality of the response [8]. Thus, network performance for such workloads is often measured in terms of 99.9 percentile of flow completion time [16, 17].

In this section, we argue that the above two properties of data center workloads necessitate a carefully designed packet drop notification mechanism to meet the stringent latency requirements. We then make a case for direct notification, the underlying design principle of *FastLane*. Having settled the need for direct notification, we investigate existing direct notification schemes with a particular focus on the design decisions of these schemes that dramatically limit their effectiveness. Based on this investigation, we propose a series of design principles for direct notification, which we use in the next section to design *FastLane*.

2.1 Notifying Drops

As discussed above, short flows are particularly sensitive to packet drops. In the absence of sufficiently many acknowledgements, existing transport layer protocols [8–10, 21, 29, 31] typically resort to timeouts for such flows. The problem with timeouts is the undesirably high flow completion time. Timeouts are purposely set to large values to increase the certainty that the missing packet has actually been dropped. Sources must set timeouts sufficiently high to account for queueing delays and unpredictable server delays, else they will have spurious retransmissions. To be able to set smaller timeouts, we must achieve both (i) small queueing delays; *and* (ii) predictable server delays.

Reducing buffer sizes can partially mitigate large queueing delays. However, packets may still observe unpredictable queuing delays as they traverse through the many hops between the source and destination. Achieving predictable server delays is even more challenging. Recent work has shown that even highly-optimized servers can take hundreds of microseconds to respond to requests in the presence of bursts [22] — up to an order of magnitude longer than the RTTs in unloaded data centers [10].

One way of avoiding timeouts is to allow switches to notify sources when packet drops occur. This can be achieved in two ways — the switch can notify the source either by sending the notification to the destination, which must echo it back to the source, or by sending it directly to the source. A recent proposal, CP [14], explores the first approach highlighting its ability to both avoid timeouts and maintain the ACK clock. To the contrary, this paper focuses on the second approach. Sending the notification directly to the source avoids the additional overheads incurred by forwarding the notification to the destination. Specifically, direct notification avoids potentially high latencies from queuing delays along the path to the destination and the additional processing time at the destination. Our evaluation (§4) shows that avoiding this overhead can lead to significant performance improvements in high percentile flow completion times for typical workloads and network utilizations.

2.2 Existing Alternatives

Using direct notification for improving flow completion time was proposed by ICMP Source Quench and Quantized Congestion Notification (802.1Qau) [1, 19]. 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 were 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.

2.2.1 ICMP Source Quench

Switches used ICMP source quench *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 (15 years ago), it responded to those messages in the same way as it does to ECN [28]. It reduced the congestion window but it waited for 3 duplicate acknowledgements or a timeout to retransmit the packet.

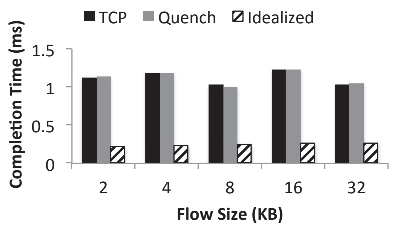


Figure 1: 99.9th percentile flow completion times.

Source quench messages suffered from two other problems. Due to having the same priority as the offending data packet, quench messages often took a long time to arrive at the source, thus diminishing potential gains [11]. Furthermore, there were no safeguards to avoid overconsumption of resources during extreme congestion.

To quantify the impact of these design decisions, we evaluated Source Quench using the workload in §4. In this workload, we have bursts of short flows (up to 32KB in length) and long flows (1 MB in length). Figure 1 shows the 99.9th percentile completion times for the short flows. We see that while Source Quench does not perform significantly better than TCP, an idealized drop notification mechanism that does not have limitations of Source Quench could reduce high-percentile completion times by 81%.

2.2.2 Quantized Congestion Notification

Quantized Congestion Notification (QCN) is a direct notification scheme proposed as part of the 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 [27]), (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]. In fact, 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) [15].

2.3 Direct Notification Design Principles

Based on the lessons learned from digging deeper into 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 are left guessing the appropriate action to take. If sources respond conservatively by waiting for an indirect indicator (i.e., 3 duplicate acknowledgements or a timeout), flows will suffer large delays. If sources respond aggressively, retransmitting the packet, they risk increasing network load aggravating congestion.
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, possibly harming high-percentile flow completion times. Moreover, by hiding congestion/drop information from transport, they prevent it from making the best decision possible.

While simple, these principles are fundamental for achieving predictable flow completion times. ICMP Source Quench does not satisfy (see Table 1) Design Principles 1-4 and QCN does not satisfy principles 3-5. In the next section, we discuss how the design of *FastLane* adheres to these principles.

Principle	1	2	3	4	5
Source Quench	×	×	×	×	√
QCN	√	√	×	×	×
FastLane	√	√	√	√	√

Table 1: Design principles satisfied by Source Quench, QCN, and *FastLane*.

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. Eventually, it runs out of capacity and must drop 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 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 safeguards, capping the bandwidth and buffers used by notifications (Section 3.3).

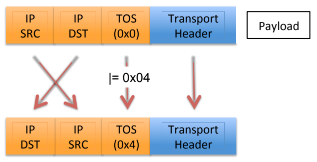


Figure 2: Transforming packets into notifications.

3.2 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 2. 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 [24].

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 (**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*.

3.3.1 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:

$$l_n = ps_n/(s_r + ps_n), \tag{1}$$

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 show how to size the buffers used by notifications next.

3.3.2 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 consider a variety

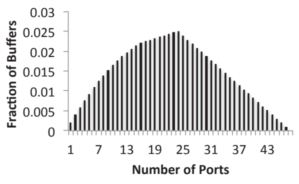


Figure 3: The fraction of a switch's buffers used by notifications when ports receive bursts simultaneously.

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, and each dropped packet will generate a notification. Then the number of bytes necessary to store the resulting notifications is approximated as follows:

$$s_b \times s_n / s_r \times (1 - 1/p_{in}), \qquad (2)$$

where s_b is the size of the burst, s_n is the size of the notification, s_r is the size of the average regular 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 s_b / p_{in} bytes 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 ports 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 3, 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 all input ports are used and 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. 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 [21, 31]. Scenarios with hundreds of sources participating in Incast will complete faster because there will be fewer rounds of timeouts and synchronized pullbacks.

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 (**Principle 5**). Here, we discuss how TCP uses notifications to improve high-percentile flow completion times. Later we present our proposed modifications to pFabric.

3.4.1 TCP

Our modifications to TCP leverage notifications to (i) quickly retransmit dropped packets and (ii) support multipath transmission.

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. 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

[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.

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 $\alpha = \frac{cwnd}{2}$. For every drop notification while in recovery mode, we exponentially decrease $bound\_sim$ according to the following equation:

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

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. The algorithm for processing drop notifications is presented in 1.

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 $entry.seqno$
        $entry.issent \leftarrow 1$
        $sim\_retx \leftarrow sim\_retx + 1$
```

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. 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 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 $bound\_sim$ packets. In this case, $bound\_sim$ starts at 1 whenever a notification arrives, increasing exponentially with every successful retransmission, in effect simulating slow start.

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

4. Evaluation

This section evaluates the performance of *FastLane* under a wide variety of network configurations and application workloads — varying short flow sizes 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 §3.

We evaluate the following protocols in this section: (i) **TCP**, using NewReno [18] (ii) **TCP-Codel**, using an early marking scheme to reduce buffer bloat [25], (iii) **TCP-Quench**, using ICMP source quench messages triggered by Codel, (iv) **TCP-Codel-FL-FH**, integrating *FastLane* and using the existing flow-hashing based load balancing scheme with TCP-Codel, (v) **TCP-Codel-FL-PS**, the same as TCP-Codel-FL-FH, but with packet scatter to more evenly balance load, (vi) **TCP-Codel-CP**, the same as TCP-Codel-FL-PS, but with notifications sent to the receiver and echoed back as in CP [14], (vii) **DCTCP**, using fine-grained marking and rate reduction to reduce buffer consumption [8], (viii) **pFabric**, using shallow buffers, fine-grained timeouts, and fine-grained prioritization to reduce flow completion times [10], and (ix) **pFabric-FL**, using *FastLane* to assist pFabric in preventing timeouts. For all protocols, we send data in the first RTT, similar to TCP FastOpen [26].

4.1 Methodology

We now describe the network and protocol configuration, and application workloads used in our evaluation.

Network Configurations. Our NS-3 [5] based simulator 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-based protocols and 64KB per port when running pFabric. These numbers are based on the amount of buffering typically available in TOR switches [2] and pFabric's buffer calculation [10], respectively. Based on [22], we model server processing delays as taking $5\mu s$ per packet, processing up to 16 packets in parallel.

For our Click-based implementation, we use a 16-server, full bisection bandwidth, FatTree topology running on Emulab [4]. Each link in the topology is 1 Gig. Given the reduced link speeds, we scale buffers to 64KB per port.

Timeouts. For our simulations, we set the timeout for TCP-based protocols to be $1ms$ and for pFabric to be $250\mu s$. $1ms$ timeouts for TCP-based protocols is 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 [22, 30]. However, for our Click-based implementation, we use the traditional datacenter timeout value of $10ms$ [8].

Notifications and Load balancing. When using *FastLane*, we institute bandwidth and buffer caps on notifications. Based on our analysis in §3, 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, as typically observed in datacenters [8]. Similar to partition-aggregate workflows, the sources initiate small requests 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 and follow an all-to-all traffic pattern [8].

4.2 Simulation Results

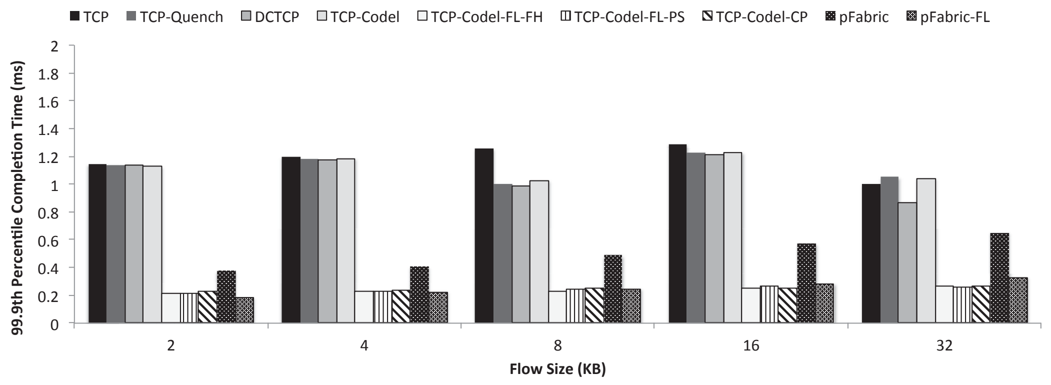
We start by evaluating, in §4.2.1, the performance of *FastLane* across a range of utilizations (defined as as the average load on the core) for a workload where 10% of the load is caused by short request-response workflows and 90% of the load is caused by long workflows. This is the distribution typically seen in production datacenters [12]. Then, in §4.2.2, we keep the utilization constant at 60% and vary the fraction of the load caused by the short request-response workflows. Finally, in §4.2.3, we evaluate *FastLane*'s sensitivity to (i) bandwidth and buffer caps, (ii) smaller buffer sizes, and (iii) varying amounts of server latency.

4.2.1 Varying Utilization

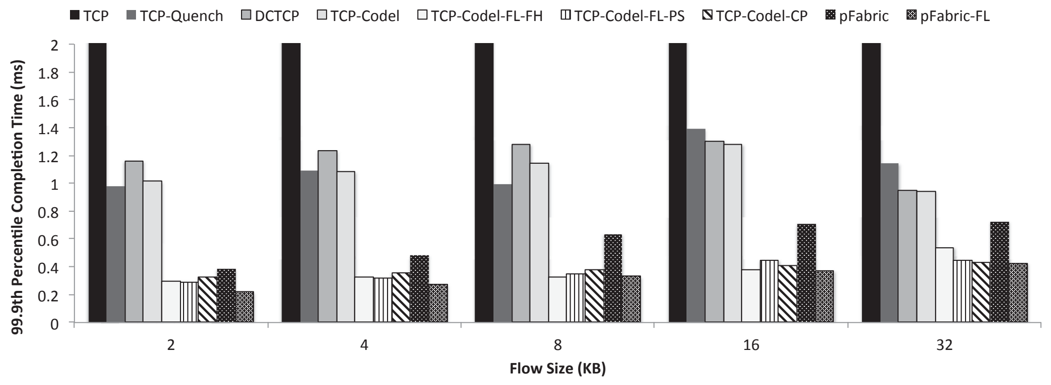
Figure 4 shows the 99.9th percentile flow completion times for all the evaluated protocols, for network utilizations of 20% and 80%; results for other utilizations are very similar. We observe that for the evaluated workloads and flow sizes: (i) TCP-Quench, TCP-Codel, and DCTCP achieve similar flow completion times; and (ii) *FastLane* effectively assists TCP-Codel and pFabric.

Note that, for most protocols, the high-percentile flow completion times for short flows do not increase significantly as the utilization increases from 20% to 80%. Intuitively, the protocols other than TCP are successful at reducing buffer utilization for long flows. As a result, while we observed a significant increase in average completion time of long flows, the high-percentile flow completion times for short flows do not change significantly since the short flows represent to at most 8% of the utilizaiton.

Quantitatively, we see that at 20% utilization, using FastLane with packet scatter reduces the 2KB flow completion times for TCP-Codel from $1.12ms$ to $0.21ms$ (an 81% reduction) and for pFabric from $0.38ms$ to $0.18ms$



(a) Flow Completion times for short flows at 20% utilization



(b) Flow Completion times for short flows at 80% utilization. Note: TCP's 99.9th percentile flow completion is often > 2 ms.

Figure 4: 99.9th percentile flow completion time for varying network utilization.

(a 52% reduction). At 80% utilization, the respective improvements are very similar.

From Figure 4, it may seem as though packet scatter does not provide significant benefits over flow hashing (see TCP-Codel-FL-PS and TCP-Codel-FL-FH). 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%. Since pFabric already employs packet scatter, *FastLane* does not effect its long flow performance, other than at 80% utilization where it reduces average completion times by up to 38%.

Also note, from Figure 4, that the direct notification of FastLane and indirect notification of CP offer similar performance benefits for this particular setup. However, direct notification can offer significant performance benefits in many real-world scenarios. For example, when the drop occurs on a heavily congested path, sending the notification along this same path to the receiver significantly increases the time taken to inform the source, inflating flow completion times. Paths can experience heavy congestion due to many reasons including uneven load balancing, topological asymmetries, and failures. In Figure 5, we evaluate the performance of FastLane against CP in presence of one such event — the failure of a core to agg link, degrading it from 10Gbps to 1Gbps. In this Figure, we see that 99.9th percentile flow completion times can reduce by an additional 82% when notifications are sent directly instead of being forwarded to the receiver.

Based on results in Figure 4, we focus on FastLane's ability to reduce flow completion times of both TCP-Codel and pFabric in the remainder of the section. However, as discussed earlier, FastLane is indeed intended to assist any transport protocol for efficient handling of packet drops.

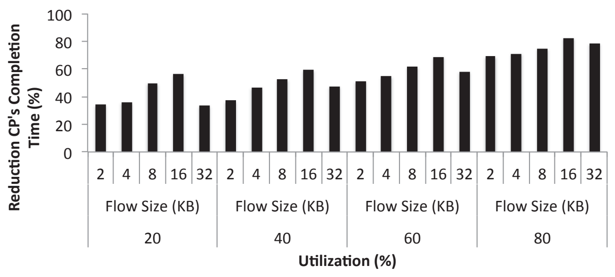


Figure 5: Reduction in CP's 99.9th percentile flow completion time.

4.2.2 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(a) and Figure 6(b)). Even when 50% of the load is due to short flows, *FastLane* provides significant benefit to TCP-Codel (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 increase under this traffic mix because the more bursty workload leads more flows to experience timeouts, providing *FastLane* more opportunities to help. Similar observations apply to FastLane's improvement to pFabric (Figure 6(b)).

For long flows, results for TCP-Codel are 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.

4.2.3 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: We now evaluate the sensitivity of *FastLane* 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). In this experiment, we do not allow use of extra resources to accurately understand the effect of the cap.

Figures 7(a) and 7(b) show the 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-Codel 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. We conclude 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: We now evaluate how *FastLane* performs with smaller buffer sizes. We start with the default TCP-Codel 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 8(a), we report the results for *FastLane* when assisting TCP-Codel. 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-Codel's 99.9th percentile flow completion times do not degrade as we reduce buffer sizes. Without *FastLane*, TCP-Codel'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.

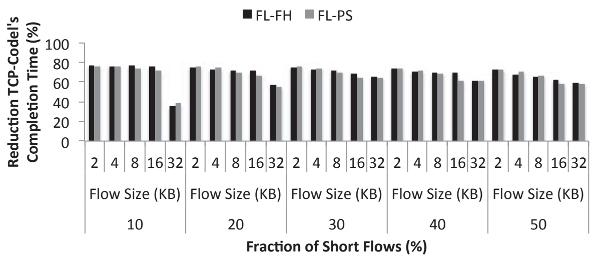
Figure 8(b) shows the results for the same experiment performed with pFabric. *FastLane* is not able to prevent the 99.9th percentile completion times of 8, 16 and 32KB flows from increasing. Average long flow completion times suffer as well, increasing by approximately 5× for both *FastLane* and unaided pFabric as we reduce buffers from 64KB to 16KB. However, we note that pFabric already tries to use the minimum buffering possible. Second as these numbers are normalized to what each flow would achieve in Figure 6(b), *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-Codel'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].

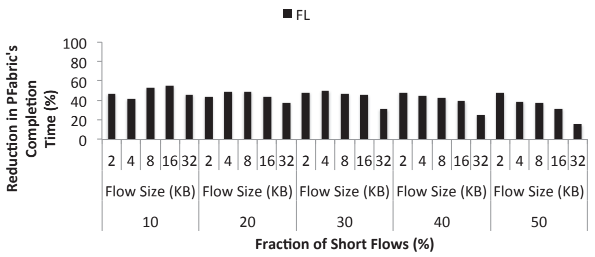
Server Parallelism: Our simulations have a server model that processes 16 packets in parallel. As server hardware varies greatly, we explore how different amounts of parallelism affect flow completion times. Figure 9 reports the reduction in 99.9th percentile flow completion times for TCP-Codel and pFabric as a function of server parallelism. *FastLane*'s performance improvement does not diminish as the amount of parallelism increases.

4.3 Implementation Results

We now discuss the implementation results for *FastLane*. For ease of implementation, we disabled the more advanced features of Linux TCP (i.e., SACK, DSACK, Timestamps, FRTO, Cubic). However, we retained ECN with Codel-based marking. To ensure that *FastLane* provides

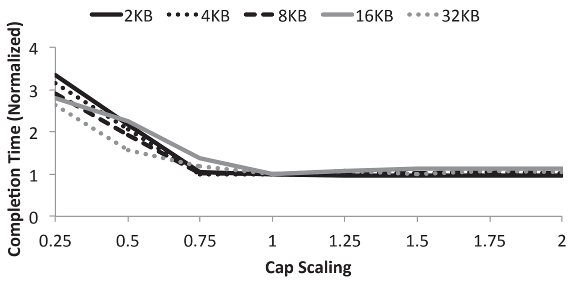


(a) TCP-Codel when assisted by *FastLane* with flow hashing (FL-FH) and with packet scatter (FL-PS)



(b) pFabric when assisted by *FastLane* (FL)

Figure 6: Reduction in 99.9th percentile flow completion time for varying fraction of short flows.



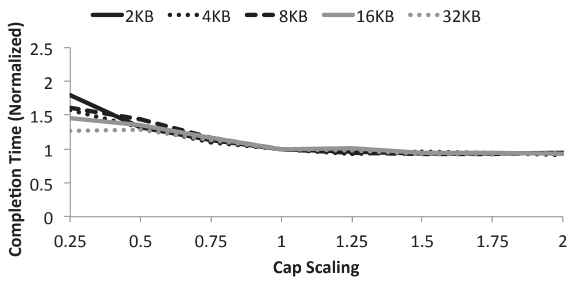
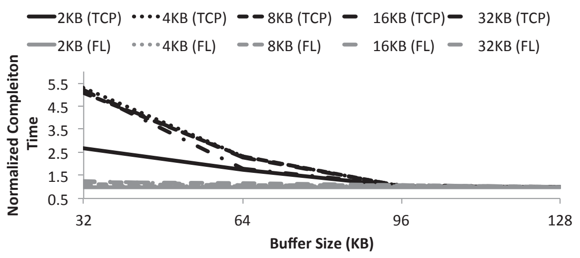


Figure 7: *FastLane*'s sensitivity to the bandwidth and buffer caps when aiding TCP-Codel (left) and pFabric (right).



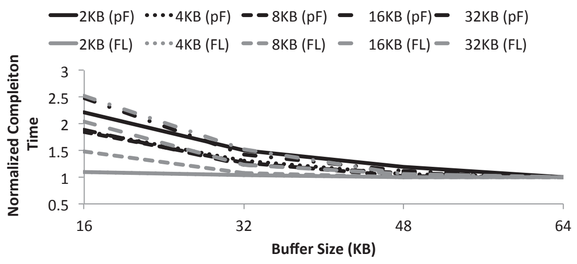


Figure 8: Reduction in 99.9th percentile completion time for varying buffer sizes for TCP-Codel (left) and pFabric (right) with and without *FastLane*. In this figure, TCP refers to TCP-Codel.

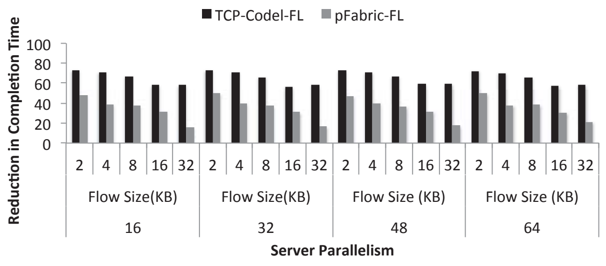


Figure 9: 99.9th percentile reduction in flow completion time with varying server parallelism.

useful functionality beyond that provided by SACK, DSACK, Timestamps, FRTO, and Cubic, we also report how *FastLane* compares to TCP with these features enabled.

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 4.2.1). 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 frontend and backend servers, with frontend servers requesting data from backend servers.

4.3.1 Varying Utilization

Figure 10 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

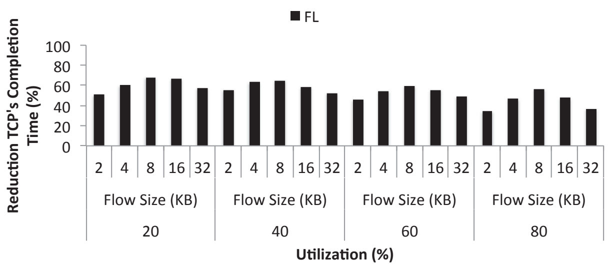


Figure 10: (Implementation) Reduction in TCP's 99.9th percentile flow completion time when assisted by *FastLane*.

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.

Table 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 10, 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.

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

4.3.2 Long Flows

Our implementation setup allows us to evaluate the flow completion times of longer flows, while maintaining manageable runtimes. Table 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 utilizations. 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.

5. Related Work

Researchers have proposed a few types of protocols to reduce the completion time of flows in datacenter networks. The first type focuses on minimally modifying network hardware. Examples of these protocols include DCTCP, HULL, and D2TCP [8, 9, 29]. *FastLane* can improve upon these proposals by reducing the cost of drops.

Alternatively, other proposals such as D^3 and PDQ have opted instead to rely on network modifications to support explicit reservations [21, 31]. 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 transmit in the first RTT, thus reducing flow completion times.

Finally, [32] proposes to orchestrate the datacenter bridging protocols [3] into a stack. DeTail, like other lossless interconnects [6], 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 10Gbps switches [2].

	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 3: (Implementation) Reduction in average completion time of long flows

6. Conclusion

In this paper, we presented *FastLane*, an in-network drop notification mechanism for reducing the high percentile flow completion time of short flows. *FastLane* allows switches to generate packet drop notifications and transmit them directly to the respective sources. Sources are, thus, informed of packet drops *as quickly as theoretically possible*, allowing them to respond in-time.

Using extensive implementation and simulation results, we demonstrated that *FastLane* significantly reduces the 99.9th percentile flow completion time of short flows (as much as by 81%) using minimal bandwidth and buffer resources. We have extended TCP and pFabric to leverage the functionality of *FastLane*, demonstrating the transport-agnostic nature of our approach.

Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata and VMware.

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

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

[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., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM* (2013).

[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 *IMC* (2010).

[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).

[14] Cheng, P., Ren, F., Shu, R., and Lin, C. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *NSDI* (2014).

[15] 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.

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

[17] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007).

[18] Floyd, S., and Henderson, T. The NewReno modification to TCP's fast recovery algorithm, 1999.

[19] Gont, F. Deprecation of ICMP source quench messages, 2012.

[20] 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).

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

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

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

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

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

[26] Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., and Raghavan, B. TCP fast open. In *CoNext* (2011).

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

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

[29] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware datacenter TCP (D2TCP). In *SIGCOMM* (2012).

[30] 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).

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

[32] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. H. DeTail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM* (2012).