FastLane: An Agile Congestion Signaling Mechanism for Improving Datacenter Performance

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

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-113
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-113.html

May 20, 2013

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, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

FastLane: An Agile Congestion Signaling Mechanism for Improving Datacenter Performance

David Zats[∓], Anand P. Iyer[∓], Randy Katz[∓], Ion Stoica[∓], Amin Vahdat[◇]
[∓] University of California, Berkeley [◇] Google / University of California, San Diego

ABSTRACT

The drive towards richer, more interactive content places increasingly stringent latency requirements on datacenters. A critical component of meeting these is ensuring that the network responds agilely to congestion, bounding network latency and improving high-percentile flow completion times.

We propose a new approach to rapidly detecting and responding to congestion. We introduce FastLane, a congestion signaling mechanism that allows senders to respond more quickly. By delivering signals to senders with high probability and low latency, FastLane allows them to retransmit packets sooner, avoiding resource-wasting timeouts. It also enables senders to make more informed decisions by differentiating between out-of-order delivery and packet loss. We demonstrate through simulation and implementation that FastLane reduces high-percentile flow completion times by over 80% by effectively managing congestion hot-spots. These benefits come at minimal cost—FastLane consumes no more than 2% of bandwidth and 5% of buffers.

1. INTRODUCTION

Many protocols have been proposed to improve worst-case, high-percentile flow completion times in datacenters [7,8,20,27,28,30]. These solutions focus on improving completion times in the presence of challenging network conditions such as congestion and packet loss. All of them try to overcome the effects of a sender taking too long to learn of and respond to congestion. Given this commonality, can we address the root cause?

Traditionally, transport protocols employ host-based solutions to detect and respond to congestion and loss. Timeouts and the receipt of three duplicate acknowledgements are used as *indirect* indicators of packet drops [15]. In response, transports typically retransmit packets and reduce their overall sending rate (to mitigate congestion). But, the indirect nature of these indicators inhibits transports from taking these and other actions rapidly.

Recent proposals have acknowledged the benefits of having the network provide senders direct congestion notifications. DCTCP [7] and D2TCP [27] have demonstrated that ECN [14] can be effective in improving high-percentile completion times. Switches set a flag in passing packets during congestion. Once the packet arrives at the receiver, it is echoed back to the sender. Upon receiving the echoed packet, the sender learns of the congestion in the network and throttles its rate. While helpful, ECN takes a long time to propagate to the receiver, to be echoed back, and to finally arrive at the sender.

We argue that an *agile congestion signaling mechanism* is the key to improving high-percentile performance in datacenter environments. Our proposal is based on the observation that *the fastest possible signaling must originate at the congested switch and flow directly back to the source.* To ensure low-latency and high probability of delivery, we protect congestion signals, providing them priority access to buffers and transmission. These signals provide transport protocols a direct and rapid indicator of congestion, allowing them to be more agile.

One potential concern with this scheme is whether the congestion signals sent by switches will exacerbate the congestion in the network on the reverse path. Fortunately, as we will show in this paper this is not the case. Indeed, in a wide variety of scenarios the bandwidth overhead due to congestion notifications is only a few percent (typically less than 2%). Furthermore, recent measurements have shown that most of the links in a datacenter are not congested [12]: edge links are rarely congested, and no more than 25% of the core links are "highly" utilized at any point in time.

To instantiate our scheme, we need to address two key challenges. First, we must determine what information to include in the congestion signal. This information should be (i) small enough to minimize resource usage, (ii) descriptive enough to provide the sender sufficient information, and (iii) simple enough so switches can generate signals quickly. Second, to succeed where prior proposals have failed (i.e., ICMP Source Quench), we must also ensure that congestion signals do not consume too many resources under any circumstances.

Addressing these challenges leads to a solution with many important benefits. First, traditional acknowledgements are no longer needed to indicate congestion. As such, we can reduce their rate significantly, as they are

now only used to implement flow control. This reduction is so large that it often leads to an overall drop in network load, offsetting the signaling overhead. Second, improving congestion response times allows us to safely start with a larger initial window, dramatically reducing the completion time of latency-critical short flows. Finally, congestion signals allow transport protocols to differentiate between out-of-order delivery and packet loss, which improves their ability to leverage multiple paths.

In this paper, we introduce FastLane, a lightweight, transport-agnostic congestion signaling mechanism that realizes these benefits. In doing so, we make the following contributions:

1. We present the design and implementation of FastLane. To underline the benefits of our proposal, we demonstrate how TCP can be extended to take advantage of it.

2. We evaluate our proposal in a number of scenarios using testbed experiments and simulations. Results from our evaluation indicate that FastLane can achieve over 80% reduction in 99.9^{th}%-ile flow completion time.

3. We analyze the overhead of FastLane and show that the worst case bound is small. Further, we experimentally show that the improvements achieved by FastLane remain even when we cap the bandwidth and buffers used by congestion signals to 2% and 5%, respectively.

The remainder of this paper is structured as follows. In the following section, we demonstrate the need for a congestion signaling mechanism, describing how it would allow transport protocols to improve their performance. In Section 3, we describe the mechanisms employed by FastLane. The details of our implementation are described in Section 4. We evaluate FastLane and report both implementation and simulation results in Section 5. We discuss the generality of FastLane in Section 6. We contrast our approach with prior work in Section 7 and conclude in Section 8.

2. THE NEED FOR A CONGESTION SIGNAL

In this section, we begin by describing the high-percentile performance requirements imposed by datacenter applications. Recent transport protocols have proposed increasingly complex actions to meet these requirements. We analyze how obtaining congestion signals from the network would help them be more effective. In the context of these actions, we describe the value of obtaining congestion signals in a timely manner and the mechanisms necessary to do so. We conclude by discussing what information the signal should provide. This decision has far-reaching consequences. Different approaches have different restrictions on how frequently the sender can be informed and provide transport varying amounts of information.

2.1 Performance Requirements

Datacenter networks are expected to meet strict performance requirements. Hundreds of intra-datacenter flows may be required to construct a single web-page [24]. The worst-case performance is critically important as workflows (i.e., partition-aggregate) can only complete when the last flow has arrived.

To make matters worse, these flows are typically small. Measurements from Microsoft's production datacenters indicate that latency-sensitive flows typically range from 2-20 KB in length [7]. As discussed in DeTail [30], such small flows pose extra challenges for traditional transport protocols as they do not have sufficient information (i.e., duplicate acknowledgements) to recover quickly from packet loss.

2.2 The Utility of Additional Information

To address these problems, recent efforts have proposed giving protocols the following capabilities:

- Reducing background flow transmission rates to prevent them from utilizing precious switch buffers (DCTCP, HULL, D2TCP) [7, 8, 27].
- Moving traffic away from congested links to those that are free (MPTCP, Hedera) [6, 26].
- Quenching a small number of flows so others can complete on time (D^3, PDQ) [20, 28].

All of these would benefit from receiving congestion signals from the network. Background flow transmission rates could be dropped and raised much more quickly in response to congestion events. This would further decrease the timeouts experienced by short, latency-sensitive flows when they are present while allowing background flows to consume more bandwidth when absent. Similarly these signals would allow traffic to be moved more quickly from congested links to lightly utilized ones. Finally they could make flow quenching operations simpler and more effective by letting transports know where congestion is being experienced and what flows are being affected.

2.3 The Value of Timely Delivery

The transport actions described would improve if the congestion signal arrived sooner. As shown in Figure 1, congestion signals are traditionally forwarded to the receiver before being echoed back to the sender. This is true for both indirect signals as well as direct ones (i.e., ECN). They incur many unnecessary network delays

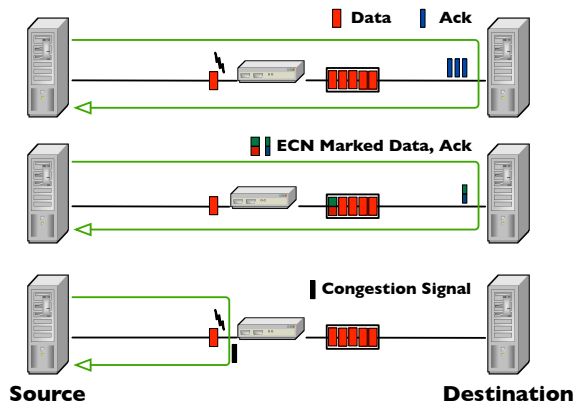


Figure 1: Different ways of signaling, with each incurring varying delays for the source to detect congestion. The *fastest* possible way is for the switch to signal the source directly.

(e.g. queueing, transmission, processing) along the path to the receiver as well as processing delays at the receiver itself. Signals then travel the extended path back to the sender. Not only are these delays large, but they are also highly variable.

In Figure 1, we see that FastLane significantly reduces delays by having switches send signals directly to the sources. To minimize queueing delays, signals also receive the highest priority. As a result, they arrive at senders as quickly as possible.

2.4 Congestion Signal Alternatives

Now that we have described the value of providing low-latency congestion signals to the transport layer, we discuss what information they should contain. Based on the relative strengths and weaknesses of various options, it becomes apparent which approach leads to high-percentile latency improvements.

2.4.1 *Periodic Congestion Summaries*

Switches already collect congestion statistics via SNMP counters. These could be used to send periodic congestion summaries to end-hosts.

This approach would be of limited use for achieving high-percentile performance for latency-sensitive flows. The workflows (i.e., partition-aggregate) commonly used in production datacenters can cause flash congestion [7, 30]. This approach is unlikely to inform sources in a timely manner. Additionally, it requires one of two unfavorable options: (i) switches can either maintain flow state to know to which sources to send notifications, or (ii) they can broadcast this information to all the end-hosts in the datacenter.

2.4.2 *Rate Reduction Notifications*

As previously proposed by ICMP Source Quench [17], switches can transmit rate reduction notifications. Since these messages are sent in response to arriving packets, they have the advantage of not requiring switches to maintain flow state. But these notifications do not provide sufficient information for senders to make more informed decisions.

Ambiguity arises because senders do not know which, if any, of a flow's packets have been dropped. This precludes transports from taking early actions such as retransmitting dropped packets on another path. Instead, transports obtaining rate reduction notifications must continue to rely on traditional, indirect indicators, to make these decisions.

2.4.3 *Dropped Packet Notifications*

It would be best for switches to transmit dropped packet notifications to senders for every packet loss. Senders could then make any of the decisions discussed earlier. They could decide how much to reduce the rate of the transmission by based on the priority/deadline of the flow. If the flow cannot meet its deadline, the sender could abandon it in hopes that it will help others complete on time. Finally, a sender could determine that a certain path is experiencing far too much congestion and retransmit the packet on another one. Congestion signals do not mandate any of these specific actions. These decisions can by made by senders, depending on the transport protocol used and the workload it is optimized for.

However, dropped packet notifications face the following challenges:

- The signal should be delivered to the sender with *high-probability* and *low-latency* so that transport can react *rapidly*.
- The mechanism needs be *low-overhead*, both in terms of signal generation and the load placed on the network.
- Finally, the information conveyed should require only *simple processing* by the sender. The network stack should be able to digest and process this information quickly to enable fast turnarounds.

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

3. FASTLANE

In the previous section, we discussed why drop notifications would help transport protocols reduce high-percentile flow completion times. Here we begin by providing an overview of the congestion notifications provided by FastLane and the transport response.

Later, we describe how we address the key challenges with FastLane. We present an approach that ensures

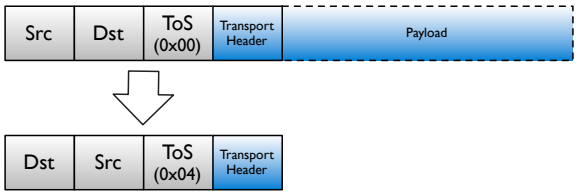


Figure 2: The notification is created by a simple transformation of the to-be dropped packet: flip source and destination, set a TOS bit and truncate at the transport header.

control notifications do not consume too many resources. We then show how the switch can perform simple operations to transmit these packets at line rate. We conclude by describing how transports can take advantage of FastLane by supporting multiple paths and reducing the number of acknowledgments.

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 buffers. At this point, the switch must start dropping packets (either arriving ones or ones already enqueued). This is where FastLane kicks in. For every dropped packet, it sends a notification back to the source, informing it which packet was lost.

A key design point with FastLane is the contents of this notification. One option would be to simply include the source and destination ports at the beginning of the transport header. This would informed the source as to which flow was contributing to congestion. However, it would not informed the source as to which packet had actually been dropped. As described later in this section, sources need to know the specific packet dropped to provide multipath support and to reduce the number of acknowledgements. So we opted to include the entire transport header as depicted in Figure 2.

Transports must also be able to differentiate notifications form other packets. One approach would be to set a flag in the transport header. But this would required the switch to have transport-specific knowledge. Instead, FastLane sets a TOS bit in the IP header (in a manner similar to ECN) that is then used by the source to identify notifications.

Finally, transports must know the length of the packet that was dropped to be able to reconstruct it. There are two approaches to addressing this problem: (i) the total length in the notification's IP header can be set to that of the dropped packet or (ii) transports can add a new header option to every packet that contains this value. In either case, the value would be echoed back in the notification.

Once the source receives and identifies the notification, it must respond appropriately, throttling its rate and retransmitting the packet. The details of source reaction are transport-specific, with different transports responding differently. We provide an example of how a TCP NewReno source would react in Algorithm 1. TCP first checks to see whether the notification is for a data packet. As shown in Line 5, this is simply performed by subtracting the TCP and IP header lengths from the total length (stored in the IP header in this example) to get the data length. If the data length is greater than zero, the dropped packet contained data.

For data packets, we can use the sequence number (in the TCP header) and the data length to reconstruct the packet (Line 7). The reconstructed packet is then transmitted. Since a lost data packet is likely an indicator of congestion, we also check to see if the congestion window should be cut in half (as is traditionally done). As shown in Line 8, we cut the window in half for the first congestion notification received within the window. This ensures that the window is not cut in half many times for a single congestion event.

The notification may have not been for a data packet. Other packets have two distinct properties that require

Algorithm 1 End-host response

```
1:  w ← 0                                              ▷ End of window
2:  r ← false                                          ▷ In recovery
3:  i ← 0                                              ▷ Window inflation
4:  function OnRxNotify(c)
5:      d ← totlen(c) − (len(iphdr) + len(tcphdr))
6:      if d > 0 then
7:          senddata(seqno(c), d)                      ▷ Resend data
8:          if r = false then                          ▷ Reduce window
9:              r ← true
10:             h ← highestsent
11:             cwnd ← cwnd/2
12:             ssthresh ← cwnd
13:         end if
14:     else
15:         if ackno(c) ≥ lastacktx then
16:             sendempty(flags(c))                    ▷ Send non-data
17:         end if
18:     end if
19: end function
20: function OnAckRx(a)
21:     if r = true && ackno(a) ≥ w then
22:         r ← false
23:     end if
24:     if newack(a) then
25:         i ← outoforder(a)
26:     else
27:         i ← max(outoforder(a), i)
28:     end if
29: end function
```

special consideration. They may contain flags (e.g., SYN or FIN) and the cumulative nature of acknowledgments means they may not have to be retransmitted. Line 15 shows how we address these issues. We first check to see if we have already transmitted a packet acknowledging a higher sequence number. If so, we do not send another packet. Otherwise, we reconstruct the packet, copying the flags from the congestion notification.

3.2 Controlling Resource Consumption

A common concern when sending notifications in response to congestion is that the notifications and/or response to them not exacerbate the congestion. We first provide some intuition about the worst-case overhead incurred when sending notifications. Then we describe how it can be effectively reduced. We conclude by providing example modifications to TCP intended to prevent senders from responding too aggressively.

To obtain some intuition about the notification overhead, lets begin by assuming the packets in a datacenter are dropped according to a probability of p, (irrespective of size). Then the overhead can be approximated by the following equation:

$$\frac{n_{data} + n_{ack}}{data + ack + n_{data} + n_{ack}} \tag{1}$$

Where *data* and *ack* are the loads due to data and acknowledgments, respectively. n_{data} and n_{ack} are the loads due to the notifications sent in response to data and acknowledgment drops, respectively. Assuming a data packet size of 1400 B and an acknowledgment size of 64 B, this equation becomes:

$$\frac{64p + 64(1-p)p}{1400 + 64(1-p) + 64p + 64(1-p)p} \tag{2}$$

Which simplifies to:

$$\frac{64p(2-p)}{1464 + 64(1-p)p} \tag{3}$$

Figure 3 depicts the overhead, plotted as a function of p. We see that at the maximal drop rate of 100%, the overhead due to notifications is under 4.5%. The maximal value depends heavily on the size of data packets. As described in a recent measurement study, packet sizes in datacenters are typically bimodal, with data packets clustering around 1400 B [12], providing the expectation of low overhead.

This equation suggests that notifications will not significantly contribute to congestion, especially given that they are transmitted in the direction away from the congested link. To ensure that edge cases do not arise, we also propose adding a strict 2% cap on the bandwidth that notifications may use. We chose this value based on a sensitivity analysis performed in Section 5.

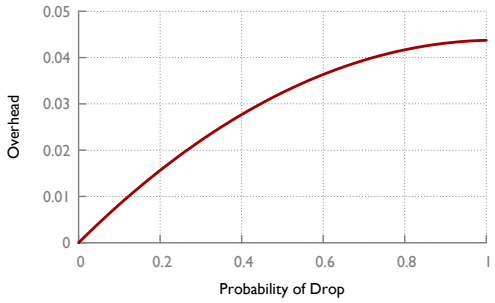


Figure 3: Theoretical overhead of FastLane as a function of drop probability. The worst-case is 4.5%, but our cap restricts it to 2%.

We expect that in normal network operation, this cap will never be reached. A 2% cap implies that over 25% of packets are being dropped. If the network approaches such a congested state, performance will be so degraded that it is likely best to drop notifications and have the sources timeout.

One of the ways to prevent reaching this cap is to ensure the transport protocol does not respond too aggressively to notifications. Choosing the aggressiveness of the response represents a tradeoff. On the one hand, transport protocols should be sufficiently aggressive as to use available resources. This is especially true in multipath environments where many paths may exist for packets to take. On the other hand, transport protocols should not be so aggressive as to push the network into an unstable state.

Our modifications to TCP strike a balance between these two extremes. We retransmit packets for which notifications have been received instantaneously. But, we do not send any new packets until the complete window has been acknowledged. This represents an appropriate tradeoff as the availability of explicit drop notifications means that we do not have to keep sending new data to prevent a timeout.

3.3 Efficient Notification Transmission

To ensure that we can send notifications for every drop, a key requirement is for switches to be able to send them at line rate. To achieve this goal, we rely on well-specified packet-manipulation operations that can be performed in the data plane.

As the packet must be sent back to the source, we need to swap the source and destination IP addresses. It may also seem that transforming the packet requires modifying the transport headers (i.e., swapping the source and destination ports), but this is actually unnecessary. By leaving this operation to the end-host, FastLane remains transport-agnostic. However, as mentioned earlier, we must also set a flag in the IP header, indicating that the packet is a notification.

As these operations are light-weight, switches should have no problem performing them at line-rate. While they do require recomputing the IP checksum, it must already be updated as the TTL field is typically decremented at each hop. These operations may violate the transport-layer checksum (if it uses a pseudo-header as TCP does). Requiring the switch to recalculate the checksum is onerous, so we opted to have end-hosts ignore it for notifications. We argue that this decision is safe as corrupted packets are still likely to be detected by link-layer checksums. In the low probability condition where the corruption goes undetected, a packet may be unnecessarily retransmitted.

Once we have transformed the packet, we must decide on which port to transmit the notification. As forwarding lookups are one of the most time-consuming operations in processing a packet, we opted to have the switch forward the packet to the port on which it arrived.

3.4 TCP Optimization: Supporting Multiple Paths

As mentioned earlier, one of the advantages of FastLane is that it allows transports to differentiate between out-of-order delivery and losses. Here we show the additional steps required for TCP to leverage the multiple paths commonly available in datacenters.

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).

A question here is how much should the congestion window be increased by. As shown in Line 24 of Algorithm 1, the answer depends on the type of acknowledgement. If the acknowledgement is a new one (i.e., it cumulatively acknowledges new segments), 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.

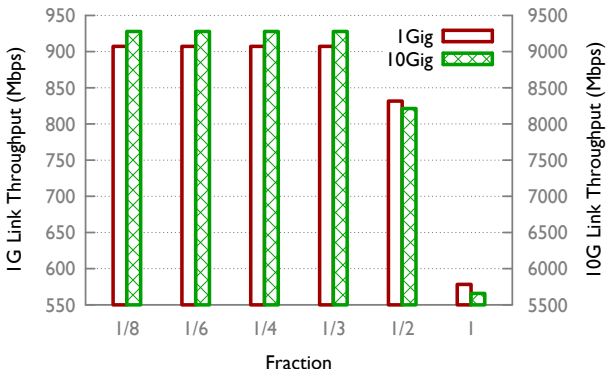


Figure 4: Performance of FastLane when the ACK is generated at the receipt of different fractions of the window. We see that it is possible to reduce ACKs to only once every $1/3^{rd}$ of a window without significantly impacting the throughput.

As mentioned earlier, a concern with more aggressively using available resources is that the transport protocol may overdrive the network. We balance these competing desires by not inflating the window whenever the connection is in slow start.

3.5 TCP Optimization: Reducing Acknowledgments

While cumulative acknowledgments pose the limitation described earlier, they are also beneficial. They allow us to reduce the number of of acknowledgements simply by sending fewer of them. Because of FastLane, acknowledgements are now only used to move or grow the congestion window. As they are not used to respond to congestion, we can reduce the number of them without increasing congestion response-times.

Sending fewer acknowledgements does have potential downsides. If notifications are dropped and the sender times out, packets may be retransmitted unnecessarily. Also, forward and reverse path asymmetries may lead to drops in throughput. In scenarios where notification drops are exceedingly rare and these throughput drops are insignificant, sending fewer acknowledgements may be an appropriate tradeoff to make.

To determine the fraction of the window that must be acknowledged, we setup a simple simulation on a 16-server FatTree topology with 10Gbps links. We run an uncontended flow between two servers in different pods, increasing the fraction of the window that is received before an acknowledgment is generated. Since we start with a larger initial window size, we assume that the window is at least one bandwidth-delay product and our flows start outside of slow start (as proposed by pFabric [9]). We also keep the congestion window constant through this simulation to ensure that short flows would not be harmed by this approach.

Figure 4 demonstrates the results for a 1 MB flow. We see that throughput drops off when the fraction of the window is larger than 1/3. To ensure the generality of our approach, we also repeated this simulation with the 16-server FatTree having 1Gbps links and a much larger bandwidth-delay product. As shown in the figure, we obtained similar results.

The value of 1/3 may come as a surprise. It is primarily because acknowledgments are smaller than data packets and hence require reduced transmission delays. Thus the reverse path takes less time.

For all of the experiments in Section 5, when using FastLane, TCP will only send an acknowledgement when 1/3 of the window has been received. Note that the packets received do not have to be in-order. To enable this functionality, the destination must know when it has received 1/3 of the sender's window. We address this problem by introducing a new TCP option that informs the destination how many packets it should receive before sending an acknowledgment. When calculating this value, we also compare it to the size of source's transmit buffer and take the minimum of the two. Towards the end of a flow, a source may have fewer than 1/3 of a window of data remaining. We want to appropriately handle this case to ensure that we do not wait needlessly for a delayed acknowledgment.

4. IMPLEMENTATION

Having described the congestion notification mechanism provided by FastLane and addressed its challenges, we now discuss the details of our implementation.

We implemented all of our proposed changes for TCP to take advantage of FastLane in Linux kernel version 3.2. To allow the end-host to process and respond to notifications, we modified the operating system's TCP/IP stack. The receiving hook at layer 4 (`tcp_v4_do_recv()`) uses a TOS bit to identify a notification and processes it in a separate routine.

The processing begins with retrieving the socket context corresponding to the received notification. We use the processing described in Section 3 to determine the type of packet that has been dropped. For data packets, we walk through the `write_queue` for the socket until we find the corresponding packet. For other packets, we simply copy over the flags from the notification and retransmit based on TCP's socket context. In either case, we call the routine `tcp_transmit_skb()` to send the packet. We deliberately avoid using the retransmission routine in the kernel (`tcp_retransmit_skb()`) so as to avoid book-keeping these retransmissions.

As discussed earlier, we use two 4-byte options in the TCP header to help inflate the congestion window at the sender and reduce the number of acknowledgments generated by the receiver. The first option informs the sender of the number of out-of-order bytes received, and is computed at the receiver when sending an acknowledgment. Fortunately, the kernel stores the out-of-order packets for a flow in an out-of-order queue in the TCP socket. To compute the number of out-of-order bytes, we iterate over this queue, adding the bytes in each packet buffered. The second option is used by the sender to inform the receiver when to send an `ACK`. To calculate this option, we require the state of the sender's current transmission window. All packets carry these options, except those for the initial connection establishment. Adding options required modifying the expected TCP header length, which is set in different places for the sender and the receiver. The receiver sets the header length it expects once it receives a SYN and forks a new socket; the sender sets it upon the receipt of the SYN-ACK.

Finally, we modified a number of default values in the kernel (e.g., the initial timeout value, maximum and minimum delayed ack timer values, etc). Our modifications, except the changes to the defaults are easily disabled using a TCP socket-level option. The defaults are maintained for standard TCP processing to ensure fairness.

5. EVALUATION

We now present the 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. In doing so, we strive to capture the efficacy of FastLane in meeting the goals described in Section 2.

We report on our experiments from a 16-server FatTree topology running on Emulab [4]. We use Click to provide the switch functionality [23]. To achieve a realistic oversubscription factor of 4 while maintaining multiple paths, we rate limit the links to speeds lower than those commonly observed in datacenters. Links between the aggregate and core switches run at 250 Mbps and the links between top-of-rack and aggregate switches run at 500 Mbps. Servers are connected to top-of-rack switches by Gigabit links. Given the reduced link speeds, we appropriately scale buffers to 32KB per port (shared across all ports). We use these measurements to validate our NS-3 [5] based simulator. Finally, we present our results using the validated simulator, for a larger-scale, 128-server FatTree topology. This topology uses 10 Gig links, maintains an oversubscription factor of 4 and uses 128KB buffers per port (as found in current switches [2]).

Our overall experimental strategy is as follows: we compare long-tail transport performance to that achieved when the transport protocol is assisted by FastLane. As FastLane allows the transport protocol to more effectively use multiple paths, we have switches use packet scatter when FastLane is enabled. We report 99.9^{th} percentile flow completion times for frontend/backend and

all-to-all communication patterns.

All experiments use request-response workflows. Requests are initiated by a 10 byte packet to a server. We classify these requests into two categories: high-priority and low-priority. A high priority request results in a response that can be a flow of size 2, 4, 8, 16, or 32 KB, with equal probability. This spans the range of high-priority flows typically observed in datacenters [7]. A low priority request generates a 1 MB background flow. We ensure the continuous presence of background flows in all the scenarios by engaging every server in one of these flows on average.

We evaluate how both TCP-Cubic [18] and TCP-NewReno [15] perform in these environments. TCP-Cubic is the most-recent, commonly-used transport available in Linux while TCP-NewReno is well-established, with a well-tested simulation model. In all cases, high priority requests are strictly prioritized over low-priority background ones. As discussed by HULL (an extension to DCTCP) [7,8], this baseline provides comparable performance to recently proposed latency-optimized transports.

We use the same workload and topology in the implementation to validate our simulator. Our simulations then explore a broader set of environments to understand the limits of FastLane, the effects of bandwidth capping, and the impact of signal latency. The topology and these scenarios sometimes dictate a different choice of parameters than supported in the implementation testbed (discussed later).

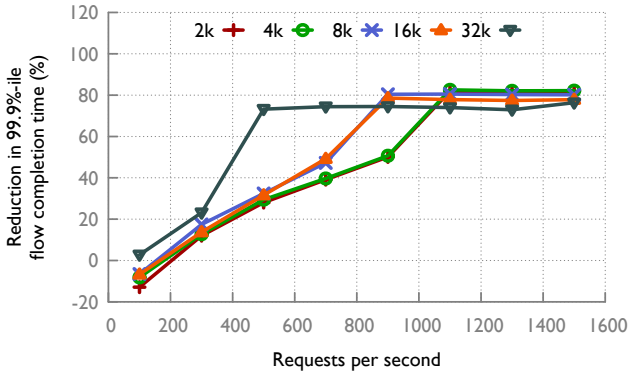
In this section, we begin by presenting the implementation results for a balanced frontend/backend workload in the presence and absence of a failed link. Section 5.2 validates the simulator by running the same topology and workload in NS-3 and comparing the results.

We then present our simulation results for larger topologies in Section 5.3. These explore performance under an all-to-all communication pattern, as well as a more extreme 3:1 frontend to backend one. Once we have evaluated failure-free performance, we return to the all-to-all communication pattern and explore the impact of failures. For all of these cases, we report the overhead incurred by FastLane. We conclude our simulation by examining FastLane's sensitivity to various bandwidth caps and assess the impact of timeliness of the congestion signal by investigating the effect of delay on flow completion times.

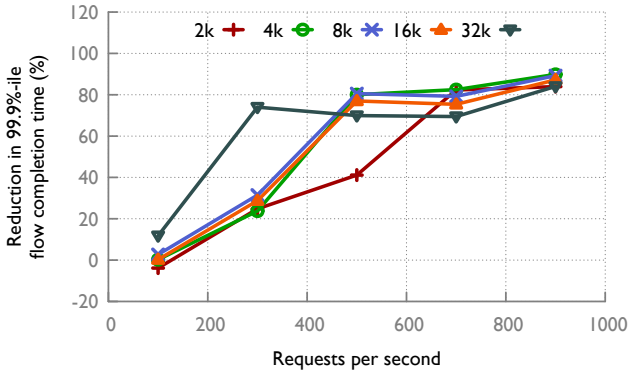
5.1 Experiments from the Implementation

In our implementation testbed, we classify half the servers as frontend and half as backend. Frontends generate requests to randomly chosen backends according to a Poission distribution. Given the topology, we set TCP timeouts to 10ms.

To evaluate the performance of short-flows, we run 5



(a) No link failures



(b) Core-to-agg link missing. We show utilizations only up to 900 requests per second since TCP fails to complete beyond this (FastLane completes successfully for all loads).

Figure 5: FastLane performs well under different loads and flow sizes, with almost an order of magnitude improvement over TCP-Cubic in some cases.

minute long experiments. During each of these experiments, frontends generate requests at one of the following rates per second—100, 300, 500, 700, 900, 1100, 1300 and 1500. Thus, each 5 minute run results in a total of (`requests-per-sec` × 8 × 5 × 60) flows overall. Since each request results in a random high priority response from 5 different flow sizes, the total flows per flow size is (`requests-per-sec` × 8 × 60). The idea here is that these request rates result in different overall utilization levels. In all these experiments, we also run 1MB background flows continuously (i.e., each frontend requests a 1MB flow back-to-back, each time randomly selecting a backend server).

Figure 5a shows the results of this experiment. The relative flow completion time improves for flows in all cases, except when the rate is 100 requests per second (this corresponds to a very low utilization by high-priority traffic [≈ 4%]). We do not believe this is fundamental to our scheme, but is an artifact of our testbed—the NICs in our end-hosts are unable to prioritize flows. Thus background flows end up interfering with the foreground flows under light utilizations. This is easily rectified in

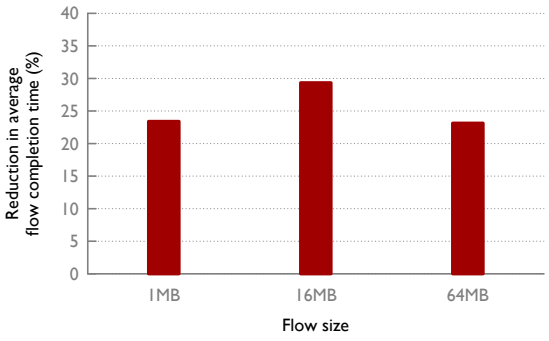


Figure 6: Long flows also see an improvement with FastLane. Although not as much as short flows (which is the focus of FastLane), the average long flow experiences a flow completion time reduction of 25%.

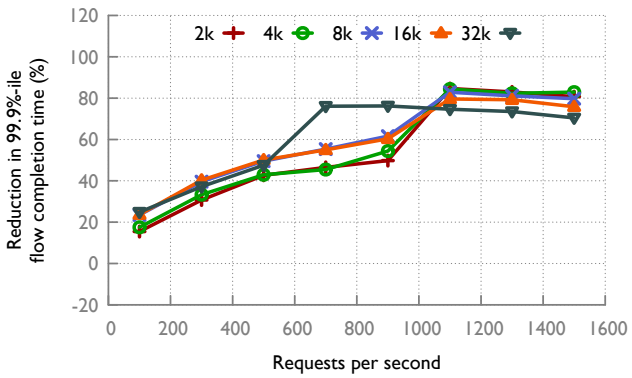


Figure 7: Simulation results for the same setting as those for Figure 5a. The mismatch at low utilization is due to caveats with the simulation environment (see Section 5.2)

real switches and newer NICs that can distinguish flow priorities correctly. We see increasing improvement with utilization and/or flow size, as we expect. While we do not show the improvements incurred by the 1MB background flow, their average flow completion time improves by approximately 37%; making FastLane improve the combined flow completion time even at the lowest load (100 requests per second).

Link failures are common in datacenter environments [16, 29]. Figure 5b presents the results when a core to agg link in the first pod is disconnected. Here, we see much higher benefits—in most cases, the completion time is cut down by more than half and in certain cases, we report improvements of almost a magnitude (because we report percentage reduction, the magnitude improvements are not immediately obvious from the figure. Hence we also report the ratio of completion times in Table 1 as a reference). Since TCP flows fail to complete at high utilizations due to extreme timeouts, we show only up to a rate of 900 requests per second. Due to the extreme amount of congestion in this experiment, the improvement in average flow completion time for the background flows is not as much as in the earlier case—but still they see a modest improvement of 26%.

Initially, it may seem like packet scattering used by FastLane is the main reason for these improvements in the presence of persistent hotspots. However, previous work shows that packet scatter performs poorly in the presence of failure [26]. Thus, rapid notifications are critical for improvement in these scenarios— notifications allow FastLane to prevent timeouts and quickly retransmit dropped packets even under the highest utilizations.

Datacenter protocols must also consider large flows. To evaluate how FastLane performs in the presence of longer flows, we run an experiment. In this experiment, frontends generate back-to-back requests for one of the following three background flows with equal probability: 1MB, 16MB or 64MB. To account for foreground flows, each frontend also generates foreground requests at a rate of 100 requests per second. Figure 6 presents the reduction in average flow completion times compared to TCP. We notice improvements here too, although not as high with short-flows. This is obvious, since long flows present a better setting for TCP compared to short flows.

5.2 Simulator Validation

We compare our implementation and simulation using the same settings as in 5.1. Figure 7 depicts the results and should be compared to Figure 5a.

We see a good match between simulation and implementation at medium to high utilization levels. At low utilization, the results do not match as well. The reason is explained by our inaccurate model of end-host behavior. At low utilization, the end-host dominates. That is, notifications take time to process; this is not captured in the simulation.

5.3 Experiments from the Simulator

Having validated the simulation environment, we now turn our attention to simulation results.

Since it is possible to control the environment much more finely, our simulations assume that high-priority requests are more bursty. They are generated according to a log-normal distribution with $\sigma = 2$. Given the use of higher-speed links, we set transport timeouts to 1ms. Unless stated otherwise, the network bandwidth and buffer size used by FastLane is capped to 2% and 5% respectively.

Since our simulation environment gives us the flexibility to test larger topologies, we present the results for a larger number of scenarios:

5.3.1 *Normal behavior*

We first evaluate how FastLane performs when the

req/s	min	mean	max
100	0.89	0.94	1.03
300	1.14	1.19	1.30
500	1.39	1.90	3.73
700	1.64	2.21	3.91
900	2.00	3.55	5.11
1100	3.85	4.95	5.74
1300	3.69	4.86	5.58
1500	4.23	4.97	5.61

no link failure

req/s	min	mean	max
100	0.96	1.03	1.13
300	1.31	1.87	3.85
500	1.70	3.90	5.14
700	3.27	4.70	5.71
900	6.21	7.89	9.86
1100	-	-	-
1300	-	-	-
1500	-	-	-

core-agg link missing

(a) by rate

size	min	mean	max
2k	0.89	2.94	5.53
4k	0.92	3.01	5.74
8k	0.94	3.23	5.13
16k	0.94	2.96	4.67
32k	1.03	3.21	4.23

no link failure

size	min	mean	max
2k	0.96	3.17	6.21
4k	1.00	4.58	9.86
8k	1.03	4.35	9.30
16k	1.00	3.72	7.79
32k	1.13	3.57	6.26

core-agg link missing

(b) by flow size

Table 1: Relative performance of FastLane over TCP-Cubic at indicated request rates and flow sizes. The numbers indicate the ratio of 99.9^{th}%-ile completion time for TCP to that for FastLane. We see approximately an order of magnitude improvement in link failure cases.

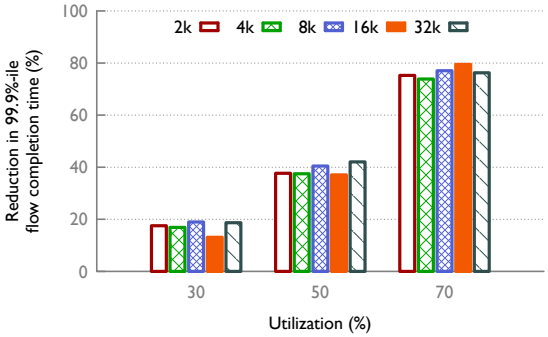


Figure 8: All-to-all scenario where every server generates random request to every other server in a 128-server FatTree topology.

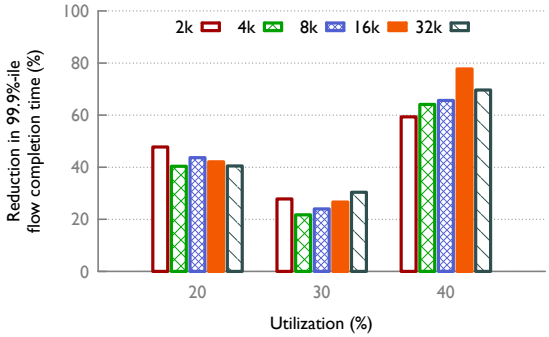


Figure 9: Frontend-Backend scenario where 75% of the servers act as frontends to the rest (backends). The utilization is the average core utilization, the backends experience twice the indicated load.

network operates smoothly, without hotspots due to link failures. We evaluate two settings:

All-to-all. In this experiment, we make every server randomly query every other server with equal probability. Hence, all 128 servers are engaged in requests.

The results are shown in Figure 8. As we expect, FastLane provides greater improvement at higher utilizations, achieving up to 80% reduction in flow completion time. Flow size does not significantly impact the improvements attained.

Frontend-Backend. In this experiment, we evaluate how FastLane performs when all the traffic is sent to or received from one pod in a FatTree, creating a hotspot. For this, we classify 75% of the servers (96) as frontend and the rest (32) as backend. Due to the high concentration of traffic at the backend, we reduce the average utilization. Fig 9 depicts the results.

Note that the utilization in this figure indicates the *average* core utilization. The utilization on the core links connected to backend servers is two times higher than the average. FastLane performs well in this case too, achieving significant benefits for all utilizations.

5.3.2 *Performance in the presence of failures*

As mentioned earlier, link failures are common in datacenters [26, 29]. To evaluate how FastLane performs in their presence, we repeat the all-to-all experiment described above, but induce a core-to-agg link failure. The removed link causes a hot-spot around it with heavy congestion.

Fig 10 shows the results of this experiment. We notice significant improvement compared to TCP even at low utilizations—for instance, at 30% utilization, TCP's 99.9^{th} percentile completion time for the 2k flow is 1.02ms compared to FastLane's 0.17ms. The improvements at higher utilizations are in the range 30-100X. This is because TCP experiences extreme timeouts due to congestion and is unable to recover. FastLane, on the other hand is able to continue because of its ability to quickly identify and react to drops. In these scenarios, quick detection and reaction are essential to avoid

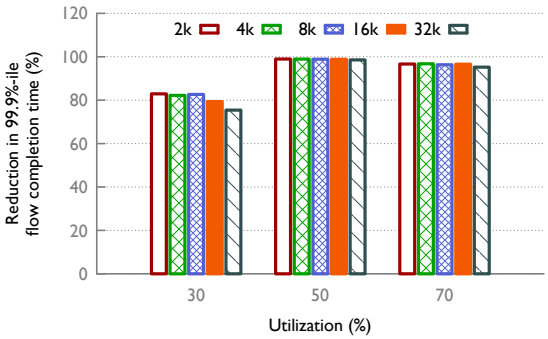


Figure 10: All-to-all scenario with a missing core-to-agg link. The ability of FastLane to quickly detect drops is reflected in its performance.

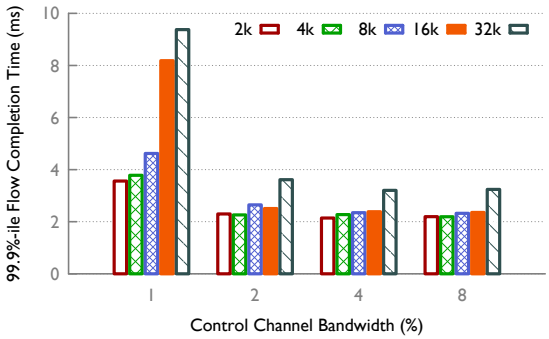


Figure 11: FastLane's perforance under varying resource caps. A 2% bandwidth cap is sufficient for reasonable performance even under extreme congestions. For reference, TCP's numbers for 2k, 4k, 8k, 16k and 32k flows are 68, 70, 72, 73 and 76 ms respectively.

congestion collapse.

5.3.3 *Overhead Analysis*

An important question that still remains unanswered is the overhead for these improvements. Recall that in Section 3, we showed that theoretically, the overhead of control notifications does not go beyond 4.5% (and above 2% when capped). But how much is it in practice? We answer this question now.

To show the overhead, we present the number of control packets generated in each of the simulation experiments discussed earlier in Table 2. Each row in the table represents the utilization; for TCP we present the number of ACKs generated during the experiment, while for FastLane we present the number of ACKs and notifications separately.

We notice that FastLane generates a fair number of notifications during the heavy congestion incurred by link failures. However, FastLane can mask this load by using intelligence at the transport layer. Specifically, FastLane is able to reduce the number of ACKs significantly, thus reducing the total number of packets in all cases.

It is easy to understand why the number of acknowledgments is high at low utilizations. Recall that continuous background flows are present in all experiments. More of these flows complete when there are fewer foreground flows.

5.3.4 *Sensitivity to the Cap*

In all of our simulations, we restrict the bandwidth and buffers allocated to FastLane to 2% and 5%, respectively. We impose this restriction as a safeguard. An interesting question is how does FastLane perform under varying amounts of signaling resource availability.

To answer this, we consider the worst-case scenario. Intuitively, a congestion signaling mechanism's demand for resources is the highest during extreme congestion periods, such as permanent hot-spots due to link failures. Hence, we repeat the failed link experiment presented in Section 5.3.2. We ran this experiment at the highest average utilization (70%) four times, each time capping the network bandwidth available for notifications to one of 1%, 2%, 4% or 8%. We also appropriately scaled the buffers allocated. Figure 11 depicts the 99.9^{th} percentile flow completion time for FastLane in each of these cases.

We see a sharp drop in the flow completion time when the cap is 2% compared to a 1% cap. We also notice that capping the bandwidth to values above 2% results in similar performance. Recall that this is a scenario of congestion collapse. In comparison, TCP's 99.9^{th} percentile flow completion time is over 70 ms! Even at these extreme conditions and under stringent resource caps, FastLane performs well.

5.3.5 *Effect of delaying notification*

Finally, we return to the need for low-latency delivery of notifications. Again, we consider the worst-case scenario by repeating the all to all experiment with a core-to-agg link missing. We run this experiment multiple times. For each, notifications incur a different artificially injected delay at the switch generating them. The results are shown in Figure 12.

In the figure, the X-axis is the delay injected at the switch before sending out the notification. Here we notice that slight delays incur no penalty. But delays $500\mu s$ and greater result in a sharp degradation of performance. Delays greater than $500\mu s$ are likely to occur when using traditional signaling mechanisms (i.e., ECN). Marked packets may wait up to 1 ms in every congested queue (recall that switches have shared-memory queues that can grow to be quite large). Once delivered to the receiver, the signal must be echoed back. The process of echoing the signal will also take a significant amount of time due to operating system processing delays. This

	All-to-All			All-to-All w/ Missing Link			Frontend-Backend		
	TCP ACKs	FastLane ACKs	FastLane Notifications	TCP ACKs	FastLane ACKs	FastLane Notifications	TCP ACKs	FastLane ACKs	FastLane Notifications
Low	2285466	766968	79	1895906	747672	27136	5680823	1323168	3251
Medium	1803619	706438	384	1675683	754914	95870	3286658	919166	7529
High	1675764	713635	10436	1686677	932640	519416	1858443	733847	195271

Table 2: Total control packets generated during simulation experiments. Although notifications increase load for FastLane, it is able to mask this load by significantly reducing ACKs; hence resulting in an overall reduction in total packets. Thus, FastLane's improvements *do not* have to come at the cost of increased overheads.

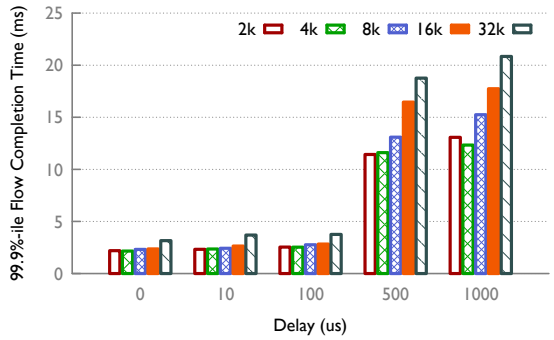


Figure 12: FastLane's performance under varying delays for sending notification. These results substantiate the need for the low-latency delivery of congestion notification signals.

validates our decision to have notifications sent directly back to the source with the highest priority possible.

6. DISCUSSION

In this paper, we focused on integrating FastLane into TCP. Many other protocols can exploit FastLane. Traditionally, transport protocols are not directly notified as to which packets have been dropped. Instead they must make educated guesses, based on timeouts and out-of-order packet arrival. An explicit drop notification sent by the switch directly to the source is far more precise and timely, greatly reducing completion times.

MPTCP [26] can be trivially extended to use this additional information. In response to a drop notification, MPTCP either places corresponding packets on other subflows or retransmits them on the original one as necessary. Even if MPTCP decides to retransmit packets on the original subflow, it will have received the notification and hence performed the retransmission sooner than would have been otherwise possible.

These extensions to MPTCP are likely to yield significant performance benefits, especially for short flows. MPTCP typically uses 4-8 subflows with an initial window size of 10 packets each. As a result, the first 40 - 80 packets (60 -120KB) of any flow are likely to take pre-determined paths, calculated by flow hashing. These packets will only be re-injected into other subflows if the receive window is not sufficiently large. Providing MPTCP precise drop notifications may allow it to overcome this limitation.

Other recent proposals also benefit. pFabric [9] performs well in the absence of failures. But pFabric's decision to forego retransmission on three duplicate acknowledgments means that it can perform poorly when link failures cause a permanent hotspot. A few packets from a window can traverse through a hot-spot and be dropped. A timeout occurs and the sender may unnecessarily retransmit large portions of its window. On the other hand, FastLane informs pFabric specifically which packets were dropped, avoiding this problem. This is particularly helpful when there are many concurrent flows with the same priority.

7. RELATED WORK

Here we first describe how FastLane compares with prior work in datacenter networks. Later we touch upon other areas, namely Internet Protocols and Wireless Networks, discussing each in turn.

7.1 Datacenter Networks

Researchers have proposed an extensive set of transport modifications for datacenter networks. They largely fall into two groups: (i) those using end-host solutions that avoid/minimize modifications to network elements and (ii) those requiring extensive protocol-specific changes to support explicit resource reservations. DCTCP, HULL, and D2TCP fall into the first category [7, 8, 27] while D^3 and PDQ fall into the second [20, 28]. FastLane focuses on improving congestion response times, thereby allowing host-based mechanisms to be more effective. Its transport-agnostic nature increases the chance for adoption.

Industry has adopted standardized Ethernet link-layer improvements, such as Quantized Congestion Notifications (802.1Qau [1]). Switches directly notify end-hosts about which flows are contributing to congestion. The decisions to use congestion notifications instead of drop notifications and to rate limit at the NIC forego many of the advantages of FastLane. Hardware resources are limited, so a set of unrelated flows may share a rate-

limiter. As a result, flows not contributing to congestion may be slowed down. The decision to avoid informing transport prevent many of the optimizations described in this paper.

[30] proposes to orchestrate the datacenter bridging protocols [3] into a stack, DeTail. While achieving high levels of performance, DeTail depends on end-hosts to respond to congestion notifications in a timely manner. If they do not, the network may experience head-of-line blocking, significantly degrading performance. Also, DeTail 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].

7.2 Internet Protocols

There is a rich history of transport protocols in the Internet [10,13,15,21]. Given the need to make minimal assumptions about underlying networks, these approaches primarily focus on learning of packet drops indirectly. As a result, they resort to making assumptions such as: *the packets of every flow will traverse a single path* in order to improve performance. These assumptions are inappropriate for datacenter networks where many paths exist between a source and destination.

There has been effort to directly provide transport protocols more information, notably the ICMP Source Quench message [17]. This approach failed primarily because the condition of the router sending the notification was poorly defined (i.e., it could send the notification early, before a packet was dropped). Also this approach was seen as unfair to TCP because other protocols could make the choice of ignoring these messages.

These problems have either been addressed by FastLane or are not a concern in datacenter networks. We have clearly specified the condition under which switches send notifications and the single administrative domain allows datacenter operators to control the types of transport-layer protocols used.

7.3 Wireless Networks

Improving transport protocols over wireless networks is a well explored topic in the literature, where the solutions try to provide reasoning for the loss—such as link-failures (e.g. [19]) or channel-induced losses (e.g. [11])—explicitly through the use of control packets so that the sender may react appropriately. Flush [22] and RCRT [25] use negative acknowledgments (NACK) for end-to-end loss recovery in the context of multihop wireless sensor networks.

However, the focus of these proposals is not optimizing flow completion times. They merely tried to *fix* the incorrect behavior in the transport protocol. In the face of actual congestion, the protocol behavior remains the same.

8. CONCLUSION

In this paper, we presented FastLane, an agile congestion signaling mechanism for improving high-percentile datacenter networking performance. The key motivation behind our proposal is the fact that the fastest possible signaling must be done by the congested switch to the source directly and with high probability. By doing so, FastLane tries to minimize the delay incurred by senders in detecting and responding to congestion.

We demonstrated the efficacy of our work by modifying TCP to take advantage of FastLane. The testbed experiments and simulations show that the rapid signaling mechanism helps achieve significant reduction in worst-case flow completion times, often over 80% and in some cases reaching an order of magnitude. We showed that the benefits do not have to come at a large cost—FastLane maintains large improvements even when capping the bandwidth and buffers to 2% and 5%, respectively. Further, we show how transport protocols can mask these already low overheads to achieve an *overall reduction* in the load.

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.

9. 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, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

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

[7] 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).

[8] 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).

[9] 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.

[10] Allman, M., Paxson, V., and Stevens, W. RFC 2581: TCP congestion control, 1999.

[11] Balakrishnan, H., Seshan, S., Amir, E., and Katz, R. Improving tcp/ip performance over wireless networld. In *Mobicom* (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] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. Tcp vegas: new techniques for congestion detection and avoidance. In *SIGCOMM* (1994).

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

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

[16] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 350–361.

[17] Gont, F. Deprecation of icmp source quench messages, 2012. http://tools.ietf.org/html/rfc6633.

[18] Ha, S., Rhee, I., and Xu, L. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev. 42* (July 2008).

[19] Holland, G., and Vaidya, N. Analysis of tcp performance over mobile ad hoc networks. In *Mobicom* (1999), ACM.

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

[21] Jacobson, V., and Braden, R. T. Tcp extensions for long-delay paths, 1988.

[22] Kim, S., Fonseca, R., Dutta, P., Tavakoli, A., Culler, D., Levis, P., Shenker, S., and Stoica, I. Flush: a reliable bulk transport protocol for multihop wireless networks. In *Sensys* (2007), ACM.

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

[24] 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).

[25] Paek, J., and Govindan, R. Rcrt: rate-controlled reliable transport for wireless sensor networks. In *Sensys* (2007), ACM.

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

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

[29] Wu, X., Turner, D., Chen, C.-C., Maltz, D. A., Yang, X., Yuan, L., and Zhang, M. Netpilot: automating datacenter network failure mitigation. 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. 419–430.

[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.