Towards a More Stable Network Infrastructure

Radhika Mittal

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-103
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-103.html

August 7, 2018

Towards a More Stable Network Infrastructure

By

Radhika Mittal

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Chair
Professor Sylvia Ratnasamy
Professor Dacher Keltner

Summer 2018

Towards a More Stable Network Infrastructure

Abstract

Towards a More Stable Network Infrastructure

by

Radhika Mittal

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

There have been many recent proposals to change the network infrastructure in order to meet different performance objectives. These changes are often difficult to deploy, either requiring specialized network switching hardware or greatly complicating network management. Rather than continuing to add new features to the network in an adhoc manner, we advocate a more principled approach for meeting different performance objectives, that leads to a more stable network infrastructure. This approach is based on the following two questions:

(1) Can we avoid making changes to the network infrastructure by finding solutions that only change the end-points? Here, we focus on congestion control for both wide-area and datacenter networks, showing how the end-points can be updated to achieve near-optimal performance using commodity switches, and on redesigning RDMA NICs to eliminate their reliance on the in-network mechanism for loss avoidance.

(2) When infrastructure changes are needed, can we make them universal in nature? Here, we focus on packet scheduling, examining whether we can have a universal packet scheduling algorithm that can mimic all others. We show, both theoretically and practically, that we can have an almost-universal packet scheduling algorithm that can closely mimic other scheduling algorithms and can achieve a variety of network-wide performance objectives.

To my father and my grandmother,
who are guiding me from beyond.

Contents

Acknowledgments

The six years of PhD have seen me grow tremendously,
And not just academically, but also personally.
In this, many people played an important role.
This barely rhyming poem attempts to thank them all.

The biggest thanks goes to Sylvia and Scott,
My wonderful advisors, who taught me a lot.
Experimenting, writing, presenting and avoiding haste,
But most importantly, developing a good taste.
I would run to them, for matters big or small,
With their deep wisdom, they could resolve them all.

I would like to thank my other collaborators too,
From each of whom, I learned something new.
Thanks to Justine, my co-author for RC3,
For keeping me calm during my initial state of worry.
Thanks to the team at Google, and Nandita especially,
For giving me the opportunity to work on TIMELY.
Thanks to Panda and Arvind, the most amazing collaborators,
And to Alex, Eitan, and Rachit, along with some others.
(While I only listed co-authors for the works in this thesis,
My thanks also extends to other collaborators beyond these.)

For all the fun, food, gossips along with technical debates,
My next set of thanks goes to my amazing lab-mates:
Colin, Kay, Panda, Shivaram, and Amin,
Aisha, Michael, Sangjin, Murphy, and Justine.
Silvery, Peter, Emmanuel, Wen, Ethan, and Gautam,
Wenting, Qifan, Chang, Anwar, Zafar, and Yotam.
A special thank you to Amin, Panda and Kay,
For the pep talks on work and life we had day-to-day.

What I also very heavily exploited, I must acknowledge,
Are Amin's technical depth and Panda's breadth of knowledge.

For all the IT help and for being the nicest admin,
A big thank you respectively to Jon and Carlyn.

For making life outside work also enjoyable and cheerful,
To my many friends in Berkeley, I am very thankful:
Gautam, Bharath, Nikunj, Anurag, Shubham, TD, Sukanya,
Sreeta, Shromona, Neeraja, Vivek, Moitrayee, and Aishwarya.

Now for some people, thanks to whom I am here,
For their motivation, inspiration, guidance or steer.
My teachers in school, and IIT professors,
Arka and Gautam, my college predecessors.
Ranveer and Aman at MSR introduced research to me,
And thus paved a path for me to pursue a PhD.

My family's support and their love ever-growing,
Their belief in me, is what keeps me going.
It wasn't easy for them to send me so far away,
But they didn't let their emotions get in the way.
To have such a family, I feel truly blessed,
My gratitude to them cannot be fully expressed.

For my best friend and husband, are these last few lines,
Who has stood by me through my laughter and whines.
I am thankful to Saurabh for joining me in this ride,
I couldn't have done much without him by my side.

Chapter 1

Introduction

The vast majority of our day-to-day applications run over computer networks, both across wide-area Internet and within datacenters, leading to an intense emphasis on network performance. This performance not only affects user satisfaction, but also has a significant impact on the revenues generated by online service providers such as Google, Microsoft and Facebook [129]. As a result, there have been many proposals for adding sophisticated features in network switches to meet different performance requirements. Such features include various scheduling policies [27, 111, 117, 30, 145, 11, 101], explicit rate signalling [34, 72], mechanisms to achieve losslessness [60], among others.

While these features promise improved performance, they often come at a high cost, either requiring specialized network switching hardware or greatly complicating network management. Rather than continuing to add new features to the network in an adhoc manner, we argue for a more principled approach: First of all, echoing the classical end-to-end principle [113], complexity should be added into the network only when the required goals cannot be met solely from the end-points. Secondly, in such cases where it is necessary to add network support, we should look for *universal* solutions, where a single mechanism implemented in network switches can handle a wide range of requirements. In many cases, the networking community has not explored the former before adding new features to the network, while the notion of universality had not even been formally defined. This dissertation, therefore, focuses on the following two questions:

(1) Can we avoid making changes to the network infrastructure? We explore this question in the context of congestion control (for both wide area and datacenter networks) and in the context of RDMA deployment in today's datacenters.

(2) When infrastructure changes are needed, can we make them universal in nature? Here, we focus on packet scheduling, exploring whether we can have a *universal packet scheduling algorithm*.

We begin with providing some relevant background in §1.1, before elaborating more on these questions in §1.2 and §1.3 respectively.

1.1 Relevant Background

The following is a very simplified view of how data is transferred across a computer network: Computers, phones, datacenter-servers, and other such *endhost* devices are connected to one another by a *network* of *switches*. The endhost's operating system typically implements a software network stack which transmits/receives data over a hardware device (called a *network interface card* or NIC) that interfaces with the physical network links. The data to be transferred is split into multiple *packets*. Packets belonging to the same connection make up a *flow*. The data payload in the packet is encapsulated within *packet headers* that contain the meta-data needed by the network to process and forward the packet. Switches in the network forward the packets along an appropriate route to the destination. Switches can only forward the packets at a finite rate determined by the *bandwidth* of their outgoing links. Therefore, the packets that arrive at a switch while it is busy forwarding another packet are *queued* in a switch *buffer*. When the buffer becomes full a packet is *dropped*. Most network communication happens over *reliable* connections, where the receiver acknowledges the data it receives and the sender needs to retransmit the dropped (or *lost*) packets. The time interval between a sender sending a packet and then receiving the corresponding acknowledgement is called a *round-trip time* or an RTT.

Network performance primarily depends on two main classes of algorithms:
(i) Congestion control algorithms that run at the endhosts and control the rate at which data packets are injected into the network. Typical congestion control algorithms use a feedback loop based on congestion signals such as packet loss, round trip times, or explicit signals set by the switches.
(ii) Scheduling algorithms that run inside the switches and determine the order in which the queued up packets in the buffer are to be transmitted on the outgoing link. While many scheduling algorithms have been proposed in the past [27, 111, 117, 30, 145, 11, 101], low-end commodity switches today only support a fixed number (typically eight) of priority queues, with simple first-in-first-out (FIFO) scheduling within each queue.

1.2 Can we avoid network infrastructure changes?

As mentioned before, we explore this question in the context of congestion control and RDMA deployment, and propose the following end-point based solutions that eliminate the need for infrastructure changes.

RC3: Recursively cautious congestion control for wide-area networks. A good congestion control algorithm needs to satisfy two conflicting goals (i) efficient use of network bandwidth, and (ii) not harming other flows that are sharing the network. The former requires a more aggressive behavior, while the latter requires a more cautious one. Conventional approaches aim to achieve these two conflicting goals by using a single mechanism of finding the appropriate sending rate. As a result, TCP [40], the most widely deployed congestion control algorithm, starts cautiously, sending a very small amount of data at first and exponentially increasing its sending rate after every round-trip, until the flow starts experiencing packet drops. This cautious ramp-up leads to

significant wastage of network capacity, especially in wide-area networks with large RTTs and increasing bandwidths. To avoid this wasted capacity, prior proposals [34, 72] advocated explicit rate signalling that required making significant changes to the switches. RC3 adopts a different approach that leverages the priority queuing support present in almost all currently deployed switches. It decouples the two conflicting goals by *sending additional data aggressively* (to use all of the available network capacity), but *at a lower priority* than the regular traffic (so as not to harm the other flows in the network). This reduces the average flow completion by 40% when compared to regular TCP, while performing better than prior proposals that require explicit rate signaling.

TIMELY: RTT-based congestion control for the datacenter. The conventional use of packet drops as the sole congestion signal results in high queuing delay, since the sending rate is reduced only after the switch buffer gets full. This negatively impacts performance, particularly for short interactive flows in datacenters with stringent latency requirements. It is, therefore, desirable to react quickly to congestion as soon as switch buffer queues begin to grow. This makes RTT a natural choice for a congestion signal, which provides fine-grained information about the queuing within the network, without requiring any switch support. However, it was considered to be too noisy to be used effectively in datacenters (where RTTs can be of the order of a few microseconds). As a result, the datacenter community moved towards using explicit congestion notification (ECN) [8, 134, 146], which requires the switches to set a bit in the packet headers when the queuing exceeds a certain (carefully tuned) threshold. We, instead, show that accurate RTT measurements at microsecond granularity are possible in today's datacenters and develop *TIMELY*, one of the first RTT-based congestion control algorithm for datacenters. TIMELY updates the sending rate based on the RTT gradient, to achieve ultra-low latency while maintaining near-optimal throughput.

IRN: Improved RoCE NIC for deploying RDMA in datacenters. Current datacenter requirements of low latency, high throughput and negligible CPU utilization can no longer be met by the software packet processing stack in the operating systems. Therefore, leading enterprises are moving towards using RDMA, where packet processing is offloaded to specialized NICs. RoCE (RDMA over Converged Ethernet) has emerged as the canonical means for deploying RDMA over the Ethernet fabric in datacenters [146, 47]. To achieve good performance, RoCE requires a lossless network which is, in turn, achieved by configuring the switches to enable Priority Flow Control (PFC) [60] within the network. Deployment experiences with PFC have shown how it greatly complicates network management and brings with it a host of problems such as head-of-the-line blocking, congestion spreading, and occasional deadlocks. Rather than trying to fix these issues, we take a step back and question the need for PFC in the first place. We show that the need for PFC is merely an artifact of current RoCE NIC designs rather than a fundamental requirement. We propose an *improved RoCE NIC* (IRN) design that makes a few incremental changes to the RoCE NIC for better handling of packet losses. We show that IRN (without PFC) outperforms RoCE (with PFC) by 6-83% for typical network scenarios. Thus, not only does IRN eliminate the need for PFC, it *improves* performance in the process.

1.3 Can we have a universal packet scheduling algorithm?

The previous section briefly presented some examples on how network infrastructure changes can be avoided. However, network performance depends on multiple factors, for some of which infrastructure support is inevitable. Packet scheduling is one such factor.

Packet scheduling algorithms play a key role in achieving various performance goals. As a result, there is a large and active research literature on novel packet scheduling algorithms. These include mechanisms for achieving fairness [30, 117, 145, 101], reducing tail latency [27], meeting deadlines [114], minimizing flow completion times [92, 11], among many others. Each of these scheduling algorithms must be implemented in the switch hardware, making it difficult to support *different scheduling algorithms* for *different performance requirements*. This led us to the following question: do we really need different scheduling algorithms for different requirements, or can we instead have a *universal packet scheduling* algorithm?

More precisely, we analyze (both theoretically and empirically) whether there is a single packet scheduling algorithm that, at a network-wide level, can perfectly match the results of *any* given scheduling algorithm. We find that in general the answer is "no". However, we show theoretically that the classical Least Slack Time First (LSTF) [77] scheduling algorithm comes closest to being universal and demonstrate empirically that LSTF can closely mimic a wide range of scheduling algorithms. We then evaluate whether LSTF can be used *in practice* to meet various network-wide objectives by looking at popular performance metrics (such as average flow completion time, tail packet delays, and fairness); we find that LSTF performs comparable to the state-of-the-art for each of them. We also discuss how LSTF can be used in conjunction with active queue management schemes (such as CoDel [96] and ECN [106]) without changing the core of the network.

1.4 Dissertation Plan

The rest of this dissertation is divided into five chapters: The following three chapters focus on the three examples that explore the first question on avoiding changes to the network infrastructure as discussed in §1.2: Chapter 2 presents RC3, our wide-area congestion control scheme (adapted from [92]); Chapter 3 presents TIMELY, our RTT-based datacenter congestion control scheme (adapted from [91]), and Chapter 4 presents IRN, our new NIC design for deploying RDMA in datacenters (adapted from [93]). Then, in Chapter 5, we focus on the second question from §1.3, detailing both the theoretical and practical aspects of universal packet scheduling (adapated from [90]). Finally, we conclude and discuss some future work in Chapter 6.

Chapter 2

Recursively Cautious Congestion Control

In this chapter we present RC3 as the first example of how network infrastructure changes can be avoided, in the context of congestion control for wide-area networks.

We begin by noting two facts about wide area networks. First, modern ISPs run their networks at a relatively low utilization [42, 66, 94]. This is not because ISPs are incapable of achieving higher utilization, but because their networks must be prepared for link failures which could, at any time, reduce their available capacity by a significant fraction. Thus, most ISP networks are engineered with substantial headroom, so that ISPs can continue to deliver high-quality service even after failures.

Second, TCP congestion control is designed to be *cautious*, starting from a small window size and then increasing every round-trip time until the flow starts experiencing packet drops [40]. The need for fairness requires that all flows follow the same congestion-control behavior, rather than letting some be cautious and others aggressive. Caution, rather than aggression, is the better choice for a uniform behavior because it can more easily cope with a heavily overloaded network; if every flow started out aggressively, the network could easily reach a congestion-collapsed state with a persistently high packet-drop rate.

These decisions – underloaded networks and cautious congestion control – were arrived at independently, but interact counter-productively. When the network is underloaded, flows will rarely hit congestion at lower speeds. However, the caution of today's congestion control algorithms requires that flows spend significant time ramping up rather than aggressively assuming that more bandwidth is available, thus resulting in wastage of available network capacity.

In recent years there have been calls to increase TCP's initial window size to alleviate this problem but, as we shall see later, this approach brings only limited benefits. Other proposals include adding mechanisms for explicit rate signaling in the switches [72, 34], where switches compute the appropriate sending rate and put this value in the packet headers. This allows the senders to start sending at the appropriate rate and avoid wasting network capacity due to cautious probing. However, such schemes require major changes to the switches and therefore face significant deployment hurdles.

In this chapter we propose a new approach called *recursively cautious congestion control* (RC3) that retains the advantages of caution while enabling it to efficiently utilize the available bandwidth.

The idea builds on a perverse notion of quality-of-service, called WQoS, in which we assume ISPs are willing to offer *worse* service if certain ToS bits are set in the packet header (and the mechanisms for doing so – priority queues, are present in almost all currently deployed switches). While traditional calls for QoS – in which better service is available at a higher price – have foundered on worries about equity (should good Internet service only be available to those who can pay the price?), pricing mechanisms (how do you extract payments for the better service?), and peering (how do peering arrangements cope with these higher-priced classes of service?), in our proposal we are only asking ISPs to make several worse classes of service available that would be treated as regular traffic for the purposes of charging and peering. Thus, we see fewer institutional barriers to deploying WQoS. Upgrading an operational network is a significant undertaking, and we do not make this proposal lightly, but our point is that many of the fundamental sources of resistance to traditional QoS do not apply to WQoS.

The RC3 approach is quite simple. RC3 runs, at the highest priority, the same basic congestion control algorithm as normal TCP. However, it also runs congestion control algorithms at each of the k worse levels of service; each of these levels sends only a fixed number of packets, with exponentially larger numbers at lower priority levels. As a result, all RC3 flows compete fairly at every priority level, and the fact that the highest priority level uses the traditional TCP algorithms ensures that RC3 does not increase the chances of congestion collapse. Moreover, RC3 can immediately “fill the pipe" with packets (assuming there are enough priority levels), so it can leverage the bandwidth available in underutilized networks.

We implemented RC3 in the Linux kernel and in the NS-3 network simulator. We find through experiments on both real and simulated networks that RC3 provides strong gains over traditional TCP, averaging 40% reduction in flow completion times over all flows, with strongest gains – of over 70% – seen in medium to large sized flows.

In the rest of this chapter, we explain RC3’s design (§2.1), provide an analytical model to understand its performance benefits (§2.2), detail its Linux kernel implementation (§2.3) and present our evaluation using simulations as well as real test-bed (§2.4). We end with discussing RC3’s deployability and future applicability (§2.5) and some related work (§2.6) before concluding (§2.7).

2.1 RC3 Design

We now discuss RC3’s design in detail, starting with an overview followed by presenting a simple example of RC3 in action.

2.1.1 Overview

RC3 runs two parallel control loops: one transmitting at normal priority and obeying the cautious transmission rate of traditional TCP, and a second “recursive low priority” (RLP) control loop keeping the link saturated with low priority packets.

In the primary control loop, TCP proceeds as normal, sending packets in order from index 0 in the byte stream, starting with slow-start and then progressing to normal congestion-avoidance

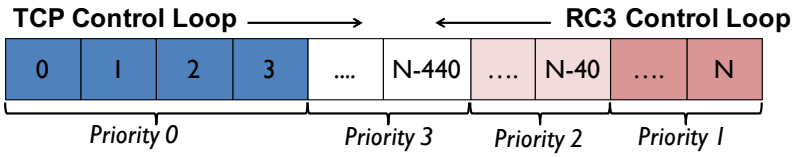


Figure 2.1: Packet priority assignments.

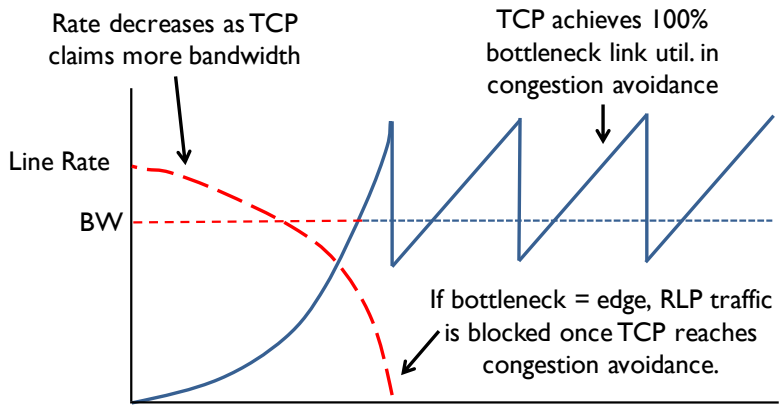


Figure 2.2: Congestion window and throughput with RC3.

behavior after the first packet loss. The packets sent by this default TCP are transmitted at 'normal' priority – priority 0 (with lower priorities denoted by higher numbers).

In the RLP control loop, the sender transmits additional traffic from the same buffer as TCP to the NIC.[1] To minimize the overlap between the data sent by the two control loops, the RLP sender starts from the very *last* byte in the buffer rather than the first, and works its way towards the beginning of the buffer, as illustrated in Figure 2.1. RLP packets are sent at low priorities (priority 1 or greater): the first 40 packets (from right) are sent at priority 1; the next 400 are sent at priority 2; the next 4000 at priority 3, and so on.[2] The RLP traffic can only be transmitted when the TCP loop is not transmitting, so its transmission rate is the NIC capacity minus the normal TCP transmission rate.

RC3 enables TCP selective ACK (SACK) to keep track of which of low priority (and normal priority) packets have been accepted at the receiver. When ACKs are received for low priority packets, no new traffic is sent and no windows are adjusted. The RLP control loop transmits each low priority packet once and once only; there are no retransmissions. The RLP loop starts sending packets to the NIC as soon as the TCP send buffer is populated with new packets, terminating when its 'last byte sent' crosses with the TCP loop's 'last byte sent'. Performance gains from RC3 are seen only during the slow-start phase; for long flows where TCP enters congestion avoidance,

[1] As end-hosts support priority queuing discipline, this traffic will never pre-empt the primary TCP traffic.

[2] RC3 requires the packets to be exponentially divided across the priority levels to accommodate large flows within feasible number of priority bits. The exact number of packets in each priority level has little significance, as we shall see in § 2.4.1.

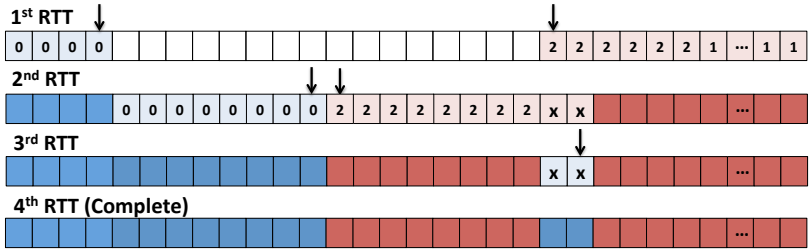


Figure 2.3: Example RC3 transmission from §2.1.2.

TCP will keep the network maximally utilized with priority 0 traffic, assuming appropriately sized buffers [14]. If the bottleneck link is the edge link, high priority packets will pre-empt any packets sourced by the RLP directly at the end host NIC; otherwise the low priority packets will be dropped elsewhere in the network. Figure 2.2 illustrates how the two loops interact: as the TCP sender ramps up, the RLP traffic has less and less 'free' bandwidth to take advantage of, until it eventually is fully blocked by the TCP traffic.

Since the RLP loop does not perform retransmissions, it can leave behind 'holes' of packets which have been transmitted (at low priority) but never ACKed. Because RC3 enables SACK, the sender knows exactly which segments are missing and the primary control loop retransmits only those segments.[3] Once the TCP 'last byte sent' crosses into traffic that has already been transmitted by the RLP loop, it uses this information to retransmit the missing segments and ensure that all packets have been received. We walk through transmission of a flow with such a 'hole' in the following subsection.

2.1.2 Example

We now walk through a toy example of a flow with 66 packets transmitted over a link with an edge-limited delay-bandwidth product of 50 packets. Figure 2.3 illustrates our example.

In the first RTT, TCP sends the first 4 packets at priority 0 (from left); after these high priority packets are transmitted, the RLP loop sends the remaining 62 packets to the NIC – 40 packets at priority 1 and 22 packets at priority 2 (from right), of which 46 packets are transmitted by the NIC (filling up the entire delay-bandwidth product of 50 packets per RTT).

The 21st and 22nd packets from the left (marked as Xs), sent out at priority 2, are dropped. Thus, in the second RTT, ACKs are received for all packets transmitted at priority 0 and for all but packets 21 and 22 sent at lower priorities. The TCP control loop doubles its window and transmits an additional 8 packets; the RLP sender ignores the lost packets and the remaining packets are transmitted by the NIC at priority 2.

In the third RTT, the sender receives ACKs for all packets transmitted in the second RTT and TCP continues to expand its window to 16 under slow start. At this point, the TCP loop sees that all packets except 21st and 22nd have been ACKed. It, therefore, transmits only these two packets.

[3]Enabling SACK allows selective retransmission for dropped low priority packets. However, RC3 still provides significant performance gains when SACK is disabled, despite some redundant retransmissions.

Finally, in the fourth RTT the sender receives ACKs for the 21st and 22nd packets as well. As all data acknowledgements have now been received by the sender, the connection completes.

2.2 Performance Model

Having described RC3 design in §2.1, we now model our expected reduction in Flow Completion Time (FCT) for a TCP flow using RC3 as compared to a basic TCP implementation. We quantify gains as ((FCT with TCP) - (FCT with RC3)) / (FCT with TCP) – i.e. the percentage reduction in FCT [34]. Our model is very loose and ignores issues of queuing, packet drops, or the interaction between flows. Nonetheless, this model helps us understand some of the basic trends in performance gains. We extensively validate these expected gains in §2.4 and see the effects of interaction with other flows.

Basic Model: Let BW be the capacity of the bottleneck link a flow traverses, and u be the utilization level of that link. We define A, the available capacity remaining in the bottleneck link as $A = (1-u) \times BW$. Since RC3 utilizes *all* of the available capacity, a simplified expectation for FCTs under RC3 is $RTT + \frac{N}{A}$, where RTT is the round trip time and N is the flow size.

TCP does not utilize all available capacity during its slow start phase; it is only once the congestion window grows to $A \times RTT$, that the link is fully utilized. The slow start phase, during which TCP leaves the link partially idle, lasts $log(min(N, A \times RTT)/i)$ RTTs, with i being the initial congestion window of TCP. This is the interval during which RC3 can benefit TCP.

In Figure 2.4, the solid line shows our expected gains according to our model. Recall that i denotes the initial congestion window under TCP. For flow sizes $N < i$, RC3 provides no gains over a baseline TCP implementation, as in both scenarios the flow would complete in $RTT + \frac{N}{A}$. For flow sizes $i < N < A \times RTT$, the flow completes in 1 RTT with RC3, and $log(N/i)$ RTTs with basic TCP in slow start. Consequently, the reduction in FCT increases with N over this interval.

Once flow sizes reach $N > A \times RTT$, basic TCP reaches a state where it can ensure 100% link utilization after $log(A \times RTT/i)$ RTTs. Therefore, the improvements from RC3 become a smaller fraction of overall FCT with increasingly large flows; this reduction roughly follows $\frac{log(A \times RTT/i) \times RTT \times A}{N}$ (ignoring a few constants in the denominator).

Parameter Sensitivity: The above model illustrates that improvements in FCTs due to RC3 are dependent primarily on three parameters: the flow size (N), the effective bandwidth-delay product ($A \times RTT$), and the choice of the initial congestion window (i). Peak improvements are observed when N is close to $A \times RTT$, because under these conditions the flow completes in 1 RTT with RC3 and spends its entire life time in slow start without RC3. When the delay-bandwidth product increases, both the optimal flow size (for performance improvement) increases, and the maximum improvement increases.

Adjusting *i*: There are several proposals [12, 35] to adjust the default initial congestion window in TCP to 10 or even more packets. Assume we adjusted a basic TCP implementation to use a new value, some i' as its initial congestion window. The dotted line in Figure 2.4 illustrates the

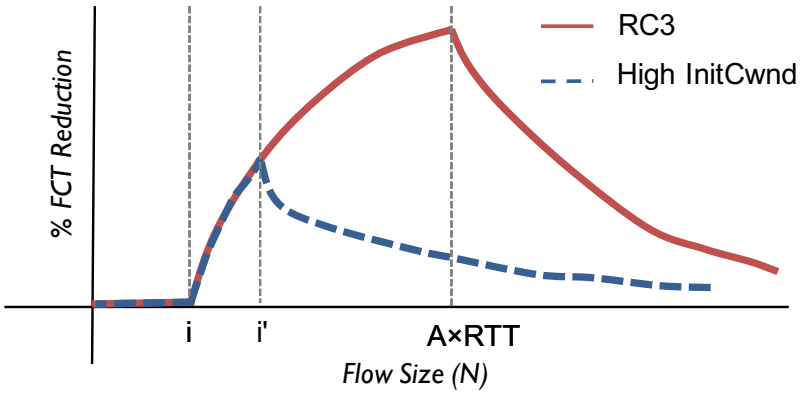


Figure 2.4: Performance gains as predicted by a simple model for RC3 and an increased initial congestion window.

gains from such an i'. When i' increases, the amount of time spent in slow start decreases to $log(min(N, A \times RTT)/i') \times RTT$. Flows of up to i' packets complete in a single RTT, but unless $i' = A \times RTT$ (hundreds of packets for today's WAN connections), adjusting the initial congestion window will always under-perform when compared to RC3. However, there is good reason not to adjust i' to $A \times RTT$: without the use of low priorities, as in RC3, sending a large amount of traffic without cautious probing can lead to an increase in congestion and overall *worse* performance. Our model does not capture the impact of queuing and drops, however, in §2.4.1 we show via simulation how increasing the initial congestion window to 10 and 50 packets penalizes small flows in the network.

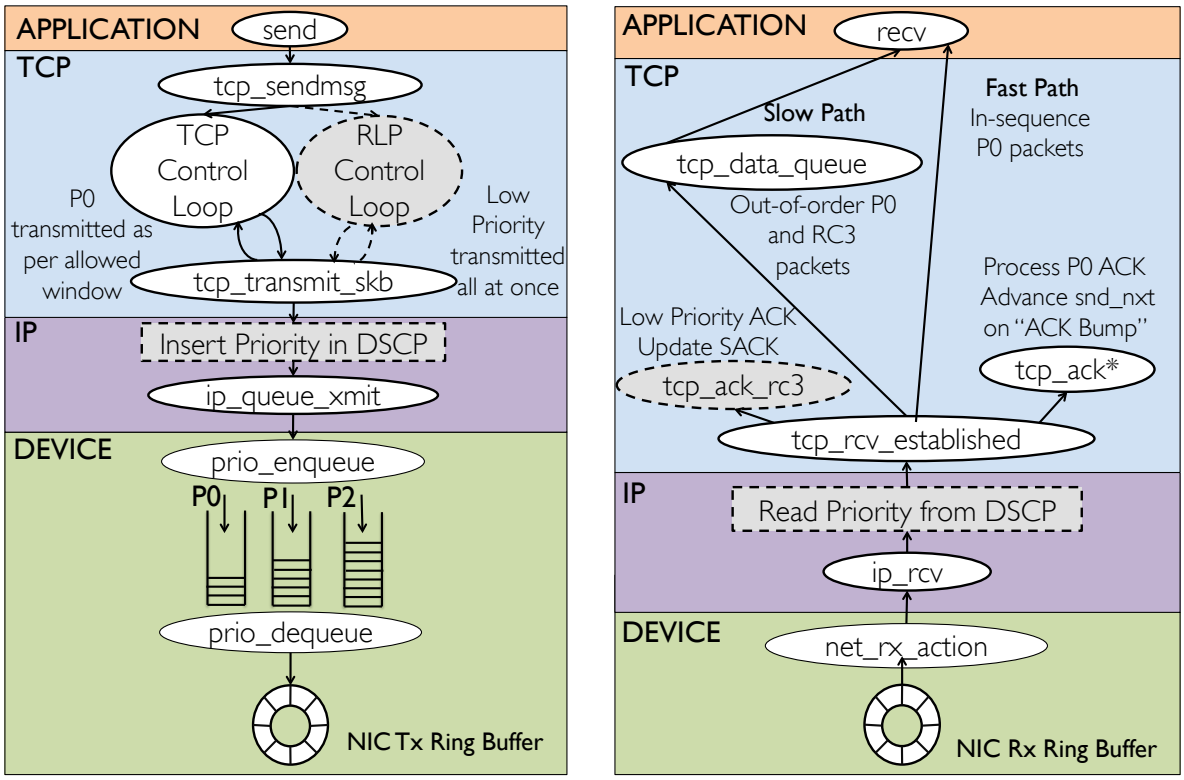
2.3 RC3 Linux Implementation

We implemented RC3 as an extension to the Linux 3.2 kernel on a server with Intel 82599EB 10Gbps NICs. Our implementation cleanly sits within the TCP and IP layers and requires minimal modifications to the kernel source code. Because our implementation does not touch the core TCP congestion control code, different congestion control implementations can easily be 'swapped out' while still retaining compatibility with RC3. After describing our implementation in §2.3.1, we discuss how RC3 interacts with other components of the networking stack in §2.3.2, including application buffering, QoS support, hardware performance optimizations, and SACK extensions.

2.3.1 Extending TCP/IP in the Linux Kernel

We briefly provide high-level context for our implementation, describing the TCP/IP stack under the Linux 3.2 kernel. We expect that RC3 can be easily ported to other implementations and operating systems as well, but leave this task to future work.

Figure 2.5 illustrates the kernel TCP/IP architecture at a very high level, along with our RC3 extensions shaded in gray. The TCP/IP stack in the Linux kernel is implemented as follows. When



(a) Sending Data

(b) Receiving Data and Acks

Figure 2.5: Modifications to Linux kernel TCP stack.

an application calls *send()*, the *tcp_sendmsg* function is invoked; this function segments the send buffer into 'packets' represented by the socket buffer (*skb*) datastructure. By default, each *skb* represents one packet to be sent on the wire. After the data to be transmitted has been segmented in to *skb*s, it is passed to the core TCP logic, and then forwarded for transmission through the network device queue to the NIC.

On the receiver side, packets arrive at the NIC and are forwarded up to a receive buffer at the application layer. As packets are read in, they are represented once again as *skb* datatypes. Once packets are copied to *skb*s, they are passed up the stack through the TCP layer. Data arriving in-order is sent along the 'fast-path', directly to the application layer receive buffer. Data arriving out of order is sent along a 'slow path' to an out of order receive queue, where it waits for the missing packets to arrive, before being forwarded up in-order to the application layer.

We now describe how we extend these functions to support RC3.

Sending Data Packets. RC3 extends the send-side code in the networking stack with two simple changes, inserting only 72LOC in the TCP stack and 2LOC in the IP stack. The first change, in the TCP stack, is to invoke the RLP control loop once the data has been segmented in the *tcp_sendmsg* function. We leave all of the segmentation and core TCP logic untouched – we merely add a function call in *tcp_sendmsg* to invoke the RLP loop, as shown in Fig. 2.5.

The RLP loop then reads the TCP write queue iteratively from the tail end, reading in the packets one by one, marking the appropriate priority in the packet buffer, and then sending out the packet. The field *skb→priority* is assigned according to the sequence number of the packet: the RLP loop subtracts the packet's sequence number from the tail sequence number and then divides this value by the MSS. If this value is ≤ 40, the packet is assigned priority 1, if the value is ≤ 400 it is assigned priority 2, and so on. After the priority assignment, the *skb* packets are forwarded out via the *tcp_transmit_skb* function.

Our second change comes in the IP layer as packets are attached to an IP header; where we ensure that *skb→priority* is not overwritten by the fixed priority assigned to the socket, as in the default case. Instead, the value of *skb→priority* is copied to the DSCP priority bits in the IP header.

Overall, our changes are lightweight and do not interfere with core congestion control logic. Indeed, because the TCP logic is isolated from the RC3 code, we can easily enable TCP CUBIC, TCP New Reno, or any other TCP congestion control algorithms to run alongside RC3.

Receiving Data Packets and ACKs. Extending the receive-side code with RC3 is similarly lightweight and avoids modifications to the core TCP control flow. Our changes here comprise only of 46 LOC in the TCP stack and 1 LOC in the IP stack.

Starting from bottom to top in Figure 2.5, our first change comes as packets are read in off the wire and converted to *skb*s – here we ensure that the DSCP priority field in the IP header is copied to the *skb* priority field; this change is a simple 1 LOC edit.

Our second set of changes which lie up the stack within TCP. These changes separate out low priority packets from high priority in order to ensure that the high priority ACKing mechanism (and therefore the sender's congestion window) and other TCP variables remain unaffected by the low priority packets. We identify the low priority packets and pass them to the out of order 'slow path' queue, using the unmodified function *tcp_data_queue*. We then call a new function, *tcp_send_ack_rc3*, which sends an ACK packet for the new data at the same priority the data arrived on, with the cumulative ACK as per the high priority traffic, but SACK tags indicating the additional low priority packets received. The priority is assigned in the field *skb→priority*, and the packets are sent out by calling *tcp_transmit_skb*.

The other modifications within the TCP receive code interpose on the handling of ACKs. We invoke *tcp_ack_rc3* on receiving a low priority ACK packet, which simply calls the function to update the SACK scoreboard (which records the packets that have been SACKed), as per the SACK tags carried by the ACK packet. We also relax the SACK validation criteria to update the SACK "scoreboard" to accept SACKed packets beyond *snd_nxt*, the sequence number up to which data has been sent out by the TCP control loop.

Typically when a new ACK is received, the stack double-checks that the received ACK is at a value less than *snd_nxt*, discarding the ACKs that do not satisfy this constraint. We instead tweak the ACK processing to update the *snd_nxt* value when a high-priority ACK is received for a sequence number that is greater than *snd_nxt*: such an ACK signals that the TCP sender has "crossed paths" with traffic transmitted by the RLP and is entering the cleanup phase. We advance the send queue's head and update *snd_nxt* to the new ACKed value and then allow TCP to continue as usual; we call this jump an "ACK bump."

While these changes dig shallowly in to the core TCP code, they do not impact our compatibility with various congestion control schemes.

2.3.2 Specific Implementation Features

We now discuss how RC3 interacts with some key features at all layers of the networking stack, from software to NIC hardware to switches and switches.

Socket Buffer Sizes. The default send and receive buffer sizes in Linux are very small - 16KB and 85KB respectively. Performance gains from RC3 are maximized when the entire flow is sent out in the first RTT itself. This requires us to make the send and receive buffers as big as the maximum flow size (up to a few MBs in most cases). Window scaling is turned on by default in Linux, and hence we are not limited by the 64KB receive window carried by the TCP header.

RC3 is nonetheless compatible with smaller send buffer sizes: every call to *tcp_sendmsg* passes a chunk of data to the RLP control loop, which treats that chunk, logically, as a new flow as far as priority assignment is concerned. We include a check to break the RLP control loop to ensure that the same packet is not transmitted twice by subsequent calls to *tcp_sendmsg*. Indeed, this behavior can help flows which resume from an application-layer imposed idle period.

Using QoS Support. RC3 is only effective if priority queuing is supported at both endhosts and the switches in the network.
Endhosts: We increase the Tx queue length at the software interface, to ensure that it can store all the packets forwarded by the TCP stack. The in-built traffic control functionality of Linux is used to set the queuing discipline (*qdisc*) as *prio* and map the packet priorities to queue 'bands'. The *prio* qdisc maintains priority queues in software, writing to a single NIC ring buffer as shown in Figure 2.5. Thus, when the NIC is free to transmit, a packet from band N is dequeued only if all bands from 0 to $N-1$ are empty. Up to 16 such bands are supported by the Linux kernel, which are more than enough for RC3.[4]
Switches: All modern switches today support QoS, where flow classes can be created and assigned to a particular priority level. Incoming packets can then be mapped to one of these classes based on the DSCP field. The exact mechanism of doing so may vary across different vendors. Although the ISPs may use the DSCP field for some internal prioritization, all we require them to do is to read the DSCP field of an incoming packet, assign a priority tag to the packet which can be recognized by their switches, and then rewrite the priority in the DSCP field when the packet leaves their network.

Compatibility with TSO/LRO. *TCP Segmentation Offload* (TSO) and *Large Receiver Offload* (LRO) are two performance extensions within the Linux kernel that improve throughput through batching. TSO allows the TCP/IP stack to send packets comprising of multiple MSSes to the NIC, which then divides them into MSS sized segments before transmitting them on the link. LRO is the receiver counterpart which amasses the incoming segments into larger packets at the driver, before

[4] 16 priority levels is sufficient to support RC3 flow sizes on the order of a petabyte!

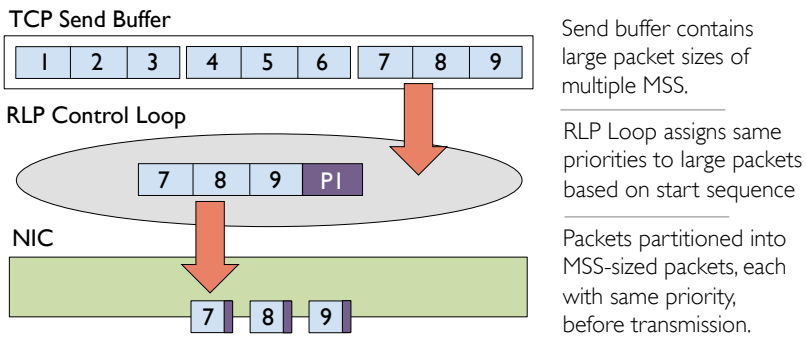


Figure 2.6: RC3 combined with TSO.

sending them higher up in the stack. TSO and LRO both improve performance by amortizing the cost of packet processing across packets in batches. Batched packets reduce the average per-packet processing CPU overhead, consequently improving throughput.

Figure 2.6 illustrates how RC3 behaves when TSO is enabled at the sender and a larger packet, comprising of multiple MSSes is seen by the RLP control loop. At first glance, TSO stands in the way of RC3: RC3 requires fine-grained control over individual packets to assign priorities, and the over sized packets passing through under TSO hinder RC3's ability to assign priorities correctly when it includes data from packets that should be assigned different priority classes. Rather than partitioning data within these extra large packets, we simply allow RC3 to label them according to the lowest priority of any data in the segment. This means that we might not strictly follow the RC3 design in §2.1 while assigning priorities for some large packets. However, such a situation can arise only when the MSSes in a large packet overlap with the border at which priority levels switch. Since the traffic is partitioned across priority level exponentially, such cases are infrequent. Further, the largest TSO packet is comprised of at most 64KB. Therefore, no more than 43 packets would be improperly labeled at the border between priority levels.

TSO batching leads to a second deviation from the RC3 specification, in that segments within a large packet are sent in sequence, rather than in reverse order. For example, in Figure 2.6, the segments in the packet are sent in order (7,8,9) instead of (9,8,7). Hence, although RC3 still processes *skb* packets from tail to front, the physical packets sent on the wire will be sent in short in-order bursts, each burst with a decreasing starting sequence number. Allowing the forward sequencing of packets within a TSO batch turns out to be useful when LRO is enabled at the receiver, where batching at the driver happens *only* if the packets arrive in order.

As we will show in §2.4.2, combining RC3 with TSO/LRO reduces the OS overhead of processing RC3 packets by almost 50%, and consequently leads to net gains in FCTs.

SACK Enhancements. Although RC3 benefits when SACK is enabled, it is incompatible with some SACK enhancements. Forward Acknowledgment (FACK) [83], is turned on by default in Linux. It estimates the number of outstanding packets by looking at the SACKed bytes. RC3 SACKed packets may lead the FACK control loop to falsely believe that all packets between the highest cumulative ACK received and the lowest SACK received are in flight. We therefore disable FACK to avoid the RC3 SACKed bytes from affecting the default congestion control behavior. Do-

ing so does not penalize the performance in most cases, as the Fast Recovery mechanism continues transmitting unsent data after a loss event has occurred and partial ACKs are received, allowing lost packets to be efficiently detected by duplicate ACKs. DSACK [37] is also disabled, to avoid the TCP control loop from inferring incorrect information about the ordering of packets arriving at the receiver based on RC3 SACKs.

2.4 Experimental Evaluation

We now evaluate RC3 across several dimensions. In §2.4.1, we evaluate RC3 extensively using NS-3 simulations - comparing RC3's FCT reductions with the model we described in §2.2; evaluating RC'3 robustness and fairness, and comparing RC3's FCT reductions relative to other designs. We evaluate our Linux RC3 implementation in §2.4.2.

2.4.1 Simulation Based Evaluation

We implement RC3 using NS-3 network simulator [99] and evaluate it across a wide range of simulation settings.[5] Our primary simulation topology models a simplified Internet-2 network topology [63] consisting of ten switches, each attached to ten end hosts, with 1Gbps bottleneck bandwidth and 40ms RTT. It runs at 30% average link utilization [42, 66, 94]. The queue buffer size is equal to the bandwidth-delay product (RTT$\times$BW) in all cases, which is 5MB for our baseline. The queues do priority dropping and priority scheduling. All senders transmit using RC3 unless otherwise noted. Flow sizes are drawn from an empirical traffic distribution [13]; with Poisson inter-arrivals.

For most experiments we present RC3's performance relative to a baseline TCP implementation. Our baseline TCP implementation is TCP New Reno [40] with SACK enabled [82, 17] and an initial congestion window of 4 [12]; maximum segment size is set to 1460 bytes while slow start threshold and advertised received window are set to infinity.

Baseline Simulation. We first investigate the baseline improvements using RC3 and compare them to our modeled results from §2.2.

Validating the Model: Figure 2.7 compares the gains predicted by our model (§2.2) with the gains observed in our simulation. The data displayed is for 1Gbps bottleneck capacity, 40ms average RTT, and 30% load. Error bars plotting the standard deviation across 10 runs are shown; they sit very close to the average. For large flows, the simulated gains are slightly lower than predicted; this is the result of queuing delay which is not included in our model. For small flows – four packets or fewer – we actually see *better* results than predicted by the model. This is due to large flows completing sooner than with regular TCP, leaving the network queues more frequently vacant and thus decreasing average queuing delay for short flows. Despite these variations, the simulated

[5] We have made our simulator code available at http://netsys.github.io/RC3/.

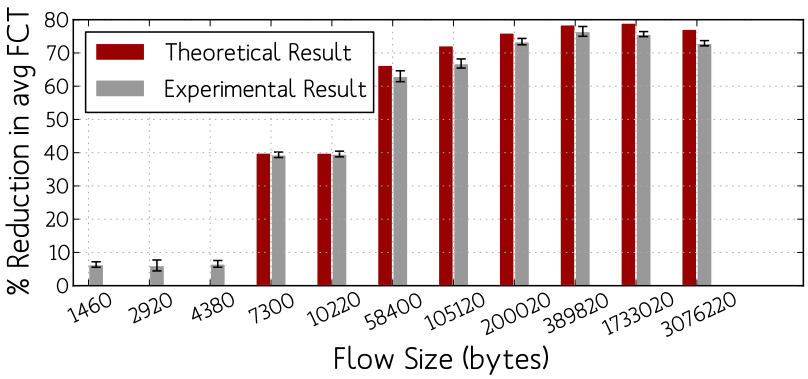


Figure 2.7: Reduction in FCT as predicted by model vs simulations. (RTT×BW = 5MB, 30% average link utilization)

and modeled results track each other quite closely: for all but the smallest flows, we see gains of 40–75%.

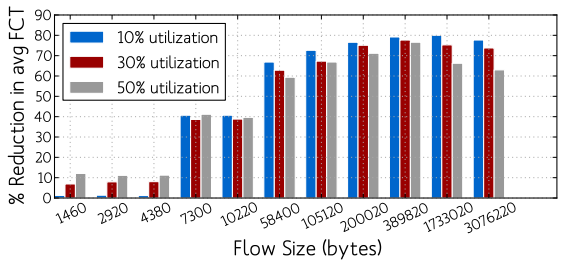
Link Load: Figure 2.8(a) shows FCT performance gains comparing RC3 to the base TCP under uniform link load of 10%, 30%, or 50%. RTT×BW is fixed at 5MB across all experiments. As expected, performance improvements decrease for higher average link utilization. For large flows, this follows from the fact that the available capacity ($A = (1-u) \times BW$) reduces with increase in utilization u. Thus, there is less spare capacity to be taken advantage of in scenarios with higher link load. However, for smaller flows, we actually see the opposite trend. This is once again due to reduced average queuing delays, as large flows complete sooner with most packets having lower priorities than the packets from the smaller flows.

RTT×BW: Figure 2.8(b) shows the FCT reduction due to RC3 at varying RTT×BW. In this experiment we adjusted the average RTTs and the bottleneck bandwidth capacities to achieve RTT×BW of 500KB (40ms×100Mbps), 5MB (40ms×1Gbps and 400ms×100Mbps) and 50MB(400ms×1Gbps). As discussed in §2.2, the performance improvement increases with increasing RTT×BW, as the peak of the curve in Figure 2.4 shifts towards the right. The opposite trend for very short flows is repeated here as well.

Summary: Overall, RC3 provides strong gains, reducing flow completion times by as much as 80% depending on simulation parameters. These results closely track the results predicted by the model presented in §2.2.

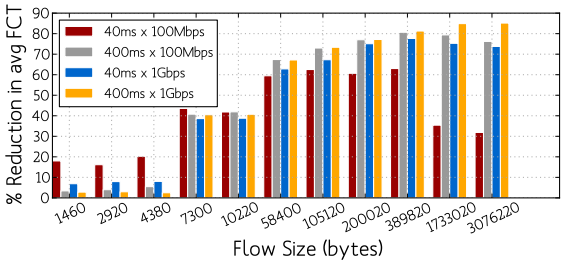
Robustness. In the previous section, we evaluated RC3 within a single context. We now demonstrate that these results are robust, inspecting RC3 in numerous contexts and under different metrics. Many of the results in this section are summarized in Table 2.1.

Topology: We performed most of our experiments on a simulation topology based off a simplified model of the Internet-2 network. To verify that our results were not somehow biased by this topology, we repeated the experiment using simulation topologies derived from the Telstra network,



		Average Over Flows	Average Over Bytes
10%	Regular FCT (s)	0.125	0.423
	RC3 FCT (s)	0.068	0.091
	% Reduction	**45.56**	**78.36**
30%	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
50%	Regular FCT (s)	0.15	0.498
	RC3 FCT (s)	0.088	0.176
	% Reduction	**41.44**	**64.88**

(a) Varying load, with RTT×BW fixed at 5MB



		Average Over Flows	Average Over Bytes
100Mbps 40ms	Regular FCT (s)	0.167	0.691
	RC3 FCT (s)	0.11	0.442
	% Reduction	**33.98**	**36.05**
100Mbps 400ms	Regular FCT (s)	0.948	3.501
	RC3 FCT (s)	0.567	0.783
	% Reduction	**40.29**	**77.62**
1Gbps 40ms	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
1Gbps 400ms	Regular FCT (s)	0.971	3.59
	RC3 FCT (s)	0.558	0.569
	% Reduction	**42.45**	**84.17**

(b) Varying RTT×BW with 30% load

Figure 2.8: Reduction in FCT as network load and RTT×BW is varied.

the Red Clara academic network, and the complete ESNet [84, 21], keeping the average delay as 40ms and the bottleneck bandwidth as 1Gbps. The second to fourth rows in Table 2.1 present these results. All three topologies provided results similar to our initial Internet-2 experiments: average FCTs for Telstra improved by 47.07%, for Red Clara, by 42.78%, and for ESNet by 33.91%.

Workload Distribution: Our baseline experiments use an empirical flow size distribution [13]. A noticeable property of the flow size distribution in our experiments is the presence of of very large flows (up to a few MBs) in the tail of the distribution. We repeated the Internet-2 experiment with an empirical distribution from a 2000 [62] study, an era when average flow sizes were smaller than today. The fifth row of Table 2.1 presents these results. Here we saw that the average FCT improved by only 13.83% when averaged over all flows. When averaging FCT gains weighted by bytes, however, we still observe strong gains for large flows resulting in a reduction of 66.73%.

Link Heterogeneity: We now break the assumption of uniform link utilization and capacity: in this experiment we assigned core link bandwidths in the Internet-2 topology to a random value between 500Mbps and 2Gbps. The results are presented in the last row of Table 2.1. We observed that FCTs in the heterogenous experiment were higher than in the uniform experiment, for both TCP and RC3. Nevertheless, the penalty to TCP was worse, resulting in a stronger reduction in flow completion times, when averaged across flows.

		Average Over Flows	Average Over Bytes
Default: Internet-2	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
Telstra Topology	Regular FCT (s)	0.159	0.510
	RC3 FCT (s)	0.084	0.111
	% Reduction	**47.07**	**78.13**
RedClara Topology	Regular FCT (s)	0.17	0.429
	RC3 FCT (s)	0.097	0.098
	% Reduction	**42.78**	**77.16**
ESNet Topology	Regular FCT (s)	0.207	0.478
	RC3 FCT (s)	0.137	0.0976
	% Reduction	**33.91**	**79.58**
2000 Workload	Regular FCT (s)	0.0871	0.238
	RC3 FCT (s)	0.0704	0.079
	% Reduction	**13.83**	**66.73**
Link Heterogeneity	Regular FCT (s)	0.159	0.541
	RC3 FCT (s)	0.087	0.141
	% Reduction	**45.35**	**73.89**

Table 2.1: RC3's performance across different experimental scenarios.

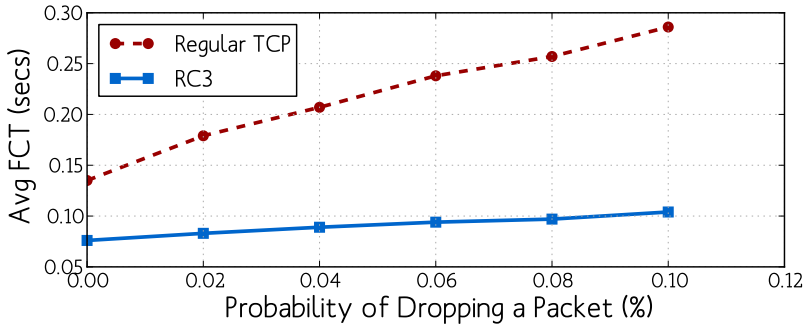
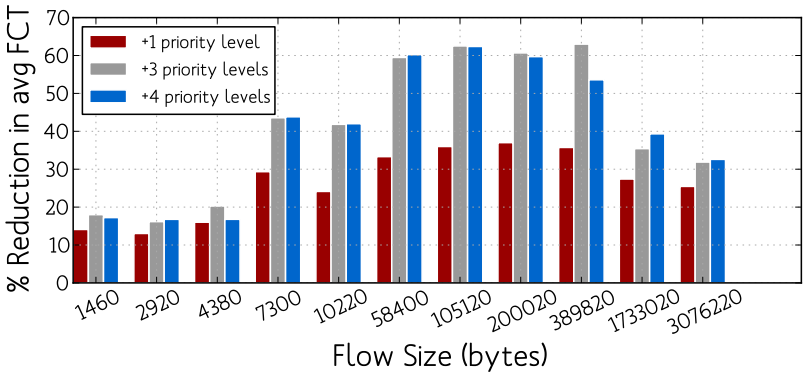


Figure 2.9: Average FCTs with increasing arbitrary loss rate. (RTT×BW = 5MB, 30% network load)

Loss Rate: Until now all loss has been the result of queue overflows; we now introduce random arbitrary loss and investigate the impact on RC3. Figure 2.9 shows flow completion times for TCP and RC3 when arbitrary loss is introduced for 0.02-0.1% of packets. We see that loss strongly penalizes TCP flows, but that RC3 flows do not suffer nearly so much as TCP. RC3 provides even stronger gains in such high loss scenarios because each packet essentially has two chances at transmission. Further, since the RLP loop ignores ACKs and losses, low priority losses do not slow the sending rate.

Priority Assignment: Our design assigns packets across multiple priorities, bucketed exponentially with 40 packets at priority 1, 400 at priority 2, and so on. We performed an experiment to investigate the impact of these design choices by experimenting with an RC3 deployment when 1, 3, or 4 additional priority levels were enabled; the results of these experiments are plotted in Fig. 2.10.



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.167	0.691
1 RC3 Priority Level	RC3 FCT (s)	0.126	0.496
	% Reduction	**24.55**	**28.22**
3 RC3 Priority Levels (40, 400, 4000)	RC3 FCT (s)	0.11	0.442
	% Reduction	**33.98**	**36.05**
4 RC3 Priority Levels (10, 100, 1000, 10000)	RC3 FCT (s)	0.112	0.434
	% Reduction	**32.94**	**37.19**

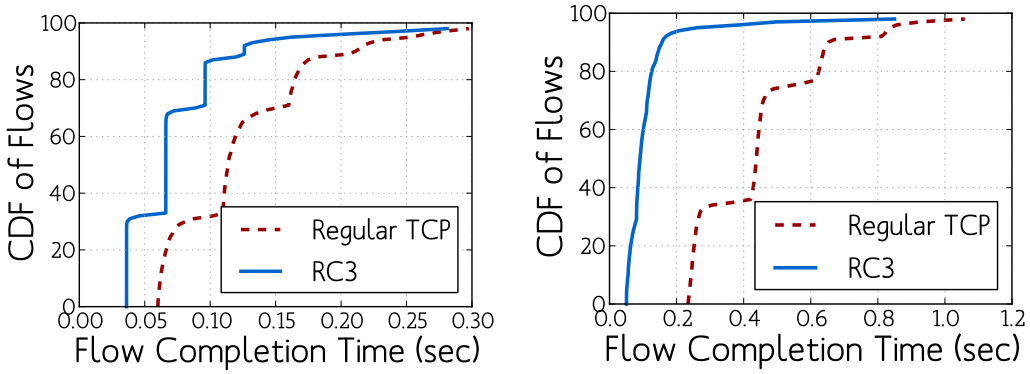
Figure 2.10: Reduction in FCT with varying priority levels. (RTT×BW = 500KB, 30% network load)

We see that dividing traffic over multiple priority levels provides stronger gains than with only one level of low priority traffic. The flows which benefit the most from extra priorities are the medium-sized flows which, without RC3, require more than one RTT to complete during slow start. A very slight difference is seen in performance gains when bucketing packets as (10, 100, 1000, 10000) instead of (40, 400, 4000).

Application Pacing: Until now, our model application has been a file transfer where the entire contents of the transfer are available for transmission from the beginning of the connection. However, many modern applications 'pace' or 'chunk' their transfers. For example, after an initial buffering phase YouTube paces video transfers at the application layer, transmitting only a few KB of data at a time proportional to the rate that the video data is consumed. To see the effect of RC3 on these type of flows, we emulated a YouTube transfer [107] with a 1.5MB buffer followed by 64KB chunks sent every 100ms. Ultimately, RC3 helped these video connections by decreasing the amount of time spent in buffering by slightly over 70% in our experimental topology. This means that the time between when a user loads the page and can begin video playback decreases while using RC3. However, in the long run, large videos did not complete transferring the entire file any faster with RC3 because their transfer rate is dominated by the 64KB pacing.

Performance at the Tails: Our previous results discuss reduction in the *average* flow completion times; in Figures 2.11(a) and (b) we plot the full cumulative distribution of FCTs from our Internet-2 experiments for two representative flow sizes, 7.3KB and 1.7MB.[6] We see in these results that

[6]The 'jumps' or 'banding' in the CDF are due to the uniform link latencies in the simulation topologies. Paths of



(a) Flow size 7.3KB (b) Flow size 1.7MB

Figure 2.11: Cumulative Distribution of FCTs. (RTT×BW = 5MB, 30% average link utilization)

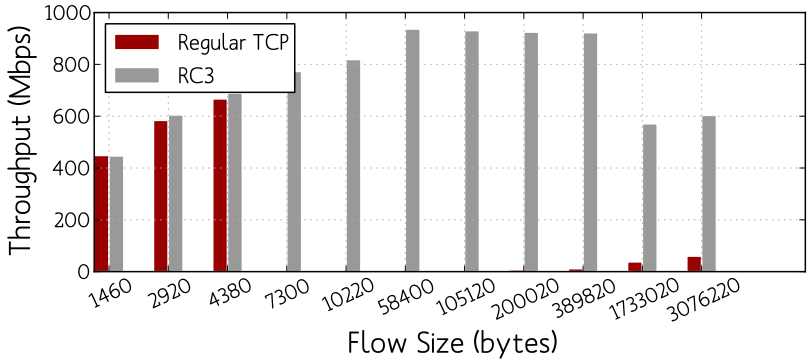


Figure 2.12: Median Flow Throughput

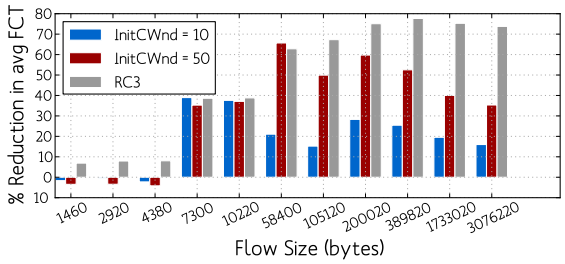
performance improvements are provided across the board at all percentiles; even the 1^{st} and 99^{th} percentiles improve by using RC3.

Summary: In this section, we examined RC3 in numerous contexts, changing our experimental setup, looking at alternative application models, and investigating the tail distribution of FCTs under RC3 and TCP. In all contexts, RC3 provides benefits over TCP, typically in the range of 30-75%. Even in the worst case context we evaluated, when downlink rather than uplink capacities bottlenecked transmission, *RC3 still outperformed baseline TCP by 10%*.

RC3 and Fairness. In this subsection we ask, *is RC3 fair?* We evaluate two forms of fairness: how RC3 flows of different sizes interact with each other, and how RC3 interacts with concurrent TCP flows.

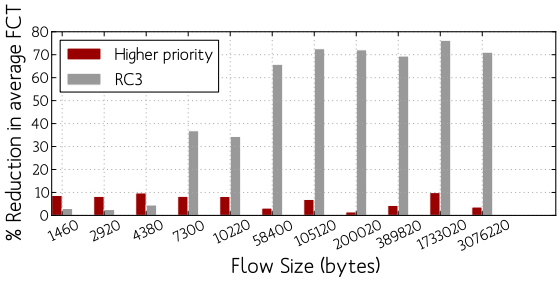
RC3 with RC3: It is well-known that TCP in the long run is biased in that its bandwidth allocations

two hops had an RTT of 40, paths of three hops had an RTT of 60, and so on. A flow which completes in some k RTTs while still under slow start thus completes in approximately $k * RTT$ time. This created fixed steps in the CDF, as per the RTTs.



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.135	0.443
i = 10	FCT (s)	0.117	0.362
	% Reduction	**13.21**	**17.87**
i = 50	FCT (s)	0.102	0.272
	% Reduction	**24.33**	**38.24**
RC3	FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**

(a) Increased InitCwnd



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.126	0.435
QoS	FCT (s)	0.119	0.411
	% Reduction	**5.33**	**5.64**
RC3	FCT (s)	0.078	0.12
	% Reduction	**38.31**	**72.43**

(b) Traditional QoS

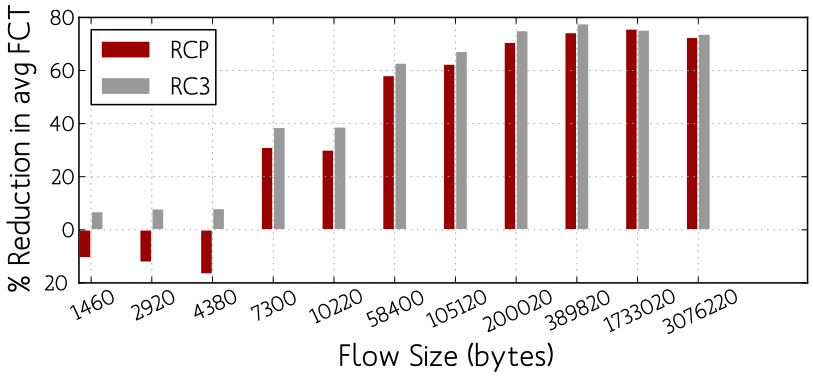
Figure 2.13: RC3 as compared to other alternatives that require no switch changes (RTT$\times$BW = 5MB, 30% average link utilization)

benefit longer flows over short ones. We calculated the effective throughput for flows using TCP or RC3 in our baseline experiments (Figure 2.12). TCP achieves near-optimal throughput for flow sizes less than 4 packets, but throughput is very low for medium-sized flows and only slightly increases for the largest (multiple-MB) flows. RC3 maintains substantially high throughput for all flow sizes, having a slight relative bias towards medium sized flows.

RC3 with TCP: To evaluate how RC3 behaves with concurrent TCP flows, we performed an experiment with mixed RC3 and TCP flows running concurrently. We allowed 50% of end-hosts in our simulations (say in set *A*) to use RC3, while the remaining 50% (set *B*) used regular TCP. Overall, FCTs for both RC3 and TCP were lower than in the same setup where all flows used regular TCP. Thus, RC3 is not only fair to TCP, but in fact improves TCP FCTs by allowing RC3 flows to complete quickly and 'get out of the way' of the TCP flows.

RC3 In Comparison. We now compare the performance gains of RC3 against some other proposals to reduce TCP flow completion times.

Increasing Initial Congestion Window: Figure 2.13(a) compares the performance gains obtained from RC3 with the performance gains from increasing the baseline TCP's initial congestion window (InitCwnd) to 10 and 50. For most flow sizes, especially larger flows, RC3 provides stronger improvements than simply increasing the initial congestion window. When averaging across all flows, RC3 provides a 44% reduction in FCT whereas increasing the InitCwnd to 10 reduces the FCT by only 13% and 50 reduces it by just 24%. Further, for small flow sizes ($\leq$ 4 packets), increasing the InitCwnd actually introduces a *penalty* due to increased queuing delays. RC3 never



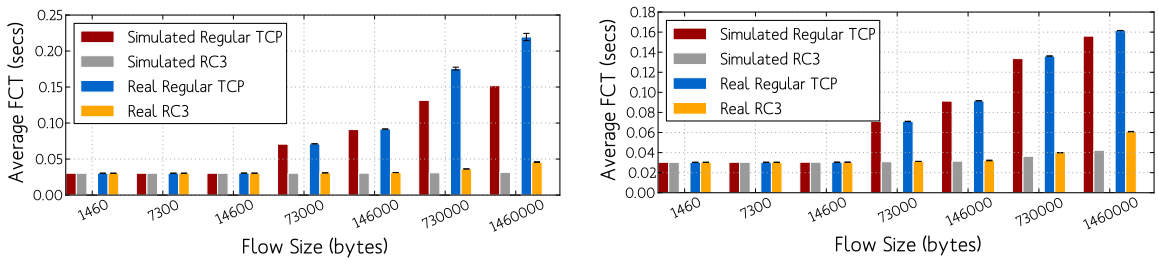
		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.135	0.443
RCP	FCT (s)	0.088	0.117
	% Reduction	**33.86**	**73.58**
RC3	FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**

Figure 2.14: RC3 as compared to RCP, a scheme that requires explicit switch support (RTT$\times$BW = 5MB, 30% average link utilization)

makes flows do worse than they would have under traditional TCP. These results confirm our expectations from §2.2.

Traditional QoS: An alternate technique to improve FCTs is to designate certain flows as 'critical' and send those flows using unmodified TCP, but at higher priority. We annotated 10% of flows as 'critical'; performance results for the critical flows alone are showed in Fig. 2.13(b). When the 'critical' 10% of flows simply used higher priority, their average FCT reduces from 0.126 seconds to 0.119 seconds; while the non-critical flows suffered a very slight ($<$2%) penalty. When we repeated the experiment, but used RC3 for the critical flows (leaving the rest to use TCP), the average FCT reduced from 0.126 seconds to 0.078 seconds, as shown in Figure 2.13(b) . Furthermore, non-critical flows showed a slight ($<$1%) *improvement*. This suggests that it is better to be able to send an unrestricted amount of traffic, albeit at low priority, than to send at high priority at a rate limited by TCP.

RCP: Finally, we compare against RCP, an alternative transport protocol to TCP. With RCP, switches calculate average fair rate and signal this to flows; this allows flows to start transmitting at an explicitly allocated rate from the first (post-handshake) RTT, overcoming TCP's slow start penalty. We show the performance improvement for RCP and RC3 in Fig. 2.14. While for large flows, the two schemes are roughly neck-to-neck, RCP actually imposes a penalty for the very smallest (1-4 packet) flows, in part because RCP's explicit rate allocation enforces *pacing* of packets according to the assigned rate, whereas with traditional TCP (and RC3), all packets are transmitted back to back. These results show that RC3 can provide FCTs which are usually comparable or even better than those with RCP. Further, as RC3 can be deployed on legacy hardware



(a) 20ms×10Gbps (b) 20ms×1Gbps

Figure 2.15: FCTs for implementation vs. simulation

and is friendly with existing TCP flows, it is a more deployable path to comparable performance improvements.

Summary: RC3 outperforms traditional QoS and increasing the initial congestion windows; performance with RC3 is on par with RCP without requiring substantial changes to switches.

2.4.2 Evaluating RC3 Linux Implementation

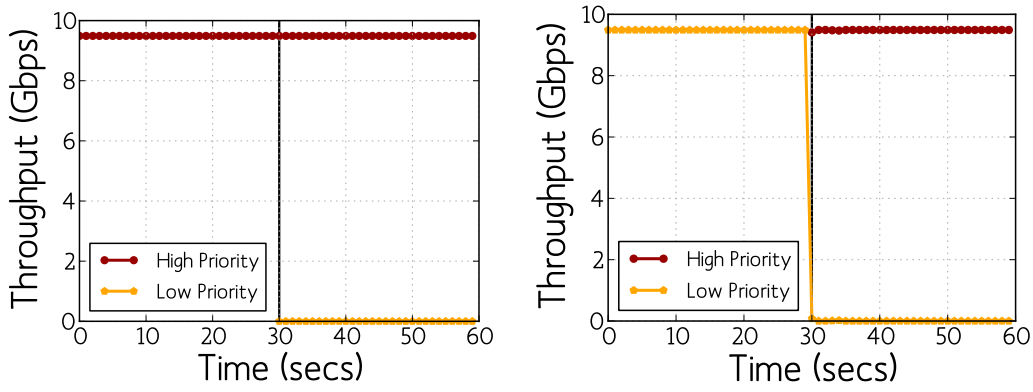
We now evaluate RC3 using our implementation in the Linux kernel. We extended the Linux 3.2 kernel as described in §2.3. We did our baseline experiments using both New Reno and CUBIC congestion control mechanisms. We set the send and receive buffer sizes to 2GB, to ensure that an entire flow fits in a single window. We keep the default initial congestion window of 10 [35] in the kernel unchanged.

Our testbed consists of two servers each with two 10Gbps NICs connected to a 10Gbps Arista datacenter switch. As the hosts were physically adjacent, we used *netem* to increase the observed link latency to 10ms, which reflects a short WAN latency.

Baseline. Flows with varying sizes were sent from one machine to another. Figure 2.15(a) shows FCTs with RC3 and baseline TCP implementation in Linux compared to RC3 and baseline TCP NS-3 simulations (with the initial congestion windows set to 10), both running with 10Gbps bandwidth and 20ms RTT. The figure reflects averages over 100 samples.

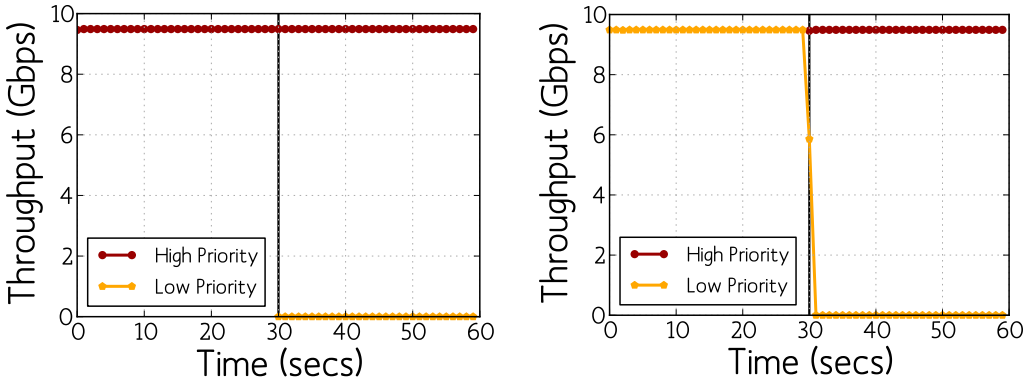
Overall, RC3 continues to provide strong gains over the baseline TCP design, however, our results in implementation do not tightly match our simulated results from NS. The baseline TCP implementation in Linux performs *worse* than in simulation because of delayed ACK behavior in Linux: when more than two segments are ACKed together, it still only generates an increase in congestion window proportional to two packets being ACKed. This slows down the rate at which the congestion window can increase. The RC3 FCT is slightly higher in Linux than in simulation for large flows because of the extra per-packet overhead in receiving RC3 packets: recall from §2.3 that RC3 packets are carried over the Linux 'slow path' and thus have slightly higher per-packet overhead.

In Figure 2.15(b), we repeat the same experiment with only 1Gbps bandwidth set by the token bucket filter queuing discipline (retaining 10ms latency through *netem*). In this experiment, results



(a) Low Priority Starts after High Priority (b) High Priority Starts after Low Priority

Figure 2.16: Correctness of Priority Queuing in Linux



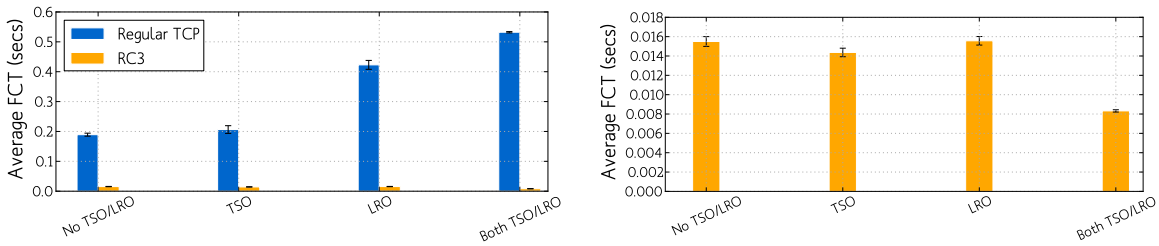
(a) Low Priority Starts after High Priority (b) High Priority Starts after Low Priority

Figure 2.17: Correctness of the Priority Queuing in the Switch

track our simulations more closely. TCP deviates little from the baseline because the arrival rate of the packets ensures that at most two segments are ACKed by the receiver via delayed ACK, and thus the congestion window increases at the correct rate. Overall, we observe that RC3 in implementation continues to provide gains proportional to what we expect from simulation.

While these graphs show the result for New Reno, we repeated these experiments using TCP CUBIC and the FCTs matched very closely to New Reno, since both have the same slow start behavior.

Endhost Correctness. Priority queuing is widely deployed in the OS networking stack, NICs, and switches, but is often unused. We now verify the correctness of the prio queuing discipline in Linux. We performed our experiments with iPerf [65] using the default NIC buffer size of 512 packets and with segment offload enabled to achieve a high throughput of 9.5Gbps. All packets in an iPerf flow were assigned the same priority level – this experiment *does not* use RC3 itself. All flows being sent to a particular destination port were marked as priority 1 by changing the DSCP



(a) Comparing FCTs for Regular TCP with RC3 (b) Zooming in to observe trends for RC3 FCT

Figure 2.18: FCTs for Regular TCP and RC3 with TSO/LRO (20ms×10Gbps)

field in the IP header. We connected two endhosts directly, with one acting as the iPerf client sending simultaneous flows to the connected iPerf server (a) with a low priority flow beginning after a high priority flow has begun, and (b) with a high priority flow beginning after a low priority flow has begun. Figure 2.16 shows that the priority queuing discipline behaves as expected.

Switch Correctness. We extended our topology to connect three endhosts to the switch, two of which acted as iPerf clients, sending simultaneous flows as explained above to the third endhost acting as the iPerf server. Since the two senders were on different machines, prioritization was done by the switch. Figure 2.17 shows that priority queuing at the switch behaves as expected.

Segment and Receiver Offload. In §2.3.2 we discussed how RC3 interacts with segment and receiver offload; we now evaluate the performance of RC3 when combined with these optimizations. For this experiment, we used the same set up, as our baseline and sent a 1000 packet flow without TSO/LRO, with each enabled independently, and with both TSO and LRO enabled. Figure 2.18 plots the corresponding FCTs excluding the connection set-up time.

For baseline TCP, we see that TSO and LRO each cause a performance penalty in our test scenario. TSO hurts TCP because the increased throughput also increases the number of segments being ACKed with one single delayed ACK, thus slowing the rate at which the congestion window increases. LRO aggravates the same problem by coalescing packets in the receive queue, once again leading them to be ACKed as a batch.

In contrast, RC3's FCTs improve when RC3 is combined with TSO and LRO. TSO and LRO do little to change the performance of RC3, when enabled independently, but when combined they allow chunks of packets to be processed together in batches at the receiver. This reduces the overhead of packet processing by almost 50%, resulting in better overall FCTs.

2.5 Discussion

Deployment Incentives. For RC3 to be widely used requires ISPs to opt-in by enabling priority queuing that already exists in their switches. As discussed in the introduction, we believe that giving worse service, rather than better service, for these low priority packets alleviates some of the concerns that has made QoS so hard to offer (in the wide area) today. WQoS is safe and backwards

compatible because regular traffic will never be penalized and pricing remains unaffected. Moreover, since RC3 makes more efficient use of bandwidth, it allows providers to run their networks at higher utilization, while still providing good performance, resulting in higher return in investment for their network provisioning.

Partial Support. Our simulations assume that all switches support multiple priorities. If RC3 is to be deployed, it must be usable even when the network is in a state of partial deployment, where some providers but not all support WQoS. When traffic crosses from a network which supports WQoS to a network which does not, a provider has two options: either drop all low priority packets before they cross in to the single-priority domain (obviating the benefits of RC3), or allow the low priority packets to pass through (allowing the packets to subsequently compete with normal TCP traffic at high priority). Simulating this latter scenario, we saw that average FCTs still improved for all flows, from using RC3 when 20% of switches did not support priorities; when 50% of switches did not support priorities small flows experienced a 6-7% FCT penalty, medium-sized flows saw slightly weaker FCT reductions (around 36%), and large flows saw slightly stronger FCT reductions (76-70%).

Middleboxes. Middleboxes which keep tight account of in-flight packets and TCP state are a rare but growing attribute of today's networks. These devices directly challenge the deployment of new protocols; resolving this challenge for proposals like RC3 and others remains an open area of research [56, 36, 110, 97, 105].

Datacenters. As we've shown via model (§2.2) and simulation (§2.4), the benefits of RC3 are strongest in networks with large bandwidth-delay product. Today's datacenter networks typically do not fit this description: with microsecond latencies, bandwidth-delay product is small and thus flows today can very quickly reach 100% link utilization. Nevertheless, given increasing bandwidth, bandwidth-delay product may not remain small forever. In simulations on a fat-tree datacenter topology with (futuristic) 100Gbps links, we observed average FCT improvements of 45% when averaged over flows, and 66% when averaged over bytes. Thus, while RC3 is not a good fit for datacenters today, it may be in the future.

Future. Outside of the datacenter, bandwidth-delay product is already large – and increasing. While increasing TCP's initial congestion window may mitigate the problem in the short term, given the inevitable expansion of available bandwidth, the problem will return again and again with any new choice of new initial congestion window. Our solution, while posing some deployment hurdles, has the advantage of being able to handle future speeds without further modifications.

2.6 Related Work

Switch-assisted Congestion Control. Observing TCP's sometimes poor ability to ensure high link utilization, some have moved away from TCP entirely, designing protocols which use explicit signaling for bandwidth allocation. RCP [34] and XCP [72] are effective protocols in this space.

Along similar lines, TCP QuickStart [39] uses an alternate slow-start behavior, which actively requests available capacity from the switches using a new IP Option during the TCP handshake. Using these explicitly supplied rates, a connection can skip slow start entirely and begin sending at its allocated rate immediately following the TCP handshake. Unlike RC3, these algorithms require new switch capabilities.

Alternate TCP Designs. There are numerous TCP designs that use alternative congestion avoidance algorithms to TCP New Reno [20, 38, 143, 128, 49]. TCP CUBIC [49] and Compound TCP [128] are deployed in Linux and Windows respectively. Nevertheless, their slow-start behaviors still leave substantial wasted capacity during the first few RTTs – consequently, they could just as easily be used in RC3's primary control loop as TCP New Reno. Indeed, in our implementation we also evaluated TCP CUBIC in combination with RC3.

TCP FastStart [100] targets back-to-back connections, allowing a second connection to re-use cached Cwnd and RTT data from a prior connection. TCP Remy [141] uses machine learning to generate the congestion control algorithm to optimize a given objective function, based on prior knowledge or assumptions about the network. RC3 improves flow completion time even from cold start and without requiring any prior information about the network delay, bandwidth or other parameters.

TCP-Nice [136] and TCP-LP [75] try to utilize the excess bandwidth in the network by using more aggressive back-off algorithms for the low-priority background traffic. RC3 also makes use of the excess bandwidth, but by explicitly using priority queues, with a very different aim of reducing the flow completion time for all flows.

Use of Low Priorities. pFabric [11] is a contemporary proposal for datacenters that also uses many layers of priorities and ensures high utilization. However, unlike RC3, pFabric's flow scheduling algorithm is targeted exclusively at the datacenter environment.

2.7 Conclusion

We presented *recursively cautious congestion control* (RC3), which uses multiple levels of priority to ensure safe, fair sharing of network resources while allowing flows to make full use of link capacity from the first RTT, without probing. RC3 can be deployed using today's infrastructure. In common wide-area scenarios, RC3 results in over 40% reduction in average flow completion times. The benefits of using RC3 are likely to increase in future, as the bandwidth-delay product in the networks are pushed to larger values.

Chapter 3

TIMELY:RTT-based Datacenter Congestion Control

In this chapter we present TIMELY, our second example that shows how we can avoid the need for changing or configuring network switches; this time in the context of datacenter congestion control.

Datacenter networks run tightly-coupled computing tasks that must be responsive to users, e.g., thousands of back-end computers may exchange information to serve a user request, and all of the transfers must complete quickly enough to let the complete response to be satisfied within 100 ms [46]. To meet these requirements, datacenter transports must simultaneously deliver high bandwidth ($\gg$Gbps) and utilization at low latency ($\ll$msec), even though these aspects of performance are at odds. Consistently low latency matters because even a small fraction of late operations can cause a ripple effect that degrades application performance [29]. As a result, datacenter transports must strictly bound latency and packet loss.

Since traditional loss-based transports do not meet these strict requirements, new datacenter transports [8, 9, 140, 57, 102, 68], take advantage of network switch support to signal the onset of congestion (e.g., DCTCP [8] and its successors use ECN), introduce flow abstractions to minimize completion latency, cede scheduling to a central controller, and more. However, in this work we take a step back in search of a simpler, immediately deployable design.

The crux of our search is the congestion signal. An ideal signal would have several properties. It would be fine-grained and timely to quickly inform senders about the extent of congestion. It would be discriminative enough to work in complex environments with multiple traffic classes. And, it would be easy to deploy.

Surprisingly, we find that a well-known signal, properly adapted, can meet all of our goals: delay in the form of RTT measurements. RTT is a fine-grained measure of congestion that comes with every acknowledgment. It effectively supports multiple traffic classes by providing an inflated measure for lower-priority transfers that wait behind higher-priority ones. Further, it requires no support from network switches.

Delay has been explored in the wide-area Internet since at least TCP Vegas [20], and some modern TCP variants use delay estimates [127, 138]. But this use of delay has not been with-

out problems. Delay-based schemes tend to compete poorly with more aggressive, loss-based schemes, and delay estimates may be wildly inaccurate due to host and network issues, e.g., delayed ACKs and different paths. For these reasons, delay is typically used in hybrid schemes with other indicators such as loss.

Moreover, delay has not been used as a congestion signal in the datacenter because datacenter RTTs are difficult to measure at microsecond granularity. This level of precision is easily overwhelmed by host delays such as interrupt processing for acknowledgments. DCTCP eschews a delay-based scheme saying "the accurate measurement of such small increases in queuing delay is a daunting task."[8]

Our insight is that recent NIC advances do allow datacenter RTTs to be measured with sufficient precision, while the wide-area pitfalls of using delay as a congestion signal do not apply. Recent NICs provide hardware support for high-quality timestamping of packet events [133, 139, 33, 23, 87], plus hardware-generated ACKs remove unpredictable host response delays. Meanwhile, datacenter host software can be controlled to avoid competition with other transports, and multiple paths have similar, small propagation delays.

In this chapter, we show that delay-based congestion control provides excellent performance in the datacenter:

1. We experimentally demonstrate how multi-bit RTT signals measured with NIC hardware are strongly correlated with network queuing (§3.1).
2. We present *Transport Informed by MEasurement of LatencY* (TIMELY): an RTT-based congestion control scheme (§3.2 and §3.3). TIMELY uses rate control and is designed to work with NIC offload of multi-packet segments for high performance. Unlike earlier schemes [138, 20], we do not build the queue to a fixed RTT threshold. Instead, we use the rate of RTT variation, or the *gradient*, to predict the onset of congestion and hence keep the delays low while delivering high throughput.
3. We evaluate TIMELY with an OS-bypass messaging implementation using hundreds of machines on a Clos network topology (§3.4 and §3.5). Turning on TIMELY for RDMA transfers on a fabric with PFC (Priority Flow Control) lowers 99 percentile tail latency by $9X$. This tail latency is $13X$ lower than that of DCTCP running in an optimized kernel.

3.1 Value of RTT as a congestion signal in datacenters

Existing datacenter transports use signals from network switches to detect the onset of congestion and run with low levels of latency and loss [8, 134, 9]. We argue that network queuing delay derived from RTT measurements, without the need for any switch support, is a superior congestion signal.

RTT directly reflects latency. RTTs are valuable because they directly measure the quantity we care about: end-to-end latency inflated by network queuing. Signals derived from queue occupancy such as ECN fail to directly inform this metric. An ECN mark on a packet simply indicates that the queue measure corresponding to the packet exceeds a threshold. Rich use of QoS in the datacenter

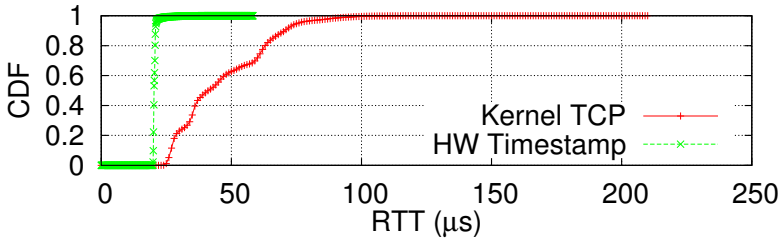


Figure 3.1: RTTs measured by hardware timestamps have a much smaller random variance than that by kernel TCP stack.

means it is not possible to convert this threshold into a single corresponding latency. Multiple queues with different priorities share the same output link, but the ECN mark only provides information about those with occupancy exceeding a threshold. Low priority traffic can experience large queuing delays without necessarily building up a large queue. In such circumstances, queuing delay reflects the state of congestion in the network which is not reflected by queue occupancy of low priority traffic. Further, an ECN mark describes behavior at a single switch. In a highly utilized network, congestion occurs at multiple switches, and ECN signals cannot differentiate among them. The RTT accumulates information about the end-to-end path. It includes the NIC, which may also become congested but is ignored by most schemes.

RTT can be measured accurately in practice. A key practical hurdle is whether RTTs can be measured accurately in datacenters where they are easily 1000X smaller than wide-area latencies. Many factors have precluded accurate measurement in the past: variability due to kernel scheduling; NIC performance techniques including offload (GSO/TSO, GRO/LRO); and protocol processing such as TCP delayed acknowledgements. This problem is severe in datacenters where each of these factors is large enough to overwhelm propagation and queuing delays.

Fortunately, recent NICs provide hardware support to solve these problems [133, 139, 33, 23, 87] and can accurately record the time of packet transmission and reception without being affected by software-incurred delays. These methods must be used with care lest they overly tax the NIC. We describe our use of NIC support later in this chapter. These NICs also provide hardware-based acknowledgements for some protocols.

The combination of these features lets us take precise RTT measurements to accurately track end-to-end network queues. The following experiment shows this behavior: we connect two hosts to the same network via 10 Gbps links and send 16 KB ping-pong messages without any cross traffic and on a quiescent network. Since there is no congestion, we expect the RTT measurements to be low and stable. Figure 3.1 compares the CDF of RTTs measured using NIC hardware timestamps and RTTs measured via the OS TCP stack. The CDF for RTTs using NIC timestamps is nearly a straight line, indicating small variance. In contrast, the RTTs measured by kernel TCP are larger and much more variable.

RTT is a rapid, multi-bit signal. Network queuing delay can be calculated by subtracting known propagation and serialization delays from an RTT. Therefore, assuming it is accurate, a single high

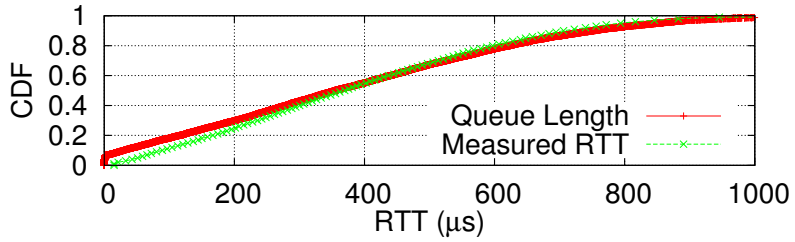


Figure 3.2: RTTs measured at end-system track closely the queue occupancy at congested link.

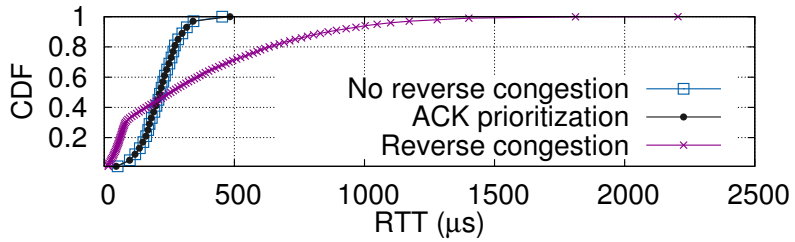


Figure 3.3: In the presence of reverse congestion, RTT measurements with ACK prioritization are indistinguishable from RTTs that do not experience any reverse path congestion.

RTT measurement immediately signals the extent of congestion. This RTT measurement works even with packet bursts for a flow sent along a single network path: the RTT of the last packet in the burst tracks the maximum RTT across packets since delays to earlier packets push out later packets. To show how well RTT tracks network queuing delay, we set up an incast experiment with 100 flows on 10 client machines simultaneously transmitting to a single server. To incorporate NIC offload, we send 64 KB messages and collect only a single RTT measurement per message on the client side. The bottleneck link is a 10 Gbps link to the server. We sample the switch queue each microsecond. Figure 3.2 shows the CDF of RTTs as measured at the end system compared to the queue occupancy measured directly at the switch and shown in units of time computed for a 10 Gbps link. The two CDFs match extremely well.

Limitations of RTTs. While we have found the RTT signal valuable, an effective design must use it carefully. RTT measurements lump queuing in both directions along the network path. This may confuse reverse path congestion experienced by ACKs with forward path congestion experienced by data packets. One simple fix is to send ACKs with higher priority, so that they do not incur significant queuing delay. This method works in the common case of flows that predominantly send data in one direction; we did not need more complicated methods.

We conducted an experiment to verify the efficacy of ACK prioritization: we started two incasts (*primary* and *secondary*) such that the ACKs of the primary incast share the same congested queue as the data segments of the secondary incast. Figure 3.3 shows the CDF of RTTs from the primary incast for the following three cases: 1) no congestion in ACK path (no secondary incast); 2) in the presence of congestion in ACK path; and 3) ACKs from primary incast are prioritized to higher

QoS queue in the presence of reverse congestion. We find that the reverse congestion creates noise in RTT measurements of the primary incast and elongates the tail latencies. Throughput of the primary incast is also impacted (and hence the smaller RTTs in the lower percentiles). With ACK prioritization, the RTTs measured in the primary incast are indistinguishable from those measured in the absence of reverse path congestion.

In future work, it would be straightforward to calculate variations in one-way delay between two packets by embedding a timestamp in a packet (e.g., TCP timestamps). The change in queuing delay is then the change in the arrival time minus send time of each packet. This method needs only clocks that run at the same rate, which is a much less stringent requirement than synchronized clocks.

The other classic shortcoming of RTTs is that changing network paths have disparate delays. It is less of a problem in datacenters as all paths have small propagation delays.

3.2 TIMELY Framework

TIMELY provides a framework for rate control that is independent of the transport protocol used for reliability. Figure 3.4 shows its three components: 1) RTT measurement to monitor the network for congestion; 2) a computation engine that converts RTT signals into target sending rates; and 3) a control engine that inserts delays between segments to achieve the target rate. We implement TIMELY in host software with NIC hardware support, and run an independent instance for each flow.

3.2.1 RTT Measurement Engine

We assume a traditional transport where the receiver explicitly ACKs new data so that we may extract an RTT. We define the RTT in terms of Figure 3.5, which shows the timeline of a message: a segment consisting of multiple packets is sent as a single burst and then ACKed as a unit by the receiver. A *completion event* is generated upon receiving an ACK for a segment of data and includes the ACK receive time. The time from when the first packet is sent (t_{send}) until the ACK is received ($t_{completion}$) is defined as the *completion time*. Unlike TCP, there is one RTT for the set of packets rather than one RTT per 1-2 packets. There are several delay components: 1) the serialization delay to transmit all packets in the segment, typically up to 64 KB; 2) the round-trip wire delay for the segment and its ACK to propagate across the datacenter; 3) the turnaround time at the receiver to generate the ACK; and 4) the queuing delay at switches experienced in both directions.

We define the RTT to be the propagation and queuing delay components only. The first component is a deterministic function of the segment size and the line rate of the NIC. We compute and subtract it from the total elapsed time so that the RTTs input to TIMELY's rate computation engine are independent of segment size. The third component is sufficiently close to zero in our setting with NIC-based ACKs that we can ignore it. Of the remaining components, the second is the propagation delay including the packet-based store-and-forward behavior at switches. It is the

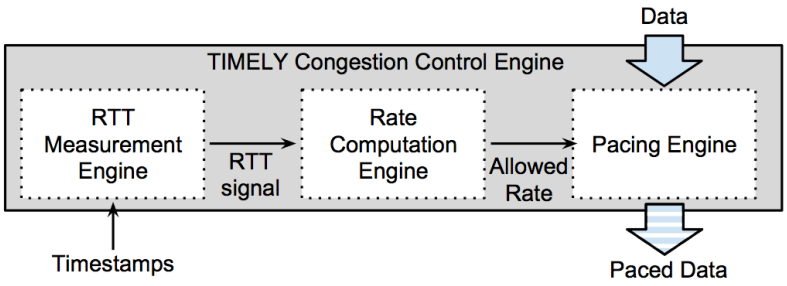


Figure 3.4: TIMELY overview.

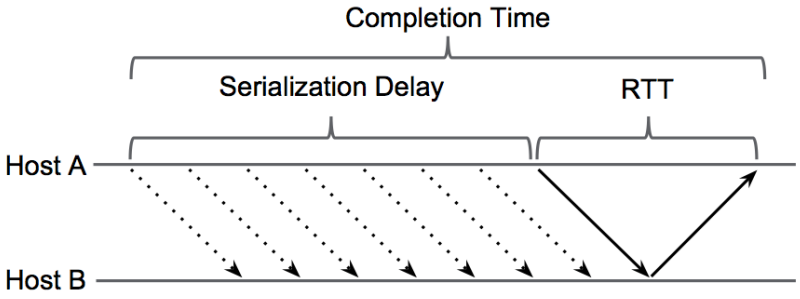


Figure 3.5: Finding RTT from completion time.

minimum RTT and fixed for a given flow. Only the last component – the queuing delay – causes variation in the RTT, and it is our focus for detecting congestion. In summary, TIMELY running on Host A (shown in Figure 3.5) computes RTT as:

$$RTT = t_{completion} - t_{send} - \frac{\text{seg. size}}{\text{NIC line rate}}$$

For context, in a 10 Gbps network, serialization of 64 KB takes 51 μs, propagation may range from 10-100 μs, and 1500 B of queuing takes 1.2 μs.

We rely on two forms of NIC support to precisely measure segment RTTs, described next.

ACK Timestamps. We require the NIC to supply the completion timestamp, $t_{completion}$. As shown in §3.1, OS timestamps suffer from variations such as scheduling and interrupts that can easily obscure the congestion signal. t_{send} is the NIC hardware time that's read by host software just before posting the segment to NIC.

Prompt ACK generation. We require NIC-based ACK generation so that we can ignore the turnaround time at the receiver. An alternative would be to use timestamps to measure the ACK turnaround delay due to host processing delay. We have avoided this option because it would require augmenting transport wire formats to include this difference explicitly.

Fortunately, some modern NICs [133, 139, 33, 23, 87] provide one or both features, and our requirements are met naturally with a messaging implementation that timestamps segment acknowledgement. A specific implementation of TIMELY in the context of RDMA is described in §3.4. We believe our design is more generally applicable to TCP with some care to work with

the batching behavior of the NIC, correctly associate an ACK with the reception of new data, and compensate for ACK turnaround time.

3.2.2 Rate Computation Engine

This component implements our RTT-based congestion control algorithm as detailed in §3.3. The interface to the rate computation engine is simple. Upon each completion event, the RTT measurement engine provides the RTT in microseconds to the rate computation engine. While this is the only required input, additional timing information could also be useful, e.g., the delay incurred in the NIC. There is no requirement for packet-level operation; in normal operation we expect a single completion event for a segment of size up to 64 KB due to NIC offload. The rate computation engine runs the congestion control algorithm upon each completion event, and outputs an updated target rate for the flow.

3.2.3 Rate Control Engine

When a message is ready to be sent, the rate control engine breaks it into segments for transmission, and sends each segment to the scheduler in turn. For runtime efficiency, we implement a single scheduler that handles all flows. The scheduler uses the segment size, flow rate (provided by the rate computation engine), and time of last transmission to compute the send time for the current segment with the appropriate pacing delay. The segment is then placed in a priority queue in the scheduler. Segments with send times in the past are serviced in round-robin fashion; segments with future send times are queued. After the pacing delay has elapsed, the rate control engine passes the segment to the NIC for immediate transmission as a burst of packets. Data is first batched into 64 KB segments, following which the scheduler computes the pacing delay to insert between two such batched segments. Note that 64 KB is the maximum batching size and is not a requirement, e.g., the segment sizes for a flow that only has small messages to exchange at any given time will be smaller than 64 KB. We later present results for segment sizes smaller than 64 KB as well.

TIMELY is rate-based rather than window-based because it gives better control over traffic bursts given the widespread use of NIC offload. The bandwidth-delay product is only a small number of packet bursts in datacenters, e.g., 51 μs at 10 Gbps is one 64 KB message. In this regime, windows do not provide fine-grained control over packet transmissions. It is easier to directly control the gap between bursts by specifying a target rate. As a safeguard, we limit the volume of outstanding data to a static worst-case limit.

3.3 TIMELY Congestion Control

Our congestion control algorithm runs in the rate computation engine. In this section, we describe our environment and key performance metrics, followed by our gradient-based approach and algorithm.

3.3.1 Metrics and Setting

The datacenter network environment is characterized by many bursty message workloads from tightly-coupled forms of computing over high bandwidth, low-latency paths. It is the opposite of the traditional wide-area Internet in many respects. Bandwidth is plentiful, and it is flow completion time (e.g., for a Remote Procedure Call (RPC)) that is the overriding concern. For short RPCs, the minimum completion time is determined by the propagation and serialization delay. Hence, we attempt to minimize any queuing delay to keep RTTs low. The latency tail matters because application performance degrades when even a small fraction of the packets are late [29]. Consistent low-latency implies low queuing delay and near zero packet loss, since recovery actions may greatly increase message latency. Longer RPCs will have larger completion times because of the time it takes to transmit more data across a shared network. To keep this added time small, we must maintain high aggregate throughput to benefit all flows and maintain approximate fairness so that no one flow is penalized.

Our primary metrics for evaluation are tail (99th percentile) RTT and aggregate throughput, as they determine how quickly we complete short and long RPCs (assuming some fairness). When there is a conflict between throughput and packet RTT, we prefer to keep RTT low at the cost of sacrificing a small amount of bandwidth. This is because bandwidth is plentiful to start with, and increased RTT directly impacts the completion times of short transfers. In effect, we seek to ride the throughput/latency curve to the point where tail latency becomes unacceptable. Secondary metrics are fairness and loss. We report both as a check rather than study them in detail. Finally, we prefer a stable design over higher average, but oscillating rates for the sake of predictable performance.

3.3.2 Delay Gradient Approach

Delay-based congestion control algorithms such as FAST TCP [138] and Compound TCP [127] are inspired by the seminal work of TCP Vegas [20]. These interpret RTT increase above a baseline as indicative of congestion: they reduce the sending rate if delay is further increased to try and maintain buffer occupancy at the bottleneck queue around some predefined threshold. However, Kelly et al. [74] argue that it is not possible to control the queue size when it is shorter in time than the control loop delay. This is the case in datacenters where the control loop delay of a 64 KB message over a 10 Gbps link is at least 51 μs, and possibly significantly higher due to competing traffic, while one packet of queuing delay lasts 1 μs. The most any algorithm can do in these circumstances is to control the *distribution* of the queue occupancy. Even if controlling the queue size were possible, choosing a threshold for a datacenter network in which multiple queues can be a bottleneck is a notoriously hard tuning problem.

TIMELY's congestion controller achieves low latencies by reacting to the *delay gradient* or derivative of the queuing with respect to time, instead of trying to maintain a standing queue. This is possible because we can accurately measure differences in RTTs that indicate changes in queuing delay. A positive delay gradient due to increasing RTTs indicates a rising queue, while a negative gradient indicates a receding queue. By using the gradient, we can react to queue growth

Algorithm 1 TIMELY congestion control.

Input: new_rtt
Result: Enforced rate
new_rtt_diff = new_rtt - prev_rtt
prev_rtt = new_rtt
rtt_diff = (1 - α) · rtt_diff + α · new_rtt_diff ▷ α: EWMA weight parameter
normalized_gradient = rtt_diff / minRTT
if new_rtt < T_{low} **then**
    rate ← rate + δ ▷ δ: additive increment step
    return;
end if
if new_rtt > T_{high} **then**
    rate ← rate · $\left(1 - \beta \cdot \left(1 - \frac{T_{high}}{new\_rtt}\right)\right)$ ▷ β: multiplicative decrement factor
    return;
end if
if normalized_gradient ≤ 0 **then**
    rate ← rate + N · δ
        ▷ N = 5 if gradient<0 for five completion events (HAI mode); otherwise N = 1
else
    rate ← rate · (1 - β · normalized_gradient)
end if

without waiting for a standing queue to form – a strategy that helps us achieve low latencies.

Delay gradient is a proxy for the rate mismatch at the bottleneck queue. We are inspired by RCP, XCP, PI, and QCN [34, 72, 55, 68] that find explicit feedback on the rate mismatch has better stability and convergence properties than explicit feedback based only on queue sizes; the latter can even cause the queue to be *less* accurately controlled. The key difference is that all of these prior controllers operate at point queues in the network, while TIMELY achieves similar benefits by using the end-to-end delay gradient.

The model we assume is N end hosts all sending data at a total rate $y(t)$ into a bottleneck queue with drain rate C, i.e. the outgoing rate is $\leq C$. We denote the queuing delay through the bottleneck queue by $q(t)$. If $y(t) > C$, the rate at which the queue builds up is $(y(t) - C)$. Since queued data drains at a rate C, the queuing delay gradient is given by $\frac{dq(t)}{dt} = \frac{(y(t)-C)}{C}$. The gradient is dimensionless. It is positive for $y(t) > C$ and signals how quickly the queue is building. The negative gradient when $y(t) < C$, signals how quickly the queue is draining. Hence, the delay gradient measured through RTT signals acts as an indicator for the rate mismatch at the bottleneck. This reasoning holds as long as there is some non-zero queue in the network. When there is zero queuing or queues are not changing in size, the measured gradient is also zero. TIMELY strives to match the aggregate incoming rate $y(t)$ to the drain rate, C, and so adapts its per-connection rate, $R(t)$, in proportion to the measured error of $\frac{(y(t)-C)}{C} = \frac{dq(t)}{dt} = \frac{d(RTT)}{dt}$.

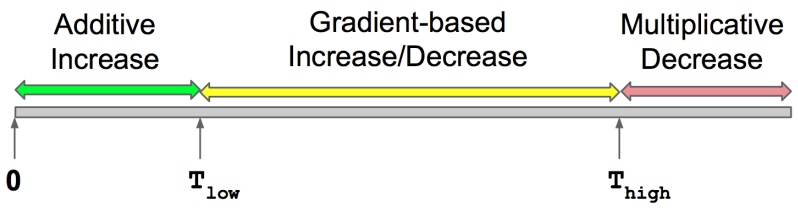


Figure 3.6: Gradient tracking zone with low and high RTT thresholds.

3.3.3 The Main Algorithm

Algorithm 1 shows pseudo-code for our congestion control algorithm. TIMELY maintains a single rate $R(t)$ for each connection and updates it on every completion event using RTT samples. It employs gradient tracking, adjusting the rate using a smoothed delay gradient as the error signal to keep throughput close to the available bandwidth. Additionally, we employ thresholds to detect and respond to extreme cases of under utilization or overly high packet latencies. Figure 3.6 shows the gradient zone along with the two thresholds. When the RTT is in the nominal operating range, the gradient tracking algorithm computes the delay gradient from RTT samples and uses it to adjust the sending rate.

Computing the delay gradient. We rely on accurate RTT measurements using NIC timestamps (§3.2). To compute the delay gradient, TIMELY computes the difference between two consecutive RTT samples. We normalize this difference by dividing it by the minimum RTT, obtaining a dimensionless quantity. In practice, the exact value of the minimum RTT does not matter since we only need to determine if the queue is growing or receding. We therefore use a fixed value representing the wire propagation delay across the datacenter network, which is known ahead of time. Finally, we pass the result through an EWMA filter. This filter allows us to detect the overall trend in the rise and fall in the queue, while ignoring minor queue fluctuations that are not indicative of congestion.

Computing the sending rate. Next, TIMELY uses the normalized gradient to update the target rate for the connection. If the gradient is negative or equals zero, the network can keep up with the aggregate incoming rate, and therefore there is room for a higher rate. In this case, TIMELY probes for more bandwidth by performing an additive increment for the connection: $R = R + \delta$, where δ is the bandwidth additive increment constant. When the gradient is positive, the total sending rate is greater than network capacity. Hence, TIMELY performs a multiplicative rate decrement β, scaled by the gradient factor:

$$R = R\left(1 - \beta \frac{d(RTT(t))}{dt}\right)$$

The delay gradient signal, which is based on the total incoming and outgoing rates, is common for all connections along the same congested path. The well-known AIMD property ensures that our algorithm achieves fairness across connections [24]. Connections sending at a higher rate observe a stronger decrease in their rate, while the increase in rate remains same for all connections.

While the delay gradient is effective in normal operation, situations with significant under-

utilization or high latency require a more aggressive response. Next we discuss how TIMELY detects and responds to these situations.

Need for RTT low threshold T_{low}. The ideal environment for our algorithm is one where packets are perfectly paced. However, in practical settings, the TIMELY rate is enforced on a segment granularity that can be as large as 64 KB. These large segments lead to packet bursts, which result in transient queues in the network and hence RTT spikes when there is an occasional collision. Without care, the core algorithm would detect a positive gradient due to a sudden RTT spike and unnecessarily infer congestion and back-off. We avoid this behavior by using a low threshold T_{low} to filter RTT spikes; the adjustment based on delay gradient kicks in for RTT samples greater than the threshold. T_{low} is a (nonlinear) increasing function of the segment size used in the network, since larger messages cause more bursty queue occupancy. We explore this effect in our evaluation, as well as show how fine-grained pacing at the hosts can reduce burstiness and hence the need for a low threshold.

Need for RTT high threshold T_{high}. The core gradient algorithm maintains close to the bottleneck link throughput while building very little queue. However, in theory, it is possible for the gradient to stay at zero while the queue remains at a high, fixed level. To remove this concern, T_{high} serves as an upper bound on the tolerable end-to-end network queue delay, i.e., the tail latency. It provides a way to reduce the rate independent of gradient value if the latency grows, a protection that is possible because we operate in a datacenter environment with known characteristics. If the measured RTT is greater than T_{high}, we reduce the rate multiplicatively:

$$R = R\left(1 - \beta\left(1 - \frac{T_{high}}{RTT}\right)\right)$$

Note that we use the instantaneous rather than smoothed RTT. While this may seem unusual, we can slow down in response to a single overly large RTT because we can be confident that it signals congestion, and our priority is to maintain low packet latencies and avoid loss. We tried responding to average RTT as a congestion indicator, and found that it *hurts* packet latency because of the extra delay in the feedback loop. By the time the average rose, and congestion control reduces the rate, queuing delay has already increased in the network. Our finding is inline with [54] which shows through a control theoretic analysis that the averaged queue length is a failing of RED AQM. We show in §3.5 how T_{high} lets us ride to the right along the throughput-delay tradeoff curve.

Hyperactive increase (HAI) for faster convergence. Inspired by the *max probing* phase in TCP BIC, CUBIC [49] congestion control, and QCN [68], we include an *HAI* option for faster convergence as follows: if TIMELY does not reach the new fair share after a period of slow growth, i.e., the gradient is negative for several consecutive completion times, then HAI switches to a faster additive increase in which the rate is incremented by $N\delta$ instead of δ. This is useful when the new fair share rate has dramatically increased due to reduced load in the network.

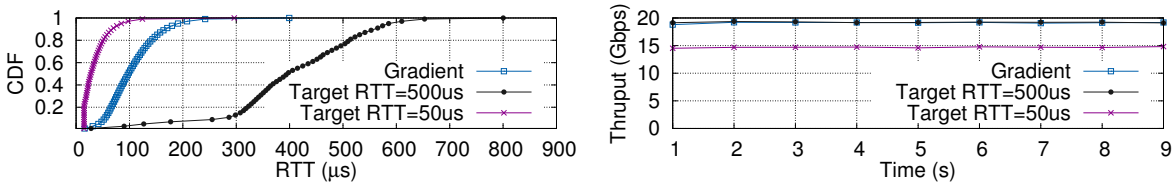


Figure 3.7: Comparison of gradient (low and high thresholds of 50 μs and 500 μs) with target-based approach (T_{target} of 50 μs and 500 μs).

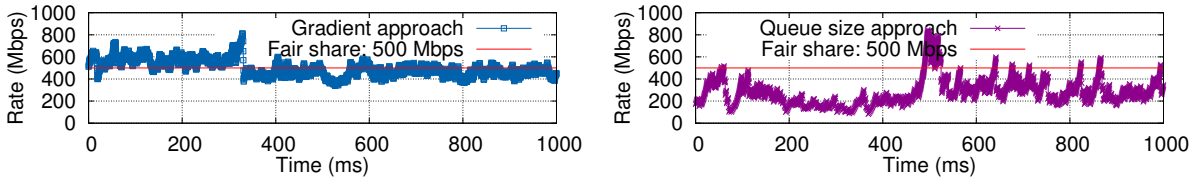


Figure 3.8: Per-connection rates in the gradient approach are smooth (top) while those in the queue-size based approach (with T_{target} = 50μs) are more oscillatory (bottom).

3.3.4 Gradient versus Queue Size

We highlight through an experiment how the gradient approach differs from a queue size based scheme. If we set the same value for T_{low} and T_{high}, then TIMELY congestion control reduces to a queue size based approach (similar to TCP FAST algorithm; FAST in turn is a scaled, improved version of Vegas). Denote T_{target} as the value of this single RTT threshold, i.e., $T_{target} = T_{low} = T_{high}$. Then the rate is increased additively and decreased multiplicatively with the decrease factor scaled by the queue size above T_{target}.

Figure 3.7 compares the RTT and throughput for the gradient and queue size based approach for an incast traffic pattern. (See §3.5 for experimental details.) We use low and high thresholds of 50 μs and 500 μs for gradient, versus T_{target} of 50 μs and 500 μs for the queue-sized approach. We see that the queue size approach can maintain either low latency or high throughput, but finds it hard to do both. By building up a standing queue up to a high T_{target} of 500 μs, throughput is optimized, but at the cost of latency due to queuing. Alternatively, by keeping the standing queue at a low T_{target} of 50 μs, latency is optimized, but throughput suffers as the queue is sometimes empty. By operating on the rising and falling queue, the gradient approach predicts the onset of congestion. This lets it deliver the high throughput of a high queue target while keeping the tail latency close to that of a low target.

Furthermore, as shown in Figure 3.8, the connection rates oscillate more in the queue-size approach, as it drives the RTT up and down towards the target queue size. The gradient approach maintains a smoother rate around the fair share. Similar results are shown in control theory terms for AQMs using the queue-size approach and gradient approach [55].

The main take-away is that T_{low} and T_{high} thresholds effectively bring the delay within a target range and play a role similar to the target queue occupancy in many AQM schemes. Using the delay gradient improves stability and helps keep the latency within the target range.

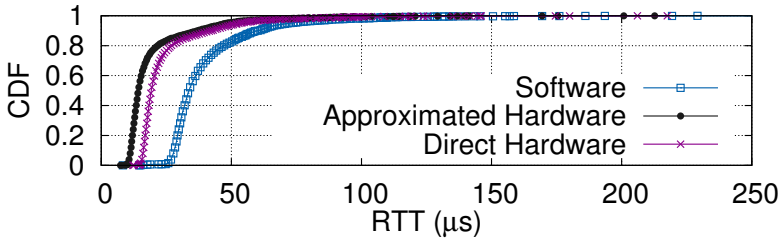


Figure 3.9: Comparison of the accuracy of NIC and SW timestamps.

3.4 Implementation

Our implementation is built on 10 Gbps NICs with OS-bypass capabilities. The NICs support multi-packet segments with hardware-based ACKs and timestamps. We implemented TIMELY in the context of RDMA (Remote Direct Memory Access) as a combination of NIC functionality and host software. We use RDMA primitives to invoke NIC services and offload complete memory-to-memory transfers to the NIC. In particular, we mainly use RDMA Write and Read to take a message from local host memory and send it on the network as a burst of packets. On the remote host, the NIC acknowledges receipt of the complete message and places it directly in the remote memory for consumption by the remote application. The local host is notified of the acknowledgement when the transfer is complete. We describe below some of the notable points of the implementation.

Transport Interface. TIMELY is concerned with the congestion control portion of the transport protocol; it is not concerned with reliability or the higher-level interface the transport exposes to applications. This allows the interface to the rest of the transport to be simple: message send and receive. When presented with a message at the sender, TIMELY breaks it into smaller segments if it is large and sends the segments at the target rate. A message is simply an ordered sequence of bytes. The segment is passed to the NIC and then sent over the network as a burst of packets. On the remote host, the NIC acknowledges receipt of the complete segment. At the receiver, when a segment is received it is passed to the rest of the transport for processing. This simple model supports transports ranging from RPCs to bytestreams such as TCP.

Using NIC Completions for RTT Measurement. In practice, using NIC timestamps is challenging. Our NIC only records the absolute timestamp of when the multi-packet operation finishes and therefore our userspace software needs to record a timestamp of when the operation was posted to the NIC. This requires a scheme to map host clock to NIC clock, as well as calibration. We record host (CPU) timestamps when posting work to the NIC and build a calibration mechanism to map NIC timestamps to host timestamps. A simple linear mapping is sufficient. The mechanism works well because the probability of being interrupted between recording the host send timestamp and actually handing the message to the NIC is fairly low. Figure 3.9 compares RTTs obtained from NIC HW timestamps, the calibration mechanism, and pure software only timestamps. Note that TIMELY does not spin, so interrupts and wakeups are included in the software timestamp num-

bers. It clearly demonstrates that the calibration mechanism is just as accurate as using only NIC timestamps. Furthermore, the SW timestamps have a large variance, which increases as load on the host increases.

We consider any NIC queuing occurring to be part of the RTT signal. This is important because NIC queuing is also indicative of congestion and is handled by the same rate-based controls as network queuing — even if the NIC were to give us an actual send timestamp, we would want the ability to observe NIC queuing.

RDMA rate control. For RDMA Writes, TIMELY on the sender directly controls the segment pacing rate. For RDMA Reads, the receiver issues read requests, in response to which the remote host performs a DMA of the data segments. In this case, TIMELY cannot directly pace the data segments, but instead achieves the same result by pacing the read requests: when computing the pacing delay between read requests, the rate computation engine takes into account the data segment bytes read from the remote host.

Application limited behavior. Applications do not always have enough data to transmit for their flows to reach the target rate. When this happens, we do not want to inadvertently increase the target rate without bound because the network appears to be uncongested. To prevent this problem, we let the target rate increase only if the application is sending at more than 80% of the target rate, and we also cap the maximum target rate at 10 Gbps. The purpose of allowing some headroom is to let the application increase its rate without an unreasonable delay when it does have enough data to send.

Rate update frequency. TIMELY's rate update equation assumes that there is at most one completion event per RTT interval. The transmission delay of a 64 KB message on a 10 Gbps link is 51 μs; with a minimum RTT of 20 μs, there can be at most one completion event in any given minimum RTT interval. However, for small segment sizes, there can be multiple completion events within a minimum RTT interval. In such a scenario, we want to update the rate based on the most recent information. We do so by updating the rate for every completion event, taking care to scale the updates by the number of completions per minimum RTT interval so that we do not overweigh the new information.

For scheduler efficiency, the rate control engine enforces rate updates lazily. When the previously computed send time for a segment elapses, the scheduler checks the current rate. If the rate has decreased, we recompute the send time, and re-queue the segment if appropriate. Otherwise, the scheduler proceeds to send the segment to the NIC.

Additional pacing opportunities. By default, the NIC sends a segment as a burst of packets at the link line rate. We explore another possibility: using NIC rate-limiters to transmit a burst of packets at less than the line rate. The rationale is to supplement the pacing engine that spreads packets of the flow over time with NIC-sized units of work. With hardware rate-limiters [112], it is feasible to offload part of the responsibility for pacing to the NIC. However due to hardware constraints, re-configuring pacing rate every few RTTs is not always feasible. Instead, we use a hybrid approach: software pacing of large segments and hardware pacing at fixed rate below the

link rate, e.g. 5 Gbps on a 10 Gbps link. At these high-rates, the purpose of NIC pacing is to insert gaps in the burst so that multiple bursts mix at switches without causing latency spikes. In this case, the rate control engine compensates for the NIC pacing delays by treating it as a lower transmission line rate.

3.5 Evaluation

We evaluate a real host-based implementation of TIMELY at two scales. First, we examine the basic properties of the congestion controller such as throughput, fairness, packet latency, and timing accuracy in an incast setting. For these microbenchmarks, we use a small-scale testbed with a rack of equipment. Second, we run TIMELY on a larger scale testbed of a few hundred machines in a classic Clos network topology [5, 118]. Along with running the traffic workload, hosts collect measurements of per-connection throughputs, RPC latencies, and RTTs (we established in §3.1 that host RTTs correspond well with queuing delays measured at the switches). All links are 10 Gbps unless mentioned otherwise. The OS used in all experiments is Linux.

To place our results in context, we compare TIMELY with two alternatives. First, we use OS-bypass messaging over a fabric with Priority Flow Control (PFC) as commonly used for low loss and latency in FCoE, e.g. DCB [28]. The RDMA transport is in the NIC and sensitive to packet drops, so PFC is necessary because drops hurt performance badly. We add TIMELY to this RDMA setting to observe its benefits; we check that pause message counts are low to verify that there is sufficient switch buffering for TIMELY to work and PFC is not an inadvertent factor in our experimental results. Second, we compare against an optimized kernel stack that implements DCTCP [8] running on the same fabric without the use of PFC. We choose DCTCP as a point of comparison because it is a well-known, modern datacenter transport that has been deployed and proven at scale.

Henceforth we refer to: 1) *DCTCP*, for kernel DCTCP over a fabric without PFC; 2) *PFC*, for OS-bypass messaging over a fabric with PFC; 3) *TIMELY*, for OS-bypass messaging with TIMELY over a fabric with PFC.

Unless mentioned otherwise, we use the following parameters for TIMELY: segment size of 16 KB, T_{low} of 50 μs, T_{high} of 500 μs, additive increment of 10 Mbps, and a multiplicative decrement factor (β) of 0.8.

3.5.1 Small-Scale Experiments

We use an incast traffic pattern for small-scale experiments (unless otherwise specified) since it is a key congestion scenario for datacenter networks [135]. To create incast, 10 client machines on a single rack send to a single server on the same rack. Each client runs 4 connections, i.e., 40 total concurrent connections. Each connection sends 16 KB segments at a high enough aggregate rate to saturate the server bandwidth of 2x10G link which is the bottleneck for the experiment. This is a demanding workload for testing congestion control: while there are many connections present in the datacenter, the number of connections limited by network capacity is normally small.

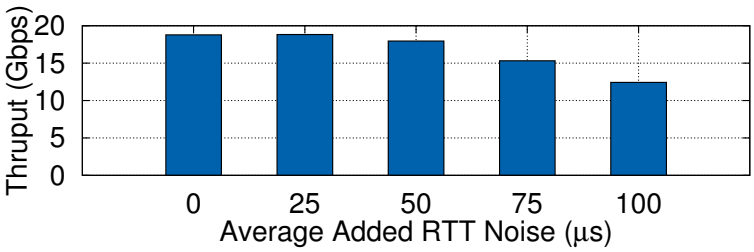


Figure 3.10: Impact of RTT noise on TIMELY throughput.

Metric	DCTCP	PFC	TIMELY
Total Throughput (Gbps)	19.5	19.5	19.4
Avg. RTT (us)	598	658	61
99-percentile RTT (us)	1490	1036	116

Table 3.1: Overall performance comparison with DCTCP and PFC

Required RTT measurement accuracy. To evaluate the accuracy of RTT samples required by TIMELY, we add noise to the measured RTTs and observe the impact on throughput. We add random noise uniformly distributed in the range of $[0,x]$ μs to each RTT sample, where x is set to 0, 50, 100, 150, 200. Figure 3.10 shows the total throughput measured on the server at different noise levels. Average noise of 50 μs causes visible throughput degradation, and higher noise leads to more severe performance penalties. A T_{low} value lower than 50 μs lowers the tolerance to RTT noise even further. Note that this level of noise is easily reachable by software timestamping (due to scheduling delays, coalescing, aggregation, etc.). Hence, accurate RTT measurement provided by NIC support is the cornerstone of TIMELY.

Comparison with PFC. The last two columns in Table 3.1 compare TIMELY with OS-bypass messaging over a fabric with conventional RDMA deployment over PFC. While the throughput is slightly lower with TIMELY, the median and tail RTTs are lower by more than order of magnitude and pauses are not triggered at all. To quantify the fairness of bandwidth allocation across connections, we also compute a Jain fairness index [67] of 0.953 for TIMELY and 0.909 for PFC. Our design is more fair, and has a high enough index to meet our fairness needs.

Comparison with DCTCP. We next compare TIMELY with DCTCP. To review briefly: with DCTCP senders emit ECN capable packets; switches mark each packet queued beyond a fixed threshold; receivers return all ECN marks to the sender; and senders adapt their rate based on the fraction of packets with ECN marks in a window. Note that of necessity our comparison is for two different host software stacks, as DCTCP runs in an optimized kernel without PFC support whereas TIMELY is used with OS-bypass messaging. We did not implement DCTCP in OS-bypass environment due to NIC firmware limitations on processing ECN feedback [146]. We set the

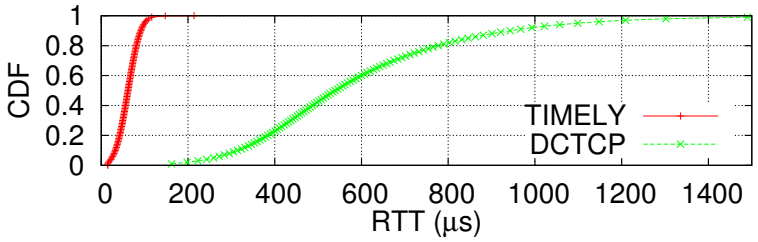


Figure 3.11: CDF of RTT distribution

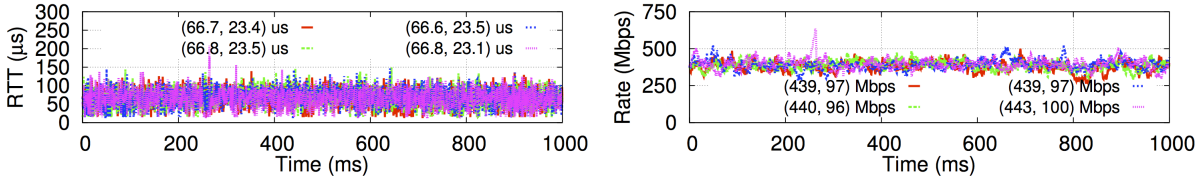


Figure 3.12: RTTs and sending rates of a sample of connections for TIMELY. The legend gives the mean and standard deviation for each connection.

switch ECN marking threshold to $K = 80$ KB. This is less than the DCTCP author recommendation of $K = 65$ packets for 10 Gbps operation as we are willing to sacrifice a small amount of throughput to ensure consistently low latency. The second and the last columns in Table 3.1 summarizes the results averaged across three runs of ten minutes, with the RTT distribution shown in Figure 3.11. TIMELY keeps the average end-to-end RTT 10X lower than DCTCP (60 μs vs. 600 μs). More significantly, the tail latency drops by almost 13X (116 μs vs. 1490 μs). No loss nor PFC packets were observed. These latency reductions do not come at the cost of throughput.

Per-session Performance. Next, we break out connections to show that TIMELY also delivers excellent performance for individual connections. Figure 3.12 shows a timeline of the observed RTT and throughput for a sample for four individual connections using TIMELY. Each datapoint represents a single completion event. The fair share for each connection is 500 Mbps. We see that the throughput is close to the fair share and the RTT remains consistently low.

Varying T_{low}. Our performance is influenced by algorithm parameters, which we explore starting with the low threshold. Our purpose is to highlight factors that affect the thresholds, not to tune parameters. By design no more than a default setting is necessary. The low threshold exists to absorb the RTT variation during uncongested network use. The expected variation is related to the maximum segment size, since as segments grow larger the effect on the RTT of occasional segment collisions increases. Figure 3.13 shows how the bottleneck throughput and RTT times vary with different values of T_{low} for segments of size 16 KB, 32 KB and 64 KB.

We see that decreasing the low threshold reduces the network delay. This is because a lower threshold allows the use of the RTT gradient more often to modulate the rate in response to queue build-ups. But lower thresholds eventually have an adverse effect on throughput. This is best seen

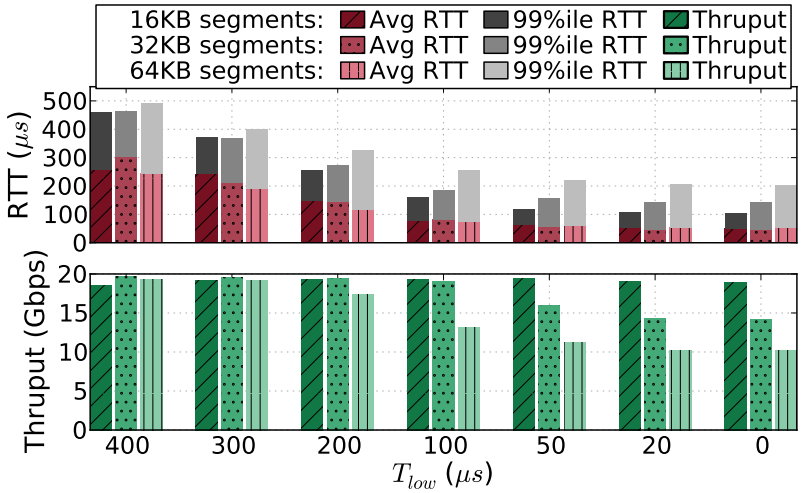


Figure 3.13: Throughput and RTT varying with T_{low} and segment size.

with bursty traffic. For 16 KB segment size, when the burstiness is relatively low, a T_{low} of just 50 μs gives us the highest throughput (19.4 Gbps), with the throughput being only slightly lower (18.9 Gbps) without any T_{low}. However, as we increase the segment size and hence the burstiness, the throughput falls quickly when the threshold becomes too small. For 32 KB segments, the tipping point is a T_{low} of 100 μs. For the most demanding workload of 64 KB segments, the transition is between 200–300 μs. Such large bursts make it difficult to obtain both high throughput and low delay. This is unsurprising since each segment is sent as a long series of back-to-back packets at wire speed, e.g., 64 KB is at least 40 packets.

Smoothing Bursts with Fine-Grained Pacers. The Rate Control Engine described in §3.2.3 introduces a pacing delay between segments. To further mitigate the burstiness with large segments of 64 KB, while still enabling NIC offload, we explore fine-grained pacing. In this model, in addition to the host software pacing segments, the NIC hardware uses pacing to send the packets in each segment at a configurable rate lower than line rate. Pacing allows packets to mix more easily in the network. Programmable NICs such as NetFPGA [130] allow a pacing implementation. Prior work such as HULL [9] has also made use of NIC pacing, and fine-grained pacing queuing disciplines such as FQ/pacing are in Linux kernels [132].

We repeat the incast experiment using 64 KB segments, this time with NIC pacing, with two values of T_{low}: 0 μs and 50 μs. We are not able to implement a dynamic fine-grained pacing rate at this time and so use static rates. When computing RTTs from the completion times, we subtract the serialization delay introduced by NIC pacers to allow for comparison, e.g., pacing a 64 KB message at 1 Gbps introduces a serialization delay of 512 μs. Figure 3.14 shows the results for different NIC pacing rates. As expected, the reduced burstiness due to pacing leads to increase in throughput and decrease in delay, with larger throughput increases for greater pacing. The most benefit is seen at 700 Mbps: 18.9 Gbps throughput for T_{low} = 50 μs and 18.4 Gbps in the absence of any T_{low} (as opposed to 11.2 Gbps and 10.2 Gbps respectively without any pacing). Note that

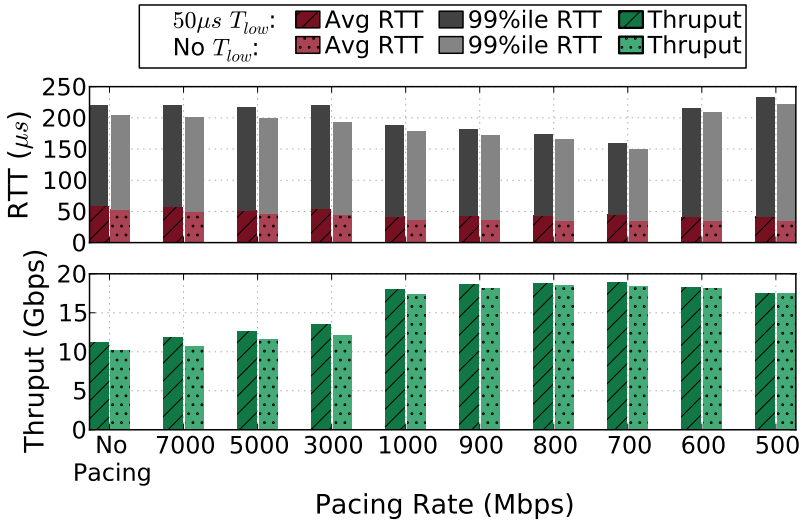


Figure 3.14: Throughput and RTT varying with pacing rate for 64 KB segments.

this also means single flow performance is capped at 700 Mbps, unless the pacing rate is adjusted dynamically. There is a slight dip in the throughput and rise in delay beyond this level, as pacing approaches the fair share and has a throttling effect.

We note that NIC hardware pacing is not an absolute requirement in TIMELY design; but rather helps navigate the tradeoff between lower network tail latency and higher CPU overhead that can be caused by software pacing with smaller segments.

Varying T_{high}. TIMELY employs a high threshold to react quickly to large RTTs. This threshold matters less than the low threshold because it only comes into play for large RTTs, but it becomes useful as the level of connection multiplexing grows and the likelihood of RTT spikes increase.

Figure 3.15 shows the effect on throughput as the high threshold is reduced for different numbers of competing connections. In our earlier runs with four connections per client, the 99-percentile RTTs are around 100 μs. This means that any $T_{high} > 100\mu$s has little effect. As the load climbs to 7 connections per client, the 99-percentile RTT settles close to 200 μs. Then there is a drop in RTT as we reduce T_{high} to 100 μs and below. For 10 connections per client, the 99-percentile RTTs remain close to 500 μs for T_{high} of 500 μs or more, and decrease as T_{high} falls. The throughput is quite good for T_{high} down to 100 μs, but is significantly lower at 50 μs for all three connection levels. These results show that a high threshold down to at most 200 μs helps to reduce tail latency without degrading throughput.

Hyper active increment (HAI). HAI helps to acquire available bandwidth more quickly. To show this, we perform an incast with a change in the offered load. The incast starts with 10 clients and 10 connections per client. After an initial period for convergence, every client simultaneously shuts down 9 of its 10 connections, thus increasing the fair share rate of the remaining connection by 10X. Figure 3.16 shows how HAI ramps up connection throughput from an initial fair rate of 200 Mbps to 1.5 Gbps within 50 ms, and reaches the new fair share of 2 Gbps in 100 ms. In

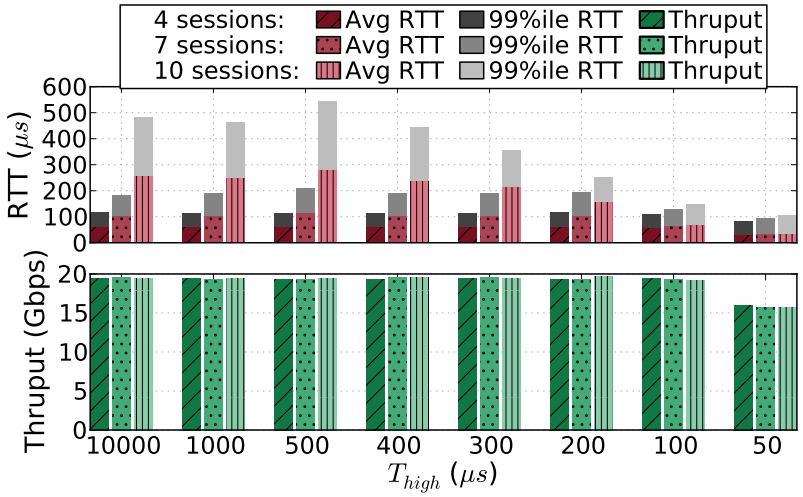


Figure 3.15: Total throughput and RTT varying with T_{high} for different number of connections. Segment Size = 16 KB, T_{low} = 50 μs.

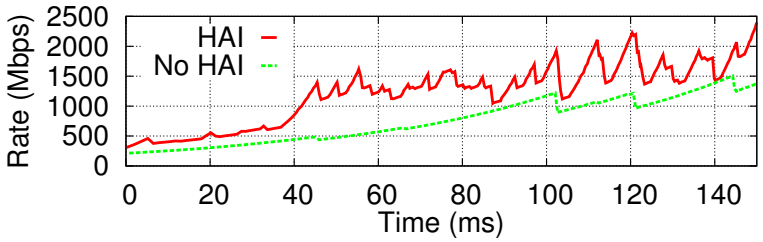


Figure 3.16: HAI quickly acquires available bandwidth.

contrast, a fixed additive increment only achieves 1.5 Gbps after 140 ms. We find that a HAI threshold of five successive RTTs strikes a good balance between convergence and stability.

3.5.2 Large-Scale Experiments

We investigate TIMELY's large-scale behavior with experiments run on a few hundred machines in a classic Clos topology [5, 118]. We show that TIMELY is able to maintain predictable and low latency in large all-to-all and incast network congestion scenarios. The experiment generates RPCs between client server pairs. To stress TIMELY and create increased burstiness, we used 64 KB RPCs and segment sizes.

Longest path uniform random. In this traffic pattern, a client picks a server from a set of servers with the longest path through the network. Clients issue 64 KB requests. The server replies with a payload of the same size. The benchmark collects goodput and RPC latency. RPC latency is computed at the client from the time the request is sent to the server, to when it receives a response from the server.

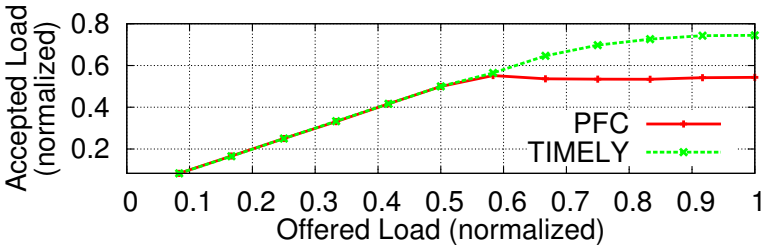


Figure 3.17: Accepted versus offered load (64 KB messages).

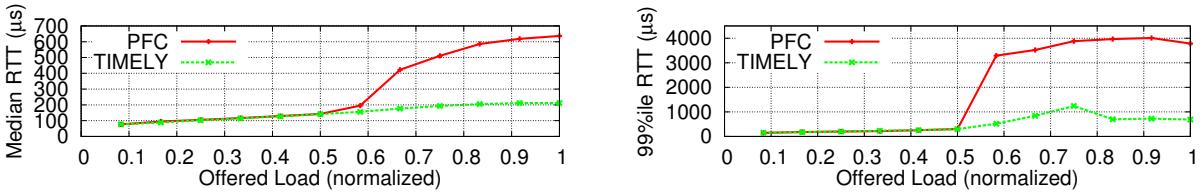


Figure 3.18: Median and 99-percentile RTTs as measured by pingers under different offered load.

Figure 3.17 shows normalized throughput (to the maximum offered load used in the experiment) as observed by the application for increasing offered loads on the x-axis. The saturation point of the network (the point at which accepted load is less than the offered load) is higher for TIMELY as it is able to send more traffic by minimizing queuing and thereby also pause frames per second.

Figure 3.18 shows RTT versus load. TIMELY reduces the median and 99-percentile RTT by 2X and 5X respectively compared to PFC. This results in a corresponding reduction of RPC median latency of about 2X (shown in Figure 3.19). Without TIMELY, beyond saturation the network queuing increases in an attempt to reach the offered load. With TIMELY, low network queuing is maintained by moving queuing from the shared network to the end-host (where it is included in RPC latency but not in the RTTs). Therefore, the 99-percentile of RPC latency reduction effect diminishes as the offered load increases beyond the saturation point.

Network imbalance (incast). To stress TIMELY's ability to mitigate congestion, we designed an experiment with a background load of longest path uniform random traffic and then added an incast load. We use three levels of background load: low (0.167 of the maximum offered load in Figure 3.17), medium (0.3) and high (0.5). Figure 3.20 shows the normalized throughput and 99-percentile RTT for this experiment. We normalize throughput to that of the background load. We know from Figure 3.17 that TIMELY and PFC throughput are the same for uniform random traffic before network saturation. When we add an incast, without TIMELY, throughput falls by 13% to 54%, depending on the background network load, primarily due to head of line blocking created by PFCs. This observation is confirmed with RTT measurements in Figure 3.20, which show that TIMELY is able to keep queuing relatively low, preventing congestion spreading [144, 6], by rate limiting only the flows passing along the congested path. The overall throughput for TIMELY remained the same during the incast.

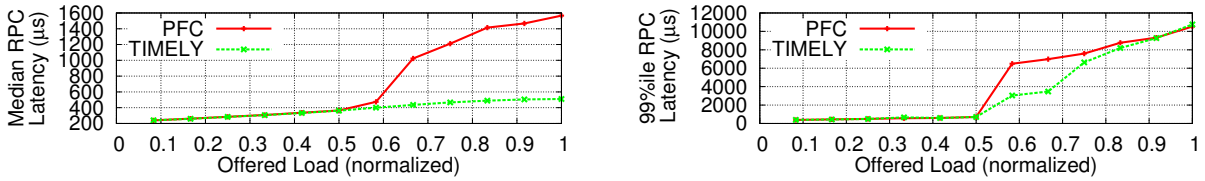


Figure 3.19: Median and 99-percentile RPC latencies.

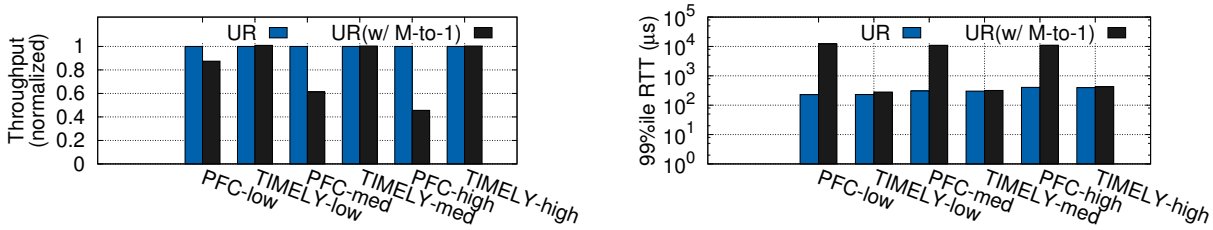


Figure 3.20: Adding one 40-to-1 pattern to longest path uniform random ((a) Normalized throughput (b) 99-percentile RTT).

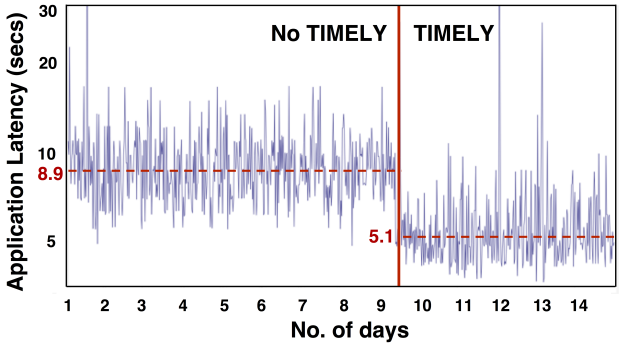


Figure 3.21: Application-level benchmark.

Application level benchmark. Figure 3.21 shows RPC latency of a datacenter storage benchmark (note that the y-axis is on a log-scale). Without TIMELY, the application is limited in the amount of data it can push through the network while keeping the 99th percentile RTT low enough. With TIMELY, the application is able to push at higher utilization levels without suffering negative latency consequences. Therefore, the drop in application data unit latency (in seconds) is really a reflection of the increased throughput that the application is able to sustain during query execution.

3.6 Related Work

Datacenter congestion control is a deeply studied topic [8, 9, 140, 57, 134, 11]. TIMELY focuses on the same problem.

RED [41] and CoDel [96] drop packets early, prompting senders to reduce transmission rates to avoid the large standing queues associated with tail drops. However, loss still drives up latencies for the flows that experience packet drops. To avoid packet drops, many schemes rely on switch-support in the form of ECN, in which packets are marked to indicate congestion [106]. ECN marks are often combined across multiple packets [8, 134, 9] to provide fine-grained congestion information, but our experiments in §3.1 show that ECN has inherent limitations. There have also been other proposals that rely on switch support to mitigate congestion such as QCN [59] (fine-grained queue occupancy information) and pFabric [11] (fine-grained prioritization).

TIMELY belongs to a different class of algorithms that use delay measurements to detect congestion, which requires no switch-support. We take inspiration from TCP Vegas, FAST, and Compound [20, 138, 127]. These proposals are window-based and maintain a queue close to the minimum RTT. In contrast, TIMELY is a rate-based algorithm that employs a gradient approach and does not rely on measuring the minimum RTT. We show that it works well with NIC support, despite infrequent RTT signals.

A recent scheme, DX [76], independently identified the benefits of using delay as congestion signal for high throughput and low latency datacenter communications. DX implements accurate latency measurements using a DPDK driver for the NIC and the congestion control algorithm is within the Linux TCP stack. DX algorithm is similar to the conventional window-based proposals, with an additive increase and a multiplicative decrease that's proportional to the average queuing delay.

CAIA Delay Gradient [50] (CDG) proposes a delay gradient algorithm for TCP congestion control for wide-area networks. Its key goal is to figure out co-existence with loss based congestion control. Hence the nature of its algorithms are different from those in TIMELY.

Link-layer flow control is used for low-latency messaging in Infiniband and Data Center Bridging (DCB) networks. However, problems with Priority Flow Control (PFC), including head of line blocking and pause propagation or congestion spreading, are documented in literature [144, 6]. Some recent proposals aim to overcome these issues with PFC using ECN markings to maintain low queue occupancy. TCP-Bolt [122] uses modified DCTCP algorithm within the kernel TCP stack. DCQCN [146] uses a combination of ECN markings with a QCN-inspired rate-based congestion control algorithm implemented in the NIC. Evaluations demonstrate that it addresses some of the HoL blocking and unfairness problems with PFC, thus making RoCE more viable for large-scale deployment. TIMELY uses RTT signal, is implemented in host software with support of NIC timestamping, and is applicable to both OS-bypass and OS-based transports. Comparison of TIMELY and DCQCN in terms of both congestion control and CPU utilization is an interesting future work.

Congestion can also be avoided by scheduling transmissions using a distributed approach [140, 57] or even a centralized one [102]. However, such schemes are yet to be proven at scale, and are more complex than a simple delay-based approach.

Finally, load-sensitive routing such as Conga [7] and FlowBender [69] can mitigate congestion hotspots by spreading traffic around the network, thereby increasing throughput. However, host-based congestion control is still required to match offered load to the network capacity.

3.7 Conclusion

Conventional wisdom considers delay to be an untrustworthy congestion signal in datacenters. Our experience with TIMELY shows the opposite – when delay is properly adapted, RTT strongly correlates with queue buildups in the network. We built TIMELY, which takes advantage of modern NIC support for timestamps and fast ACK turnaround to perform congestion control based on precise RTT measurements. We found TIMELY can detect and respond to tens of microseconds of queuing to deliver low packet latency and high throughput, even in the presence of infrequent RTT signals and NIC offload. As datacenter speeds scale up by an order of magnitude, future work should focus on how effective RTTs continue to be for congestion control, alongside rethinking the nature of delay based algorithms.

Chapter 4

Revisiting Network Support for RDMA

In this chapter, we discuss our third and final example on how good end-point based solutions can eliminate the need for complicated in-network mechanisms, this time in the context of deploying RDMA in datacenters.

Datacenter networks offer higher bandwidth and lower latency than traditional wide-area networks. However, traditional endhost networking stacks, with their high latencies and substantial CPU overhead, have limited the extent to which applications can make use of these characteristics. As a result, several large datacenters have recently adopted RDMA, which bypasses the traditional networking stacks in favor of direct memory accesses.

RDMA over Converged Ethernet (RoCE) has emerged as the canonical method for deploying RDMA in Ethernet-based datacenters [146, 47]. The centerpiece of RoCE is a NIC that (i) provides mechanisms for accessing host memory without CPU involvement and (ii) supports very basic network transport functionality. Early experience revealed that RoCE NICs only achieve good end-to-end performance when run over a lossless network, so operators turned to Ethernet's Priority Flow Control (PFC) mechanism to achieve minimal packet loss. The combination of RoCE and PFC has enabled a wave of datacenter RDMA deployments.

However, the current solution is not without problems. In particular, PFC adds management complexity and can lead to significant performance problems such as head-of-the-line blocking, congestion spreading, and occasional deadlocks [146, 47, 122, 58, 115]. Rather than continue down the current path and address the various problems with PFC, we take a step back and ask whether it was needed in the first place. To be clear, current RoCE NICs require a lossless fabric for good performance. However, the question we raise is: *can the RoCE NIC design be altered so that we no longer need a lossless network fabric?*

We answer this question in the affirmative, proposing a new design called IRN (for Improved RoCE NIC) that makes two incremental changes to current RoCE NICs (i) more efficient loss recovery, and (ii) basic end-to-end flow control to bound the number of in-flight packets (§4.2). We show, via extensive simulations on a RoCE simulator obtained from a commercial NIC vendor, that IRN performs better than current RoCE NICs, and that IRN does not require PFC to achieve high performance; in fact, IRN often performs better without PFC (§4.3). We detail the extensions to the RDMA protocol that IRN requires (§4.4) and use comparative analysis and FPGA synthesis

to evaluate the overhead that IRN introduces in terms of NIC hardware resources (§4.5). Our results suggest that adding IRN functionality to current RoCE NICs would add as little as 3-10% overhead in resource consumption, with no deterioration in message rates.

A natural question that arises is how IRN compares to iWARP? iWARP [108] long ago proposed a similar philosophy as IRN: handling packet losses efficiently in the NIC rather than making the network lossless. What we show is that iWARP's failing was in its design choices. The differences between iWARP and IRN designs stem from their starting points: iWARP aimed for full generality which led them to put the full TCP/IP stack on the NIC, requiring multiple layers of translation between RDMA abstractions and traditional TCP bytestream abstractions. As a result, iWARP NICs are typically far more complex than RoCE ones, with higher cost and lower performance (§4.1). In contrast, IRN starts with the much simpler design of RoCE and asks what minimal features can be added to eliminate the need for PFC.

More generally: while the merits of iWARP vs. RoCE has been a long-running debate in industry, there is no conclusive or rigorous evaluation that compares the two architectures. Instead, RoCE has emerged as the de-facto winner in the marketplace, and brought with it the implicit (and still lingering) assumption that a lossless fabric is necessary to achieve RoCE's high performance. Our results are the first to rigorously show that, counter to what market adoption might suggest, iWARP in fact had the right architectural philosophy, although a needlessly complex design approach.

Hence, one might view IRN and our results in one of two ways: (i) a new design for RoCE NICs which, at the cost of a few incremental modifications, eliminates the need for PFC and leads to better performance, or, (ii) a new incarnation of the iWARP philosophy which is simpler in implementation and faster in performance.

4.1 Background

We begin with reviewing some relevant background.

4.1.1 Infiniband RDMA and RoCE

RDMA has long been used by the HPC community in special-purpose Infiniband clusters that use credit-based flow control to make the network lossless [61]. Because packet drops are rare in such clusters, the RDMA Infiniband transport (as implemented on the NIC) was not designed to efficiently recover from packet losses. When the receiver receives an out-of-order packet, it simply discards it and sends a negative acknowledgement (NACK) to the sender. When the sender sees a NACK, it retransmits all packets that were sent after the last acknowledged packet (i.e., it performs a go-back-N retransmission).

To take advantage of the widespread use of Ethernet in datacenters, RoCE [125, 126] was introduced to enable the use of RDMA over Ethernet.[1] RoCE adopted the same Infiniband transport

[1] We use the term RoCE for both RoCE [125] and its successor RoCEv2 [126] that enables running RDMA, not just over Ethernet, but also over IP-routed networks.

NIC	Throughput	Latency
Chelsio T-580-CR (iWARP)	3.24 Mpps	2.89 μs
Mellanox MCX416A-BCAT (RoCE)	14.7 Mpps	0.94 μs

Table 4.1: An iWARP and a RoCE NIC's raw performance for 64B RDMA Writes on a single queue-pair.
design (including go-back-N loss recovery), and the network was made lossless using PFC.

4.1.2 Priority Flow Control

Priority Flow Control (PFC) [60] is Ethernet's flow control mechanism, in which a switch sends a pause (or X-OFF) frame to the upstream entity (a switch or a NIC), when the queue exceeds a certain configured threshold. When the queue drains below this threshold, an X-ON frame is sent to resume transmission. When configured correctly, PFC makes the network lossless (as long as all network elements remain functioning). However, this coarse reaction to congestion is agnostic to *which* flows are causing it and this results in various performance issues that have been documented in numerous papers in recent years [146, 47, 122, 58, 115]. These issues range from mild (e.g.unfairness and head-of-line blocking) to severe, such as "pause spreading" as highlighted in [47] and even network deadlocks [122, 58, 115]. In an attempt to mitigate these issues, congestion control mechanisms have been proposed for RoCE (e.g.DCQCN [146] and TIMELY [91]) which reduce the sending rate on detecting congestion, but are not enough to eradicate the need for PFC. Hence, there is now a broad agreement that PFC makes networks harder to understand and manage, and can lead to myriad performance problems that need to be dealt with.

4.1.3 iWARP vs RoCE

iWARP [108] was designed to support RDMA over a fully general (i.e., not loss-free) network. iWARP implements the entire TCP stack in hardware along with multiple other layers that it needs to translate TCP's byte stream semantics to RDMA segments. Early in our work, we engaged with multiple NIC vendors and datacenter operators in an attempt to understand why iWARP was not more broadly adopted (since we believed the basic architectural premise underlying iWARP was correct). The consistent response we heard was that iWARP is significantly more complex and expensive than RoCE, with inferior performance [109].

We also looked for empirical datapoints to validate or refute these claims. We ran RDMA Write benchmarks on two machines connected to one another, using Chelsio T-580-CR 40Gbps iWARP NICs on both machines for one set of experiments, and Mellanox MCX416A-BCAT 56Gbps RoCE NICs (with link speed set to 40Gbps) for another. Both NICs had similar specifications, and at the time of purchase, the iWARP NIC cost $760, while the RoCE NIC cost $420. Raw NIC performance values for 64 bytes batched Writes on a single queue-pair are reported in Table 4.1. We find that iWARP has 3$\times$ higher latency and 4$\times$ lower throughput than RoCE.

These price and performance differences could be attributed to many factors other than transport design complexity (such as differences in profit margins, supported features and engineering effort) and hence should be viewed as anecdotal evidence as best. Nonetheless, they show that our

conjecture (in favor of implementing loss recovery at the endhost NIC) was certainly not obvious based on current iWARP NICs.

Our primary contribution is to show that iWARP, somewhat surprisingly, did in fact have the right philosophy: explicitly handling packet losses in the NIC leads to better performance than having a lossless network. However, efficiently handling packet loss does not require implementing the entire TCP stack in hardware as iWARP did. Instead, we identify the incremental changes to be made to current RoCE NICs, leading to a design which (i) does not require PFC yet achieves better network-wide performance than both RoCE and iWARP (§4.3), and (ii) is much closer to RoCE's implementation with respect to both NIC performance and complexity (§4.5) and is thus significantly less complex than iWARP.

4.2 IRN Design

We begin with describing the transport logic for IRN. For simplicity, we present it as a general design independent of the specific RDMA operation types. We go into the details of handling specific RDMA operations with IRN later in §4.4.

Changes to the RoCE transport design may introduce overheads in the form of new hardware logic or additional per-flow state. With the goal of keeping such overheads as small as possible, IRN strives to make *minimal* changes to the RoCE NIC design in order to eliminate its PFC requirement, as opposed to squeezing out the best possible performance with a more sophisticated design (we evaluate the small overhead introduced by IRN in §4.5).

IRN, therefore, makes two key changes to current RoCE NICs, as described in the following subsections: (1) improving the loss recovery mechanism, and (2) basic end-to-end flow control (termed *BDP-FC*) which bounds the number of in-flight packets by the bandwidth-delay product of the network. We justify these changes by empirically evaluating their significance, and exploring some alternative design choices in §4.3.3. Note that these changes are orthogonal to the use of explicit congestion control mechanisms (such as DCQCN [146] and TIMELY [91]) that, as with current RoCE NICs, can be *optionally* enabled with IRN.

4.2.1 Improved Loss Recovery

As discussed in §4.1, current RoCE NICs use a go-back-N loss recovery scheme. In the absence of PFC, redundant retransmissions caused by go-back-N loss recovery result in significant performance penalties (as evaluated in §4.3). Therefore, the first change we make with IRN is a more efficient loss recovery, based on selective retransmission (inspired by TCP's loss recovery), where the receiver does not discard out of order packets and the sender selectively retransmits the lost packets, as detailed below.

Upon every out-of-order packet arrival, an IRN receiver sends a NACK, which carries both the cumulative acknowledgment (indicating its expected sequence number) and the sequence number of the packet that triggered the NACK (as a simplified form of selective acknowledgement or SACK).

An IRN sender enters loss recovery mode when a NACK is received or when a timeout occurs. It also maintains a bitmap to track which packets have been cumulatively and selectively acknowledged. When in the loss recovery mode, the sender selectively retransmits lost packets as indicated by the bitmap, instead of sending new packets. The first packet that is retransmitted on entering loss recovery corresponds to the cumulative acknowledgement value. Any subsequent packet is considered lost only if another packet with a higher sequence number has been selectively acked. When there are no more lost packets to be retransmitted, the sender continues to transmit new packets (if allowed by BDP-FC). It exits loss recovery when a cumulative acknowledgement greater than the *recovery sequence* is received, where the recovery sequence corresponds to the last regular packet that was sent before the retransmission of a lost packet.

SACKs allow efficient loss recovery only when there are multiple packets in flight. For other cases (e.g.for single packet messages), loss recovery gets triggered via timeouts. A high timeout value can increase the tail latency of such short messages. However, keeping the timeout value too small can result in too many spurious retransmissions, affecting the overall results. An IRN sender, therefore, uses a low timeout value of RTO_{low} only when there are a small N number of packets in flight (such that spurious retransmissions remains negligibly small), and a higher value of RTO_{high} otherwise. We discuss how the values of these parameters are set in §4.3, and how the timeout feature in current RoCE NICs can be easily extended to support this in §4.5.

4.2.2 BDP-based Flow Control

The second change we make with IRN is introducing the notion of a basic end-to-end packet level flow control, called BDP-FC, which bounds the number of outstanding packets in flight for a flow by the bandwidth-delay product (BDP) of the network, as suggested in [11]. This is a static cap that we compute by dividing the BDP of the longest path in the network (in bytes) [2] with the packet MTU set by the RDMA queue-pair (typically 1KB in RoCE NICs). An IRN sender transmits a new packet only if the number of packets in flight (computed as the difference between current packet's sequence number and last acknowledged sequence number) is less than this BDP cap.

BDP-FC improves the performance by reducing unnecessary queuing in the network. Furthermore, by strictly upper bounding the number of out-of-order packet arrivals, it greatly reduces the amount of state required for tracking packet losses in the NICs (discussed in more details in §4.5).

As mentioned before, IRN's loss recovery has been inspired by TCP's loss recovery. However, rather than incorporating the entire TCP stack as is done by iWARP NICs, IRN: (1) decouples loss recovery from congestion control and does not incorporate any notion of TCP congestion window control involving slow start, AIMD or advanced fast recovery, (2) operates directly on RDMA segments instead of using TCP's byte stream abstraction, which not only avoids the complexity introduced by multiple translation layers (as needed in iWARP), but also allows IRN to simplify

[2] As in [11], we expect this information to be available in a datacenter setting with known topology and routes. IRN does not require a fully precise BDP computation and over-estimating the BDP value would still provide the required benefits to a large extent without under-utilizing the network.

its selective acknowledgement and loss tracking schemes. We discuss how these changes effect performance towards the end of §4.3.

4.3 Evaluating IRN's Transport Logic

We now confront the central question of this chapter: *Does RDMA require a lossless network?* If the answer is yes, then we must address the many difficulties of PFC. If the answer is no, then we can greatly simplify network management by letting go of PFC. To answer this question, we evaluate the network-wide performance of IRN's transport logic via extensive simulations. Our results show that IRN performs better than RoCE, without requiring PFC. We test this across a wide variety of experimental scenarios and across different performance metrics. We end this section with a simulation-based comparison of IRN with Resilient RoCE [116] and iWARP [108].

4.3.1 Experimental Settings

Simulator. Our simulator, obtained from a commercial NIC vendor, extends INET/OMNET++ [1, 2] to model the Mellanox ConnectX4 RoCE NIC [85]. RDMA queue-pairs (QPs) are modelled as UDP applications with either RoCE or IRN transport layer logic, that generate flows (as described later). We define a flow as a unit of data transfer comprising of one or more messages between the same source-destination pair as in [91, 146]. When the sender QP is ready to transmit data packets, it periodically polls the MAC layer until the link is available for transmission. The simulator implements DCQCN as implemented in the Mellanox ConnectX-4 ROCE NIC [116], and we add support for a NIC-based TIMELY implementation. All switches in our simulation are input-queued with virtual output ports, that are scheduled using round-robin. The switches can be configured to generate PFC frames by setting appropriate buffer thresholds.

Default Case Scenario. For our default case, we simulate a 54-server three-tiered fat-tree topology, connected by a fabric with full bisection-bandwidth constructed from 45 6-port switches organized into 6 pods [10]. We consider 40Gbps links, each with a propagation delay of 2μs, resulting in a bandwidth-delay product (BDP) of 120KB along the longest (6-hop) path. This corresponds to $\sim$110 MTU-sized packets (assuming typical RDMA MTU of 1KB).

Each end host generates new flows with Poisson inter-arrival times [11, 92]. Each flow's destination is picked randomly and size is drawn from a realistic heavy-tailed distribution derived from [15]. Most flows are small (50% of the flows are single packet messages with sizes ranging between 32 bytes-1KB representing small RPCs such as those generated by RDMA based key-value stores [32, 71]), and most of the bytes are in large flows (15% of the flows are between 200KB-3MB, representing background RDMA traffic such as storage). The network load is set at 70% utilization for our default case. We use ECMP for load-balancing [47]. We vary different aspects from our default scenario (including topology size, workload pattern and link utilization) in §4.3.4.

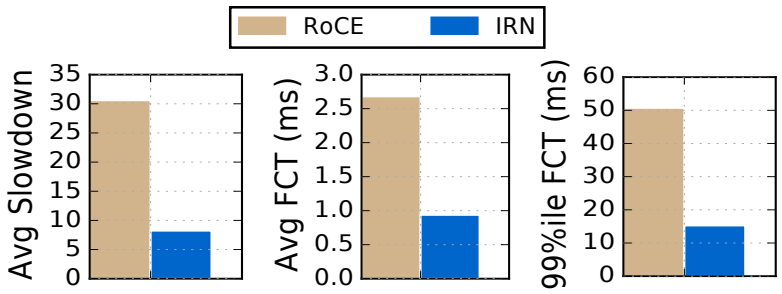


Figure 4.1: Comparing IRN and RoCE's performance.

Parameters. RTO_{high} is set to an estimation of the maximum round trip time with one congested link. We compute this as the sum of the propagation delay on the longest path and the maximum queuing delay a packet would see if the switch buffer on a congested link is completely full. This is approximately 320μs for our default case. For IRN, we set RTO_{low} to 100μs (representing the desirable upper-bound on tail latency for short messages) with N set to a small value of 3. When using RoCE without PFC, we use a fixed timeout value of RTO_{high}. We disable timeouts when PFC is enabled to prevent spurious retransmissions. We use buffers sized at twice the BDP of the network (which is 240KB in our default case) for each input port [11, 14]. The PFC threshold at the switches is set to the buffer size minus a headroom equal to the upstream link's bandwidth-delay product (needed to absorb all packets in flight along the link). This is 220KB for our default case. We vary these parameters in §4.3.4 to show that our results are not very sensitive to these specific choices. When using RoCE or IRN with TIMELY or DCQCN, we use the same congestion control parameters as specified in [91] and [146] respectively. For fair comparison with PFC-based proposals [146, 122], the flow starts at line-rate for all cases.

Metrics. We primarily look at three metrics: (i) average slowdown, where slowdown for a flow is its completion time divided by the time it would have taken to traverse its path at line rate in an empty network, (ii) average flow completion time (FCT), (iii) 99%ile or tail FCT. While the average and tail FCTs are dominated by the performance of throughput-sensitive flows, the average slowdown is dominated by the performance of latency-sensitive short flows.

4.3.2 Basic Results

We now present our basic results comparing IRN and RoCE for our default scenario. Unless otherwise specified, IRN is always used without PFC, while RoCE is always used with PFC for the results presented here.

IRN performs better than RoCE. We begin with comparing IRN's performance with current RoCE NIC's. The results are shown in Figure 4.1. IRN's performance is upto 2.8-3.7$\times$ better than RoCE across the three metrics. This is due to the combination of two factors: (i) IRN's BDP-FC mechanism reduces unnecessary queuing and (ii) unlike RoCE, IRN does not experience any congestion spreading issues, since it does not use PFC. (explained in more details below).

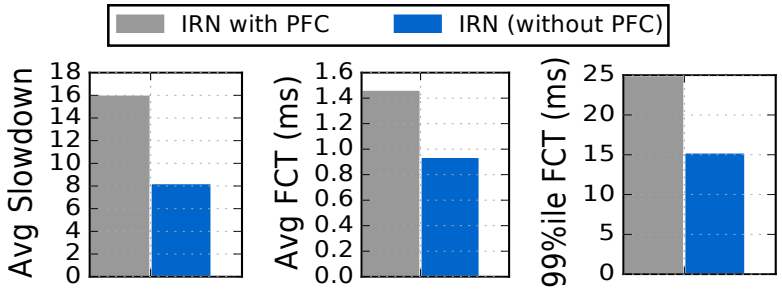


Figure 4.2: Impact of enabling PFC with IRN.

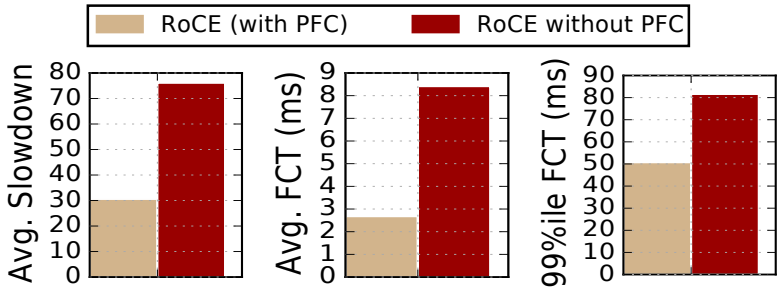


Figure 4.3: Impact of disabling PFC with RoCE.

IRN does not require PFC. We next study how IRN's performance is impacted by enabling PFC. If enabling PFC with IRN does not improve performance, we can conclude that IRN's loss recovery is sufficient to eliminate the requirement for PFC. However, if enabling PFC with IRN significantly improves performance, we would have to conclude that PFC continues to be important, even with IRN's loss recovery. Figure 4.2 shows the results of this comparison. Remarkably, we find that not only is PFC not required, but it significantly degrades IRN's performance (increasing the value of each metric by about 1.5-2×). This is because of the head-of-the-line blocking and congestion spreading issues PFC is notorious for: pauses triggered by congestion at one link, cause queue build up and pauses at other upstream entities, creating a cascading effect. Note that, without PFC, IRN experiences significantly high packet drops (8.5%), which also have a negative impact on performance, since it takes about one round trip time to detect a packet loss and another round trip time to recover from it. However, the negative impact of a packet drop (given efficient loss recovery), is restricted to the flow that faces congestion and does not spread to other flows, as in the case of PFC. While these PFC issues have been observed before [47, 146, 91], we believe our work is the first to show that *a well-design loss-recovery mechanism outweighs a lossless network*.

RoCE requires PFC. Given the above results, the next question one might have is whether RoCE required PFC in the first place? Figure 4.3 shows the performance of RoCE with and without PFC. We find that the use of PFC helps considerably here. Disabling PFC degrades performance by 1.5-3× across the three metrics. This is because of the go-back-N loss recovery used by current

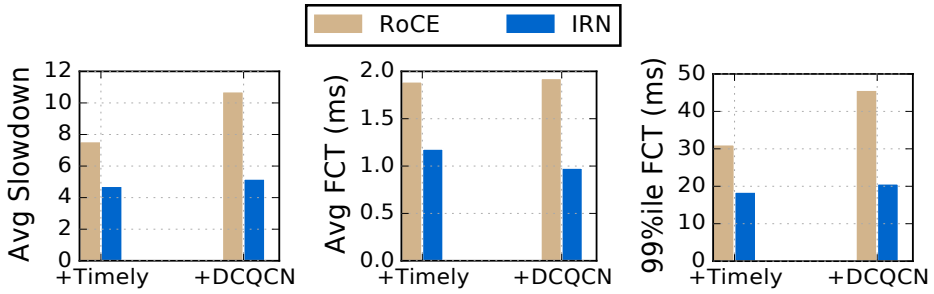


Figure 4.4: Comparing IRN and RoCE's performance with explicit congestion control (TIMELY and DCQCN).

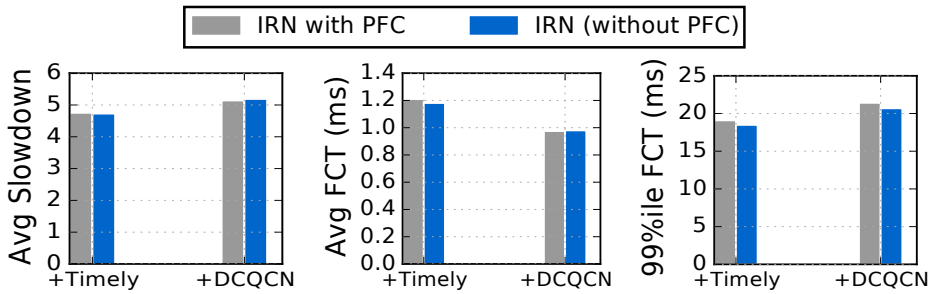


Figure 4.5: Impact of enabling PFC with IRN, when explicit congestion control (TIMELY and DCQCN) is used.

RoCE NICs, which penalizes performance due to (i) increased congestion caused by redundant retransmissions and (ii) the time and bandwidth wasted by flows in sending these redundant packets.

Effect of Explicit Congestion Control. The previous comparisons did not use any explicit congestion control. However, as mentioned before, RoCE today is typically deployed in conjunction with some explicit congestion control mechanism such as TIMELY or DCQCN. We now evaluate whether using such explicit congestion control mechanisms affect the key trends described above.

Figure 4.4 compares IRN and RoCE's performance when TIMELY or DCQCN is used. IRN continues to perform better by up to 1.5-2.2× across the three metrics.

Figure 4.5 evaluates the impact of enabling PFC with IRN, when TIMELY or DCQCN is used. We find that, IRN's performance is largely unaffected by PFC, since explicit congestion control reduces both the packet drop rate as well as the number of pause frames generated. The largest performance improvement due to enabling PFC was less than 1%, while its largest negative impact was about 3.4%.

Finally, Figure 4.6 compares RoCE's performance with and without PFC, when TIMELY or DCQCN is used.[3] We find that, unlike IRN, RoCE (with its inefficient go-back-N loss recovery)

[3]RoCE + DCQCN without PFC presented in Figure 4.6 is equivalent to Resilient RoCE [116]. We provide a direct comparison of IRN with Resilient RoCE later in this section.

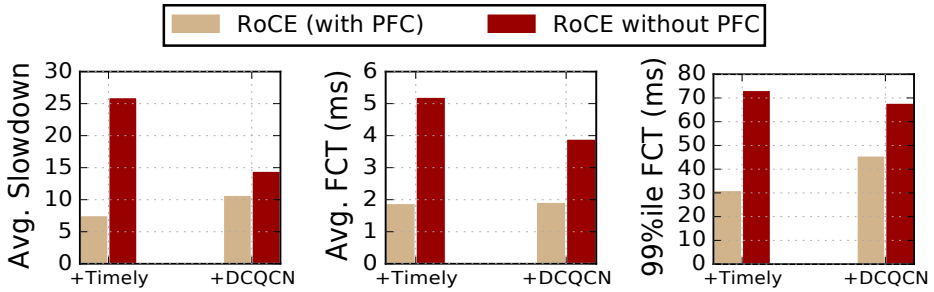


Figure 4.6: Impact of disabling PFC with RoCE, when explicit congestion control (TIMELY and DCQCN) is used.

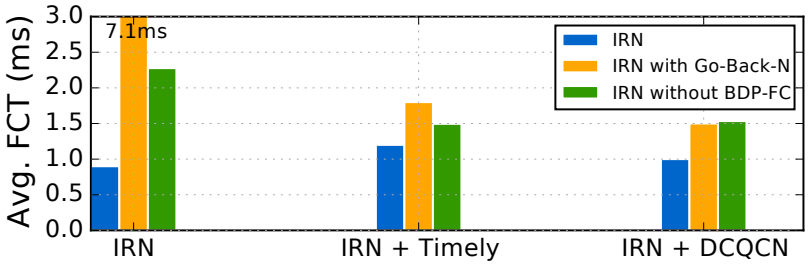


Figure 4.7: The figure shows the effect of doing go-back-N loss recovery and disabling BDP-FC with IRN. The y-axis is capped at 3ms to better highlight the trends.

requires PFC, even when explicit congestion control is used. Enabling PFC improves RoCE's performance by 1.35× to 3.5× across the three metrics.

Key Takeaways. The following are, therefore, the three takeaways from these results: (1) IRN (without PFC) performs better than RoCE (with PFC), (2) IRN does not require PFC, and (3) RoCE requires PFC.

4.3.3 Factor Analysis of IRN

We now perform a factor analaysis of IRN, to individually study the significance of the two key changes IRN makes to RoCE, namely (1) efficient loss recovery and (2) BDP-FC. For this we compare IRN's performance (as evaluated in §4.3.2) with two different variations that highlight the significance of each change: (1) enabling go-back-N loss recovery instead of using SACKs, and (2) disabling BDP-FC. Figure 4.7 shows the resulting average FCTs (we saw similar trends with other metrics). We discuss these results in greater details below.

Need for Efficient Loss Recovery. The first two bars in Figure 4.7 compare the average FCT of default SACK-based IRN and IRN with go-back-N respectively. We find that the latter results in significantly worse performance. This is because of the bandwidth wasted by go-back-N due to redundant retransmissions, as described before.

Before converging to IRN's current loss recovery mechanism, we experimented with alternative designs. In particular we explored the following questions:
(1) Can go-back-N be made more efficient? Go-back-N does have the advantage of simplicity over selective retransmission, since it allows the receiver to simply discard out-of-order packets. We, therefore, tried to explore whether we can mitigate the negative effects of go-back-N. We found that explicitly backing off on losses improved go-back-N performance for TIMELY (though, not for DCQCN). Nonetheless, SACK-based loss recovery continued to perform significantly better across different scenarios (with the difference in average FCT for TIMELY ranging from 20%-50%).
(2) Do we need SACKs? We tried a selective retransmit scheme without SACKs (where the sender does not maintain a bitmap to track selective acknowledgements). This performed better than go-back-N. However, it fared poorly when there were multiple losses in a window, requiring multiple round-trips to recover from them. The corresponding degradation in average FCT ranged from $<1\%$ up to 75% across different scenarios when compared to SACK-based IRN.
(3) Can the timeout value be computed dynamically? As described in §4.2, IRN uses two static (low and high) timeout values to allow faster recovery for short messages, while avoiding spurious retransmissions for large ones. We also experimented with an alternative approach of using dynamically computed timeout values (as with TCP), which not only complicated the design, but did not help since these effects were then be dominated by the initial timeout value.
Significance of BDP-FC. The first and the third bars in Figure 4.7 compare the average FCT of IRN with and without BDP-FC respectively. We find that BDP-FC significantly improves performance by reducing unnecessary queuing. Furthermore, it prevents a flow that is recovering from a packet loss from sending additional new packets and increasing congestion, until the loss has been recovered.
Efficient Loss Recovery vs BDP-FC. Comparing the second and third bars in Figure 4.7 shows that the performance of IRN with go-back-N loss recovery is generally worse than the performance of IRN without BDP-FC. This indicates that of the two changes IRN makes, efficient loss recovery helps performance more than BDP-FC.

4.3.4 Robustness of Basic Results

We now evaluate the robustness of the basic results from §4.3.2 across different scenarios and performance metrics.

Varying Experimental Scenario. We now evaluate the robustness of our results, as the experimental scenario is varied from our default case. The results are presented in a tabular format, and include for each the three metrics we consider (i.e. average slowdown, average FCT and tail FCT): (i) the absolute value of the metric with IRN, (ii) the ratio of the metric for IRN over RoCE + PFC (if this ratio is less than 1, then it shows that IRN without PFC performs better than RoCE with PFC), and (iii) the ratio of the metric for IRN over IRN + PFC (if this ratio is smaller than 1 or less than 1.1, i.e. close enough to 1, then it shows that IRN does not require PFC).

We present these set of results for all three cases that we consider: (a) no explicit CC, i.e. with-

Link Util.		**No explicit CC**			**With TIMELY**			**With DCQCN**		
		Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT
30%	IRN	1.85	0.0002	0.0024	1.70	0.0003	0.0059	1.69	0.0002	0.0029
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.335	0.743	0.852	0.890	0.756	0.789	0.904	0.909	0.936
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.990	1.000	0.995	0.996	0.993	0.999	1.003	1.005	1.010
50%	IRN	3.38	0.0003	0.0046	2.55	0.0006	0.0101	2.65	0.0004	0.0057
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.186	0.344	0.335	0.801	0.755	0.776	0.746	0.771	0.626
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.868	0.931	0.930	0.996	0.993	0.989	1.001	1.000	1.015
70%	IRN	8.24	0.0009	0.0153	4.73	0.0012	0.0185	5.19	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.269	0.350	0.301	0.626	0.625	0.594	0.484	0.509	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.513	0.640	0.612	0.995	0.976	0.968	1.009	1.005	0.966
90%	IRN	14.03	0.0019	0.0334	7.84	0.0019	0.0321	8.13	0.0019	0.0464
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.359	0.475	0.468	0.501	0.518	0.487	0.534	0.592	0.697
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.483	0.572	0.484	0.930	0.868	0.761	1.004	0.990	0.964

Table 4.2: Robustness of IRN with varying average link utilization levels.

Bandwidth		**No explicit CC**			**With TIMELY**			**With DCQCN**		
		Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT
10Gbps	IRN	9.41	0.0035	0.0595	5.45	0.0039	0.0616	5.88	0.0040	0.0809
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.170	0.309	0.274	0.720	0.637	0.620	0.695	0.681	0.688
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.371	0.523	0.458	0.967	0.924	0.903	0.973	0.929	0.954
40Gbps	IRN	8.24	0.0009	0.0153	4.73	0.0012	0.0185	5.19	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.269	0.350	0.301	0.626	0.625	0.594	0.484	0.509	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.513	0.640	0.612	0.995	0.976	0.968	1.009	1.005	0.966
100Gbps	IRN	7.92	0.0004	0.0064	4.84	0.0006	0.0091	5.44	0.0007	0.0187
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.408	0.476	0.413	0.385	0.489	0.432	0.424	0.597	0.658
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.629	0.728	0.705	0.988	0.986	0.964	1.011	1.022	1.051

Table 4.3: Robustness of IRN with varying average link bandwidth.

out any explicit congestion control (b) with TIMELY, and (c) with DCQCN. For ease of viewing, the results have been color coded: blue values indicate IRN performs better than IRN + PFC or RoCE + PFC for that metric, while red values highlight cases where IRN's performance is a bit worse.

Varying link utilization levels. Table 4.2 shows the robustness of our basic results as the link utilization level is varied from 30% to 90%. We find that as the link utilization increases, both the ratios for each metric (i.e. IRN over IRN + PFC and IRN over RoCE + PFC) decrease, indicating that IRN (without PFC) performs increasingly better than both IRN + PFC and RoCE + PFC.

Scale-out factor (No. of servers)		No explicit CC			With TIMELY			With DCQCN		
		Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT
6 (54 servers)	IRN	8.24	0.0009	0.0153	4.73	0.0012	0.0185	5.19	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.269	0.350	0.301	0.626	0.625	0.594	0.484	0.509	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.513	0.640	0.612	0.995	0.976	0.968	1.009	1.005	0.966
8 (128 servers)	IRN	8.93	0.0011	0.0166	4.98	0.0013	0.0195	5.36	0.0011	0.0234
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.250	0.335	0.292	0.613	0.642	0.609	0.479	0.503	0.481
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.497	0.601	0.515	1.000	0.993	0.985	1.010	0.998	0.992
10 (250 servers)	IRN	8.28	0.0010	0.0149	4.48	0.0012	0.0177	4.87	0.0010	0.0211
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.258	0.322	0.272	0.651	0.664	0.631	0.477	0.491	0.445
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.486	0.601	0.547	1.000	0.996	0.994	1.015	1.010	1.012

Table 4.4: Robustness of IRN with varying fat-tree topology size (in terms of scale out factor or arity).

Workload pattern		No explicit CC			With TIMELY			With DCQCN		
		Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT
Heavy-tailed (32B-3MB)	IRN	8.24	0.0009	0.0153	4.73	0.0012	0.0185	5.19	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.269	0.350	0.301	0.626	0.625	0.594	0.484	0.509	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.513	0.640	0.612	0.995	0.976	0.968	1.009	1.005	0.966
Uniform (500KB-5MB)	IRN	18.93	0.0116	0.0428	18.91	0.0123	0.0337	16.06	0.0109	0.0557
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.213	0.231	0.156	0.584	0.576	0.496	0.600	0.553	0.647
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.313	0.334	0.170	0.955	0.957	0.919	0.993	0.988	0.974

Table 4.5: Robustness of IRN with varying workload pattern.

This follows from the fact that the drawbacks of using PFC increases at higher link utilization due to increased congestion spreading.

Varying bandwidth Table 4.3 shows how our results vary as the bandwidth is changed from our default of 40Gbps to a lower value of 10Gbps and a higher value of 100Gbps. Here we find that as the bandwidth increases, the relative cost of a round trip required to react to packet drops without PFC also increases, thus reducing the performance gap between IRN and the two PFC enabled cases. However, even at 100Gbps, IRN (without PFC) continues to perform better than RoCE (with PFC).

Varying the scale of topology. Table 4.4 shows the robustness of our basic results as the scale of the topology is increased from our default of 6 port switches with 54 servers to 8 and 10 port switches with 128 and 250 servers respectively. Our trends remain roughly similar as we scale up

Buffer Size		No explicit CC			With TIMELY			With DCQCN		
		Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT
60KB	IRN	9.58	0.0014	0.0225	5.75	0.0013	0.0213	6.81	0.0022	0.0464
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.354	0.454	0.395	0.680	0.565	0.579	0.821	0.813	0.876
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.285	0.371	0.351	0.723	0.597	0.596	0.848	0.829	0.883
120KB	IRN	8.87	0.0012	0.0191	4.99	0.0012	0.0192	5.68	0.0014	0.0324
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.320	0.411	0.353	0.603	0.578	0.562	0.689	0.697	0.692
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.343	0.410	0.340	0.863	0.821	0.794	0.951	0.945	0.932
240KB	IRN	8.24	0.0009	0.0153	4.73	0.0012	0.0185	5.19	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.269	0.350	0.301	0.626	0.625	0.594	0.484	0.509	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.513	0.640	0.612	0.995	0.976	0.968	1.009	1.005	0.966
480KB	IRN	8.63	0.0008	0.0127	4.66	0.0012	0.0185	4.99	0.0010	0.0206
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.223	0.315	0.285	0.621	0.657	0.614	0.310	0.442	0.405
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.853	0.953	0.945	1.005	1.004	1.004	1.003	1.001	0.955

Table 4.6: Robustness of IRN with varying per-port buffer size.

the topology beyond our default set up.

Varying workload. Our default workload comprised of a heavy-tailed mix of short messages (e.g. for key-value lookups) and large messages (for storage or background applications). We also experimented with another workload pattern, comprising of medium to large sized flows with a uniform distribution, representing a scenario where RDMA is used only for storage or background tasks. Table 4.5 shows the results. We find that our key trends hold for this workload as well. Even when considering individual flow sizes in the range captured by our default workload, we did not observe any significant deviation from the key trends produced by the aggregated metrics. We also present results with an incast workload later in this section.

Varying buffer size. Table 4.6 shows the robustness of our basic results as the buffer size is varied from 60KB to 480KB. We find that as the buffer size is decreased, the drawbacks of using PFC increases, due to more pauses and greater impact of congestion spreading. [4] In general, as the buffer size is increased, the difference between PFC-enabled and PFC-disabled performance with IRN reduces (due to fewer PFC frames and packet drops), while the benefits of using IRN over RoCE+PFC increases (because of greater relative reduction in queuing delay with IRN due to BDP-FC). We expect to see similar behaviour in shared buffer switches.

Varying parameters. We finally present the robustness of our results as we vary some of IRN's parameters from their default values. In particular, Table 4.7 captures the effect of over-estimating the RTO_{high} value to 2× and 4× the ideal, and Table 4.8 shows results with higher N values for

[4] Decreasing just the PFC threshold below its default value also has a similar effect.

RTO_{high}		No explicit CC			With TIMELY			With DCQCN		
		Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT
$320\mu s$	IRN	8.24	0.0009	0.0153	4.73	0.0012	0.0185	5.19	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.269	0.350	0.301	0.626	0.625	0.594	0.484	0.509	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.513	0.640	0.612	0.995	0.976	0.968	1.009	1.005	0.966
$640\mu s$	IRN	8.38	0.0010	0.0162	4.77	0.0012	0.0188	5.23	0.0010	0.0206
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.274	0.369	0.319	0.631	0.631	0.605	0.488	0.503	0.452
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.522	0.675	0.649	1.003	0.987	0.987	1.018	0.995	0.964
$1280\mu s$	IRN	8.74	0.0011	0.0194	4.79	0.0012	0.0194	5.24	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.285	0.426	0.382	0.634	0.644	0.623	0.489	0.510	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.544	0.779	0.776	1.008	1.007	1.016	1.020	1.007	0.968

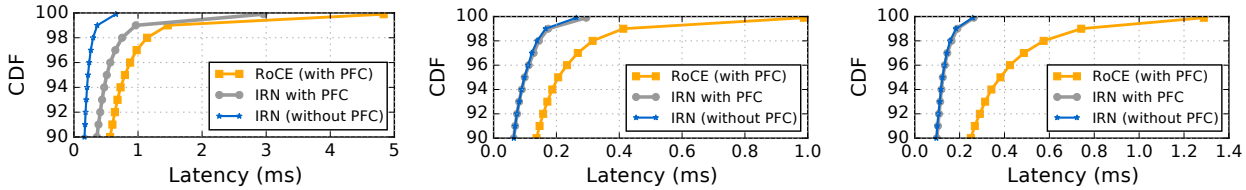
Table 4.7: Robustness of IRN to higher RTO_{high} value.

N for using RTO_{low}		No explicit CC			With TIMELY			With DCQCN		
		Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT	Avg Slow-down	Avg FCT	99%ile FCT
3	IRN	8.24	0.0009	0.0153	4.73	0.0012	0.0185	5.19	0.0010	0.0207
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.269	0.350	0.301	0.626	0.625	0.594	0.484	0.509	0.453
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.513	0.640	0.612	0.995	0.976	0.968	1.009	1.005	0.966
10	IRN	8.26	0.0010	0.0157	4.75	0.0012	0.0187	5.13	0.0010	0.0208
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.270	0.356	0.310	0.628	0.629	0.600	0.478	0.500	0.456
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.515	0.651	0.629	0.999	0.983	0.979	0.997	0.989	0.974
15	IRN	8.25	0.0010	0.0154	4.72	0.0012	0.0187	5.22	0.0010	0.0218
	$\frac{\text{IRN}}{\text{RoCE+PFC}}$	0.270	0.357	0.303	0.624	0.628	0.602	0.487	0.516	0.477
	$\frac{\text{IRN}}{\text{IRN+PFC}}$	0.514	0.653	0.616	0.992	0.981	0.981	1.015	1.019	1.019

Table 4.8: Robustness of IRN to higher N value for using RTO_{low}.

using RTO_{low}. We find that changing these parameters produces very small differences over our default case results, showing that IRN is fairly robust to how its parameters are set.

Summarizing Overall Results: Across all of these experimental scenarios, we find that:
(a) IRN (without PFC) always performs better than RoCE (with PFC), with the performance improvement ranging from 6% to 83% across different cases.
(b) When used without any congestion control, enabling PFC with IRN always degrades perfor-



(a) No CC (b) TIMELY (c) DCQCN

Figure 4.8: The figures compare the tail latency for single-packet messages for IRN, IRN with PFC, and RoCE (with PFC), across different congestion control algorithms.

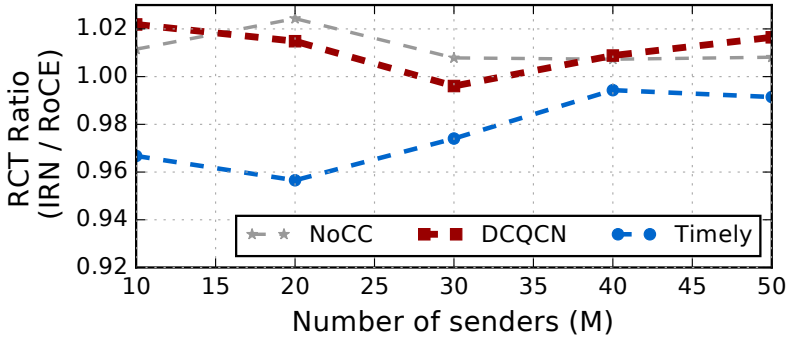


Figure 4.9: The figure shows the ratio of request completion time of incast with IRN (without PFC) over RoCE (with PFC) for varying degree of fan-ins across congestion control algorithms.

mance, with the maximum degradation across different scenarios being as high as 2.4×.
(c) Even when used with TIMELY and DCQCN, enabling PFC with IRN often degrades performance (with the maximum degradation being 39% for TIMELY and 20% for DCQCN). Any improvement in performance due to enabling PFC with IRN stays within 1.6% for TIMELY and 5% for DCQCN.

Tail latency for small messages. We now look at the tail latency (or tail FCT) of the single-packet messages from our default scenario, which is another relevant metric in datacenters [91]. Figure 4.8 shows the CDF of this tail latency (from 90%ile to 99.9%ile), across different congestion control algorithms. Our key trends from §4.3.2 hold even for this metric. This is because IRN (without PFC) is able to recover from single-packet message losses quickly due to the low RTO_{low} timeout value. With PFC, these messages end up waiting in the queues for similar (or greater) duration due to pauses and congestion spreading. For all cases, IRN performs significantly better than RoCE.

Incast. We now evaluate incast scenarios, both with and without cross-traffic. The incast workload without any cross traffic can be identified as the best case for PFC, since only valid congestion-causing flows are paused without unnecessary head-of-the-line blocking.
Incast without cross-traffic. We simulate the incast workload on our default topology by striping 150MB of data across M randomly chosen sender nodes that send it to a fixed destination node [11]. We vary M from 10 to 50. We consider the request completion time (RCT) as the metric for incast

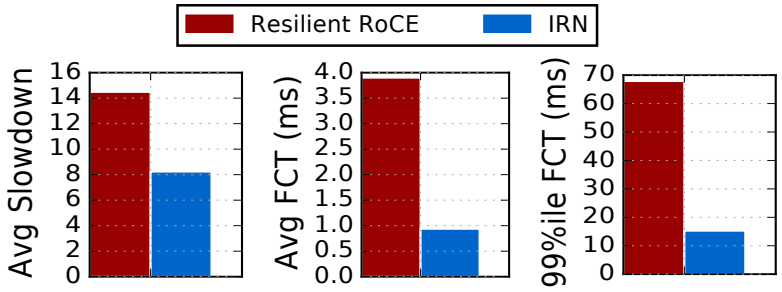


Figure 4.10: The figures compares resilient RoCE (RoCE+DCQCN without PFC) with IRN.

performance, which is when the last flow completes. For each M, we repeat the experiment 100 times and report the average RCT. Figure 4.9 shows the results, comparing IRN with RoCE. We find that the two have comparable performance: any increase in the RCT due to disabling PFC with IRN remained within 2.5%. The results comparing IRN's performance with and without PFC looked very similar. We also varied our default incast setup by changing the bandwidths to 10Gbps and 100Gbps, and increasing the number of connections per machine. Any degradation in performance due to disabling PFC with IRN stayed within 9%.

Incast with cross traffic. In practice we expect incast to occur with other cross traffic in the network [91, 47]. We started an incast as described above with $M = 30$, along with our default case workload running at 50% link utilization level. The incast RCT for IRN (without PFC) was always lower than RoCE (with PFC) by 4%-30% across the three congestion control schemes. For the background workload, the performance of IRN was better than RoCE by 32%-87% across the three congestion control schemes and the three metrics (i.e.the average slowdown, the average FCT and the tail FCT). Enabling PFC with IRN generally degraded performance for both the incast and the cross-traffic by 1-75% across the three schemes and metrics, and improved performance only for one case (incast workload with DCQCN by 1.13%).

Window-based congestion control. We also implemented conventional window-based congestion control schemes such as TCP's AIMD and DCTCP [8] with IRN and observed similar trends as discussed in §4.3.2. In fact, when IRN is used with TCP's AIMD, the benefits of disabling PFC were even stronger, because it exploits packet drops as a congestion signal, which is lost when PFC is enabled.

Summary. Our key results i.e. (1) IRN (without PFC) performs better than RoCE (with PFC), and (2) IRN does not require PFC, hold across varying realistic scenarios, congestion control schemes and performance metrics.

4.3.5 Comparison with Resilient RoCE.

A recent proposal on Resilient RoCE [116] explores the use of DCQCN to avoid packet losses in specific scenarios, and thus eliminate the requirement for PFC. However, as observed previously in Figure 4.6, DCQCN may not always be successful in avoiding packet losses across all realistic

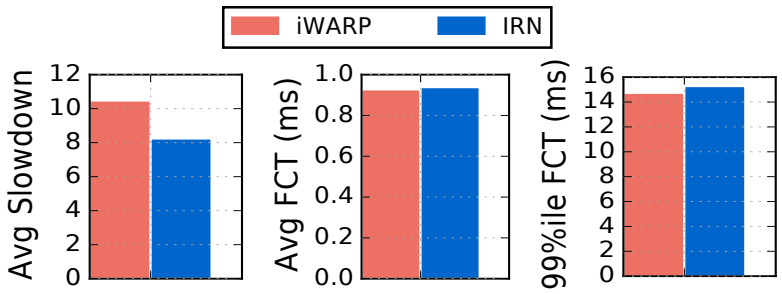


Figure 4.11: The figures compares iWARP's transport (TCP stack) with IRN.

scenarios with more dynamic traffic patterns and hence PFC (with its accompanying problems) remains necessary. Figure 4.10 provides a direct comparison of IRN with Resilient RoCE. We find that IRN, even without any explicit congestion control, performs significantly better than Resilient RoCE, due to better loss recovery and BDP-FC.

4.3.6 Comparison with iWARP.

We finally explore whether IRN's simplicity over the TCP stack implemented in iWARP impacts performance. We compare IRN's performance (without any explicit congestion control) with full-blown TCP stack's, using INET simulator's in-built TCP implementation for the latter. Figure 4.11 shows the results for our default scenario. We find that absence of slow-start (with use of BDP-FC instead) results in 21% smaller slowdowns with IRN and comparable average and tail FCTs. These results show that in spite of a simpler design, IRN's performance is better than full-blown TCP stack's, even without any explicit congestion control. Augmenting IRN with TCP's AIMD logic further improves its performance, resulting in 44% smaller average slowdown and 11% smaller average FCT as compared to iWARP. Furthermore, IRN's simple design allows it to achieve message rates comparable to current RoCE NICs with very little overheads (as evaluated in §4.5). An iWARP NIC, on the other hand, can have up to 4× smaller message rate than a RoCE NIC (§4.1). Therefore, IRN provides a simpler and more performant solution than iWARP for eliminating RDMA's requirement for a lossless network.

4.4 Implementation Considerations

We now discuss how one can incrementally update RoCE NICs to support IRN's transport logic, while maintaining the correctness of RDMA semantics as defined by the Infiniband RDMA specification [61]. Our implementation relies on extensions to RDMA's packet format, e.g.introducing new fields and packet types. These extensions are encapsulated within IP and UDP headers (as in RoCEv2) so they only effect the endhost behavior and not the network behavior (i.e. no changes are required at the switches). We begin with providing some relevant context about different RDMA operations before describing how IRN supports them.

4.4.1 Relevant Context

The two remote endpoints associated with an RDMA message transfer are called a *requester* and a *responder*. The interface between the user application and the RDMA NIC is provided by *Work Queue Elements* or WQEs (pronounced as *wookies*). The application posts a WQE for each RDMA message transfer, which contains the application-specified metadata for the transfer. It gets stored in the NIC while the message is being processed, and is expired upon message completion. The WQEs posted at the requester and at the responder NIC are called Request WQEs and Receive WQEs respectively. Expiration of a WQE upon message completion is followed by the creation of a *Completion Queue Element* or a CQE (pronounced as *cookie*), which signals the message completion to the user application. There are four types of message transfers supported by RDMA NICs:

Write. The requester *writes* data to responder's memory. The data length, source and sink locations are specified in the Request WQE, and typically, no Receive WQE is required. However, Write-with-Immediate operation requires the user application to post a Receive WQE that expires upon completion to generate a CQE (thus signaling Write completion at the responder as well).

Read. The requester *reads* data from responder's memory. The data length, source and sink locations are specified in the Request WQE, and no Receive WQE is required.

Send. The requester *sends* data to the responder. The data length and source location is specified in the Request WQE, while the sink location is specified in the Receive WQE.

Atomic. The requester reads and atomically updates the data at a location in the responder's memory, which is specified in the Request WQE. No Receive WQE is required. Atomic operations are restricted to single-packet messages.

4.4.2 Supporting RDMA Reads and Atomics

IRN relies on per-packet ACKs for BDP-FC and loss recovery. RoCE NICs already support per-packet ACKs for Writes and Sends. However, when doing Reads, the requester (which is the data sink) does not explicitly acknowledge the Read response packets. IRN, therefore, introduces packets for *read (N)ACKs* that are sent by a requester for each Read response packet. RoCE currently has eight unused opcode values available for the reliable connected QPs, and we use one of these for *read (N)ACKs*. IRN also requires the Read responder (which is the data source) to implement timeouts. New timer-driven actions have been added to the NIC hardware implementation in the past [116]. Hence, this is not an issue.

RDMA Atomic operations are treated similar to a single-packet RDMA Read messages.

Our simulations from §4.3 did not use ACKs for the RoCE (with PFC) baseline, modelling the extreme case of all Reads. Therefore, our results take into account the overhead of per-packet ACKs in IRN.

4.4.3 Supporting Out-of-order Packet Delivery

One of the key challenges for implementing IRN is supporting out-of-order (OOO) packet delivery at the receiver – current RoCE NICs simply discard OOO packets. A naive approach for handling OOO packet would be to store all of them in the NIC memory. The total number of OOO packets with IRN is bounded by the BDP cap (which is about 110 MTU-sized packets for our default scenario as described in §4.3.1) [5]. Therefore to support a thousand flows, a NIC would need to buffer 110MB of packets, which exceeds the memory capacity on most commodity RDMA NICs.

We therefore explore an alternate implementation strategy, where the NIC DMAs OOO packets directly to the final address in the application memory and keeps track of them using bitmaps (which are sized at BDP cap). This reduces NIC memory requirements from 1KB per OOO packet to only a couple of bits, but introduces some additional challenges that we address here. Note that partial support for OOO packet delivery was introduced in the Mellanox ConnectX-5 NICs to enable adaptive routing [86]. However, it is restricted to Write and Read operations. We improve and extend this design to support all RDMA operations with IRN.

We classify the issues due to out-of-order packet delivery into the following categories.

First packet issues. For some RDMA operations, critical information is carried in the first packet of a message, which is required to process other packets in the message. Enabling OOO delivery, therefore, requires that some of the information in the first packet be carried by all packets.

In particular, the RETH header (containing the remote memory location) is carried only by the first packet of a Write message. IRN requires adding it to every packet.

WQE matching issues. Some operations require every packet that arrives to be matched with its corresponding WQE at the responder. This is done implicitly for in-order packet arrivals. However, this implicit matching breaks with OOO packet arrivals. A work-around for this is assigning explicit WQE sequence numbers, that get carried in the packet headers and can be used to identify the corresponding WQE for each packet. IRN uses this workaround for the following RDMA operations:

Send and Write-with-Immediate: It is required that Receive WQEs be consumed by Send and Write-with-Immediate requests in the same order in which they are posted. Therefore, with IRN every Receive WQE, and every Request WQE for these operations, maintains a *recv_WQE_SN* that indicates the order in which they are posted. This value is carried in all Send packets and in the last Write-with-Immediate packet, [6] and is used to identify the appropriate Receive WQE. IRN also requires the Send packets to carry the relative offset in the packet sequence number, which is used to identify the precise address when placing data.

Read/Atomic: The responder cannot begin processing a Read/Atomic request *R*, until all packets

[5] For QPs that only send single packet messages less than one MTU in size, the number of outstanding packets is limited to the maximum number of outstanding requests, which is typically smaller than the BDP cap [70, 71].

[6] A Receive WQE is consumed only by the last packet of a Write-with-immediate message, and is required to process all packets for a Send message.

expected to arrive before *R* have been received. Therefore, an OOO Read/Atomic Request packet needs to be placed in a Read WQE buffer at the responder (which is already maintained by current RoCE NICs). With IRN, every Read/Atomic Request WQE maintains a *read_WQE_SN*, that is carried by all Read/Atomic request packets and allows identification of the correct index in this Read WQE buffer.

Last packet issues. For many RDMA operations, critical information is carried in last packet, which is required to complete message processing. Enabling OOO delivery, therefore, requires keeping track of such last packet arrivals and storing this information at the endpoint (either on NIC or main memory), until all other packets of that message have arrived. We explain this in more details below.

A RoCE responder maintains a *message sequence number (MSN)* which gets incremented when the last packet of a Write/Send message is received or when a Read/Atomic request is received. This MSN value is sent back to the requester in the ACK packets and is used to expire the corresponding Request WQEs. The responder also expires its Receive WQE when the last packet of a Send or a Write-With-Immediate message is received and generates a CQE. The CQE is populated with certain meta-data about the transfer, which is carried by the last packet. IRN, therefore, needs to ensure that the completion signalling mechanism works correctly even when the last packet of a message arrives before others. For this, an IRN responder maintains a 2-bitmap, which in addition to tracking whether or not a packet *p* has arrived, also tracks whether it is the last packet of a message that will trigger (1) an MSN update and (2) in certain cases, a Receive WQE expiration that is followed by a CQE generation. These actions are triggered only after all packets up to *p* have been received. For the second case, the *recv_WQE_SN* carried by *p* (as discussed before) can identify the Receive WQE with which the meta-data in *p* needs to be associated, thus enabling a *premature CQE* creation. The premature CQE can be stored in the main memory, until it gets delivered to the application after all packets up to *p* have arrived.

Application-level Issues. Certain applications (for example FaRM [32]) rely on polling the last packet of a Write message to detect completion, which is incompatible with OOO data placement. This polling based approach violates the RDMA specification and is more expensive than officially supported methods (FaRM [32] mentions moving on to using the officially supported Write-with-Immediate method in the future for better scalability). IRN's design provides all of the Write completion guarantees as per the RDMA specification.

More precisely, the RDMA specification (Sec o9-20 [61]) clearly states that an application shall not depend on the contents of an RDMA Write buffer at the responder, until one of the following has occurred: (1) arrival and completion of the last RDMA Write request packet when used with Immediate data; (2) arrival and completion of a subsequent Send message; (3) update of a memory element by a subsequent Atomic operation. IRN design guarantees that meeting any of these conditions would automatically imply that previous Writes have completed. As discussed before, it supports Write-with-Immediate, where a CQE for the request is not released to the responder's application until all packets up until the last packet of the request have been received. Likewise, the CQE for a subsequent send will not be released to the application until all previous packets

have arrived. The Atomic request packet will wait in the Read/Atomic WQE buffer without being processed until all previous packets have arrived.

OOO data placement can also result in a situation where data written to a particular memory location is overwritten by a restransmitted packet from an older message. Typically, applications using distributed memory frameworks assume relaxed memory ordering and use application layer *fences* whenever strong memory consistency is required [4, 121]. Therefore, both iWARP and Mellanox ConnectX-5, in supporting OOO data placement, expect the application to deal with the potential memory over-writing issue and do not handle it in the NIC or the driver. IRN can adopt the same strategy. Another alternative is to deal with this issue in the driver, by enabling the fence indicator for a newly posted request that could potentially overwrite an older one.

4.4.4 Other Considerations

Simultaneously Tracking Reads and Writes. Currently, the packets that are sent and received by a requester use the same packet sequence number (*PSN*) space. This interferes with loss tracking and BDP-FC. IRN, therefore, splits the *PSN* space into two different ones (1) *sPSN* to track the request packets sent by the requester, and (2) *rPSN* to track the response packets received by the requester. This decoupling remains transparent to the application and is compatible with the current RoCE packet format.

Support for Shared Receive Queues. IRN can be extended to support Shared Receive Queues (SRQ) as follows.
For Send. As mentioned previously, with IRN, a Responder QP allots *recv_WQE_SN* to the Receive WQEs in the order they get posted. When the QP uses SRQ, the *recv_WQE_SN* can be allotted as and when new Receive WQEs are dequeued from the SRQ by the QP. Suppose we start with *recv_WQE_SN* of 0. The first send packet arrives in order with *recv_WQE_SN* of 0. IRN will dequeue one Receive WQE from SRQ and allot it *recv_WQE_SN* of 0. Suppose after this some intermediate send requests are lost and a Send packet with *recv_WQE_SN* of 4 arrives. IRN will then dequeue four Receive WQEs from SRQ, allot them *recv_WQE_SN* of 1,2,3 and 4 and will use the fourth Receive WQE to process the packet.
For Write-with-Immediate: Write-with-Immediate do not require the Receive WQE to process incoming packets. The Receive WQE just needs to be expired when the entire message is received. If there are no outstanding Sends (or already dequeued Receive WQEs waiting at the QP), then the first available Receive WQE is dequeued from the SRQ and expired to generate a completion event, when all packets of the Write-with-Immediate request have been received. If there are outstanding Receive WQEs at the QP, the first Receive WQE that is outstanding is expired. The fact that IRN checks for message completion using the bitmap *in order* guarantees that the first Receive WQE is the correct WQE that needs to be expired.

Support for End-to-End Credits. For messages that need a Receive WQE, an end-to-end credit scheme is used, where the acks piggy-back the information about the number of Receive WQEs (or credits) remaining. When the responder runs out of credits, it can still send the first packet

of a Send message or all packets of a Write-with-Immediate message as a probe. If the receiver has new WQEs the operation executes successfully and the new credit is communicated via the acknowledgement packet. Otherwise, an RNR (receiver not ready) NACK is sent which results in go-back-N.

This can be supported with IRN as well. Although when an out-of-sequence probe packet is received without credits (with no Receive WQE), it should be dropped at the receiver. For example, if there is only one Receive WQE at the responder and the requester sends two Send messages (the first as a valid message, the second as a probe). If the first message is lost, the second message should be dropped instead of placing it in first message's memory address (which would be wrong to do) or sending an RNR NACK (which would be ill-timed). The first message will be sent again due to loss recovery and credit update process will get back on track.

NACKs due to Other Errors. This is a generalization of the case described above. Current RoCE NICs generate a NACK for out-of-sequence packets, the requester treats them as errors and does a go-back-n on receiving them. With IRN, we consider out-of-sequence NACKs as normal behaviour and treat them differently (as described in §4.2). But NACKs can still be generated for other reasons such as RNR. IRN will do a go-back-N on receiving such a NACK. If an out-of-sequence packet will result in generation of such an error NACK at the responder, IRN will discard that packet at the responder, without processing it and without sending a NACK.

Supporting Send with Invalidate. This operation is used to invalidate the use of a remote memory region. If this packet arrives and is executed before previous Writes on the invalidated region, the Write operation would die. To avoid this, IRN can enforce a fence before the Send with Invalidate operations.

4.5 Evaluating Implementation Overheads

We now evaluate IRN's implementation overheads over current RoCE NICs along the following three dimensions: in §4.5.1, we do a comparative analysis of IRN's memory requirements; in §4.5.2, we evaluate the overhead for implementing IRN's packet processing logic by synthesizing it on an FPGA; and in §4.5.3, we evaluate, via simulations, how IRN's implementation choices impact end-to-end performance.

4.5.1 NIC State Overhead

Mellanox RoCE NICs support several MBs of cache to store various metadata including per-QP and per-WQE contexts. The additional state that IRN introduces consumes a total of only 3-10% of the current NIC cache for a couple of thousands of QPs and tens of thousands of WQEs, even when considering large 100Gbps links. We present a breakdown this additional state below.

Additional Per-QP Context.
State variables: IRN needs 52 bits of additional state for its transport logic: 24 bits each to track the packet sequence to be retransmitted and the recovery sequence, and 4 bits for various flags. Other per-flow state variables needed for IRN's transport logic (e.g., expected sequence number) are already maintained by current RoCE NICs. Hence, the per-QP overhead is 104 bits (52 bits each at the requester and the responder). Maintaining a timer at the responder for Read timeouts and a variable to track in-progress Read requests in the Read WQE buffer adds another 56 bits to the responder leading to a total of 160 bits of additional per-QP state with IRN. For context, RoCE NICs currently maintain a few thousands of bits per QP for various state variables.
Bitmaps: IRN requires five BDP-sized bitmaps: two at the responder for the 2-bitmap to track received packets, one at the requester to track the Read responses, one each at the requester and responder for tracking selective acks. Assuming each bitmap to be 128 bits (i.e.sized to fit the BDP cap for a network with bandwidth 40Gbps and a two-way propagation delay of up to 24μs, typical in today's datacenter topologies [91]), IRN would require a total of 640 bits per QP for bitmaps. This is much less than the total size of bitmaps maintained by a QP for the OOO support in Mellanox ConnectX-5 NICs.
Others: Other per-QP meta-data that is needed by an IRN driver when a WQE is posted (e.g counters for assigning WQE sequence numbers) or expired (e.g. premature CQEs) can be stored directly in the main memory and do not add to the NIC memory overhead.

Additional Per-WQE Context. As described in §4.4, IRN maintains sequence numbers for certain types of WQEs. This adds 3 bytes to the per-WQE context which is currently sized at 64 bytes.

Additional Shared State. IRN also maintains some additional variables (or parameters) that are shared across QPs. This includes the BDP cap value, the RTO_{low} value, and N for RTO_{low}, which adds up to a total of only 10 bytes.

4.5.2 Packet Processing Overhead

We evaluate the implementation overhead due to IRN's per-packet processing logic, which requires various bitmap manipulations. The logic for other changes that IRN makes – e.g., adding header extensions to packets, premature CQE generation, etc. – are already implemented in RoCE NICs and can be easily extended for IRN.

We use Xilinx Vivado Design Suite 2017.2 [142] to do a high-level synthesis of the four key packet processing modules (as described below), targeting the Kintex Ultrascale XCKU060 FPGA which is supported as a bump-on-the-wire on the Mellanox Innova Flex 4 10/40Gbps NICs [88]. [7]

Synthesis Process. To focus on the *additional* packet processing complexity due to IRN, our implementation for the four modules is *stripped-down*. More specifically, each module receives the relevant packet metadata and the QP context as streamed inputs, relying on a RoCE NIC's existing implementation to parse the packet headers and retrieve the QP context from the NIC

[7] We have made our synthesis code available at https://netsys.github.io/irn-vivado-hls/.

cache (or the system memory, in case of a cache miss). The updated QP context is passed as streamed output from the module, along with other relevant module-specific outputs as described below.
(1) receiveData: Triggered on a packet arrival, it outputs the relevant information required to generate an ACK/NACK packet and the number of Receive WQEs to be expired, along with the updated QP context (e.g. bitmaps, expected sequence number, MSN).
(2) txFree: Triggered when the link's Tx is free for the QP to transmit, it outputs the sequence number of the packet to be (re-)transmitted and the updated QP context (e.g. next sequence to transmit). During loss-recovery, it also performs a look ahead by searching the SACK bitmap for the next packet sequence to be retransmitted.
(3) receiveAck: Triggered when an ACK/NACK packet arrives, it outputs the updated QP context (e.g. SACK bitmap, last acknowledged sequence).
(4) timeout: If triggered when the timer expires using RTO_{low} value (indicated by a flag in the QP context), it checks if the condition for using RTO_{low} holds. If not, it does not take any action and sets an output flag to extend the timeout to RTO_{high}. In other cases, it executes the timeout action and returns the updated QP context. Our implementation relies on existing RoCE NIC's support for setting timers, with the RTO_{low} value being used by default, unless explicitly extended.

The bitmap manipulations in the first three modules account for most of the complexity in our synthesis. Each bitmap was implemented as a ring buffer, using an arbitrary precision variable of 128 bits, with the *head* corresponding to the expected sequence number at the receiver (or the cumulative acknowledgement number at the sender). The key bitmap manipulations required by IRN can be reduced to the following three categories of known operations: (i) *finding first zero*, to find the next expected sequence number in *receiveData* and the next packet to retransmit in *txFree* (ii) *popcount* to compute the increment in MSN and the number of Receive WQEs to be expired in *receiveData*, (iii) *bit shifts* to advance the bitmap *heads* in *receiveData* and *receiveAck*. We optimized the first two operations by dividing the bitmap variables into chunks of 32 bits and operating on these chunks in parallel.

We validated the correctness of our implementation by generating input event traces for each synthesized module from the simulations described in §4.3 and passing them as input in the test bench used for RTL verification by the Vivado Design Suite. The output traces, thus, generated were then matched with the corresponding output traces obtained from the simulator. We also used the Vivado HLS tool to export our RTL design to create IP blocks for our modules.

Synthesis Results. Our FPGA synthesis report has been summarized in Table 4.9 and discussed below.
Resource Usage: The second and third columns in Table 4.9 report the percentage of flip-flops (FF) and look-up tables (LUT) used for the four modules (no BRAM or DSP48E units were consumed). We find that each of IRN's packet processing modules consume less than 1% FFs and 2% LUTs (with a total of 1.35% FFs and 4% LUTs consumed). Increasing the bitmap size to support 100Gbps links consumed a total of 2.66% of FFs and 9.5% of LUTs on the same device (though we expect the relative resource usage to be smaller on a higher-scale device designed for 100Gbps links).

Module Name	Resource Usage		Max Latency	Min Throughput
	FF	LUT		
receiveData	0.62%	1.93%	16.5 ns	45.45 Mpps
txFree	0.32%	0.95%	15.9 ns	47.17 Mpps
receiveAck	0.4%	1.05%	15.96 ns	46.99 Mpps
timeout	0.01%	0.08%	<6.3 ns	318.47 Mpps
Total Resource Usage: 1.35% FF and 4.01% LUTs				
Min Bottleneck Tpt: 45.45Mpps				

Table 4.9: Performance and resource usage for different packet processing modules on Xilinx Kintex Ultrascale KU060 FPGA.

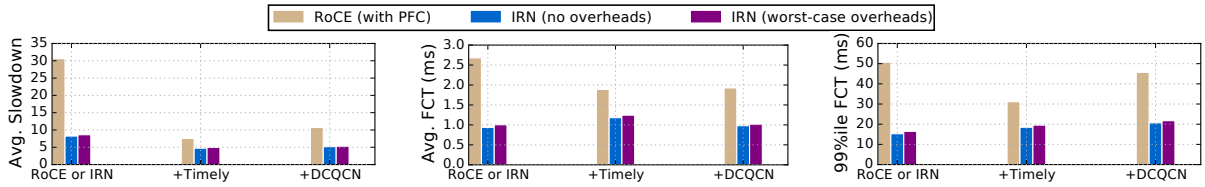


Figure 4.12: The figures show the performance of IRN with worse case overheads, comparing it with IRN without any overheads and with RoCE for our default case scenario.

Performance: The third and fourth column in Table 4.9 report the worst-case latency and throughput respectively for each module. [8] The latency added by each module is at most only 16.5ns. The *receiveData* module (requiring more complex bitmap operations) had the lowest throughput of 45.45Mpps. This is high enough to sustain a rate of 372Gbps for MTU-sized packets. It is also higher than the maximum rate of 39.5Mpps that we observed on Mellanox MCX416A-BCAT RoCE NIC across different message sizes (2 bytes - 1KB), after applying various optimizations such as batching and using multiple queue-pairs. A similar message rate was observed in prior work [71]. Note that we did not use pipelining within our modules, which can further improve throughput.

While we expect IRN to be implemented on the ASIC integrated with the existing RoCE implementation, we believe that the modest resources used on an FPGA board supported as an *add-on* in recent RDMA-enabled NICs, provides some intuition about the feasibility of the changes required by IRN. Also, note that the results reported here are far from the optimal results that can be achieved on an ASIC implementation due to two sources of sub-optimality: (i) using HLS for FPGA synthesis has been found to be up to 2× less optimal than directly using Verilog [78] and (ii) FPGAs, in general, are known to be less optimal than ASICs.

[8] The worst-case throughput was computed by dividing the clock frequency with the maximum initiation interval, as reported by the Vivado HLS synthesis tool [137].

4.5.3 Impact on End-to-End Performance

We now evaluate how IRN's implementation overheads impact the end-to-end performance. We identify the following two implementation aspects that could potentially impact end-to-end performance and model these in our simulations.

Delay in Fetching Retransmissions. While the regular packets sent by a RoCE NIC are typically pre-fetched, we assume that the DMA request for retransmissions is sent only after the packet is identified as lost (i.e. when loss recovery is triggered or when a look-ahead is performed). The time taken to fetch a packet over PCIe is typically between a few hundred nanoseconds to $<2\mu$s [3, 112]. We set a worst-case retransmission delay of 2μs for every retransmitted packet i.e. the sender QP is allowed to retransmit a packet only after 2μs have elapsed since the packet was detected as lost.

Additional Headers. As discussed in §4.4, some additional headers are needed in order to DMA the packets directly to the application memory, of which, the most extreme case is the 16 bytes of RETH header added to every Write packet. Send data packets have an extra header of 6 bytes, while Read responses do not require additional headers. We simulate the worst-case scenario of all Writes with every packet carrying 16 bytes additional header.

Results. Figure 4.12 shows the results for our default scenario after modeling these two sources of worst-case overheads. We find that they make little difference to the end-to-end performance (degrading the performance by 4-7% when compared to IRN without overheads). The performance remains 35%-63% better than our baseline of RoCE (with PFC). We also verified that the retransmission delay of 2μs had a much smaller impact on end-to-end performance (2μs is very small compared to the network round-trip time taken to detect a packet loss and to recover from it, which could be of the order of a few hundred microseconds). The slight degradation in performance observed here can almost entirely by attributed to the additional 16 bytes header in every packet. Therefore, we would expect the performance impact to be even smaller when there is a mix of Write and Read workloads.

Summary. Our analysis shows that IRN is well within the limits of feasibility, with small chip area and NIC memory requirements and minor bandwidth overhead. We also validated our analysis through extensive discussions with two commercial NIC vendors (including Mellanox); both vendors confirmed that the IRN design can be easily implemented on their hardware NICs. Inspired by our results, Mellanox is considering implementing a version of IRN in their next release.

4.6 Discussion and Related Work

Backwards Compatibility. We briefly sketch one possible path to incrementally deploying IRN. We envision that NIC vendors will manufacture NICs that support dual RoCE/IRN modes. The

use of IRN can be negotiated between two endpoints via the RDMA connection manager, with the NIC falling back to RoCE mode if the remote endpoint does not support IRN. (This is similar to what was used in moving from RoCEv1 to RoCEv2.) Network operators can continue to run PFC until all their endpoints have been upgraded to support IRN at which point PFC can be permanently disabled.

Reordering due to load-balancing. Datacenters today use ECMP for load balancing [47], that maintains ordering within a flow. IRN's OOO packet delivery support also allows for other load balancing schemes that may cause packet reordering within a flow [43, 31]. IRN's loss recovery mechanism can be made more robust to reordering by triggering loss recovery only after a certain threshold of NACKs are received.

Other hardware-based loss recovery. MELO [81], a recent scheme developed in parallel to IRN, proposes an alternative design for hardware-based selective retransmission, where out-of-order packets are buffered in an off-chip memory. Unlike IRN, MELO only targets PFC-enabled environments with the aim of greater robustness to random losses caused by failures. As such, MELO is orthogonal to our main focus which is showing that PFC is unnecessary. Nonetheless, the existence of alternate designs such as MELO's further corroborates the feasibility of implementing better loss recovery on NICs.

HPC workloads. The HPC community has long been a strong supporter of losslessness. This is primarily because HPC clusters are smaller with more controlled traffic patterns, and hence the negative effects of providing losslessness (such as congestion spreading and deadlocks) are rarer. PFC's issues are exacerbated on larger scale clusters [47, 146, 91, 58, 115].

Credit-based Flow Control. Since the focus of our work was RDMA deployment over Ethernet, our experiments used PFC. Another approach to losslessness, used by Infiniband, is credit-based flow control, where the downlink sends credits to the uplink when it has sufficient buffer capacity. Credit-based flow control suffers from the same performance issues as PFC: head-of-the-line blocking, congestion spreading, the potential for deadlocks, *etc*. We, therefore, believe that our observations from §4.3 can be applied to credit-based flow control as well.

4.7 Conclusion

As discussed earlier, the use of RDMA in datacenters has gone through several stages of evolution. First, RDMA was deployed over Infiniband, which provides a lossless fabric. As the interest in RDMA grew, there was a need to deploy it over commodity Ethernet. One approach was embodied in iWARP, which embedded a full TCP stack on the NIC and did not require a lossless network. The other approach, RoCE, started with a simpler NIC (based largely on what was used for Infiniband), and needed a lossless network for good performance. Operators started enabling PFC to achieve very low loss networks, and the RoCE+PFC approach is now the dominant one for deploying RDMA over commodity Ethernet. As the usage of RoCE+PFC increased, operators

began noticing various problems and failure modes for PFC. These problems have been described at length in several papers [146, 122, 47, 58, 115], and there is now little doubt that the use of PFC has significant drawbacks.

At this point, the ecosystem (and the relevant components of the research community) faces a critical choice. It could attempt to fix the problems with PFC which, judging by the recent literature, is the path most are pursuing. However, this path would require both new insight (while there has been some progress in addressing some of PFC's problems, they still pose significant management and operational difficulties) and requiring changes to the core of most datacenter networks.

Our work is a call to take another path, one that revisits the assumption (that largely reflects the history of how RDMA initially got deployed, rather than a sound scientific investigation) that RDMA over Ethernet requires a lossless network. We believe our results suggest that we should turn to an architecture closer to iWARP (where the NIC implements efficient loss recovery), but with an implementation closer to RoCE (incrementally updating it as needed, rather than embodying an entirely different, and presumably, more complex transport stack). This approach offers the prospect of relatively cheap yet high performance NICs, while not requiring any changes to the network infrastructure or operation.

Chapter 5

Universal Packet Scheduling

There is a large and active research literature on novel packet scheduling algorithms, from simple schemes such as priority scheduling [111], to more complicated mechanisms for achieving fairness [117, 30, 101], to schemes that help reduce tail latency [27] or flow completion time [11], and this short list barely scratches the surface of past and current work. These scheduling algorithms must be implemented in the switch hardware to keep up with high bandwidth requirements, making it difficult to support different scheduling algorithms for different performance requirements.

In this chapter, we therefore ask if there is a single *universal* packet scheduling algorithm that can mimic existing scheduling algorithms, and obviate the need for new ones. In this context, we consider a packet scheduling algorithm to be both how packets are served inside the network (based on their arrival times and their packet headers) and how packet header fields are initialized and updated; this definition includes all the classical scheduling algorithms (FIFO, LIFO, priority, round-robin) as well as algorithms that incorporate dynamic packet state [124, 123, 27].

We can define a universal packet scheduling algorithm (hereafter UPS) in two ways, depending on our viewpoint on the problem. From a theoretical perspective, we call a packet scheduling algorithm *universal* if it can replay any *schedule* (the set of times at which packets arrive to and exit from the network) produced by any other scheduling algorithm. This is not of practical interest, since such schedules are not typically known in advance, but it offers a theoretically rigorous definition of universality that (as we shall see) helps illuminate its fundamental limits (i.e., which scheduling algorithms have the flexibility to serve as a UPS, and why).

From a more practical perspective, we say a packet scheduling algorithm is universal if it can achieve different desired performance objectives (such as fairness, reducing tail latencies and minimizing flow completion times). In particular, we require that the UPS should match the performance of the best known scheduling algorithm for a given performance objective. [1]

The notion of universality for packet scheduling might seem esoteric, but we think it helps clarify some basic questions. If there exists no UPS then we should *expect* to design new scheduling

[1] For this definition of universality, we allow the header initialization to depend on the objective being optimized. That is, while the basic scheduling operations must remain constant, the header initialization can depend on whether you are seeking fairness or minimal flow completion time.

algorithms as performance objectives evolve. Moreover, this would make a strong argument for switches being equipped with programmable packet schedulers so that such algorithms could be more easily deployed (as argued in [120]; in fact, it was the eloquent argument in this paper that caused us to initially ask the question about universality).

However, if there is indeed a UPS, then it changes the lens through which we view the design and evaluation of scheduling algorithms: e.g., rather than asking whether a new scheduling algorithm meets a performance objective, we should ask whether it is easier/cheaper to implement/configure than the UPS (which could also meet that performance objective). Taken to the extreme, one might even argue that the existence of a (practical) UPS greatly diminishes the need for programmable *scheduling* hardware.[2] Thus, while the rest of the paper occasionally descends into scheduling minutiae, the question we are asking has important practical (and intriguing theoretical) implications.

This paper starts from the theoretical perspective, defining a formal model of packet scheduling and our notion of replayability in §5.1. We first prove that there is no UPS, but then show that Least Slack Time First (LSTF) [77] comes as close as any scheduling algorithm to achieving universality. We also demonstrate empirically (via simulation) that LSTF can closely approximate the schedules of many scheduling algorithms. Thus, while not a perfect UPS in terms of replayability, LSTF comes very close to functioning as one.

We then take a more practical perspective in §5.2, showing (via simulation) that LSTF is comparable to the state of the art in achieving various objectives relevant to an application's performance. We investigate in detail LSTF's ability to minimize average flow completion times, minimize tail latencies, and achieve per-flow fairness. We also consider how LSTF can be used in multitenant situations to achieve multiple objectives simultaneously, while highlighting some of its key limitations.

In §5.3, we look at how network feedback for active queue management (AQM) can be incorporated using LSTF. Rather than augmenting the basic LSTF logic (which is restricted to packet scheduling) with a queue management algorithm, we show that LSTF can, instead, be used to implement AQM at the edge of the network. This novel approach to AQM is a contribution in itself, as it allows the algorithm to be upgraded without changing internal switches.

We then discuss the feasibility of implementing LSTF (§5.4) and provide an overview of related work (§5.5) before concluding with a discussion of open questions in §5.6.

5.1 Theory: Replaying Schedules

This section delves into the theoretical viewpoint of a UPS, in terms of its ability to *replay* a given schedule.

[2]Note that the case for programmable hardware as made in recent work on P4 and the RMT switch [19, 18] remains: these systems target programmability in header parsing and in how a packet's processing pipeline is defined (i.e., how forwarding 'actions' are applied to a packet). The P4 language does not currently offer primitives for scheduling and, perhaps more importantly, the RMT switch does not implement a programmable packet scheduler; we hope our results can inform the discussion on whether and how P4/RMT might be extended to support programmable scheduling.

5.1.1 Definitions and Overview

Network Model. We consider a network of store-and-forward output-queued switches connected by links. The input load to the network is a fixed set of packets $\{p \in P\}$, their arrival times $i(p)$ (i.e.when they reach the ingress switch), and the path $path(p)$ each packet takes from its ingress to its egress switch. We assume no packet drops, so all packets eventually exit. Every switch executes a non-preemptive scheduling algorithm which need not be work-conserving or deterministic and may even involve oracles that know about future packet arrivals. Different switches in the network may use different scheduling logic. For each incoming load $\{(p, i(p), path(p))\}$, a collection of scheduling algorithms $\{A_\alpha\}$ (switch α implements algorithm A_α) will produce a set of packet output times $\{o(p)\}$ (the time a packet p exits the network). We call the set $\{(path(p), i(p), o(p))\}$ a *schedule*.

Replaying a Schedule. Applying a different collection of scheduling algorithms $\{A'_\alpha\}$ to the same set of packets $\{(p, i(p), path(p))\}$ (with the packets taking the same path in the replay as in the original schedule), produces a new set of output times $\{o'(p)\}$. We say that $\{A'_\alpha\}$ *replays* $\{A_\alpha\}$ on this input if and only if $\forall p \in P$, $o'(p) \leq o(p)$.[3]

Universal Packet Scheduling Algorithm. We say a schedule $\{(path(p), i(p), o(p))\}$ is *viable* if there is at least one collection of scheduling algorithms that produces that schedule. We say that a scheduling algorithm is *universal* if it can replay *all* viable schedules. While we allowed significant generality in defining the scheduling algorithms that a UPS seeks to replay (demanding only that they be non-preemptive), we insist that the UPS itself obey several practical constraints (although we allow it to be preemptive for theoretical analysis, but then quantitatively analyze the non-preemptive version in §5.1.3).[4] The three practical constraints we impose on a UPS are:
(1) Uniformity and Determinism: A UPS must use the same deterministic scheduling logic at every switch.
(2) Limited state used in scheduling decisions: We restrict a UPS to using only (i) packet headers, and (ii) static information about the network topology, link bandwidths, and propagation delays. It cannot rely on oracles or other external information. However, it can modify the header of a packet before forwarding it (resulting in *dynamic packet state* [124]).
(3) Limited state used in header initialization: We assume that the header for a packet p is initialized at its ingress node. The additional information available to the ingress for this initialization is limited to: (i) $o(p)$ from the original schedule[5] and (ii) $path(p)$. Later, we extend the kinds

[3] We allow the inequality because, if $o'(p) < o(p)$, one can delay the packet upon arrival at the egress node to ensure $o'(p) = o(p)$.

[4] The issue of preemption is somewhat complicated. Allowing the original scheduling algorithms to be preemptive allows packets to be fragmented, which then makes replay extremely difficult even in simple networks (with store-and-forward switches). However, disallowing preemption in the candidate UPS overly limits the flexibility and would again make replay impossible even in simple networks. Thus, we take the seemingly hypocritical but only theoretically tractable approach and disallow preemption in the original scheduling algorithms but allow preemption in the candidate UPS. In practice, when we care only about approximately replaying schedules, the distinction is of less importance, and we simulate LSTF in the non-preemptive form.

[5] Note that this ingress switch can directly observe $i(p)$ as the time the packet arrives.

of information the header initialization process can use, and find that this is a key determinant in whether one can find a UPS.

We make three observations about the above model. First, our model assumes greater capability at the edge than in the core, in keeping with common assumptions that the network edge is capable of greater processing complexity, exploited by many architectural proposals[123, 104, 22]. Second, when initializing a packet p's header, a UPS can only use the input time, output time and the path information for p itself, and must be *oblivious* [48] to the corresponding attributes for *other* packets in the network. Finally, the key source of impracticality in our model is the assumption that the output times $o(p)$ are known at the ingress. However, a different interpretation of $o(p)$ suggests a more practical application of replayability (and thus our results): *if we assign $o(p)$ as the "desired" output time for each packet in the network, then the existence of a UPS tells us that if these goals are viable then the UPS will be able to meet them.*

5.1.2 Theoretical Results

In this section we only summarize our key theoretical results. The detailed proofs are in Appendix A.

Existence of a UPS under omniscient initialization. Suppose we give the header-initialization process extensive information in the form of times $o(p,\alpha)$ which represent when p was scheduled by switch α in the original schedule. We can then insert an n-dimensional vector in the header of every packet p, where the i^{th} element contains $o(p,\alpha_i)$ with α_i being the i^{th} hop in $path(p)$. Every time a packet arrives at a switch, the switch can pop the value at the head of this vector and use that as its priority (earlier values of output times get higher priority). This can perfectly replay any viable schedule (proof in Appendix A.1), which is not surprising, as having such detailed knowledge of the internal scheduling of the network is tantamount to knowing all the scheduling decisions made by the original algorithm. For reasons discussed previously, our definition limited the information available to the output time from the network as a whole, and not from each individual switch; we call this *black-box* initialization.

Nonexistence of a UPS under black-box initialization. We can prove by counter-example (described in Appendix A.2) that *there is no UPS* under the conditions stated in §5.1.1. We provide some intuition for the counter-example later in this section. Given this impossibility result, we now ask *how close can we get to a UPS?*

Natural candidates for a near-UPS. Simple priority scheduling [6] can reproduce all viable schedules on a single switch, so it would seem to be a natural candidate for a near-UPS. However, for multihop networks it may be important to make the scheduling of a packet dependent on what has happened to it earlier in its path. For this, we consider Least Slack Time First (LSTF) [77].

In LSTF, each packet p carries its slack value in the packet header, which is initialized to

[6] By simple priority scheduling, we mean that the ingress assigns priority values to the packets and the switches simply schedule packets based on these static priority values.

$slack(p) = (o(p) - i(p) - t_{min}(p, src(p), dest(p)))$ at the ingress; where $src(p)$ is the ingress of p, $dest(p)$ is the egress of p and $t_{min}(p, \alpha, \beta)$ is the time p takes to go from switch α to switch β in an uncongested network. Therefore, the slack of a packet indicates the maximum queueing time (excluding the transmission time at any switch) that the packet could tolerate without violating the replay condition. Each switch, then, schedules the packet which has the least remaining slack at the time when its last bit is transmitted. Before forwarding the packet, the switch overwrites the slack value in the packet's header with its remaining slack (i.e.the previous slack time minus the duration for which it waited in the queue before being transmitted).

An alternate way to implement this algorithm is having a static packet header as in Earliest Deadline First (EDF) and using additional state in the switches (reflecting the value of t_{min}) to compute the priority for a packet at each switch, but here we chose to use an approach with dynamic packet state. We provide more details about EDF and prove its equivalence to LSTF in Appendix A.4.

Key Results. Our analysis shows that the difficulty of replay is determined by the number of *congestion points*, where a *congestion point* is defined as a node where a packet is forced to "wait" during a given schedule. [7] Our theorems show the following key results:
1. Priority scheduling can replay all viable schedules with no more than one congestion point per packet, and there are viable schedules with no more than two congestion points per packet that it cannot replay. (Proof in Appendix A.5.)
2. LSTF can replay all viable schedules with no more than two congestion points per packet, and there are viable schedules with no more than three congestion points per packet that it cannot replay. (Proof in Appendix A.6.)
3. There is no scheduling algorithm (obeying the aforementioned constraints on UPSs) that can replay *all* viable schedules with no more than three congestion points per packet, and the same holds for larger numbers of congestion points. (Proof in Appendix A.2.)
Main Takeaway. *LSTF is closer to being a UPS than simple priority scheduling, and no other candidate UPS can do better in terms of handling more congestion points.*

Intuition. It is clear why LSTF is superior to priority scheduling: by carrying information about previous delays in the packet header (in the form of the *remaining* slack value), LSTF can "make up for lost time" at later congestion points, whereas for priority scheduling packets with low priority might repeatedly get delayed (and thus miss their target output times).

We now provide some intuition for why LSTF works for two congestion points and not for three, by presenting an outline of the proof detailed in Appendix A.6. We define the *local deadline* of a packet p at a switch α as the time when p is scheduled by α in the original schedule. The *global deadline* of p at α is defined as the time by when p must leave α in order to meet its target

[7] For our theoretical results, we adopt a pessimistic definition of a congestion point, where a switch that falls in the path of more than one flow is a congestion point (along with switches having output link capacity less than input link capacity or non work-conserving original schedules that make a packet wait explicitly). Since this definition is independent of per-packet dynamics, the set of congestion points remains the same in the original schedule and in the replay. This pessimistic definition is not required in practice, where the difficulty of replay would depend on the number of switches in a packet's path which see significant queuing.

output time, assuming that it sees no queuing delay after α. Hence, *global deadline* is the time when p's slack at α becomes zero. We can prove that *as long as all packets arrive at a switch at or before their local deadlines during the LSTF replay, no packet can miss its global deadline at α (i.e. no packet can have a negative slack at α).* The proof for this follows from the fact that if all packets arrive at or before their *local deadline* at α, there exists a *feasible* schedule where no packet misses its *global deadline* at α (this *feasible* schedule is the same as the original schedule at α). We can now apply the standard LSTF (or EDF) optimality proof technique for a single processor [80], to show that this feasible schedule can be iteratively *transformed* to a feasible LSTF schedule at switch α.

When there are only two congestion points per packet, it is guaranteed that every packet arrives at or before its local deadline at each congestion point during the LSTF replay. A packet can never arrive after its *local deadline* at its first congestion point, because it sees no queuing before that. Moreover, the *local deadline* is the same as the *global deadline* at the last congestion point. Therefore, if a packet arrives after its *local deadline* at its second (and last) congestion point, it means that it must have already missed its *global deadline* earlier, which, again, is not possible.

However, when there are three congestion points per packet, there is no guarantee that every packet arrives at or before its local deadline at each congestion point during the LSTF replay (due to the presence of a "middle" congestion point). One can, therefore, create counterexamples where unless LSTF (or, in fact, any other scheduling algorithm) makes precisely the right choice at the first congestion point of a packet p, at least one packet will miss its target output time, due to p arriving after its *local deadline* at its middle congestion point. We present such a counter-example in Appendix A.2, where we illustrate two ways of scheduling the same set of packets (having the same input times and paths) on a given topology with three congestion points per packet, resulting in two cases. The output times for two of the packets (named a and x), which compete with each other at the first congestion point (α_0), remains the same in both cases. However, one case requires scheduling a before x at α_0 and the second case requires scheduling x before a at α_0, else a packet will end up missing its target output time at the second (or middle) congestion points of a and x respectively. Since the information available for header initialization for the two packets is the same in both cases, no deterministic scheduling algorithm with blackbox header initialization can make the *correct* choice at the first congestion point *in both cases*.

5.1.3 Empirical Results

The previous section clarified the theoretical limits on a *perfect* replay. Here we investigate, via ns-2 simulations [98], how well (a non-preemptable version of) LSTF can *approximately* replay schedules in realistic networks. [8]

Experiment Setup.

Default scenario. We use a simplified Internet-2 topology [63], identical to the one used in [92] (consisting of 10 core switches connected by 16 links). We connect each core switch to 10 edge

[8] We have made our simulator code available at http://netsys.github.io/ups/.

switches using 1Gbps links and each edge switch is attached to an end host via a 10Gbps link. The number of hops per packet is in the range of 4 to 7, excluding the end hosts. We refer to this topology as I2 1Gbps-10Gbps. Each end host generates UDP flows using a Poisson inter-arrival model, with the destination picked randomly for each flow. Our default scenario runs at 70% utilization. The flow sizes are picked from a heavy-tailed distribution [13, 15]. Since our focus is on packet scheduling, not dropping policies, we use large buffer sizes that ensure no packet drops. Note that we use higher than usual access bandwidths for our default scenario to increase the stress on the schedulers in the core switches, where the number of congestion points seen by most packets is two, three or four for 22%, 44% and 24% packets respectively. [9] We also present results for smaller (and more realistic) access bandwidths, where most packets see smaller number of congestion points (one, two or three for 18%, 46% and 26% packets respectively), resulting in better replay performance.

Varying parameters. We tested a wide range of experimental scenarios by varying different parameters from their default values. We present results for a small subset of these scenarios here: (1) the default scenario with network utilization varied from 10-90% (2) the default scenario but with 1Gbps link between the endhosts and the edge switches (I2 1Gbps-1Gbps), with 10Gbps links between the edge switches and the core (I2 10Gbps-10Gbps) and with all link capacities in the I2 1Gbps-1Gbps topology reduced by a factor of 10 (I2 / 10) and (3) the default scenario applied to two different topologies, a bigger Rocketfuel topology [84] (with 83 core switches connected by 131 links) and a full bisection bandwidth datacenter (fat-tree) topology from [11] (with 10Gbps links). Note that our other results were generally consistent with those presented here.

Scheduling algorithms. Our default case, which we expected to be hard to replay, uses completely arbitrary schedules produced by a *random* scheduler (which picks the packet to be scheduled randomly from the set of queued up packets). We also present results for more traditional packet scheduling algorithms: FIFO, LIFO, fair queuing [30], and SJF (shortest job first using priorities). We also looked at two scenarios with a mixture of scheduling algorithms: one where half of the switches run FIFO+ [27] and the other half run fair queuing, and one where fair queueing is used to isolate two classes of traffic, with one class being scheduled with SJF and the other class being scheduled with FIFO.

Evaluation Metrics. We consider two metrics. First, we measure the fraction of packets that are overdue (i.e., which do not meet the original schedule's target). Second, to capture the *extent* to which packets fail to meet their targets, we measure the fraction of packets that are overdue by more than a threshold value T, where T is one transmission time on the bottleneck link ($\approx 12\mu s$ for 1Gbps). We pick this value of T both because it is sufficiently small that we can assume being overdue by this small amount is of negligible practical importance, and also because this is the order of violation we should expect given that our implementation of LSTF is non-preemptive. While we may have many small violations of replay (because of non-preemption), one would hope

[9] To compute this, we record the number of non-empty queues (excluding the endhost queues) encountered by each packet.

Topology	Avg. Link Utilization	Scheduling Algorithm	Fraction of packets overdue	
			Total	$> T$
I2 1Gbps-10Gbps	70%	Random	0.0021	0.0002
I2 1Gbps-10Gbps	10% 30% 50% 90%	Random	0.0007 0.0281 0.0221 0.0008	0.0 0.0017 0.0002 4×10^{-6}
I2 1Gbps-1Gbps I2 10Gbps-10Gbps I2 / 10	70%	Random	0.0204 0.0631 0.0127	8×10^{-6} 0.0448 0.00001
Rocketfuel Datacenter	70%	Random	0.0246 0.0164	0.0063 0.0154
I2 1Gbps-10Gbps	70%	FIFO FQ SJF LIFO FQ/FIFO+ FQ: SJF/FIFO	0.0143 0.0271 0.1833 0.1477 0.0152 0.0297	0.0006 0.0002 0.0019 0.0067 0.0004 0.0003

Table 5.1: LSTF replay performance across various scenarios. T represents the transmission time at the bottleneck link.

that most such violations are less than T.

Results. Table 5.1 shows the simulation results for LSTF replay for various scenarios, which we now discuss.

(1) Replayability. Consider the column showing the fraction of packets overdue. In all but three cases (we examine these shortly) over 97% of packets meet their target output times. In addition, the fraction of packets that did not arrive within T of their target output times is much smaller; even in the worst case of SJF scheduling (where 18.33% of packets failed to arrive by their target output times), only 0.19% of packets are overdue by more than T. Most scenarios perform substantially better: e.g., in our default scenario with Random scheduling, only 0.21% of packets miss their targets and only 0.02% are overdue by more than T. Hence, we conclude that even without preemption LSTF achieves good (but not perfect) replayability under a wide range of scenarios.

(2) Effect of varying network utilization. The second row in Table 5.1 shows the effect of varying network utilization. We see that at low utilization (10%), LSTF achieves exceptionally good replayability with a total of only 0.07% of packets overdue. Replayability deteriorates as utilization is increased to 30% but then (somewhat surprisingly) improves again as utilization increases. This improvement occurs because with increasing utilization, the amount of queuing (and thus the average slack across packets) in the original schedule also increases. This provides more room for

slack re-adjustments when packets wait longer at queues seen early in their paths during the replay. We observed this trend in all our experiments though the exact location of the "low point" varied across settings.

(3) Effect of varying link bandwidths. The third row shows the effect of changing the relative values of access/edge *vs.* core links. We see that while decreasing access link bandwidth (I2 1Gbps-1Gbps) resulted in a much smaller fraction of packets being overdue by more than T (0.0008%), increasing the edge-to-core link bandwidth (I2 10Gbps-10Gbps) resulted in a significantly higher fraction (4.48%). For I2 1Gbps-1Gbps, packets are paced by the endhost link, resulting in few congestion points thus improving LSTF's replayability. In contrast, with I2 10Gbps-10Gbps, both the access and edge links have a higher bandwidth than most core links; hence packets (that are no longer paced at the endhosts or the edges) arrive at the core switches very close to one another and hence the effect of one packet being overdue *cascades* over to the following packets. Decreasing the absolute bandwidths in I2 / 10, while keeping the ratio between access and edge links the same as that in I2 1Gbps-1Gbps, did not produce significantly different results compared to I2 1Gbps-1Gbps, indicating that the relative link capacities have a greater impact on the replay performance than the absolute link capacities.

(4) Effect of varying topology. The fourth row in Table 5.1 shows our results using different topologies. LSTF performs well in both cases: only 2.46% (Rocketfuel) and 1.64% (datacenter) of packets fail replay. These numbers are still somewhat higher than our default case. The reason for this is similar to that for the I2 10Gbps-10Gbps topology – all links in the datacenter fat-tree topology are set to 10Gbps, while in our simulations, we set half of the core links in the Rocketfuel topology to have bandwidths smaller than the access links.

(5) Varying Scheduling Algorithms. Row five in Table 5.1 shows LSTF's ability to replay different scheduling algorithms. We see that LSTF performs well for FIFO, FQ, and the combination cases (a mixture of FQ/FIFO+ and having FQ share between FIFO and SJF); e.g., with FIFO, fewer than 0.06% of packets are overdue by more than T. However, there are two problematic cases: SJF and LIFO fare worse with 18.33% and 14.77% of packets failing replay (although only 0.19% and 0.67% of packets are overdue by more than T respectively). The reason stems from a combination of two factors: (1) for these algorithms a larger fraction of packets have a very small slack value (as one might expect from the scheduling logic which produces a larger skew in the slack distribution), and (2) for these packets with small slack values, LSTF *without preemption* is often unable to "compensate" for misspent slack that occurred earlier in the path. To verify this intuition, we extended our simulator to support preemption and repeated our experiments: with preemption, the fraction of packets that failed replay dropped to 0.24% (from 18.33%) for SJF and to 0.25% (from 14.77%) for LIFO.

(6) End-to-end (Queuing) Delay. Our results so far evaluate LSTF in terms of measures that we introduced to test universality. We now evaluate LSTF using the more traditional metric of packet delay, focusing on the queueing delay a packet experiences. Figure 5.1 shows the CDF of the ratios of the queuing delay that a packet sees with LSTF to the queuing delay that it sees in the

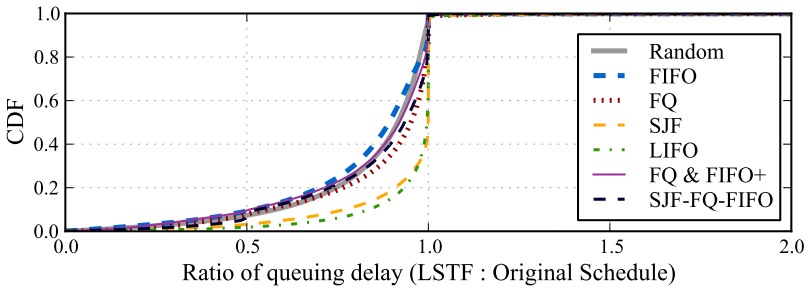


Figure 5.1: Ratio of queuing delay with varying packet scheduling algorithms, on I2 1Gbps-10Gbps topology at 70% utilization.

original schedule, for varying packet scheduling algorithms. We were surprised to see that most of the packets actually have a smaller queuing delay in the LSTF replay than in the original schedule. This is because LSTF eliminates "wasted waiting", in that it never makes packet A wait behind packet B if packet B is going to have significantly more waiting later in its path.

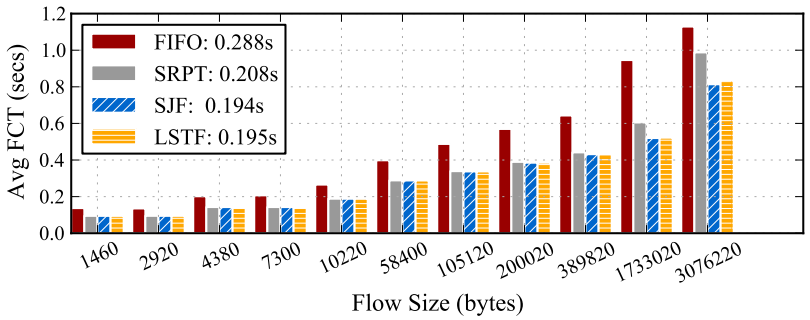
(7) Comparison with Priorities. To provide a point of comparison, we also did a replay using simple priorities for our default scenario, where the priority for a packet p is set to $o(p)$ (which seemed most intuitive to us). As expected, the resulting replay performance is much worse than LSTF: 21% packets are overdue in total, with 20.69% being overdue by more than T. For the same scenario, LSTF has only 0.21% packets overdue in total, with merely 0.02% packets overdue by more than T.

Summary. We observe that, in almost all cases, less than 1% of the packets are overdue with LSTF by more than T. The replay performance initially degrades and then starts improving as the network utilization increases. The distribution of link speeds has a bigger influence on the replay results than the scale of the topology. Replay performance is better for scheduling algorithms that produce a smaller skew in the slack distribution. LSTF replay performance is significantly better than simple priorities replay performance, with the most intuitive priority assignment.

5.2 Practical: Achieving Various Objectives

While replayability demonstrates the theoretical flexibility of LSTF, it does not provide evidence that it would be practically useful. In this section we look at how LSTF can be used *in practice* to meet the following performance objectives: minimizing average flow completion times, minimizing tail latencies, and achieving per-flow fairness.

Since the knowledge of a previous schedule is unavailable in practice, instead of using a given set of output times (as done in §5.1.3), we now use heuristics to assign the slacks in an effort to achieve these objectives. Our goal here is not to outperform the state-of-the-art for each objective in all scenarios, but instead we aim to be competitive with the state-of-the-art in most common cases.



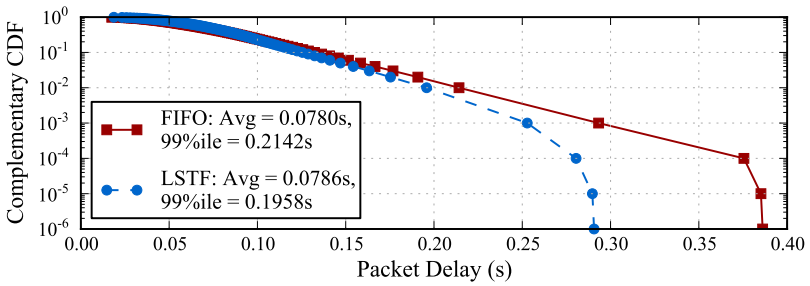
Expt. Setup	Avg FCT (s)			
	FIFO	SRPT	SJF	LSTF
I2 1Gbps-10Gbps at 30% util.	0.189	0.183	0.182	0.182
I2 1Gbps-10Gbps at 50% util.	0.212	0.189	0.185	0.185
I2 1Gbps-10Gbps at 70% util.	0.288	0.208	0.194	0.195
I2 1Gbps-1Gbps at 70% util.	0.252	0.209	0.202	0.202
I2 / 10 at 70% util.	0.899	0.658	0.620	0.621
Rocketfuel at 70% util.	0.305	0.240	0.228	0.228
Datacenter at 70% util.	0.058	0.018	0.016	0.015

Figure 5.2: The graph shows the average FCT bucketed by flow size obtained with FIFO, SRPT and SJF (using priorities and LSTF) for I2 1Gbps-10Gbps at 70% utilization. The legend indicates the average FCT across all flows. The table indicates the average FCTs for varying settings.

In presenting our results for each objective, we first describe the slack initialization heuristic we use and then present some ns-2 [98] simulation results on (i) how LSTF performs relative to the state-of-the-art scheduling algorithm and (ii) how they both compare to FIFO scheduling (as a baseline to indicate the overall impact of specialized scheduling for this objective). As our default case, we use the I2 1Gbps-10Gbps topology using the same workload as in the previous section (running at 70% average utilization). We also present aggregate results at different utilization levels and for variations in the default topology (I2 1Gbps-1Gbps and I2 / 10), for the bigger Rocketfuel topology, and for the datacenter topology (for selected objectives). The switches use non-preemptive scheduling (including for LSTF) and have finite buffers (packets with the highest slack are dropped when the buffer is full). Unless otherwise specified, our experiments use TCP flows with switch buffer sizes of 5MB for the WAN simulations (equal to the average bandwidth-delay product for our default topology) and 500KB for the datacenter simulations.

5.2.1 Average Flow Completion Time

While there have been several proposals on how to minimize flow completion time (FCT) via the transport protocol [34, 92], here we focus on *scheduling*'s impact on FCT, while using standard TCP New Reno at the endhosts. In [11] it is shown that (i) Shortest Remaining Processing Time (SRPT) is close to optimal for minimizing the mean FCT and (ii) Shortest Job First (SJF) produces results similar to SRPT for realistic heavy-tailed distribution. Thus, these are the two algorithms we use as benchmarks.



Expt. Setup	Avg Delay (s)		99%ile Delay (s)	
	FIFO	LSTF	FIFO	LSTF
I2 1Gbps-10Gbps at 30% util.	0.0411	0.0411	0.0911	0.0868
I2 1Gbps-10Gbps at 50% util.	0.0516	0.0517	0.1288	0.1195
I2 1Gbps-10Gbps at 70% util.	0.0780	0.0786	0.2142	0.1958
I2 1Gbps-1Gbps at 70% util.	0.0771	0.0771	0.2163	0.216
I2 / 10 at 70% util.	0.5762	0.5765	1.9393	1.9367
Rocketfuel at 70% util.	0.1891	0.1883	3.8139	3.7199
Datacenter at 70% util.	0.0250	0.0240	0.1352	0.1100

Figure 5.3: Tail packet delays for LSTF compared to FIFO. The graph shows the complementary CDF of packet delays for the I2 1Gbps-10Gbps topology at 70% utilization with the average and 99%ile packet delay values indicated in the legend. The table shows the corresponding results for varying settings.

Slack Initialization. We make LSTF emulate SJF by initializing the slack for a packet p as $slack(p) = fs(p) * D$, where $fs(p)$ is the size of the flow to which p belongs (in terms of the number of MSS-sized packets in the flow) and D is a value much larger than the queuing delay seen by any packet in the network. We use a value of $D = 1$ sec for our simulations.

Evaluation. Figure 5.2 compares LSTF with three other scheduling algorithms – FIFO, SJF and SRPT with *starvation prevention* as in [11]. Both SJF and SRPT have significantly lower mean FCT than FIFO. The LSTF based execution of SJF produces nearly the same results as the strict priorities based execution.

5.2.2 Tail Packet Delays

Clark et. al. [27] proposed the FIFO+ algorithm, where packets are prioritized at a switch based on the amount of queuing delay they have seen at their previous hops, for minimizing the tail packet delays in multi-hop networks. FIFO+ is *identical* to LSTF scheduling where all packets are initialized with the same slack value.

Slack Initialization. All incoming packets are initialized with the same slack value (we use an initial slack value of 1 second in our simulations). With the slack update taking place at every switch, the packets that have waited longer in the network queues are naturally given preference over those that have waited for a smaller duration.

Evaluation. We compare LSTF (which, with the above slack initialization, is identical to FIFO+)

with FIFO, the primary metric being the 99%ile end-to-end one way delay seen by the packets. Figure 5.3 shows our results. To better understand the impact of the two scheduling policies on the packet delays, our evaluation uses an open-loop setting with UDP flows. With LSTF, packets that have traversed through more number of hops, and have therefore spent more slack in the network, get preference over shorter-RTT packets that have traversed through fewer hops. While this might produce a slight increase in the average packet delay, it reduces the tail. This is in-line with the observations made in [27].

5.2.3 Fairness

Fairness is a common scheduling goal, which involves two different aspects: *asymptotic* bandwidth allocation (eventual convergence to the fair-share rate) and *instantaneous* bandwidth allocation (enforcing this fairness on small time-scales, so every flow experiences the equivalent of a per-flow pipe). The former can be measured by looking at long-term throughput measures, while the latter is best measured in terms of the flow completion times of relatively short flows (which measures bandwidth allocation on short time scales). We now show how LSTF can be used to achieve both of these goals, but more effectively the former than the latter. Our slack assignment heuristic can also be easily extended to achieve weighted fair queuing, but we do not present those results here.

Slack Initialization. The slack assignment for fairness works on the assumption that we have some ballpark notion of the fair-share rate for each flow and that it does not fluctuate wildly with time. Our approach to assigning slacks is inspired from [145]. We assign $slack = 0$ to the first packet of the flow and the slack of any subsequent packet p_i is then initialized as:

$$slack(p_i) = max\Big(0,\ slack(p_{i-1}) + \frac{size(p_i)}{r_{est}} - \big(i(p_i) - i(p_{i-1})\big)\Big)$$

where $i(p)$ is the arrival time of a packet p at the ingress, $size(p)$ is its size in bits, and r_{est} is an estimate of the fair-share rate r^* in bps. We show that the above heuristic leads to asymptotic fairness, for *any* value of r_{est} that is less than r^*, as long as all flows use the same value. The same heuristic can also be used to provide instantaneous fairness, when we have a complex mix of short-lived flows, where the r_{est} value that performs the best depends on the link bandwidths and their utilization levels. A reasonable value of r_{est} can be estimated using knowledge about the network topology and traffic matrices, though we leave a detailed exploration of this to future work.

Evaluation: Asymptotic Fairness. We evaluate the asymptotic fairness property by running our simulation on the Internet2 topology with 10Gbps edges, such that all the congestion happens at the core. However, we reduce the propagation delay to 10μs for each link, to make the experiment more scalable, while the buffer size is kept large (50MB) so that fairness is dominated by the scheduling policy and not by how TCP reacts to packet drops. We start 90 long-lived TCP flows with a random jitter in the start times ranging from 0-5ms. The topology is such that the fair share rate of each flow on each link in the core network (which is shared by up to 13 flows) is around 1Gbps. We use different values for $r_{est} \leq$ 1Gbps for computing the initial slacks and compare

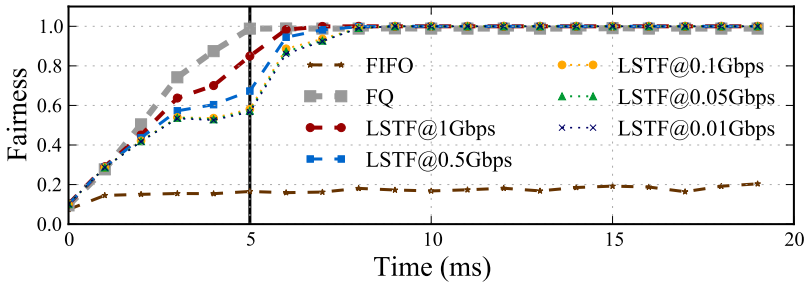


Figure 5.4: Fairness for long-lived flows on Internet2 topology. The legend indicates the value of r_{est} used for LSTF slack initialization.

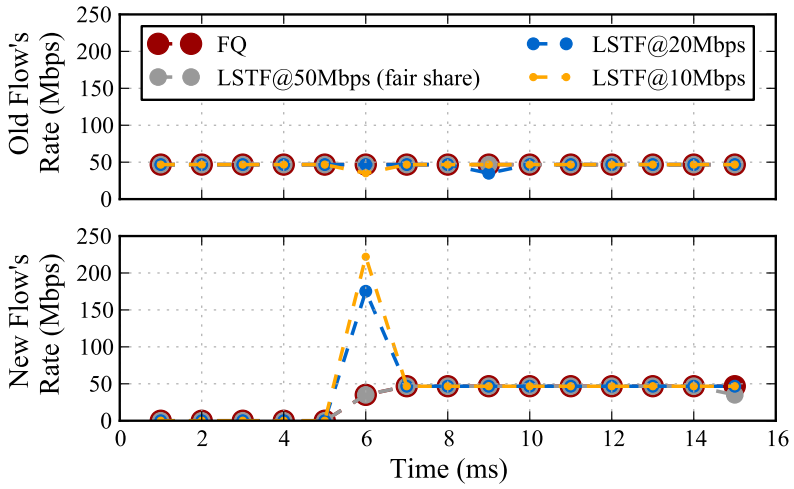
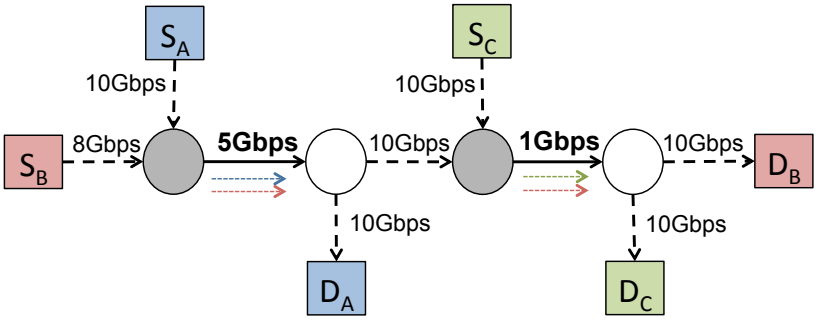


Figure 5.5: Control experiment: 20 flows share a single bottleneck link of 1Gbps and a 21st flow is added after 5ms. The graph shows the rate allocations for an old flow and the new flow with Fair Queuing and for LSTF with varying r_{est}.

our results with fair queuing (FQ). Figure 5.4 shows the fairness computed using Jain's Fairness Index [67], from the throughput each flow receives per millisecond. Since we use the throughput received by each of the 90 flows to compute the fairness index, it reaches 1 with FQ only at 5ms, after all the flows have started. We see that LSTF is able to converge to perfect fairness, even when r_{est} is 100X smaller than r^*. It converges slightly sooner when r_{est} is closer to r^*, though the subsequent differences in the time to convergence decrease with decreasing values of r_{est}.

Intuition for why it works. The reason behind why any slack assignment with $r_{est} < r^*$ leads to convergence to fairness is quite straight-forward and is explained by the control experiment shown in Figure 5.5. 20 long-lived TCP flows share a single bottleneck link of 1Gbps (giving a fair share rate of 50Mbps) and a 21st flow is added after 5ms. Since the first 20 flows have started early, the queue at the bottleneck link already contains packets belonging to these flows.

When $r_{est} = 50Mbps$, the actual queuing delay experienced by a packet is almost equal to the



r_{est} value (Mbps)			Expected Throughput (Mbps)			Achieved Throughput (Mbps)		
A	B	C	A	B	C	A	B	C
2000	100	100	4761	238	762	4762	238	763
900	100	100	4500	500	500	4499	501	500
500	100	100	4167	500	500	4166	501	500
200	100	100	3333	500	500	3333	501	500
100	100	100	2500	500	500	2500	500	501
100	100	500	2500	167	833	2500	167	834

Figure 5.6: Weighted Fairness on a multi-bottleneck topology (drawn above). The link capacities and the source/destination of each flow are indicated in the figure. Flows A and B share a 5Gbps link and then Flows B and C share a 1Gbps link.

slack value assigned to it. Therefore, at any given point of time, the first packet of each flow present in the queue will have a slack value which is approximately equal to zero. The next packet of each flow will have a higher slack value (around 1500bytes/50Mbps = 0.24ms). By the time the corresponding first packets of every flow in the queue have been transmitted, the slack values of the next packet would also have been reduced to zero and so on. It therefore produces a round-robin pattern for scheduling packets across flows, as is done by FQ. Therefore, when the 21st flow starts at 5ms, with the first packet coming in with zero slack, the next one with 0.24ms slack and so on, it immediately starts following the round-robin pattern as well.

However, when r_{est} is smaller than $50Mbps$, then the packets of the old flows already present in the queue have a higher slack value than what they actually experience in the network. The first packet of every flow in the queue therefore has a slack which is more than 0 when the 21st flow comes in at 5ms. The earlier packets of the new flow therefore get precedence over any of the existing packets of the old flows, resulting in the spike in the rate allocated to the new flow as shown in Figure 5.5. Nonetheless, with the slack of every newly arriving packet of the 21st flow being higher than the previous one and with the slack of the already queued up packet decreasing with time, the slack value of the first packet in the queue for new flow and the old flows soon *catch up* with each other and the schedule starts following a round robin pattern again. The closer r_{est} is to the fair-share rate, the sooner the slack values of the old flows and the new flow *catch up* with each other. The duration for which a packet ends up waiting in the queue is upper-bounded by the time it would have waited, had all the flows arrived at the same time and were being serviced at their fair share rate.

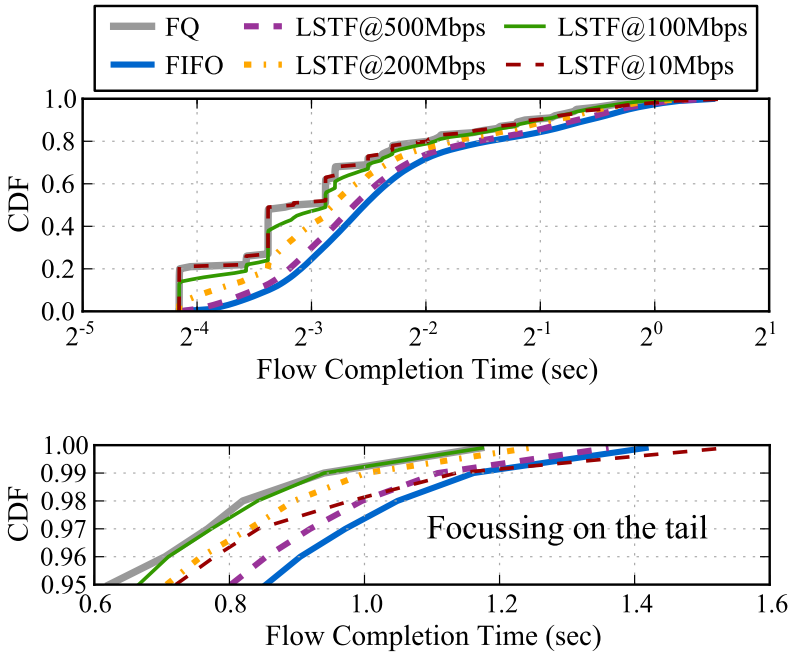


Figure 5.7: CDF of FCTs for the I2 1Gbps-10Gbps topology at 70% utilization.

Weighted Fairness with multiple-bottlenecks. One can see how the above logic can be extended for achieving weighted fairness. Moreover, when a packet sees multiple bottlenecks, the slack update (subtraction of the duration for which the packet waits) at the first bottleneck ensures that the next bottleneck takes into account the rate-limiting happening at the first one and the packets are given precedence accordingly.

We did a control experiment to evaluate weighted fairness with LSTF on a multi-bottleneck topology. We started three UDP flows with a start-time jitter between 0 and 1ms, on the topology as shown in Figure 5.6. We ran the simulation for 30ms and computed the throughput each flow received for the last 15ms. We varied the values of r_{est} used for assigning slacks to each flow, relative to one another, to assign different weights to different flows. For example, r_{est} assignment $\{A : 900Mbps, B : 100Mbps, C : 100Mbps\}$ results in Flow A getting 9 times more share on the 5Gbps link than Flow B, with Flows B and C sharing the 1Gbps link equally. We compute the expected throughput based on the assigned r_{est} values and find that the throughput actually achieved is almost the same, as shown in the table.

Evaluation: Instantaneous Fairness. As one might expect, the choice of r_{est} has a bigger impact on instantaneous fairness than on asymptotic fairness. A very high r_{est} value would not provide sufficient isolation across flows. On the other hand, a very small r_{est} value can starve the long flows. This is because the assigned slack values for the later packets of long flows with high sequence numbers would be much higher than the actual slack they experience. As a result, they will end up waiting longer in the queues, while the initial packets of newer flows with smaller slack values would end up getting a higher precedence.

Expt. Setup	Avg FCT across bytes (s)			Best r_{est}	Reasonable r_{est} Range
	FIFO	FQ	LSTF	(Mbps)	(Mbps)
I2 1Gbps-10Gbps at 30% util.	0.563	0.537	0.538	300	10-900
I2 1Gbps-10Gbps at 50% util.	0.626	0.549	0.555	200	10-800
I2 1Gbps-10Gbps at 70% util.	0.811	0.622	0.632	100	50-200
I2 1Gbps-1Gbps at 70% util.	0.766	0.630	0.652	100	50-400
I2 / 10 at 70% util.	4.838	2.295	2.759	10	10-20
Rocketfuel at 70% util.	0.964	0.796	0.824	100	50-300

Table 5.2: FCT averaged across bytes for FIFO, FQ and LSTF (with best r_{est} value) across varying settings. The last column indicates the range of r_{est} values that produce results within 10% of the best r_{est} result.

To verify this intuition, we evaluated our LSTF slack assignment scheme by running our standard workload with a mix of TCP flows ranging from sizes 1.5KB - 3MB on our default I2 1Gbps-10Gbps topology at 70% utilization, with 50MB buffer size. Note that the traffic pattern is now bursty and the instantaneous utilization of a link is often lower or higher than the assigned average utilization level. The CDF of the FCTs thus obtained is shown in Figure 5.7. As expected, the distribution of FCTs looks very different between FQ and FIFO. FQ isolates the flows from each-other, significantly reducing the FCT seen by short to medium size flows, compared to FIFO. The long flows are also helped a little by FQ, again due to the isolation provided from one-another.

LSTF performance varies somewhere in between FIFO and FQ, as we vary r_{est} values between 500Mbps to 10Mbps. A high value of r_{est} = 500Mbps does not provide sufficient isolation and the performance is close to FIFO. As we reduce the value of r_{est}, the "isolation-effect" increases. However, for very small r_{est} values (e.g. 10Mbps), the tail FCT (for the long flows) is much higher than FQ, due to the starvation effect explained before.

We try to capture this trade-off between isolation for short and medium sized flows and starvation for long flows, by using average FCT across bytes (in other words, the average FCT weighted by flow size) as our key metric. We term the r_{est} value that achieves the sweetest spot in this trade-off as the "best" r_{est} value. The r_{est} values that produce average FCT which is within 10% of the value produced by the best r_{est} are termed as "reasonable" r_{est} values. Table 5.2 presents our results across different settings. We find that (1) LSTF produces significantly lower average FCT than FIFO, performing only slightly worse than FQ (2) As expected, the best r_{est} value decreases with increasing utilization and with decreasing bandwidths (as in the case of I2 / 10 topology), while the range of reasonable r_{est} values gets narrower with increasing utilization and with decreasing bandwidths.

Thus, for instantaneous fairness, LSTF would require some estimate of the per-flow rate. We believe that this can be obtained from the knowledge of the network topology (in particular, the link bandwidths), which is available to the ISPs, and on-line measurement of traffic matrices and link utilization levels, which can be done using various tools [26, 103, 18]. However, this does impose a higher burden on deploying LSTF than on FQ or other such scheduling algorithms.

5.2.4 Limitations of LSTF: Policy-based objectives

So far we showed how LSTF achieves various performance objectives. We now describe certain policy-based objectives that are hard to achieve with LSTF.

Multi-tenancy. As network virtualization becomes more popular, networks are often called upon to support multiple tenants or traffic classes, with each having their own networking objectives. Network providers can enforce isolation across such tenants (or classes of traffic) through static bandwidth provisioning, which can be implemented via dedicated hard-wired links [45, 89] or through multiqueue scheduling algorithms such as fair queuing or round robin [30]. LSTF can work in conjunction with both of these isolation mechanisms to meet different desired performance objectives for each tenant (or class of traffic).

However, without such multiqueue support it cannot provide such isolation or fairness on a per-class or per-tenant basis. This is because for class-based fairness (which also includes hierarchical fairness) the appropriate slack assignment for a packet at a particular ingress depends on the input from other ingresses (since these packets can belong to the same class). Note, however, that if two or more classes/tenants are separated by strict prioritization, LSTF can be used to enforce the appropriate precedence order, along with meeting the individual performance objective for each class.

Traffic Shaping. Shaping or rate limiting flows at a particular switch requires non-work-conserving algorithms such as Token Bucket Filters [131]. LSTF itself is a work-conserving algorithm and cannot shape or rate limit the traffic on its own. We believe that shaping the traffic only at the edge, with the core remaining work-conserving, would also produce the desired network-wide behavior, though this requires further exploration.

5.3 Incorporating Network Feedback

Up until now we have considered packet scheduling in isolation, whereas in the Internet today switches send implicit feedback to hosts via packet drops [41, 96] (or marking, as in ECN [106]). This is often called Active Queue Management (AQM), and its goal is to reduce the per-packet delays while keeping throughput high. We now consider how we might generalize our LSTF approach to incorporate such network feedback as embodied in AQM schemes.

LSTF is just a scheduling algorithm and cannot perform AQM on its own. Thus, at first glance, one might think that incorporating AQM into LSTF would require implementing the AQM scheme in each switch, which would then require us to find a universal AQM scheme in order to fulfill our pursuit of universality. On the contrary, LSTF enables a novel edge-based approach to AQM based on the following insights: (1) As long as appropriate packets are chosen, it does not matter *where* they are being dropped (or marked) – whether it is inside the core switches or at the edge. (2) In addition to scheduling packets LSTF produces a very useful by-product, carried by the slack values in the packets, which gives us a precise measure of the one-way queuing delay seen by the packet and can be used for AQM. For obtaining this by-product, an extra field is added to the packet

header at the ingress which stores the assigned slack value (called the initial slack field), which remains untouched as the packet traverses the network. The other field where the ingress stores the assigned slack value is updated as per the LSTF algorithm; we call this the current slack field. The precise amount of queuing delay seen by the packet within the network (or the *used slack* value) can be computed at the edge by simply comparing the initial slack field and the current slack field.

We evaluate our edge-based approach to AQM in the context of (1) CoDel [96], the state-of-the-art AQM scheme for wide area networks and (2) ECN used with DCTCP [8], the state-of-the-art AQM scheme for datacenters.

5.3.1 Emulating CoDel from Edge

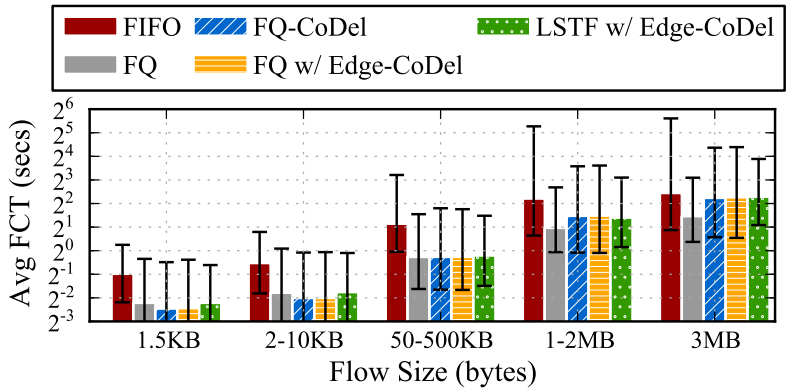
Background. In CoDel, the amount of time a packet has spent in a queue is recorded as the sojourn time. A packet is dropped if its sojourn time exceeds a fixed target (set to 5ms [95]), and if the last packet drop happened beyond a certain interval (initialized to 100ms [95]). When a packet is dropped, the interval value is reduced using a control law, which divides the initial interval value by the square root of the number of packets dropped. The interval is refreshed (re-initialized to 100ms) when the queue becomes empty, or when a packet sees a sojourn time less than the target.[10] An extension to CoDel is FQ-CoDel [53], where the scheduler round-robins across flows and the CoDel control loop is applied to each flow individually. The interval for a flow is refreshed when there are no more packets belonging to that flow in the queue. FQ-CoDel is considered to be better than CoDel in all regards , even by one of the co-developers of CoDel [73].

Edge-CoDel. We aim to approximate FQ-CoDel from the edge by using LSTF to implement per-flow fairness in switches (as in §5.2.3). We then compute the used slack value at the egress switch for every packet, as described above, and run the FQ-CoDel logic for when to drop packets for each flow, keeping the control law and the parameters (the target value and the initial interval value) the same as in FQ-CoDel. We call this approach Edge-CoDel.

There are only two things that change in Edge-CoDel as compared to FQ-CoDel. First, instead of looking at the sojourn time of each queue individually, Edge-CoDel looks at the total queuing time of the packet across the entire network. The second change is with respect to how the CoDel interval is refreshed. As mentioned before, in traditional FQ-CoDel, there are two events that trigger a refresh in the interval (i) when a packet's sojourn time is less than the target and (ii) when all the queued-up packets for a given flow have been transmitted. While Edge-CoDel can react to the former, it has no explicit way of knowing the latter. To address this, we refresh the interval if the difference in the send time of two consecutive packets (found using TCP timestamps that are enabled by default) is more than a certain threshold. Clearly, this *refresh threshold* must be greater than CoDel's target queuing delay value. We find that a refresh threshold of 2-4 times the target value (10-20ms) works reasonably well.

Evaluation. In our experiments, we compare four different schemes: (1) FIFO without AQM

[10] CoDel is a little more complicated than this, and while our implementation follows the CoDel specification [95], our explanation has been simplified, highlighting only the relevant points for brevity.

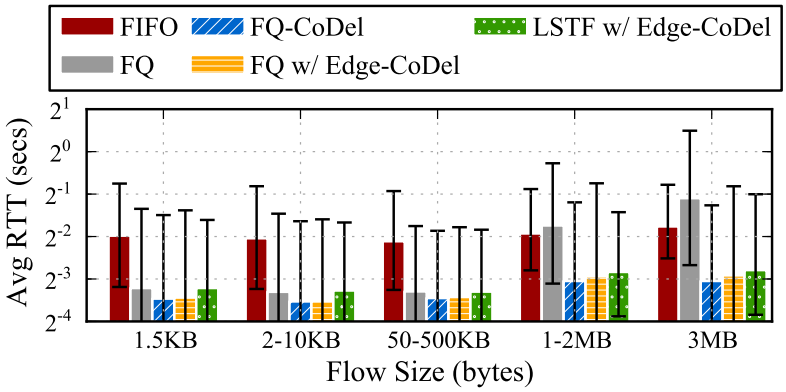


Expt. Setup	r_{est} (Mbps)	Avg FCT across bytes (s)				
		FIFO	FQ	FQ-CoDel	FQ w/ Edge-CoDel	LSTF w/ Edge-CoDel
I2 1Gbps-10Gbps at 70% util.	100	0.811	0.622	0.642	0.633	0.641
I2 1Gbps-1Gbps at 70% util.	100	0.766	0.630	0.642	0.637	0.658
I2 / 10 at 30% util.	40	0.918	0.836	0.897	0.887	0.907
I2 / 10 at 50% util.	30	1.706	1.214	1.430	1.369	1.427
I2 / 10 at 70% util.	10	4.837	2.295	3.687	3.738	3.739
I2 / 10, half RTTs at 70% util.	10	4.569	2.023	3.196	3.245	3.405
I2 / 10, double RTTs at 70% util.	10	5.098	2.769	4.243	4.125	4.389
Rocketfuel at 70% util.	100	0.964	0.796	0.840	0.813	0.835

Figure 5.8: The figure shows the average FCT values for I2 / 10 at 70% utilization (LSTF uses fairness slack assignment with $r_{est} = 10Mbps$). The error bars indicate the 10^{th} and the 99^{th} percentile values and the y-axis is in log-scale. The table indicates the average FCT (across bytes) for varying settings.

(to set a baseline), (2) FQ without AQM (to see the effects of FQ on its own), (3) FQ-CoDel (to provide the state-of-the-art comparison) (4) LSTF scheduling (with slacks assigned to meet the fairness objective using appropriate r_{est} values) in conjunction with Edge-CoDel. As we move from (3) to (4), we make two transitions – first is with respect to the scheduling done inside the network (perfect isolation with FQ vs approximate isolation with LSTF) and the second is the shift of AQM logic from inside the network to the edge. Therefore, as an incremental step in between the two transitions, we also provide results for FQ with Edge-CoDel, where switches do FQ across flows (with the slack values maintained only for book-keeping) and AQM is done by Edge-CoDel. This allows us see how well Edge-CoDel works with perfect per-switch isolation. The refresh threshold we use for Edge-CoDel in both cases is 20ms (4 times the CoDel target value). The buffer size is increased to 50MB so that AQM kicks in before a natural packet drop occurs.

The main metrics we use for evaluation are the FCTs and the per-packet RTTs, since the goal of an AQM scheme is to maintain high throughput (or small FCTs) while keeping the RTTs small. Figure 5.8 and Figure 5.9 show our results, for FCTs and RTTs respectively, for varying settings and schemes. The two graphs show the average FCT and the average RTT across flows bucketed by their size for the I2 / 10 topology at 70% utilization (where AQM produces a bigger impact compared to our default case). As expected, we find that while FQ helps in reducing the FCT



Expt. Setup	r_{est} (Mbps)	Avg RTT across bytes (s)				
		FIFO	FQ	FQ-CoDel	FQ w/ Edge-CoDel	LSTF w/ Edge-CoDel
I2 1Gbps-10Gbps at 70% util.	100	0.0756	0.0733	0.0642	0.0646	0.0661
I2 1Gbps-1Gbps at 70% util.	100	0.0716	0.0702	0.0639	0.0643	0.0666
I2 / 10 at 30% util.	40	0.0998	0.1085	0.0792	0.0798	0.0826
I2 / 10 at 50% util.	30	0.1384	0.1752	0.0901	0.0918	0.1001
I2 / 10 at 70% util.	10	0.2779	0.3752	0.1182	0.1281	0.1388
I2 / 10, half RTTs at 70% util.	10	0.2555	0.3607	0.0995	0.1131	0.1165
I2 / 10, double RTTs at 70% util.	10	0.325	0.4172	0.1591	0.1640	0.1843
Rocketfuel at 70% util.	100	0.0922	0.0991	0.0794	0.0788	0.0836

Figure 5.9: The figure shows the average RTT values for I2 / 10 at 70% utilization (LSTF uses fairness slack assignment with $r_{est} = 10Mbps$). The error bars indicate the 10^{th} and the 99^{th} percentile values and the y-axis is in log-scale. The table indicates the average RTTs (across bytes) for varying settings.

values as compared to FIFO, it results in significantly higher RTTs than FIFO for long flows. FQ-CoDel reduces the RTT seen by long flows compared to FQ (with the short flows having RTT smaller than FIFO and comparable to FQ). What is new is that, shifting the CoDel logic to the edge through Edge-CoDel while doing FQ in the switch makes very little difference as compared to FQ-CoDel. As we experiment with varying settings, we find that in some cases, FQ with Edge-CoDel results in slightly smaller FCTs at the cost of slightly higher RTTs than FQ-CoDel. We believe that this is due to the difference in how the CoDel interval is refreshed with Edge-CoDel and with in-switch FQ-CoDel. Replacing the scheduling algorithm with LSTF again produces minor differences in the results compared to FQ-CoDel. Both the FCT and the RTT are slightly higher than FQ-CoDel for almost all cases, and we attribute the differences to LSTF's *approximation* of round-robin service across flows. Nonetheless, the average FCTs obtained are significantly lower than FIFO and the average RTTs are significantly lower than both FIFO and FQ for all cases.

We next evaluate the sensitivity of Edge-CoDel to the refresh threshold. Table 5.3 shows the average FCT and RTT values for varying refresh thresholds. We find that there are very minor differences in the results as we vary this threshold, because the dominating cause for refreshing the interval is when a packet sees a queuing delay less than the CoDel target. However, the general trend is that increasing the refresh threshold increases the FCT and decreases the RTT. This

Refresh Threshold (ms)	Avg FCT across bytes (s)	Avg RTT across bytes (s)
10	3.578	0.143
20	3.739	0.139
30	3.954	0.135
40	4.079	0.132

Table 5.3: Effect of varying refresh threshold on I2/10 topology at 70% utilization running LSTF (r_{est} = 10Mbps) with Edge-CoDel.

Util.	Avg FCT (s)			Avg RTT (ms)		
	FIFO w/ No ECN	FIFO w/ ECN	LSTF w/ Edge-ECN	FIFO w/ No ECN	FIFO w/ ECN	LSTF w/ Edge-ECN
30%	0.0020	0.0011	0.0011	0.2069	0.1123	0.1077
50%	0.0219	0.0086	0.0079	0.3425	0.1601	0.1477
70%	0.0501	0.0241	0.0240	0.4497	0.2616	0.2494

Table 5.4: DCTCP performance with no ECN, ECN (in-switch) and Edge-ECN for the datacenter topology at varying utilizations.

is because with increasing refresh threshold, the interval is reset to the larger 100ms value less frequently. This results in more packet drops for the long flows, causing an increase in FCTs, but a decrease in the RTT values.

5.3.2 Emulating ECN for DCTCP from Edge

Background. DCTCP [8] is a congestion control scheme for datacenters that uses ECN marks as a congestion signal to control the queue size before a packet drop occurs. It requires the switches to mark the packets whenever the instantaneous queue size goes beyond a certain threshold K. These markings are echoed back to the sender with the acknowledgments and the sender decreases its sending rate in proportion to the ECN marked packets.

Edge-ECN. The marking process can be moved to the edge (or the receiving endhost) by simply marking a packet if its queuing delay (computed, as in §5.3.1, by subtracting the initial slack value from the current slack value) is greater than the transmission time of K packets. This transmission time is easy to compute in datacenters where the link capacities are known.

Evaluation. The results for varying utilization levels are shown in Table 5.4. We compare Edge-ECN running LSTF in the switches (with all packets initialized to the same slack value) with in-switch ECN running FIFO in the switches, both using the same unmodified DCTCP algorithm at the endhosts. We use the DCTCP default value of $K = 15$ packets as the marking threshold. We also present results for DCTCP with no ECN marks (which reduces to TCP) and FIFO scheduling, as a comparison point. We see that both in-switch ECN and Edge-ECN DCTCP have comparable performance, with significantly lower average FCTs and RTTs than no ECN TCP.

Summary. The *used slack* information available as a by-product from LSTF can be effectively used to emulate an AQM scheme from the edge of the network.

5.4 LSTF Implementation

In this section, we study the feasibility of implementing LSTF in the switches. We start with showing that given a switch that supports fine-grained priority scheduling, it is trivial to implement LSTF on it using programmable header processing mechanisms [18, 19]. We then explore two different proposals for implementing fine-grained priorities in hardware.

Using fine-grained priorities to implement LSTF. Consider a packet p that arrives at a switch α at time $i(p,\alpha)$, with slack $slack(p,\alpha)$. As mentioned in §5.1, LSTF prioritizes packets based on their remaining slack value at the time when their last bit is transmitted. This term is given by $(slack(p,\alpha) - (t - i(p,\alpha)) + T(p,\alpha))$ at any time t while p is waiting at α. $T(p,\alpha)$ is the transmission time of p at α, which is added to account for the remaining slack of p, relative to other packets, when its *last bit* is transmitted. Since t is same for all packets at any given point of time when the packets are being compared at α, the deciding term is $(slack(p,\alpha) + i(p,\alpha) + T(p,\alpha))$. With $slack(p,\alpha)$ being available in the packet header and the values of $i(p,\alpha)$ and $T(p,\alpha)$ being available at α when the packet arrives at the switch, this term can be easily computed and attached to the packet as its priority value. Right before a packet p is transmitted by the switch, its slack can be overwritten by the remaining slack value, computed by subtracting the stored priority value $(slack(p,\alpha) + i(p,\alpha) + T(p,\alpha))$ with the sum of the current time and $T(p,\alpha)$. We verified that these steps can be easily executed using P4 [18].

Implementing fine-grained priorities in hardware. Fine-grained priorities can be implemented by using specialized data-structures such as pipelined heap (p-heap) [16, 64], which can scale to very large buffers (>100MB), because the pipeline stage time is not affected by the queue size. However, p-heaps are difficult to implement and verify due to their intricate design and large chip area, thus resulting in higher costs. The p-heap implemented by Ioannou et. al. [64] using a 130nm technology node has a per-port area overhead of 10% (over a typical switching chip with minimum area of $200mm^2$ [44]) [11].

Leveraging the advancement in hardware technology over the years, Sivaraman et. al. [119] propose a simpler solution, based on bucket-sort algorithm. The area overhead reduces to only 1.65% (over a baseline single-chip shared-memory switch such as the Broadcom Trident [52]), when implemented using a 16nm technology node. While this approach is much cheaper to implement, it cannot scale to very large buffer sizes (beyond a few tens of MBs).

Thus, given these choices, it does not appear a significant challenge to implement LSTF at linespeed, though the key trade-offs between cost, simplicity and buffer limits need to be taken into consideration. To support a scale-out infrastructure, most datacenters today use a large number of inexpensive single chip shared memory switches [118], which have shallow buffers (around 10MB). The low overhead bucket-sort based approach [119] towards implementing LSTF would be ideal in such a setup. Core switches in wide area, on the other hand, have deep buffers (a few hundred MBs) and would require the more expensive p-heap based implementation [16, 64]. While

[11] 130nm technology node was developed in 2001; the overheads would be lower for an implementation using the latest technology (14nm).

they are fewer in number [79], they may cost up to millions of dollars. Supporting the slightly more expensive, but flexible LSTF implementation would, to a large extent, obviate the need for replacing these expensive switches with changing demands, resulting in long-term savings. We are also optimistic that advancements in hardware technology would further reduce the cost overheads of implementing LSTF.

5.5 Related Work

The literature on packet scheduling is vast. Here we only touch on a few topics most relevant to our work.

The real-time scheduling literature has studied the optimality of scheduling algorithms[12] (in particular EDF and LSTF) for single and multiple processors [80, 77]. Liu and Layland [80] proved the optimality of EDF for a single processor in hard real-time systems. LSTF was then shown to be optimal for single-processor scheduling, while being more effective than EDF (though not optimal) for multi-processor scheduling [77]. In the context of networking, [25] provides theoretical results on emulating the schedules produced by a single output-queued switch using a combined input-output queued switch with a smaller speed-up of at most two. To the best of our knowledge, the optimality or universality of a scheduling algorithm for a network of inter-connected resources (in our case, switches) has never been studied before.

The authors of [120] propose the use of programmable hardware in the dataplane for packet scheduling and queue management, in order to achieve various objectives. The proposal shows that there is no "silver bullet" solution, by simulating three schemes (FQ, CoDel+FQ, CoDel+FIFO) competing on three different metrics. As mentioned earlier, our work is inspired by the questions the authors raise; we adopt a broader view of scheduling in which packets can carry dynamic state leading to the results presented here. A recent proposal for programmable packet scheduling [119], developed in parallel to UPS, uses an hierarchy of priority and calendar queues to express different scheduling algorithms on a single switch hardware. The proposed solution is able to achieve better expressiveness than LSTF by allowing packet headers to be re-initialized at *every switch*. UPS assumes a stronger model, where the header initialization is restricted to the ingress switches, while the core switches remain untouched. Moreover, we provide theoretical results which shed light on the effectiveness of both of these models.

5.6 Conclusion

This chapter started with a theoretical perspective by analyzing whether there exists a single *universal* packet scheduling algorithm that can perfectly replay all viable schedules. We proved that while such an algorithm cannot exist, LSTF comes closest to being one (in terms of the number of congestion points it can handle). We then empirically demonstrated the ability of LSTF

[12] A scheduling algorithm is said to be optimal if it can (feasibly) schedule a set of tasks that can be scheduled by any other algorithm.

to approximately replay a wide range of scheduling algorithms under varying network settings. Replaying a given schedule, while of theoretical interest, requires the knowledge of viable output times, which is not available in practice.

Hence, we next considered if LSTF can be used in practice to achieve various performance objectives. We showed via simulation how LSTF, combined with heuristics to set the slack values at the ingress, can do a reasonable job of minimizing average flow completion time, minimizing tail latencies, and achieving per-flow fairness. We also discussed some limitations of LSTF (with respect to achieving class-based fairness and traffic shaping).

Noting that scheduling is often used along with AQM to prevent queue build up, we then showed how LSTF can be used to implement versions of AQM from the network edge, with performance comparable to FQ-CoDel and to DCTCP with ECN (the state-of-the art AQM schemes for wide-area and datacenters respectively).

While an initial step towards understanding the notion of a Universal Packet Scheduling algorithm, our work leaves several theoretical questions unanswered, three of which we mention here. First, we showed existence of a UPS with omniscient header initialization, and nonexistence with limited-information initialization. *What is the least information we can use for header initialization in order to achieve universality?* Second, we showed that, in practice, the fraction of overdue packets is small, and most are only overdue by a small amount. *Are there tractable bounds on both the number of overdue packets and/or their degree of lateness?* Third, while we have a formal characterization for the scope of LSTF with respect to replaying a given schedule, and we have simulation evidence of LSTF's ability to meet several performance objectives, we do not yet have any formal model for the scope of LSTF in meeting these objectives. *Can one describe the class of performance objectives that LSTF can meet?* Also, are there any new objectives that LSTF allows us to achieve?

Chapter 6

Conclusion and Future Work

In this dissertation, we argued that whenever we are looking for solutions to meet different network performance requirements, we should ask ourselves the following two questions for enabling a more stable network infrastructure: (1) Can we avoid making changes to the network infrastructure (can an end-point based solution meet the desired performance requirements)?, and (2) When infrastructure changes are needed, can we make them *universal* in nature?

We presented examples of how network infrastructure changes can be avoided in the context of congestion control for wide-area and datacenters. We then developed a framework for universality, and analyzed it in the context of packet scheduling. However, we believe that these are just the first steps towards a more stable network infrastructure, and that our line of reasoning is applicable more broadly, beyond just congestion control and packet scheduling.

For example, can we use our framework of universality to end the age-old debate about where to place the intelligence needed for a network functionality: should it go only in the network edge or should it be deployed more widely in the network core? This debate has mostly been informal (though often heated). Our work on UPS has led to a clean theoretical formulation whose fundamental question can be more generally stated as: for which kinds of functionality can intelligence in the edge *effectively simulate* a set of specialized mechanisms deployed in the core? Our results show that we can effectively simulate packet scheduling and AQM policies. But there are many other functionalities where we do not yet have a clear answer, such as network monitoring, load-balancing, specialized routing and support for information-centric networking, among others.

Furthermore, beyond just the network edge vs core argument, we believe our two questions can be used more generally for other scenarios as well. For example, they can potentially be used to determine what network stack functionality at an endhost should get offloaded to hardware. The popularly used RDMA seems to be an extreme solution where all networking functionality is offloaded. While hardware may give us more speed, it often entails compromising on flexibility and the amount of resources (such as memory). The analogous questions in this context could be: (1) Can we avoid offloading a functionality to hardware (can a software solution meet the desired performance requirements)?, and (2) When hardware offload is needed, can we design universal primitives that can handle different requirements of different higher layer applications?

Bibliography

[1] http://omnetpp.org/.

[2] https://inet.omnetpp.org.

[3] http://www.xilinx.com/support/documentation/white_papers/wp350.pdf, 2014.

[4] S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 2010.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. ACM SIGCOMM*, 2008.

[6] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. P. B. Prabhakar, and M. Seaman. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *Annual Allerton Conference*, 2008.

[7] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion aware Load Balancing for Datacenters. In *Proc. ACM SIGCOMM*, 2014.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[9] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proc. USENIX NSDI*, 2012.

[10] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing Datacenter Packet Transport. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[11] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proc. ACM SIGCOMM*, 2013.

[12] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390.

[13] M. Allman. Comments on bufferbloat. *ACM SIGCOMM Computer Communication Review*, 2013.

[14] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. ACM SIGCOMM*, 2004.

[15] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. ACM Internet Measurement Conference (IMC)*, 2012.

[16] R. Bhagwan and B. Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *Proc. IEEE Infocom*, 2000.

[17] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[19] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proc. ACM SIGCOMM*, 2013.

[20] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *ACM SIGCOMM Computer Communication Review*, 1994.

[21] CAIDA Internet Topology Data Kit. http://goo.gl/QAbecc.

[22] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proc. ACM HotSDN*, 2012.

[23] Chelsio T5 Packet Rate Performance Report. http://goo.gl/3jJL6p, Pg 2.

[24] D.-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Netw. ISDN Syst.*, 1989.

[25] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching output queueing with a combined input/output-queued switch. *IEEE Journal on Selected Areas in Communications*, 1999.

[26] B. Claise. Cisco systems NetFlow services export version 9. RFC 3954, 2004.

[27] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Proc. ACM SIGCOMM*, 1992.

[28] Data Center Bridging Task Group. http://www.ieee802.org/1/pages/dcbridges.html.

[29] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[30] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proc. ACM SIGCOMM*, 1989.

[31] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *Proc. IEEE INFOCOM*, 2013.

[32] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proc. USENIX NSDI*, 2014.

[33] Dual Port 10 Gigabit Server Adapter with Precision Time Stamping. http://goo.gl/VtL5oO.

[34] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, 2006.

[35] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An Argument for Increasing TCP's Initial Congestion Window. *ACM SIGCOMM Computer Communication Review*, 2010.

[36] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: the Virtue of Gentle Aggression. In *Proc. SIGCOMM*, 2013.

[37] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883.

[38] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, 2003.

[39] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782.

[40] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582.

[41] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1993.

[42] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-Level Traffic Measurements from the Sprint IP Backbone. *IEEE Network*, 2003.

[43] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *Proc. ACM SIGCOMM*, 2017.

[44] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013.

[45] Global Consortium to Construct New Cable System Linking US and Japan to Meet Increasing Bandwidth Demands. http://googlepress.blogspot.com/2008/02/global-consortium-to-construct-new_26.html.

[46] I. Grigorik. Optimizing the Critical Rendering Path. http://goo.gl/DvFfGo, Velocity Conference 2013.

[47] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proc. ACM SIGCOMM*, 2016.

[48] A. Gupta, M. T. Hajiaghayi, and H. Räcke. Oblivious Network Design. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, 2006.

[49] S. Ha, I. Rhee, and L. Xu. CUBIC: a New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 2008.

[50] D. A. Hayes and G. Armitage. Revisiting TCP Congestion Control using Delay Gradients. In *Networking IFIP*, 2011.

[51] D. A. Hayes and D. Ros. Delay-based Congestion Control for Low Latency. In *ISOC Workshop on Reducing Internet Latency, Sep*, 2013.

[52] High Capacity StrataXGS Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[53] T Hoeiland-Joergensen, P McKenney, D Taht, J Gettys, and E Dumazet. FlowQueue-Codel. *IETF Informational*, 2013.

[54] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. A control theoretic analysis of RED. In *Proc. IEEE Infocom*, 2001.

[55] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. On designing improved controllers for AQM routers supporting TCP flows. In *Proc. IEEE Infocom*, 2001.

[56] M. Honda, Y. Nishida, C. Raiciu, A. Greengalgh, M. Handley, and H. Tokuda. In *Proc. IMC*, 2011.

[57] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. ACM SIGCOMM*, 2012.

[58] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *Proc. ACM HotNets*, 2016.

[59] IEEE. 802.1Qau - Congestion Notification. http://www.ieee802.org/1/pages/802.1au.html.

[60] IEEE. 802.11Qbb. Priority based flow control, 2011.

[61] InfiniBand architecture volume 1, general specifications, release 1.2.1. www.infinibandta.org/specs, 2008.

[62] Internet Traffic Flow Size Analysis. http://net.doit.wisc.edu/data/flow/size/.

[63] Internet2. http://www.internet2.edu/.

[64] A. Ioannou and M. G. H. Katevenis. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-speed Networks. *IEEE/ACM Trans. Netw.*, 2007.

[65] iPerf. http://iperf.sourceforge.net/.

[66] S. Iyer, S. Bhattacharyya, N. Taft, and C. Diot. An Approach to Alleviate Link Overload as Observed on an IP Backbone. In *Proc. IEEE INFOCOM*, 2003.

[67] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. In *DEC Research Report TR-301*, 1984.

[68] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar. AF-QCN: Approximate Fairness with Quantized Congestion Notification for Multi tenanted Data Centers. In *Proc. Hot Interconnects*, 2010.

[69] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In *Proc. ACM CoNEXT*, 2014.

[70] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proc. USENIX ATC*, 2016.

[71] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proc. ACM SIGCOMM*, 2014.

[72] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proc. ACM SIGCOMM*, 2002.

[73] Kathie Nichol's CoDel presented by Van Jacobson. http://www.ietf.org/proceedings/84/slides/slides-84-tsvarea-4.pdf.

[74] F. P. Kelly, G. Raina, and T. Voice. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM Computer Communication Review*, 2008.

[75] A. Kuzmanovic and E. W. Knightly. TCP-LP: Low-priority service via end-Point congestion Control. *In IEEE/ACM ToN*, 2006.

[76] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proc. USENIX ATC*, 2015.

[77] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 1989.

[78] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proc. ACM SIGCOMM*, 2016.

[79] L. Li, D. Alderson, W. Willinger, and J. Doyle. A First-principles Approach to Understanding the Internet's Router-level Topology. In *Proc. ACM SIGCOMM*, 2004.

[80] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 1973.

[81] Y. Lu, G. Chen, Z. Ruan, W. Xiao, B. Li, J. Zhang, Y. Xiong, P. Cheng, and E. Chen. Memory Efficient Loss Recovery for Hardware-based Transport in Datacenter. In *Proc. First Asia-Pacific Workshop on Networking (APNet)*, 2017.

[82] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018.

[83] M. Mathis and J. Mahdavi. Forward acknowledgement: refining tcp congestion control. *ACM SIGCOMM Computer Communication Review*, 1996.

[84] Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[85] Mellanox ConnectX-4 Product Brief. https://goo.gl/HBw9f9, 2016.

[86] Mellanox ConnectX-5 Product Brief. https://goo.gl/ODlqMl, 2016.

[87] Mellanox for Linux. http://goo.gl/u44Xea.

[88] Mellanox Innova Flex 4 Product Brief. http://goo.gl/Lh7VN4, 2016.

[89] Microsoft Invests in Subsea Cables to Connect Datacenters Globally. http://goo.gl/GoXfxH.

[90] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. In *Proc. USENIX NSDI*, 2016.

[91] R. Mittal, V Lam, N Dukkipati, E Blem, H Wassel, M Ghobadi, A Vahdat, Y Wang, D Wetherall, and D Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. ACM SIGCOMM*, 2015.

[92] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively Cautious Congestion Control. In *Proc. USENIX NSDI*, 2014.

[93] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting Network Support for RDMA. In *Proc. ACM SIGCOMM*, 2018.

[94] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proc. USENIX NSDI*, 2008.

[95] K Nichols and V Jacobson. Controlled delay active queue management: draft-nichols-tsvwg-codel-02. *Internet Requests for Comments-Work in Progress, Tech. Rep*, 2014.

[96] K. Nichols and V. Jacobson. Controlling Queue Delay. *Queue*, 2012.

[97] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Fordy. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-compatible with TCP and TLS. In *Proc. USENIX NSDI*, 2012.

[98] NS-2. http://www.isi.edu/nsnam/ns/.

[99] NS-3. http://www.nsnam.org.

[100] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In *Proc. IEEE Global Internet Conference (GLOBECOM)*, 1998.

[101] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-node Case. *IEEE/ACM Trans. Netw.*, 1993.

[102] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: a centralized zero-queue datacenter network. In *Proc. ACM SIGCOMM*, 2014.

[103] P. Phaal, S. Panchen, and N. McKee. InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks. RFC 3176, 2001.

[104] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proc. ACM HotNets*, 2012.

[105] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proc. USENIX NSDI*, 2012.

[106] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP, 2001.

[107] A. Rao, A. Legout, Y. s. Lim, D. Towlsley, C. Barakat, and W. Dabbous. Network Characteristics of Video Streaming Traffic. In *Proc. ACM CoNeXT*, 2011.

[108] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040, 2007.

[109] RoCE vs. iWARP Competitive Analysis. http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf, 2015.

[110] C. Rotsos, H. Howard, D. Sheets, R. Mortier, A. Madhavapeddy, A. Chaudhry, and J. Crowcroft. Lost In the Edge: Finding Your Way With Signposts. In *Proc. USENIX FOCI*, 2013.

[111] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[112] S. Radhakrishnan et al. SENIC: scalable NIC for end-host rate limiting. In *Proc. USENIX NSDI*, 2014.

[113] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 1984.

[114] H. Sariowan, R. L. Cruz, and G. C. Polyzos. Sced: a generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM Trans. Netw.*, 1999.

[115] A. Shpiner, E. Zahavi, V. Zdornov, T. Anker, and M. Kadosh. Unlocking Credit Loop Deadlocks. In *Proc. ACM HotNets*, 2016.

[116] A. Shpiner, E. Zahavi, O. Dahley, A. Barnea, R. Damsker, G. Yekelis, M. Zus, E. Kuta, and D. Baram. RoCE Rocks Without PFC: Detailed Evaluation. In *Proc. ACM Workshop on Kernel-Bypass Networks (KBNets)*, 2017.

[117] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proc. ACM SIGCOMM*, 1995.

[118] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. ACM SIGCOMM*, 2015.

[119] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan. Towards Programmable Packet Scheduling. In *Proc. ACM HotNets*, 2015.

[120] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *Proc. ACM HotNets*, 2013.

[121] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011. ISBN: 1608455645, 9781608455645.

[122] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical DCB for improved data center networks. In *Proc. IEEE Infocom*, 2014.

[123] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High-speed Networks. *IEEE/ACM Trans. Netw.*, 2003.

[124] I. Stoica and H. Zhang. Providing Guaranteed Services Without Per Flow Management. In *Proc. ACM SIGCOMM*, 1999.

[125] Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A16: RDMA over Converged Ethernet (RoCE). www.infinibandta.org/specs, 2010.

[126] Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE), www.infinibandta.org/specs, 2014.

[127] K. Tan and J. Song. A compound TCP approach for high-speed and long distance networks. In *IEEE INFOCOM*, 2006.

[128] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *Proc. IEEE INFOCOM*, 2006.

[129] The Cost of Latency. http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/.

[130] The NetFPGA Project. http://netfpga.org/.

[131] Token Bucket Filters. http://lartc.org/manpages/tc-tbf.html.

[132] TSO Sizing and the FQ Scheduler. http://lwn.net/Articles/564978/.

[133] Using Hardware Timestamps with PF RING. http://goo.gl/oJtHCe, 2011.

[134] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proc. ACM SIGCOMM*, 2012.

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

[136] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp nice: a mechanism for background transfers. In *Proc. USENIX OSDI*, 2002.

[137] Vivado Design Suite User Guide. https://goo.gl/akRdXC, 2013.

[138] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. Netw.*, 2006.

[139] Who (Really) Needs Sub-microsecond Packet Timestamps? http://goo.gl/TI3r1u, 2013.

[140] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proc. ACM SIGCOMM*, 2011.

[141] K. Winstein and H. Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. In *Proc. ACM SIGCOMM*, 2013.

[142] Xilinx Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[143] L Xu, K Harfoush, and I Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *INFOCOM 2004*, 2004.

[144] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: reducing the flow completion time tail in datacenter networks. In *Proc. ACM SIGCOMM*, 2012.

[145] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. In *Proc. ACM SIGCOMM*, 1990.

[146] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proc. ACM SIGCOMM*, 2015.

Appendix A

Proofs for UPS's Theoretical Results

We now provide detailed proofs for the theoretical results presented in Chapter 5. We use the following notations:

Relevant nodes:
$src(p)$: Ingress of a packet p.
$dest(p)$: Egress of a packet p.

Relevant time notations:
$T(p,\alpha)$: Transmission time of a packet p at node α.
$o(p,\alpha)$: Time when the first bit of p is scheduled by node α in the original schedule.
$o(p) = o(p,dest(p)) + T(p,dest(p))$: Time when the last bit of p exits the network in the original schedule (which is non-preemptive).
$o'(p)$: Time when the last bit of p exits the network in the replay (which may be preemptive in our theoretical arguments).
$i(p,\alpha)$ and $i'(p,\alpha)$: Time when p arrives at node α in the original schedule and in the replay respectively.
$i(p) = i(p,src(p)) = i'(p)$: Arrival time of p at its ingress. This remains the same for both the original schedule and the replay.
$t_{min}(p,\alpha,\beta)$: Minimum time p takes to start from node α and exit from node β in an uncongested network. It therefore includes the propagation delays and the store-and-forward delays of all links in the path from α to β and the transmission delays at α and β. Handling the edge case: $t_{min}(p,\alpha,\alpha) = T(p,\alpha)$
$slack(p) = o(p) - i(p) - t_{min}(p,src(p),dest(p))$: Total slack of p that gets assigned at its ingress. It denotes the amount of time p can wait in the network (excluding the time when any of its bits are getting serviced) without missing its target output time.
$slack(p,\alpha,t) = o(p) - t - t_{min}(p,\alpha,dest(p)) + T(p,\alpha)$: Remaining slack of the last bit of p at time t when it is at node α. We derive this expression later in §A.3 of this appendix.

Other miscellaneous notations:
$path(p, \alpha, \beta)$: The ordered set of nodes and links in the path taken by p to go from α to β. The set also includes α and β as the first and the last nodes.
$path(p) = path(p, src(p), dest(p))$
$pass(\alpha)$: Set of packets that pass through node α.

A.1 Existence of a UPS under Omniscient Header Initialization

Algorithm: At the ingress, insert an n-dimensional vector in the packet header, where the i^{th} element contains $o(p, \alpha_i)$, α_i being the i^{th} hop in $path(p)$. Every time a packet p arrives at the switch, the switch pops the value at the head of the vector in p's header and uses that as the priority for p (earlier values of output times get higher priority). This can perfectly replay any schedule.

Proof: We can prove that the above algorithm will result in no overdue packets (which do not meet their original schedule's target) using the following two theorems:
Theorem 1. If for any node α, $\exists p' \in pass(\alpha)$, such that using the above algorithm, the last bit of p' exits α at time $(t' > (o(p', \alpha) + T(p', \alpha)))$, then $(\exists p \in pass(\alpha) \mid i'(p, \alpha) \leq t'$ and $i'(p, \alpha) > o(p, \alpha))$.
Proof by contradiction: Consider the first such $p^* \in pass(\alpha)$ that gets late at α (i.e. its last bit exits α at time $t^* > (o(p^*, \alpha) + T(p^*, \alpha)))$. Suppose the above condition is not true i.e. $(\forall p \in pass(\alpha) \mid i'(p, \alpha) \leq o(p, \alpha)$ or $i'(p, \alpha) > t^*)$. In other words, if p arrives at or before time t^*, it also arrives at or before time $o(p, \alpha)$. Given that all bits of p^* arrive at or before time t^*, they also arrive at or before time $o(p^*, \alpha)$. The only reason why the last bit of p^* would wait until time $(t^* > o(p^*, \alpha) + T(p^*, \alpha))$ in our work-conserving replay is if some other bits (belonging to higher priority packets) were being scheduled after time $o(p^*, \alpha)$, resulting in p^* not being able to complete its transmission by time $(o(p^*, \alpha) + T(p^*, \alpha))$. However, as per our algorithm, any packet p_{high} having a higher priority than p^* at α must have been scheduled before p^* in the original schedule, implying that $(o(p_{high}, \alpha) + T(p_{high}, \alpha)) \leq o(p^*, \alpha)$. [1] Therefore, some bits of p_{high} being scheduled after time $o(p^*, \alpha)$, implies them being scheduled after time $(o(p_{high}, \alpha) + T(p_{high}, \alpha))$. This means that p_{high} is already late and contradicts our assumption that p^* is the first packet to get late. . Hence, Theorem 1 is proved by contradiction.
Theorem 2. $\forall \alpha, (\forall p \in pass(\alpha) \mid i'(p, \alpha) \leq i(p, \alpha))$.
Proof by contradiction: Consider the first time when some packet p^* arrives late at some node α^* (i.e. $i'(p^*, \alpha^*) > i(p^*, \alpha^*)$). In other words, α^* is the first node in the network to see a late packet arrival, and p^* is the first late arriving packet. Let α_{prev} be the node visited by p^* just before arriving at α^*. p^* can arrive at a time later than $i(p^*, \alpha^*)$ at α^* only if the last bit of p^* exits α_{prev} at time $t_{prev} > o(p^*, \alpha_{prev}) + T(p^*, \alpha_{prev})$. As per Theorem 1 above, this is possible only if

[1] Given that the original schedule is non-preemptible, the next packet gets scheduled only after the previous one has completed its transmission.

some packet p' (which may or may not be the same as p^*) arrives at α_{prev} at time $i'(p', \alpha_{prev}) > o(p', \alpha_{prev}) \geq i(p', \alpha_{prev})$ and $i'(p', \alpha_{prev}) \leq t_{prev} < i'(p^*, \alpha^*)$. This contradicts our assumption that α^* is the first node to see a late arriving packet. Therefore, $\forall \alpha, (\forall p \in pass(\alpha) \mid i'(p, \alpha) \leq i(p, \alpha))$.

Combining the two theorems above: Since $\forall \alpha (\forall p \in pass(\alpha) \mid i'(p, \alpha) \leq i(p, \alpha))$, with the above algorithm, $\forall \alpha (\forall p \in pass(\alpha))$, all bits of p exit α before $(o(p, \alpha) + T(p, \alpha))$. Therefore, the algorithm can perfectly replay any viable schedule.

A.2 Nonexistence of a UPS under black-box initialization

Proof by counter-example. Consider the example shown in Figure A.1. For simplicity, assume all the propagation delays are zero, the transmission time for each congestion point (shaded in gray) is 1 unit and the uncongested (white) switches have zero transmission time. [2] All packets are of the same size.

The table illustrates two cases. For each case, a packet's arrival and scheduling time (the time when the packet is scheduled by the switch) at each node through which it passes are listed. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A, B, C, X, Y, Z\}$. The packets have the same *path* in both cases. For example, a belongs to Flow A, starts at ingress S_A, exits at egress D_A and passes through three congestion points in its path α_0, α_1 and α_2; x belongs to Flow X, starts at ingress S_X, exits at egress D_X and passes through three congestion points in its path α_0, α_3 and α_4; and so on.

The two critical packets we care about in this example are a and x, which interact with each-other at their first congestion point α_0, being scheduled by α_0 at different times in the two cases (a before x in Case 1 and x before a in Case 2). But, notice that for both cases,

1. a enters the network from its ingress S_A at congestion point α_0 at time 0, and passes through two other congestion points α_1 and α_2 before exiting the network at time $(4+1)$ [3].
2. x enters the network from its ingress S_X at congestion point α_0 at time 0, and passes through two other congestion points α_3 and α_4 before exiting the network at time $(3+1)$.

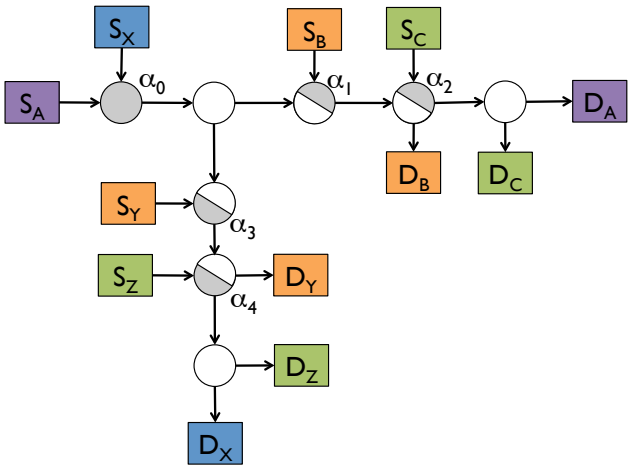
a interacts with packets from Flow C at its third congestion point α_2, while x interacts with a packet from Flow Z at its third congestion point α_4. For both cases,

1. Two packets of Flow C (c_1, c_2) enter the network at times 2 and 3 at α_2 before they exit the network at time $(2+1)$ and $(3+1)$ respectively.
2. z enters the network at time 2 at α_4 before exiting at time $2+1$.

The difference between the two cases comes from how a interacts with packets from Flow B at its second congestion point α_1 and how x interacts with packets from Flow Y at its second congestion points α_3. Note that α_1 and α_3 are the last congestion points for Flow B and Flow Y packets respectively and their exit times from these congestion points directly determine their exit times from the network.

[2] These assignments are made for simplicity of understanding. The example will hold for any reasonable value of propagation and transmission delays.

[3] $+1$ is added to indicate transmission time at the last congestion point. As mentioned before, we assume the propagation delay to the egress and the transmission time at the egress are both 0.



Node	Packet(arrival time, scheduling time)
	Case 1
α_0	$a(\mathbf{0},0)$; $x(\mathbf{0},1)$
α_1	$a(1,1)$, $b_1(2,2)$, $b_2(3,3)$, $b_3(4,4)$
α_2	$c_1(2,2)$, $c_2(3,3)$; $a(2,\mathbf{4})$
α_3	$x(2,2)$, $y_1(2,3)$, $y_2(3,4)$
α_4	$z(2,2)$, $x(3,\mathbf{3})$
	Case 2
α_0	$x(\mathbf{0},0)$; $a(\mathbf{0},1)$
α_1	$a(2,2)$, $b_1(2,3)$, $b_2(3,4)$, $b_3(4,5)$
α_2	$c_1(2,2)$, $c_2(3,3)$, $a(3,\mathbf{4})$
α_3	$x(1,1)$, $y_1(2,2)$, $y_2(3,3)$
α_4	$z(2,2)$, $x(2,\mathbf{3})$

Figure A.1: Example showing non-existence of a UPS with Blackbox Initialization. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C,X,Y,Z\}$. For simplicity assume all packets are of the same size and all links have a propagation delay of zero. All uncongested switches (white), ingresses and egresses have a transmission time of zero. The congestion points (shaded gray) have transmission times of $T = 1$ unit.

1. Three packets of Flow B (b_1, b_2, b_3) enter the network at times 2, 3 and 4 respectively at α_1. In Case 1, they leave α_1 at times $(2+1), (3+1), (4+1)$ respectively. This provides no *lee-way* for a at α_0, which leaves α_1 at time $(1+1)$, since it is required that α_1 must schedule a by at most time 3 in order for it to exit the network at its target output time. In Case 2, (b_1, b_2, b_3) leave at times $(3+1), (4+1), (5+1)$ respectively, providing lee-way for a at α_0, which leaves α_1 at time $(2+1)$.
2. Two packets of Flow Y (y_1, y_2) enter the network at times 2 and 3 respectively at α_3. In Case 1, they leave at times $(3+1), (4+1)$ respectively, providing a lee-way for x at α_0, which leaves α_3 at time $(2+1)$. In Case 2, (y_1, y_2) exit at times $(2+1), (3+1)$, providing no lee-way for x at α_0, which leaves α_3 at time $(1+1)$.

Note that the interaction of a and x with Flow C and Flow Z at their third congestion points respectively, is what ensures that their eventual exit time remains the same across the two cases

inspite of the differences in how a and x are scheduled in their previous two hops.

Thus, we can see that $i(a)$, $o(a)$, $i(x)$, $o(x)$ are the same in both cases (also indicated in bold blue). Yet, *Case 1* requires a to be scheduled before x at α_0, else packets will get delayed at α_1, since it is required that α_1 schedules a at a time no more than 3 units if it is to meet its target output time. *Case 2* requires x to be scheduled before a at α_0, else packets will be delayed at α_3, where it is required to schedule x at a time no more than 2 units if it is to meet its target output time. Since the attributes $(i(\cdot), o(\cdot), path(\cdot))$ for both a and x are exactly the same in both cases, any deterministic UPS with Blackbox Initialization will produce the same order for the two packets at α_0, which contradicts the situation where we want a before x in one case and x before a in another.

A.3 Deriving the Slack Equation

We now prove that for any packet p waiting at any node α at time t_{now}, the remaining slack of the last bit of p is given by $slack(p, \alpha, t_{now}) = o(p) - t_{now} - t_{min}(p, \alpha, dest(p)) + T(p, \alpha)$.

Let $t_{wait}(p, \alpha, t_{now})$ denote the total time spent by p on waiting behind other packets at the nodes in its path from $src(p)$ to α (including these two nodes) until time t_{now}. We define $t_{wait}(p, \alpha, t_{now})$, such that it excludes the transmission times at previous nodes which gets captured in t_{min}, but includes the local service time received by the packet so far at α itself.

$$
\begin{aligned}
slack(p, \alpha, t_{now}) =& slack(p) - t_{wait}(p, \alpha, t_{now}) + T(p, \alpha) && \text{(A.1a)}\\
=& o(p) - i(p) - t_{min}(p, src(p), dest(p)) \\
& - t_{wait}(p, \alpha, t_{now}) + T(p, \alpha) && \text{(A.1b)}\\
=& o(p) - i(p) - (t_{min}(p, src(p), \alpha) \\
& + t_{min}(p, \alpha, dest(p)) - T(p, \alpha)) \\
& - t_{wait}(p, \alpha, t_{now}) + T(p, \alpha) && \text{(A.1c)}\\
=& o(p) - t_{min}(p, \alpha, dest(p)) + T(p, \alpha) \\
& - (i(p) + t_{min}(p, src(p), \alpha) \\
& - T(p, \alpha) + t_{wait}(p, \alpha, t_{now})) && \text{(A.1d)}\\
=& o(p) - t_{min}(p, \alpha, dest(p)) + T(p, \alpha) - t_{now} && \text{(A.1e)}
\end{aligned}
$$

(A.1a) is straightforward from our definition of LSTF and how the slack gets updated at every time slice. $T(p, \alpha)$ is added since α needs to locally consider the slack of the last bit of the packet in a store-and-forward network. (A.1c) then uses the fact that for any α in $path(p)$, $(t_{min}(p, src(p), dest(p)) = t_{min}(p, src(p), \alpha) + t_{min}(p, \alpha, dest(p)) - T(p, \alpha))$. $T(p, \alpha)$ is subtracted here as it is accounted for twice when we break up the equation for $t_{min}(p, src(p), dest(p))$. (A.1e) then follows from the fact that the difference between t_{now} and $i(p)$ is equal to the total amount of time the packet has spent in the network until time t_{now} i.e. $(t_{now} - i(p) = (t_{min}(p, src(p), \alpha) - T(p, \alpha)) + t_{wait}(p, \alpha, t_{now}))$. We need to subtract $T(p, \alpha)$, since by our definition, $t_{min}(p, src(p), \alpha)$ includes transmission time of the packet at α.

A.4 LSTF and EDF Equivalence

In our network-wide extension of EDF scheduling, every switch computes a deadline (or priority) for a packet p based on the static header value $o(p)$ and additional state information about the minimum time the packet would take to reach its destination from the switch. More precisely, each switch (say α), uses $priority(p) = (o(p) - t_{min}(p, \alpha, dest(p)) + T(p, \alpha))$ to do priority scheduling, with $o(p)$ being the value carried by the packet header, initialized at the ingress and remaining unchanged throughout. EDF is equivalent to LSTF, in that for a given original schedule, the two produce exactly the same replay schedule.

Proof. Consider a node α and let $P(\alpha, t_{now})$ be the set of packets waiting at the output queue of α at time t_{now}. A packet will then be scheduled by α as follows:

With EDF: Schedule packet $p_{edf}(\alpha, t_{now})$, where

$$p_{edf}(\alpha, t_{now}) = \underset{p \in P(\alpha, t_{now})}{\text{argmin}} \ (priority(p, \alpha))$$
$$priority(p, \alpha) = o(p) - t_{min}(p, \alpha, dest(p)) + T(p, \alpha)$$

With LSTF: Schedule packet $p_{lstf}(\alpha, t_{now})$, where

$$p_{lstf}(\alpha, t_{now}) = \underset{p \in P(\alpha, t_{now})}{\text{argmin}} \ (slack(p, \alpha, t_{now}))$$
$$slack(p, \alpha, t_{now}) = o(p) - t_{min}(p, \alpha, dest(p)) + T(p, \alpha) - t_{now}$$

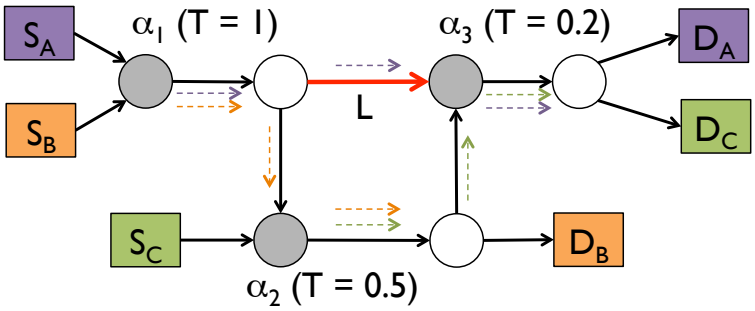
The above expression for $slack(p, \alpha, t_{now})$ has been derived before (§A.3). Thus, $slack(p, \alpha, t_{now}) = priority(p, \alpha) - t_{now}$. Since t_{now} is the same for all packets, we can conclude that:

$$\underset{p \in P(\alpha, t_{now})}{\text{argmin}} \ (slack(p, \alpha, t_{now})) = \underset{p \in P(\alpha, t_{now})}{\text{argmin}} \ (priority(p, \alpha))$$
$$\implies p_{lstf}(\alpha, t_{now}) = p_{edf}(\alpha, t_{now})$$

Therefore, at any given point of time, all nodes will schedule the same packet with both EDF and LSTF (assuming ties are broken in the same way for both EDF and LSTF, such as by using FCFS). Hence, EDF and LSTF are equivalent.

A.5 Theoretical Limits for Replay using Simple Priorities

In Figure A.2, we present an example which shows that simple priorities can fail in replay when there are two congestion points per packet, no matter what information is used to assign priorities. At α_1, we need to have $priority(a) < priority(b)$, at α_2 we need to have $priority(b) < priority(c)$



Node	Packet(arrival time, scheduling time)
α_1	$a(0,0), b(0,1)$
α_2	$b(2,2), c(2,2.5)$
α_3	$c(3,3), a(3,3.2)$

Figure A.2: Example showing replay failure with simple priorities for a schedule with two congestion points per packet. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A, B, C\}$. All packets are of the same size. For simplicity assume all links (except L) have a propagation delay of zero. L has a propagation delay of 2. All uncongested switches (white circles), ingresses and egresses have a transmission time of zero. The three congestion points – $\alpha_1, \alpha_2, \alpha_3$ have transmission times of $T = 1$ unit, $T = 0.5$ units and $T = 0.2$ units respectively.

and at α_3 we need to have $priority(c) < priority(a)$. This creates a priority cycle where we need $priority(a) < priority(b) < priority(c) < priority(a)$, which can never be possible to achieve with simple priorities.

We would also like to point out here that priority assignment for perfect replay in networks with single congestion point per packet requires detailed knowledge about the topology and the input load. More precisely, if a packet p passes through congestion point α_p, then its priority needs to be assigned as $priority(p) = o(p) - t_{min}(p, \alpha_p, dest(p)) + T(p, \alpha_p)$. The proof that this would always replay schedules with at most one congestion point per packet follows from the fact that the only scheduling decision made in a packet p's path is at the single congestion point α_p. This decision, at the single congestion point in a packet's path, is the same as what will be made with the network-wide extension of EDF, which we proved is equivalent to LSTF in §A.4. LSTF, in turn, can always replay schedules with one (or to be more precise, at most two) congestion points per packet, as we shall prove in §A.6.

Hence, in order to replay schedules with at most one congestion per packet using simple priorities, we need to know where the congestion point occurs in a packet's path, along with the final output times, to assign the priorities. In the absence of this knowledge, priorities cannot replay even a single congestion point.

A.6 Theoretical Limits for Replay using LSTF

We now prove that LSTF can replay all schedules with at most two congestion points per packet. We first present the high-level proof in §A.6.1 that uses a key condition, the proof for which is then presented in §A.6.2.

Note that we work with *bits* in our proof, since we assume a preemptive version of LSTF. Due to store-and-forward switches, the remaining slack of a packet at a particular switch is represented by the slack of the last bit of the packet (with all other bits of the packet having the same slack as the last bit).

A.6.1 LSTF can Replay up to Two Congestion Points per Packet

In order for a replay failure to occur, there must be at least one overdue packet, where a packet p is said to be overdue if $o'(p) > o(p)$. This implies that p must have spent all of its slack while waiting behind other packets at a queue in some node α at say time t, such that $slack(p,\alpha,t) < 0$. Obviously, α must be a congestion point.

Necessary Condition for Replay Failure with LSTF. If a packet p^* sees negative slack at a congestion point α when its last bit exits α at time t^* in the replay (i.e. $slack(p^*,\alpha,t^*) < 0$), then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \leq t^*$ and $i'(p,\alpha) > o(p,\alpha))$. We prove this later in §A.6.2.

We use the term "*local deadline* of p at α" for $o(p,\alpha)$, which is the time at which α schedules p in the original schedule.

Key Observation: When there are at most two congestion points per packet, then no packet p can arrive at any congestion point α in the replay, after its local deadline at α (.i.e. $i'(p,\alpha) > o(p,\alpha)$ is not possible). Therefore, by the necessary condition above, no packet can see a negative slack at any congestion point.

Proof by contradiction: Suppose that there exists α^*, which is the first congestion point (in time) that sees a packet which arrives after its local deadline at α^*. Let p^* be this first packet that arrives after its local deadline at α^* ($i'(p^*,\alpha^*) > o(p^*,\alpha^*)$). Since there are at most two congestion points per packet, either α^* is the first congestion point seen by p^* or the last (or both).
(1) If α^* is the first congestion point seen by p^*, then clearly, $i'(p^*,\alpha^*) = i(p^*,\alpha^*) \leq o(p^*,\alpha^*)$. This contradicts our assumption that $i'(p^*,\alpha^*) > o(p^*,\alpha^*)$.
(2) If α^* is not the first congestion point seen by p^*, then it is the last congestion point seen by p^*. If $i'(p^*,\alpha^*) > o(p^*,\alpha^*)$, then it would imply that p^* saw a negative slack before arriving at α^*. Suppose p^* saw a negative slack at a congestion point α_{prev}, before arriving at α^* when its last bit exited α_{prev} at time t_{prev}. Clearly, $t_{prev} < i'(p^*,\alpha^*)$. As per our necessary condition, this would imply that there must be another packet p', such that $i'(p',\alpha_{prev}) > o(p',\alpha_{prev})$ and $i'(p',\alpha_{prev}) \leq t_{prev} < i'(p^*,\alpha^*)$. This contradicts our assumption that α^* is the first congestion point (in time) that sees a packet which arrives after its corresponding scheduling time in the original schedule.

Hence, no congestion point can see a packet that arrives after its local deadline at that congestion point (and therefore no packet can get overdue) when there are at most two congestion points per packet.

A.6.2 Proof for Necessary Condition for Replay Failure with LSTF

We start this proof with the following observation:

Observation 1: If all bits of a packet p exit a switch α by time $o(p,\alpha)+T(p,\alpha)$, then p cannot see a negative slack at α.

Proof for Observation 1: As shown previously in §A.3,

$$slack(p,\alpha,t) = o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - t$$

Therefore,

$$\begin{aligned}
&slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) \\
&= o(p) - t_{min}(p,\alpha,dest(p)) + T(p,\alpha) - (o(p,\alpha)+T(p,\alpha)) \\
&\text{But, } o(p) = o(p,\alpha) + t_{min}(p,\alpha,dest(p)) + wait(p,\alpha,dest(p)) \\
&\implies slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) = wait(p,\alpha,dest(p)) \\
&\implies slack(p,\alpha,o(p,\alpha)+T(p,\alpha)) \geq 0
\end{aligned}$$

where $wait(p,\alpha,dest(p))$ is the time spent by p in waiting behind other packets in the original schedule, after it left α, which is clearly non-negative.

We now move to the main proof for the necessary condition.

Necessary Condition for Replay Failure: If a packet p^* sees negative slack at a congestion point α when its last bit exits α at time t^* in the replay (i.e. $slack(p^*,\alpha,t^*) < 0$), then $(\exists p \in pass(\alpha) \mid i'(p,\alpha) \leq t^*$ and $i'(p,\alpha) > o(p,\alpha))$.

Proof by Contradiction: Suppose this is not the case .i.e. there exists p^* whose last bit exits α at time t^*, such that $slack(p^*,\alpha,t^*) < 0$ and $(\forall p \in pass(\alpha) \mid i'(p,\alpha) > t^*$ or $i'(p,\alpha) \leq o(p,\alpha))$. We can show that if the latter condition holds, then p^* cannot see a negative slack at α, thus violating our assumption.

We take the set of all bits which exit α at or before time t^* in the LSTF replay schedule. We denote this set as $S_{bits}(\alpha,t^*)$. As per our assumption, $(\forall b \in S_{bits}(\alpha,t^*) \mid i'(p_b,\alpha) \leq o(p_b,\alpha))$, where p_b denotes the packet to which bit b belongs. Note that $S_{bits}(\alpha,t^*)$ also includes all bits of p^*, since they all arrive before time t^*.

We now prove that no bit in $S_{bits}(\alpha,t^*)$ can see a negative slack (and therefore p^* cannot see a negative slack at α), leading to a contradiction. The proof comprises of two steps:

Step 1: Using the same input arrival times of each packet at α as in the replay schedule, we first construct a *feasible schedule* at α up until time t^*, denoted by $FS(\alpha,t^*)$, where by feasibility we mean that no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack.
Step 2: We then do an iterative transformation of $FS(\alpha,t^*)$ such that the bits in $S_{bits}(\alpha,t^*)$ are scheduled in the order of their *least remaining slack times*. This reproduces the LSTF replay schedule from which $FS(\alpha,t^*)$ was constructed in the first place. However, while doing the transformation we show how the schedule remains feasible at every iteration, proving that the LSTF schedule finally obtained is also feasible up until time t^*. In other words, no packet sees a negative slack at α in the resulting LSTF replay schedule up until time t^*, contradicting our assumption that p^* sees a negative slack when it exits α at time t^* in the replay.

We now discuss these two steps in details.

Step 1:. Construct a feasible schedule at α up until time t^* (denoted as $FS(\alpha,t^*)$) for which no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack.
(i) Algorithm for constructing $FS(\alpha,t^*)$: Use priorities to schedule each bit in $S_{bits}(\alpha,t^*)$, where $\forall b \in S_{bits}(\alpha,t^*) \mid priority(b) = o(p_b,\alpha)$. (Note that since both $FS(\alpha,t^*)$ and LSTF are work-conserving, $FS(\alpha,t^*)$ is just a shuffle of the LSTF schedule up until t^*. The set of time slices at which a bit is scheduled in $FS(\alpha,t^*)$ and in the LSTF schedule up until t^* remains the same, but *which* bit gets scheduled at a given time slice is different.)
(ii) In $FS(\alpha,t^*)$, all bits b in $S_{bits}(\alpha,t^*)$ exit α by time $o(p_b,\alpha)+T(p_b,\alpha)$.
Proof by contradiction: Suppose the statement is not true and consider the first bit b^* that exits after time $(o(p_{b^*},\alpha)+T(p_{b^*},\alpha))$. We term this as b^* got late at α due to $FS(\alpha,t^*)$. Remember that, as per our assumption, $(\forall b \in S_{bits}(\alpha,t^*) \mid i'(p_b,\alpha) \leq o(p_b,\alpha))$. Thus, given that all bits of p_{b^*} arrive at or before time $o(p_{b^*},\alpha)$, the only reason why the delay can happen in our work-conserving $FS(\alpha,t^*)$ is if some other higher priority bits were being scheduled after time $o(p_{b^*},\alpha)$, resulting in p_{b^*} not being able to complete its transmission by time $(o(p_{b^*},\alpha)+T(p_{b^*},\alpha))$. However, as per our priority assignment algorithm, any bit b' having a higher priority than b^* at α must have been scheduled before the first bit of p_{b^*} in the non-preemptible original schedule, implying that $(o(p_{b'},\alpha)+T(p_{b'},\alpha)) \leq o(p_{b^*},\alpha)$. Therefore, a bit b' being scheduled after time $o(p_{b^*},\alpha)$, implies it being scheduled after time $(o(p_{b'},\alpha)+T(p_{b'},\alpha))$. This contradicts our assumption that b^* is the first bit to get late at α due to $FS(\alpha,t^*)$. Therefore, all bits b in $S_{bits}(\alpha,t^*)$ exit α by time $o(p_b,\alpha)+T(p_b,\alpha)$ as per the schedule $FS(\alpha,t^*)$.
(iii) Since all bits in $S_{bits}(\alpha,t^*)$ exit by time $o(p_b,\alpha)+T(p_b,\alpha)$ due to $FS(\alpha,t^*)$, no bit in $S_{bits}(\alpha,t^*)$ sees a negative slack at α (from Observation 1).

Step 2:. *Transform* $FS(\alpha,t^*)$ into a feasible LSTF schedule for the single switch α up until time t^*.

(Note: The following proof is inspired from the standard LSTF optimality proof that shows that for a single switch, any feasible schedule can be transformed to an LSTF (or EDF) schedule [80]).

Let $fs(b,\alpha,t^*)$ be the scheduling time slice for bit b in $FS(\alpha,t^*)$. The transformation to LSTF is carried out by the following pseudo-code:

1: **while** true **do**
2: Find two bits, b_1 and b_2, such that:
$(fs(b_1,\alpha,t^*) < fs(b_2,\alpha,t^*))$ **and**
$(slack(b_2,\alpha,fs(b_1,\alpha,t^*))$
$< slack(b_1,\alpha,fs(b_1,\alpha,t^*)))$ **and**
$(i'(b_2,\alpha,t^*) \leq fs(b_1,\alpha,t^*))$
3: **if** no such b_1 and b_2 exist **then**
4: $FS(\alpha,t^*)$ is an LSTF schedule
5: **break**
6: **else**
7: $swap(fs(b_1,\alpha,t^*),fs(b_2,\alpha,t^*))$ ▷ swap the scheduling times of the two bits. [4]
8: **end if**
9: **end while**
10: Shuffle the scheduling time of the bits belonging to the same packet, to ensure that they are in order.
11: Shuffle the scheduling time of the same-slack bits such that they are in FIFO order

Line 7 above will not cause b_1 to have a negative slack, when it gets scheduled at $fs(b_2,\alpha,t^*)$ instead of $fs(b_1,\alpha,t^*)$. This is because the difference in $slack(b_2,\alpha,t)$ and $slack(b_1,\alpha,t)$ is independent of t and so:

$$slack(b_2,\alpha,fs(b_1,\alpha,t^*)) < slack(b_1,\alpha,fs(b_1,\alpha,t^*))$$
$$\implies slack(b_2,\alpha,fs(b_2,\alpha,t^*)) < slack(b_1,\alpha,fs(b_2,\alpha,t^*))$$

Since $FS(\alpha,t^*)$ is feasible before the swap, $slack(b_2,\alpha,fs(b_2,\alpha,t^*)) \geq 0$. Therefore, $slack(b_1,\alpha,fs(b_2,\alpha,t^*)) > 0$ and the resulting $FS(\alpha,t^*)$ after the swap remains feasible.

Lines 10 and 11 will also not result in any bit getting a negative slack, because all bits participating in the shuffle have the same slack at any fixed point of time in α.

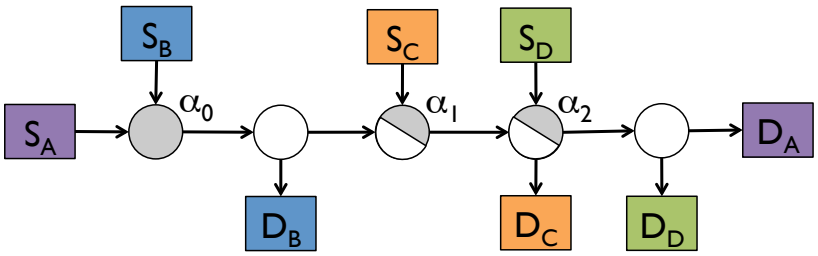
Therefore, no bit in $S_{bits}(\alpha,t^*)$ has a negative slack at α after any iteration.

Since no bit in $S_{bits}(\alpha,t^*)$ has a negative slack at α in the swapped LSTF schedule, it contradicts our statement that p^* sees a negative slack when its last bit exits α at time t^*. Hence proved that if a packet p^* sees a negative slack at congestion point α when its last bit exits α at time t^* in the replay, then there must be at least one packet that arrives at α in the replay at or before time t^* and later than the time at which it is scheduled by α in the original schedule.

A.6.3 LSTF Replay Failure Example

In Figure A.3, we present an example where a flow passes through three congestion points and a replay failure occurs with LSTF. When packet a arrives at α_0, it has a slack of 2 (since it waits behind d_1 and d_2 at α_2), while at the same time, packet b has a slack of 1 (since it waits behind a at α_0). As a result, b gets scheduled before a in the LSTF replay. a therefore arrives at α_1 with slack

[4]Note that we are working with bits here for easy expressibility. In practice, such a swap is possible under the preemptive LSTF model.



Original Schedule	
Node	*Packet(arrival time, scheduling time)*
α_0	$a(0,0), b(0,1)$
α_1	$a(1,1), c_1(2,2), c_2(3,3)$
α_2	$d_1(2,2), d_2(3,3), a(2,4)$
LSTF Replay	
Node	*Packet(arrival time, scheduling time)*
α_0	$b(0,0), a(0,1)$
α_1	$c_1(2,2), a(2,3), c_2(3,4)$
α_2	$d_1(2,2), d_2(3,3), a(4,4)$

Figure A.3: Example showing replay failure with LSTF when there is a flow with three congestion points. A packet represented by p belongs to flow P, with ingress S_P and egress D_P, where $P \in \{A,B,C,D\}$. For simplicity assume all links have a propagation delay of zero. All uncongested switches (white), ingresses and egresses have a transmission time of zero. The three congestion points (shaded gray) have transmission times of $T = 1$ unit.

1 at time 2. c_1 with a zero slack is prioritized over a. This reduces a's slack to zero at time 3, when c_2 is also present at α_1 with zero slack. Scheduling a before c_2, will result in c_2 being overdue (as shown). Likewise, scheduling c_2 before a would have resulted in a getting overdue. Note that in this failure case, a arrives at α_1 at time 2, which is greater than $o(a,\alpha_1) = 1$.

Appendix B

Experience using Different Network Simulators

I have extensively used different network simulators and have often been asked about their relative pros and cons. I decided to list them here, in case anyone else finds them useful. Note that this pros and cons list is based entirely on my experience using them for evaluating congestion control, queue management and scheduling algorithms, and might not reflect how they compare with one-another for other types of use-cases.

NS-2.
Pros:
1. The primary advantage of ns-2 is that since it has been around for a while, a large number of congestion control, queue management and scheduling algorithms have been implemented in ns-2. This makes it simpler to perform comparative analysis.
2. More extensive usage also implies that ns-2 is relatively bug-free and quite stable.
3. Extending ns-2 for specific use cases (more congestion control, scheduling, and queuing mechanisms) is fairly straight forward.

Cons:
1. The ns-2 documentation is not very good.
2. It requires knowing some Tcl for specifying the simulation set-up.

A tip: Logging results via the Tcl interface, which seems to be the recommended way, can increase the simulation completion time. The simulation will complete significantly faster if logs are captured by directly changing the C++ code-base.

NS-3.
Pros:
1. It has very good documentation.
2. Knowledge of $C++$ is enough to use ns-3. It also supports Python bindings, but I never used them.

Cons:

1. The ns-3 code-base is overly structured, making it somewhat cumbersome to add new extensions (such as a new transport protocol, or even a new packet header field).
2. Since ns-3 is relatively new, there are fewer transport schemes implemented in ns-3 (especially when compared to ns-2). So when extensive comparison with other algorithms is needed, it would require re-implementing them in ns-3, which can be non-trivial.

Common myth: ns-3 is not just another newer version of ns-2. It is completely different and has no backwards compatibility with ns-2.

OMNeT++/INeT.

Pros:

1. As opposed to ns-2 and ns-3, OMNeT++ provides a more primitive message-driven framework, which can be extended to experiment with radical communication schemes that do not comply with the structure of other network simulators.
2. OMNeT++/INeT ran significantly faster than NS-2 for some of the experiment scenarios I tried on both.

Cons:

1. It took me a while to figure out my way around the simulator. But this might just be because I was habituated to other simulators when I started with OMNeT++/INeT.
2. OMNeT++/INeT uses its own domain-specific language called NED for specifying the simulation set up. It seemingly does not support file operations, and so the topology and workload files cannot be simply passed as inputs. I ended up writing a series scripts to translate a text file containing an arbitrary topology and workload into the corresponding NED file. [1]
3. Like NS-3, OMNeT++/INeT also suffers from the disadvantage of having fewer schemes implemented in it, which makes comparative analysis more time consuming.

Overall. I believe that once enough time has been spent with any simulator, one becomes very efficient at using it and the above differences matter less. So it might just be best to start with a simulator that has the least *ramp up* time. For example, if the project requires some special features that are implemented in a particular simulator, it might make sense to use that. If it requires comparisons with particular schemes, it might be better to use a simulator in which those schemes are implemented, and so on.

[1] These scripts can be made available upon request.