UNIVERSITY OF CALIFORNIA
SANTA CRUZ

NETWORK CONGESTION AND PERFORMANCE MANAGEMENT

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

Doctor of Philosophy

in

COMPUTER SCIENCE

by

Andrew G. Shewmaker

September 2016

The Dissertation of Andrew G. Shewmaker
is approved:

Professor Carlos Maltzahn, Chair

Professor Scott Brandt

Professor Katia Obraczka

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Network Congestion and Performance Management

by

Andrew G. Shewmaker

We increasingly depend on well-behaved networks in the course of every-day activities for business, community, government, science, and recreation. And with more people demanding a greater variety of services comes sharp disagreement about which needs are most important. Unfortunately, today's technology is inadequate to guarantee the performance of dynamic and diverse workloads in such a way that we are prevented from hurting each other's goals.

No one likes waiting in traffic, whether on a road or on a computer network. Stuttering progress and slow interactive feedback annoy everyone and cost time and money. A network should aim to (1) minimize latency, (2) maximize bandwidth, (3) share resources according to agreements, (4) enable incremental deployment, and (5) minimize administrative overhead. Many technologies have been developed, but none yet satisfactorily address all five of these goals. The best performing solutions to date achieve some success in goals 1-3, but they often downplay the importance of goals 4-5. Solutions almost always require coordinated configurations across many pieces of a network, but they end up being either impractical or suffer poor performance when even a single piece of a network is not cooperating.

In this dissertation I present an overview of the multilayered problem surrounding network performance, a transport-level solution called TCP Inigo, and foundations for a clean-slate queueing discipline solution based on a new combination of recent real-time and economic scheduling algorithms. TCP Inigo uses independent delay-based algorithms on the sender and receiver. In emulated experiments with single administrative domains, a situation common in data centers or a single HPC cluster, TCP Inigo's fairness, bandwidth, and latency indices are up to 1.3× better than the best available solution. When deployed in a more complex environment, such as across administrative domains, TCP Inigo performs up to 42× better.

“Those who do not understand TCP are destined to reimplement it.”

- Jon Postel

“It’s not a big truck. It’s a series of tubes.”

- from Senator Ted Stevens’ famous explanation of congestion on the Internet

Acknowledgments

The text of this dissertation includes material that was previously published in technical reports and at the 25th International Conference on Computer Communication and Networks [142, 143, 144, 145]. Carlos Maltzahn and Scott Brandt directed and supervised the research which forms the basis for the dissertation, and Katia Obraczka provided much valuable advice in the area of networking. Also, John Bent's suggestions improved the discussion of the historical context and the clarity of many figures.

Many thanks to my family, especially to Terra and my children for sacrificing time with me and for encouraging me.

Thanks to Professors Carlos Maltzahn, Katia Obraczka, and Scott Brandt for their advice and collaboration.

Thanks to Nathan DeBardeleben and Sean Blanchard for pushing me to finish. Thanks to John Bent for discussions that led to the Predictable Latency Index. Thanks to the members of the UCSC Systems Research Laboratory, Ike Nassi, and my co-workers at LANL. Special thanks to Michael Sevilla and Monique Morin for their feedback and questions. Thanks to Gary Grider, Jeff Kuehn, Cory Lueninghoener, and other leaders at LANL for their support.

And thanks to Los Alamos National Laboratory and to Google for partially funding this work with a Research Award.

Part I

Introduction

The end goal of my research is to produce formal theories and real-world implementations of flexible, general, and fine-grained performance guarantees on standard commodity network hardware. Network performance guarantees should be compatible with performance guarantees made for CPU, disk, and memory resources such that end-to-end Quality of Service (QoS) can be provided, from a client to a remote resource.

The Resource Allocation and Dispatching (RAD) model, described in § 2.3, provides an underlying framework that makes end-to-end guarantees possible, and has been previously been used successfully in the other contexts [19, 89, 90, 91, 92, 122, 124, 125] Systems built using RAD are able to satisfy performance guarantees without over-provisioning, and can perfectly virtualize the performance of resources. This dissertation will apply RAD to network resources and show how RAD enables automatic identification of unreliable components § 5.2.

End-to-end performance guarantees are a grand challenge in Operating Systems research, if there ever was one. And, before that goal can be reached a fundamental problem which has plagued computer networks for decades must be solved …*congestion*. Consider that the Internet, as well as local area networks, have been the subject of intense study by brilliant researchers for over 30 years and we are still discovering how to better detect packet loss [166, 165]. Advanced networks with credit-based flow control may block flows due to full queues in certain scenarios [55, 56, 138], so even if congestion does not cause dropped packets it can still cause huge variations in delay [139, 140]. And if we lack enough basic control of our networks to manage their queues without overflowing or blocking, then how can we expect to satisfy more difficult performance guarantees?

The contributions of this dissertation include the following innovations in congestion control. First, a sender-side algorithm inspired by Data Center TCP (DCTCP) but using Round-trip-times (RTTs) to mimic Explicit Congestion Notification (ECN). Second, a receiver-side algorithm that similarly mimics ECN using differences in One Way Delay (OWD). I call both of these algorithms by the name TCP Inigo, after the famous swordsman that could fight with either hand from *The Princess Bride* [54]. TCP Inigo's ambidextrous, delay-based algorithm's complement DCTCP's ECN-based congestion control, with the result that TCP Inigo is able to minimize congestion even when not every sender, middle-box, and receiver can be modified.

This dissertation also presents enhancements to RAD scheduling algorithms that can provide a solid foundation for clean slate queue scheduling designs. The RAD model is extended to include economic benefit, which affects the order in which reservations are admitted into a system. I describe new policies for performance and power efficiency for use with the Reduction to UNiprocessor (RUN) algorithm [131]. But more importantly, when RUN is combined with Rate Based Earliest Deadline (RBED) scheduling [19], it can then support efficient use of dynamic slack on multiple resources with sporadic workloads. Previous versions of RUN are not work-conserving, and are inappropriate for common use cases.

In addition to the two new congestion control algorithms described above, the implementation of TCP Inigo uncovered subtle implementation issues in existing DCTCP implementations. When optimized scaled arithmetic is used to calculate DCTCP's congestion ratio, it is easy to accidentally preclude the ratio from ever reaching zero (i.e. no congestion). My bug fix was accepted in to Linux and I helped write the description of the issue in section 4 of the DCTCP Internet-Draft standard [13]. Another issue I discovered is that DCTCP does not specify whether or not Slow Start should be exited upon reception of the first ECN echo. This causes an overshoot of the ideal congestion window and cause unnecessarily high RTTs. I have brought the issue up to the IETF working group, and will continue to work with them to address it.

The rest of the dissertation is organized as follows: chapter 1 gives context to this research. It shows how network performance fits into a larger picture of end-to-end performance management. It also describes congestion and why it is a difficult problem to solve.

Next, chapter 2 reviews many established and proposed techniques that improve performance in controlled environments, such as data centers, where all components (i.e. end-hosts and middle-boxes) can be modified. However, the effectiveness of those same techniques significantly degrades in uncontrolled environments when interacting with unmodified components across network borders. Established systems with multiple owners and long histories, like most networks, favor incremental evolution over dramatic change because upgrades are costly and decisions to upgrade are made independently. Even data centers and supercomputers, which are prime examples of scenarios where new network technologies can be leveraged, must communicate frequently with external systems, and end-to-end arguments [14, 136] should be considered.

Part I of the dissertation ends with chapter 3, which describes the strengths and weaknesses of different strategies for demonstrating the effectiveness of network research.

Part II begins with a presentation of TCP Inigo in chapter 4, and continues with real-time and economic-based scheduling algorithmic foundations for a clean slate queueing discipline in chapter 5. Future work is outlined in chapter 6, and the dissertation concludes with chapter 7.

Chapter 1

Background

Network performance cannot be studied in isolation. From Figure 1.1 it should be obvious there needs to be explicit relationships between the network and other resources. Each hop in the diagram— from permanent storage to server cache, from server cache to network, throughout the network, and so on and so forth to the user application—needs to match speeds and feeds. And this dissertation is intended to be read in light of its place in a larger end-to-end Quality of Service (QoS) project. Traditionally, network QoS involves coarse-grained priority classes, but UCSC's project includes the ability to provide performance guarantees to individual flows.

1.1 Congestion

The most famous study of congestion occurred before the World Wide Web, when Van Jacobson and others saved the Internet from collapse [70]. It is a wonderful example of doing the simplest thing that might work. It created a stable foundation for the Internet, and it

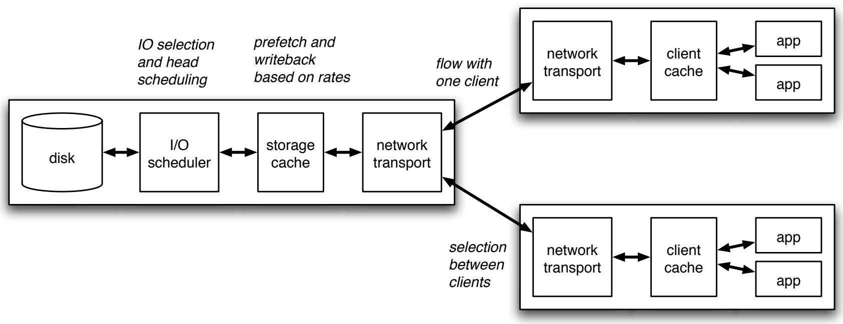

Figure 1.1: Radon is a component of a larger Storage QoS project

turned the Transmission Control Protocol (TCP) into one of the most successful protocols ever invented. But for all TCP's success, it leaves much to be desired.

Most modern networks operate on the principle of statistical multiplexing, which is a scientific way of saying, "Let's all just go for it when we're ready because we probably won't hit each other." Smarter network technologies exist, but there are economic advantages to simple hardware. A real life example of statistical multiplexing occurs regularly on college campuses when a bicyclist flies through a crosswalk at breakneck speed. In many cases, it does not appear possible that the cyclist could either properly judge the speed of traffic or avoid a collision if a vehicle moved into their path. It seems clear that the cyclist is betting with their life that the crosswalk is usually empty. The story of an Ethernet packet is similar when it is sent through a switch with overflowing queues.

Now consider a second analogy of how a series of TCP packets, a *flow*, behave. A traditional TCP flow is like a delivery train with a blind and deaf engineer. The engineer's only method of determining if they should slow down or stop is by feeling collisions. The train has a long way to go and a deadline to meet, so the engineer keeps increasing speed as a general rule.

When the train shudders from hitting something, the engineer slows to half the current speed.

Now, we are fortunate that our networks use buffers to temporarily store packets. Out of a pair of contending packets, one passes through immediately, and the other usually suffers delay, not destruction. Therefore buffering saves us much grief, but it has a cost and it has limits. We call it *congestion*.

Congested networks remain a perennial concern in data centers and the Internet. For businesses, the long tail of variations in delay can cost money [38], and congestion made worse by bufferbloat [53] creates pain for every-day users. Network congestion can block flows from passing through a busy port on a switch, and it can cause performance collapse in a worst case scenario such as servers simultaneously sending data to the same client (i.e. *incast*) in a storage network. Additionally, the network will come under more pressure as the number of users and the speed of storage increase.

High Performance Computing (HPC) systems often run tightly coupled simulations that are highly sensitive to delay variability since global progress is only as fast as the slowest task. Some practitioners believe that moving to a fine-grained asynchronous programming model will solve their problems. However, the ability for individual tasks to continue alone with work is limited, and congestion can degrade performance regardless of programming model. This effect stems from pushing inadequate scheduling technology to loads higher than it is designed. The X-Stack Program, which researches ways to make scientific computing achieve Exascale performance says, "Congestion management and flow control mechanisms are of particular concern at very large scale ..." even though HPC programmers are anticipating the use of fine-grained asynchronous communication to help [20]. Application-level congestion management alone simply is not enough [96].

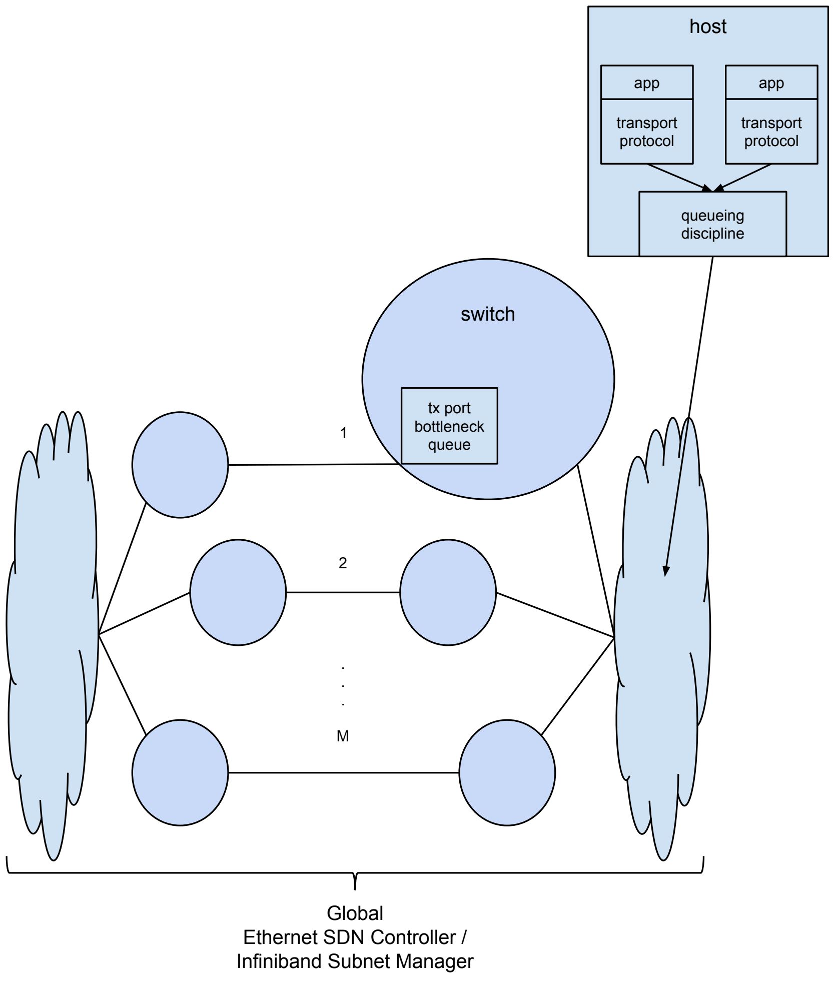

Figure 1.2: Network QoS Layers

Providing QoS on networks is complex because several independent resources must be managed in concert: transmission and reception queues on the communicating hosts and the transmission queues on the switches. As Figure 1.2 shows, flow is affected by the route it takes, the bottlenecks on that route, and the manner in which a host sends its data.

Table 1.1 summarizes the scope of the resource management problem for various topologies of both wired and wireless networks. The simplest case is one in which two hosts

Table 1.1: Scope of resource management problem for various types of networks.

Network Description	Complexity
two linked hosts	$2 \times \textit{host port}$
n chained hosts	$2(n-1) \times \textit{host port}$
d dimensional P2P fabric	$2dn \times \textit{host port}$
switched	$2 \times \textit{host port} + \textit{switch port}$
hub or wireless	$2 \times \textit{host port} + \textit{collision domain}$
wireless P2P fabric	$dn \times (2 \times \textit{host port} + \textit{collision domain})$

are directly connected to each other, and even then each host must schedule use of its network interface between competing processes and threads. The resource management problem quickly increases in difficulty as the topology of a network becomes more complex, but the distinguishing feature of network resource management is its distributed nature. Scheduling decisions cannot rely on global knowledge without incurring prohibitive communication overhead, making uncoordinated transmissions (i.e. statistical multiplexing) generally preferred over other strategies.

1.2 Types of Networks

Current network technology, no matter how fast or expensive, lacks a compelling solution to congestion and end-to-end performance guarantees (i.e. QoS). Of course, networks vary in capability, from the cheap and unreliable to the expensive and robust. TCP Inigo, one of the primary contributions of this research, is described in chapter 4 and applies to almost any network, since its techniques do not require special support. Even the fastest, most expensive networks, found in supercomputers and clusters used for scientific simulation must use TCP at some point even if they use higher performance protocols internally. Note that this dissertation primarily concerns itself with wired networks. Although the ideas concerning delay still apply, wireless technology brings another set of problems which are out of scope for this research.

The three major classes of storage networks are Network Attached Storage (NAS), the Storage Area Network (SAN), and the distributed file system. NAS is the most common and least expensive storage network, where one or more servers individually provide a file system interface over a standard Ethernet network. More expensive SANs are composed of storage arrays connected with a high performance network such as Fibre Channel and appear as a local device to a host. Distributed file systems come in Wide Area Network and Local Area Network (LAN) variants. Wide area systems serve large numbers of users and operate over a large variety of technologies. In contrast, local area systems are designed to provide a high performance parallel file system for a set of well-defined users. In general, all networks are grown over time even if they were initially designed.

The market is pressuring vendors to unify storage products with converging technologies that provide reliable, or nearly lossless, transport (e.g. Fibre Channel Over Ethernet and Converged Enhanced Ethernet). However, the adoption of new standards often takes the better part of a decade, and it is an uneven process. Priority Flow Control (PFC) was accepted by industry, but can deadlock. The Quantized Congestion Notification standard alleviated fears of PFC deadlock, but has rarely been implemented. The much older Explicit Congestion Notification (ECN) standard is often unused even though it is supported [82]. So, most Ethernet network still suffer packet drops.

Figure 1.3 demonstrates congestion in a simple switch model, which is applicable even if advanced link level flow control features are available. Packets contending for the same

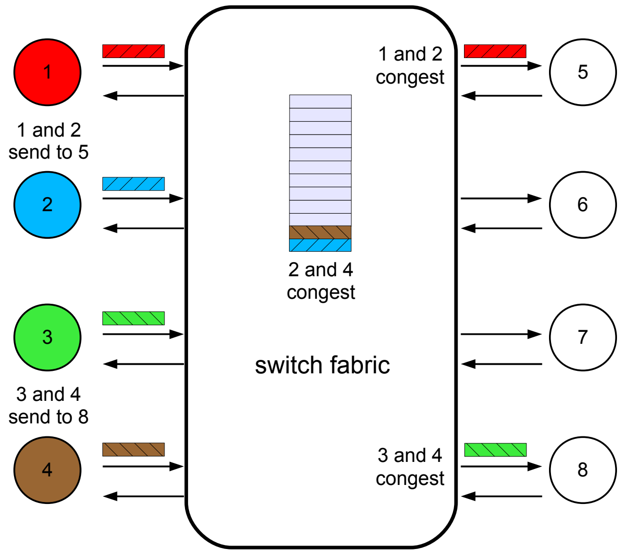

Figure 1.3: Multiple flows harm each other with congestion in a simple switch model.

destination port are queued. Continuous contention may cause previously isolated streams to interfere with each other. In the worst case, the queue will overflow and packets will be lost. Distributed file systems experience a particular case of congestion called incast [81, 121] where a file spread among many servers is sent in simultaneous bursts to a client, which can overflow a switch buffer with little or no warning signs.

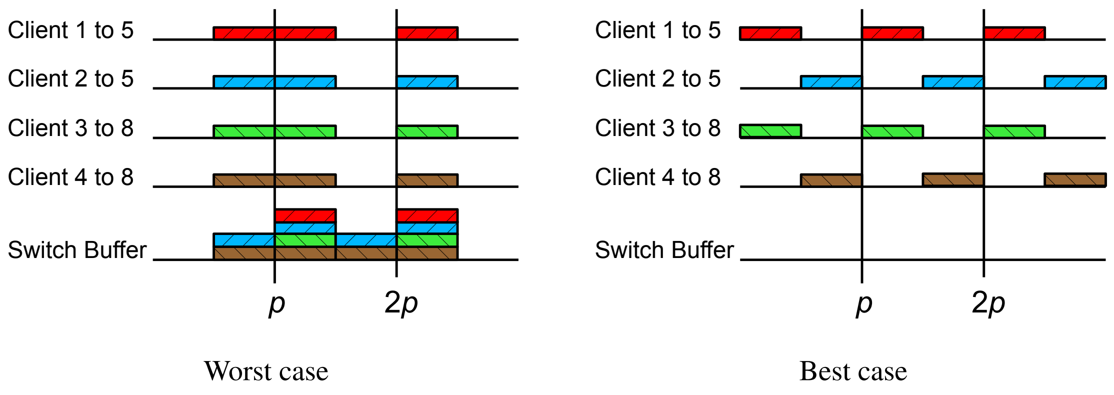

Figure 1.4: Time-series plot of a switch buffer with and without congestion

Figure 1.4 shows the best-case behavior of a switch performing statistical multiplexing on the right and the worst-case behavior on the left, over three periods. Assuming that each client is using half of the period to transmit, the switch will never exceed four units of buffer space, and each packet will always be served within the period of its arrival. On the other hand, in the best case, transmissions from each client interleave perfectly, the switch uses no buffer space at all, and the latency for individual packets is minimal.

High speed network architectures exist that provide reliable transport using credit-based flow control and multiple levels of QoS. When flow control is implemented throughout the network, some of the worst effects of congestion are mitigated (i.e. prevention of packet drops). However, high speed networks are not immune. They suffer from a form of congestion called the Parking Lot Problem [55, 56, 160]. Therefore, it is important to supplement flow control with improved congestion control. Also, their priority-based QoS is only appropriate for static network flows as opposed to the dynamic flows seen in many environments.

Neither converged enterprise networks nor more exotic high speed networks provide hard latency or throughput guarantees without significant over-provisioning. This research focuses on providing flexible, general, and fine-grained performance guarantees on a local area distributed file system.

1.3 Real-world Considerations

Regardless of whether or not a transport is reliable, there are other real-world considerations. For example, the maximum transmit unit (MTU) and buffer sizes should increase with speed of the network, otherwise the number of interrupts the hosts' CPUs must handle will exceed its capabilities. While the network adapter may offload some of the protocol processing overhead from the CPU, but this does not always result in a positive result for all workloads and it can complicate debugging problems in the network. In addition to the capabilities of hardware, the topology of the network also has implications with regard to making performance guarantees.

A Network Interface Controller (NIC) affects the performance that can be achieved and the amount of overhead it imposes on a host computer. A large amount of effort has gone into interrupt moderation techniques, where one interrupt is delivered for a group of packets in order to reduce the amount of work required by the CPU. While this can successfully increase throughput, it can also cause an undesirable increase in latency variation. In Linux, drivers can be written to conform to either the softnet or the "New API" [135]. NAPI solved several problems with softnet, including avoiding interrupt livelock [106] and packet re-ordering, by switching between interrupt driven and polling modes depending on the number of packets being received. The Linux kernel is continuing to evolve, traditional interrupt handling with top and bottom halves will likely be replaced by the real-time Linux tree's threaded interrupt handlers. Also, significant work has already been merged to allow hardware with multiple queues to scale its processing over multiple cores.

While small packets can be transferred much faster than large packets, they inefficiently use system resources. Overall performance can be maximized if the Maximum Transmission Unit (MTU) of the NIC is a multiple of the operating system's memory page size. This allows the PCI bus to use its maximum transfer size without requiring an extra transfer to handle the last part of a packet.

NIC designers have added the ability to offload various pieces of the work generally handled by the operating system. A TCP Offload Engine (TOE) implements an entirely separate TCP stack in hardware. Generic Segmentation Offload (GSO), refers to a NIC with the ability to take a large buffer and split the data into packets on behalf of the operating system. The complement of GSO is Generic Receive Offload (GRO), which merges received packets. One of the primary benefits of both GRO and GSO is a reduction of the load on the system bus. When segmenting or merging packets the hardware must also support offloading the checksum operation, so the main CPU also does less work. Furthermore, there are security and performance pitfalls with any of these advanced hardware features, so many drivers allow them to be disabled. What may be beneficial on a multi-user system may impede performance on a router or hinder packet filtering functionality. One of the more useful features from a resource management standpoint is the availability of hardware generated timestamps for each packet.

Network performance is largely determined by a protocol's flow control, which manages the rate at which a stream injects data into the network when there is no congestion, and congestion control, which adapts the rate, burstiness (i.e. dispersion), and timing of packet transmissions when congestion is detected. TCP/IP is the most widely deployed end-to-end network protocol, but its congestion control algorithms do not provide performance guarantees. A TCP sender continuously tries to increase throughput by increasing the window (burst) size and uses packet loss as a congestion signal to throttle the sender drastically. Even for a single connection, this results in a sawtooth pattern for throughput over time and a large variance in packet delays, as a switch's queue continually overflows and drains. Feng, et al. [28] show that the flaws in TCP's congestion control dramatically worsen as the number of streams scale up in local area distributed system because the streams tend to respond to congestion in lock step.

Many researchers have sought to improve or replace TCP, but change has proven to be difficult because TCP actually does a decent job in many situations. TCP is fairly robust, the Internet community desires new protocols to be friendly with older versions, and it is difficult to get buy-in for new ideas from a significant portion of the Internet community. The current default variant used by the Linux kernel is called CUBIC [59, 58] because of the function it uses to modify its window size. It is intended mostly to enhance behavior on high bandwidth networks with large delays.

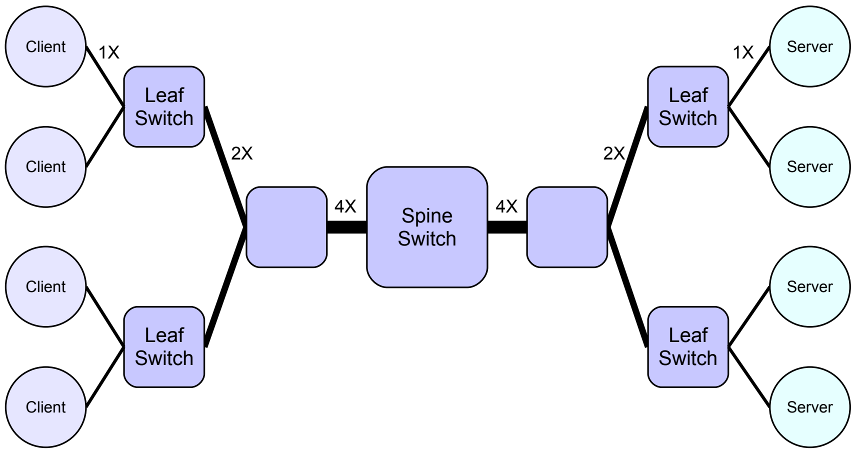

Figure 1.5: Canonical Fat-tree storage network.

Figure 1.5 depicts a canonical network–a closed, full bisection bandwidth, Fat-Tree [88] network composed of standard Gigabit Ethernet switches. If the network is segmented into equal parts, then the links connecting the switches allow all pairs of hosts to communicate at their full link bandwidth.

Real-world enterprise networks may provide less than full bisection bandwidth or redundant routes, but a basic Fat-Tree is a reasonable starting point for congestion and QoS research. For now, it is assumed that the network is tightly controlled and that the only significant source of packet loss is due to a buffer overflow. After achieving guaranteed performance at the single switch level, further research must explore the affects of more complex topologies. A good congestion control algorithm should be able to converge quickly to fair shares of bandwidth, and minimize losses and delay variation. With QoS, a good congestion control algorithm should be able to guarantee up to the bisection bandwidth of a Fat-Tree network.

In summary, scheduling network resources is a difficult theoretical problem because many resources must be scheduled coherently and centralized control includes too much overhead to be efficient. Even when the theoretical problem of scheduling resources is not compounded by extraneous administrative and economic factors, network hardware designers cannot afford to provide separate queues to each flow. Therefore, congestion continues to be a fundamental problem preventing fine-grained performance guarantees.

Chapter 2

Previous Work

Before diving into the history of scheduling on network resources, it is important to remember some of the broader history of scheduling. Because early systems were severely resource constrained, the first scheduling algorithms focused on maximizing responsiveness (foreground processing) or minimizing context switching (background processing) using the simplest, most efficient algorithms possible. Next, schedulers were developed that dynamically adapted their parameters depending on whether a task was CPU or IO limited. As systems became less resource constrained, schedulers have been asked to meet more specific performance targets, such as when a multimedia process needs to finish a certain amount of work during each interval of time.

General systems have most often used the concepts of priorities or shares in order to meet the performance goals of different processes. However, these approaches can lead to processes being inappropriately ranked as always more important than others, even though they are only more important according to one criterion. The Resource Allocation and Dispatching (RAD) model [19], developed by Brandt, et al. more than a decade ago and described in § 2.3, describes a better model that meets the multidimensional work and time goals common in modern, dynamic systems.

The direction of the research of this dissertation has been heavily influenced by RAD. TCP Inigo, described in chapter 4, is an end-to-end protocol which is not based on RAD, but is intended to solve the problem of congestion so that RAD theory can then be applied. On the other hand the economic and real-time scheduling theory described in chapter 5, involves the direct application of RAD to queueing disciplines.

2.1 Network Scheduling

Whenever devices communicate, potentially mismatched speeds or overloaded routes commonly result in data being queued in buffers. The data in a well behaving buffer drains between peaks, but the data in a badly behaving buffer persists, taking up space needed to handle new packets and increasing delay without any benefit to the application [112].

Five general techniques used to encourage good buffering are:

1. dropping packets to implicitly signal senders to slow down
2. explicitly requesting senders to slow down
3. delaying packets to increase interleaving of packets from different sources or match upstream speeds to downstream speeds
4. selecting different routes for load-balancing

5. classifying packets in order to apply combinations of the above to different types of traffic.

These techniques can be applied in hardware, software queues immediately above the hardware, communication protocols, or in applications.

This section describes how the five general techniques listed above have been applied to the network, beginning with the bottom layer. Additionally, the section discusses how well they address the five network design goals first listed in the abstract. The goals are to:

1. minimize latency
2. maximize bandwidth
3. share resources according to agreements
4. enable incremental deployment
5. minimize administrative overhead

There is considerable practical value in being able to improve network performance while minimizing the effort needed to deploy and maintain the changes. When work done in hardware must be redone in software layers (i.e. guaranteeing end-to-end delivery across different networks), or a feature increases the complexity of network configurations, it will likely find itself in a losing tug-of-war with end-to-end arguments [14, 136]. For instance, if two workable solutions compete, where one is implemented at end-points and the other is implemented in middle-boxes, the end-point solution will be more economic and robust in the face of evolving technology.

It should be noted that *network-layer* approaches such as load balancing of routes and assistance from Software Defined Networking are out of scope for this dissertation. However, they are complementary to all of the techniques described below.

Link-layer

The *link-layer* hardware of some networks can limit the growth of buffers while providing per-hop fairness. Earlier Ethernet switches may support a limited form of link-level flow control by sending a PAUSE command to transmitting devices to indicate they should slow down for a specified amount of time. This can help in some situations, but the early flow control standards were implemented using multicast packets and do not differentiate between senders. The early standards also ignore Ethernet priorities.

Newer Ethernet standards, collectively known as either Data Center Ethernet or Converged Enhanced Ethernet, include Priority-based Flow Control – addressing the flaws in the previous Ethernet PAUSE command, Enhanced Transmission Selection – allowing different priorities to share each others' spare bandwidth, and Congestion Notification – providing upper layer protocols such as TCP with information to help them set their transmission rates. Even with those enhancements, Ethernet will probably not match the performance and reliability of more advanced technology such as Infiniband or Omnipath.

Infiniband's credit-based flow control and Ethernet's Quantized Congestion Notification (QCN) [47, 77] are two examples of more advanced link-layer solutions. Infiniband has seen success in supercomputing, but it still experiences Head-of-Line (HoL) blocking and congestion in the form of the Parking Lot Problem [55, 56, 160] when many flows are treated as one during subsequent hops in a network fabric. Reducing or eliminating HoL is being actively researched [138], and Infiniband does provide hardware-level congestion control. However, an informal survey of Department of Energy labs indicates that Infiniband's congestion control is left disabled due to unstable performance. While Infiniband has been at the forefront of high speed commodity networks, its advantages have not translated into displacing Ethernet due to its greater cost and incompatibility with existing infrastructure. With regard to Ethernet's link-layer congestion control, the only QCN-enabled hardware the authors of this paper are aware of comes from Mellanox, and that is likely because they support similar features in their Infiniband products.

In a personal communication in January 2014 the editor of the QCN standard said, "... The promise of [QCN] congestion control made more palatable the standardization of IEEE Std 802.1Qbb Priority Flow Control, to which objections were raised on the grounds that it could cause a deadlock. ..."

> "Judging by the scarcity of implementations of IEEE Std 802.1Qau, the principle benefit obtained from the standard may not have been congestion control, itself. The promise of congestion control made more palatable the standardization of IEEE Std 802.1Qbb Priority Flow Control, to which objections were raised on the grounds that it could cause a deadlock. 802.1Qau lessens the likelihood of 802.1Qbb deadlocks."

Another standardization effort is the Time-Sensitive Networking (TSN) Task Group of the IEEE 802.1 Working Group [52]. TSN is intended for audio, video, and time-sensitive control over small Ethernet networks, sacrificing bandwidth for low latency. TSN uses "Traffic SPECifications" (TSPEC) reservations (max packet size, peak data rate, max burst size), as specified by the Stream Reservation Protocol (SRP).

The admission control done using the SRP must account for the worst case possibility of every source's packets arriving at a switch port at the same time. So, if the latency requirements dictate that a buffer can only be 8 packets deep, then there can only be 8 sources. A limit of 64 sources would not be too bad since even large switches are made up of modules that limit the possible contention for any given port to at most 64:1. Maybe that ratio will increase.

TSN requires packet pacing, and the best pacer in Linux right now is the Fair Queue qdisc. However, it does not appear to be the same as the one referred to in the TSN standard. The TSN bridge traffic shaper is only required to pace all flows as an aggregate, and not on a per-flow basis. In contrast, the FQ qdisc is designed to interleave the packets from different flows to achieve efficient pacing.

Traffic Shaping and Classification

Traffic shaping and *classification* may be implemented in hardware or in the software layer immediately above it. Traffic shaping throttles bandwidth or rate-limits transmissions by delaying packets, and classification enables various techniques, such as shaping, to selectively affect applications.

Le Boudec's and Thiran's Network Calculus [86] thoroughly describes the behavior of traffic shaping using Min-plus algebra to reason about the equations representing arrival and service curves. The analysis reveals that for feasible flows, a switch's buffer requirement is the sum of the burst sizes that can arrive simultaneously for a single destination port, regardless of any other parameter. They prove that bursts from peer links must be paid for only once, since traffic becomes serialized at the first bottleneck. Of course, the next hop in a fabric may include more peer links with flows destined for the same transmit port.

Loeser and Haertig used simple cooperative traffic shaping [95] to limit the size of the transmit queues used in an Ethernet switch in order to provide real-time guarantees. With respect to a reservation, the equations used to calculate the size of its buffer is dependent on the burstiness caused by the shaping interval. The maximum delay of a packet is the sum of all buffers divided by the speed of the switch plus a switch-specific multiplexing constant. This bound assumes that the switch uses one big First In First Out (FIFO) queue and does not keep track of a separate queue for each transmit port, servicing them in a round-robin fashion. A smaller shaping interval reduces burstiness, buffer size, and maximum delay; but it also increases the CPU load. As the number of clients grow, the total amount of buffer space and the maximum packet delay grow until they reach the limit of the switch's memory. In the end, this sort of overprovisioning fails to effectively utilize the theoretical capacity of a network since the worst case is uncommon and it requires hosts to know characteristics of middle-boxes, which is impractical.

Queue Scheduling

Queueing disciplines (i.e. queue scheduling algorithms) are largely studied independently from other types of resource scheduling. Beyond FIFO and its associated "Drop Tail" semantics, the most common queue schedulers are descended from Processor Sharing [1, 80] on early time-shared computer systems. Queue scheduling gradually grew beyond Processor Sharing's early Round-robin algorithm, gaining the ability to add weights to different classes of traffic. A second family of queueing discipline is descended from Shortest Job First (SJF). SJF and its preemptive variant, Shortest Remaining Time, can suffer from starvation but works well when workloads can be described by a heavy-tailed distribution with many short flows [137, 61].

There appears to be little interaction between current research on queue scheduling and real-time processor scheduling. A handful of network papers [8, 153, 161] reference Earliest Deadline First (EDF) [93], but they are the exception to the rule, and significant advances have been made since EDF was first published.

Whereas a traffic shaper treats all packets coming through it the same way, *Active Queue Management* (AQM) manipulates packets depending on the state of the queue. When the number of packets or bytes in a queue configured with the Random Early Detection (RED) [51] AQM exceeds an upper limit, it drops or marks the packet. If the queue length is below a lower limit, then the packets are left alone. And between those thresholds, RED randomly drops or marks a packet in proportion to the queue length relative to the thresholds. Hardware support for RED is common in switches and it is able to signal TCP to lower its congestion window, however RED is difficult to configure and remains unused in practice.

Controlled Delay (CoDel) is perhaps the most successful Active Queue Management (AQM), and it [31, 108, 112, 150] solves bufferbloat by limiting the amount of time packets remain in a queue. By default, CoDel assumes an average RTT of 100ms and a target of 5ms of delay. It was designed and tested for edge routers of Internet connections, although should theoretically work with tuning for lower latency networks. The biggest issue with CoDel is that if it is not the controlling the slowest link, then it cannot control delay, and that is exactly the case when flows converge somewhere else on a network.

The Fair Queuing packet scheduler implemented in the Linux kernel, works in conjunction with TCP to set a pacing rate to improve the interleaving of flows [36]. In general AQMs and improved TCP congestion control algorithms are complementary to each other because in situations where an AQM can be deployed, it prevents non-TCP flows from over-filling buffers. Getting the maximum theoretical benefit out of AQM would require deploying it ubiquitously throughout a network. Yet even if one AQM, such as CoDel, is only deployed on the edge of networks, it provides the clear benefit of preventing a user from overfilling their own buffers. Variants like Fair Queuing with CoDel or CAKE [108] add important features such as hashing flows to independent queues and pacing packets, which make the combined AQM+TCP solutions more effective.

If the type of network that CoDel is being used on does not provide back-pressure (i.e. the network allows new packets to be inserted even when the next hop does not have adequate buffer space), then a traffic shaper must be added so that CoDel can manage the queue of the slowest link.

A network cannot afford enough hardware queues to keep each flow separate, and software AQMs end up having to dissect flows from the aggregate packet stream in order to enforce fairness or QoS. This is problematic because it creates more work (i.e. the end points higher in the stack already treat flows separately) and Network Address Translation complicates the dissection. This happen because queueing disciplines, in Linux at least, are applied after packets have been translated on egress and before they have been translated on ingress.

Transport Protocols

The Transmission Control Protocol (TCP) is the broadest deployed *transport protocol* ensuring reliable delivery of data. Applications use TCP through a socket interface, that abstracts much of the details of communication. There are other protocols that provide similar guarantees to TCP, but they are difficult to use in practice because middle-boxes regularly limit connections to either TCP or Unreliable Datagram Protocol (UDP). Even TCP's evolution has been hampered by middle-boxes which modify its headers and option fields in a fashion that prevents adoption of new options as allowed in the TCP specification.

TCP endures despite its suboptimal performance in various scenarios because it assumes little about the details of the networks it traverses, yet is able to provide adequate performance in most cases, as long as packet loss generally results from congestion instead of the reliability of the connection. Even though TCP does not know the speed of a network or where bottlenecks lie, it leverages the idea of using acknowledgments (ACKs) from the receiver as a clock in an effort to follow the principle of 'conservation of packets' [70]. A *Slow-start* phase starts the ACK clock, quickly discovering bandwidth by doubling the congestion window—a limit on amount of data sent every Round-trip-time (RTT). A retransmit timer based on a good RTT and variance estimator ensures data is only resent if it should have arrived long ago. And a *Congestion Avoidance* phase responds to a signal of congestion once every RTT by halving the congestion window and then slowly probing for bandwidth.

Delay has a difficult history of being used effectively within TCP. It cannot be the only signal used by congestion control, although it should be used [63]. TCP Vegas [17, 65] was an earlier delay-based congestion control variant that showed promise when tested by itself, but suffered badly when attempting to co-exist with loss-based TCPs. Other delay-based protocols followed Vegas and have met with mixed success; and FAST TCP [76], Compound TCP [151], and others have not become ubiquitous either. Even so, delay-based protocols or enhancements are actively being pursued by Facebook with New Vegas [146] and Google with various projects, including

The RACK [166, 165] algorithm is particularly interesting with regard to this dissertation because it uses a similar threshold—$1.25RTT_{min}$ versus $1.17RTT_{min}$—to make decisions. However, RACK is purposely decoupled from congestion control. It is used only for detecting when packets are lost. Also, its threshold has a default and lower bound of one millisecond.

TCP Santa Cruz [119] modeled queueing in a switch by summing the RFDs over an interval and dividing that by the average packet service time during that same interval. TCP Santa Cruz was implemented in the NS-2 network simulator, and so did not have to contend with noisy delay measurements.

Probe Control Protocol (PCP) [12], a congestion control protocol outside the family of TCP, uses probe packets to detect if the network can currently support a specific load and converges to a desired throughput using short, paced, high-rate bursts. PCP is shown to outperform traditional TCP in various ways including response time and loss rate, and recovers from incast after some packet loss. Despite all of those positive features, PCP never progressed beyond a simulated prototype.

Matthew Mathis and Bob Briscoe have put forward an interesting variant of TCP, Relentless TCP [100]. They observe that existing interactions between AQMs and TCP senders make it difficult to control traffic. Instead of halving the congestion window, Relentless TCP simply reduces the transmission window by the number of segments dropped instead of halving the window once per RTT if one or more packets is dropped. This simple and direct effect makes it straightforward for a traffic controller to exactly adjust a flow's rate. Since it is not friendly to other TCPs, Relentless TCP has not been widely deployed.

There have been many attempts to create a replacement for TCP, including including XCP [78], RCP [41], and many others. While many good ideas appear in XCP and RCP, they tend to run into fundamental problems. There is an enormous amount of inertia behind TCP, and it is easy to screw up something that TCP has already figured out. Furthermore, many ideas are on the wrong side of the End-to-end argument [14, 136]. Clean slate designs can produce valuable insight, but in practice, it appears that the most valuable research works toward evolving TCP rather than replacing it.

Standard ECN support [130] directs a sender to halve its window once per RTT upon seeing an acknowledgment (ACK) marked with Congestion Exists (CE), whereas DCTCP [4] tracks the ratio of bytes marked with CE to the total number of bytes ACKed in order to estimate the extent of congestion. A congestion ratio of 1 causes DCTCP to halve its window, and smaller ratios cause it to back off correspondingly less.

When ECN is not supported by the receiver, DCTCP falls back to basic TCP Reno. If the receiver supports ECN, but was not modified to accurately convey ECN with delayed ACKs, then DCTCP will under-estimate the extent of congestion. Kato developed a one-sided variant of DCTCP [79]. However, it compromises the performance of DCTCP when the receiver has been modified.

New TCP header options could allow a sender to detect if a receiver has been modified. Unfortunately, that type of solution can run into problems when middle-boxes manipulate headers without properly supporting new or rarely used options. The primary tool at present for ensuring DCTCP senders talk to modified receivers is to configure per-route congestion control. While this works for homogeneous subnets, it is an increasingly complex and infeasible solution when communication occurs between a wide variety of hosts controlled by other organizations.

Switches must also be configured to mark ECN appropriately for use with DCTCP. Configuring for DCTCP is simpler than for RED [51], and it can use the common support for RED in Ethernet hardware. However, there are many situations where DCTCP cannot be easily deployed. A cloud provider may not be able to force all tenants to use a buffer-friendly TCP, they may consider configuring separate switch queues to be impractical, or setting per-route congestion control on the application side may not be fine-grained enough for the applications running on their cloud.

In cases like these, it would be good to fall back to behavior as similar as possible, but with fewer requirements. Based on our observations thus far, we submit that existing RTT measurements are adequate to the task. Up until this point, congestion control algorithms using existing delay measurements have not resulted in low bottleneck queue depths on switches and tight latency distributions similar to DCTCP.

Some newer TCP congestion control variants [4, 87] reduce latency, increase bandwidth, and share resources more fairly than older TCPs with the help of Explicit Congestion Notification (ECN) or improved timestamping. However, configuring switch queues to mark ECN appropriately or separating low latency protocols from aggressive legacy traffic is not always feasible. And the inertia and variety of networks makes modifying both senders and receivers, altering drivers, or adding new TCP options a daunting challenge.

While change may be well worth it in some cases, networks tend to resist change. Consider the slow uptake of IPV6, ECN, RED, and FQ_CoDel [112]. Even when hardware and software support become common, network administrators and application developers do not change their configurations quickly (or at all) to take advantage of them.

The difficulty of adopting significantly different protocol headers should not be underestimated. It has taken 20 years for IPV6 to reach 10% adoption because the annoyances of using IPV4 were not enough to encourage the upgrading of every client, server, and middle-box (e.g. router, firewall, load balancer, and management system). Increasing pressure from address exhaustion might be enough to finally drive widespread change by 2020 [154].

Many enhancements have been proposed to improve or leverage DCTCP, including RTT-fairness through sub-window adjustments [5], improved fairness [77], ultra-low latency with phantom queues [7], deadline-awareness [153], minimizing flow completion times [110], sender-side only DCTCP [79], application to wireless networks [156] stability enhancements [27], elimination of Slow Start in conjunction with Data Center Bridging [148], and various ideas for deployability enhancements [142].

Academia is not alone in trying to take DCTCP further. The Internet Engineering Task Force (IETF) is discussing DCTCP's vulnerability to ACK-loss, along with the ways they might improve congestion notification and DCTCP [24, 82, 37]. Apple is enabling of ECN in all its software [84], which could encourage more ECN marking in routers and help make DCTCP feasible on the Internet [83]. However, those routers would need to be configured both to mark

ECN as DCTCP requires and to enable DCTCP to coexist with other TCP variants. Change of that magnitude should not be expected.

DCTCP has not eliminated the interest in other congestion control algorithms in the data center or for the Internet. CAIA Delay-Gradient (CDG) TCP [62] uses minimum and maximum RTTs to reason about congestion, with an emphasis on coexistence with loss-based congestion control in wide area networks, and it was merged into the Linux 4.2 kernel. It is a sender-side only modification, so it is easy to deploy.

Incast Congestion Control for TCP (ICTCP) [164] is one of very few receiver-side congestion control algorithms. It shows that sometimes the receiver is best able to decide how to respond to or prevent extreme congestion scenarios, such as incast, when many servers send data to one client. There approach adapts TCP Vegas' congestion control and is able to prevent incast better than DCTCP. ICTCP was implemented as a Microsoft Windows driver, which allowed it to transparently affect the behavior of virtualized guests.

Google's Chrome Project has been experimenting with the Quick UDP Internet Connection (QUIC) [126], with reduced connection and transport latency being two of its goals. It seeks to send an initial payload without requiring multiple handshakes like secure TCP connections. Forward Error Correction allows QUIC to recover from data loss without incurring retransmission latency. The original documents for QUIC indicated that it would pace packets, but the latest documentation only refer to CUBIC and New Reno style congestion control [69, 127]. This change is probably because pacing can be done more effectively in the Fair Queuing packet scheduler.

Remy [147, 162] has been used to generate congestion control protocols, and it compares favorably to many previous loss-based and delay-based TCPs in simulations. However, RemyCC results in RTTs $4-6\times$ worse than DCTCP since Remy does not yet take advantage of ECN or AQM. The Tao protocols in later Remy experiments appear to approach the performance of an omniscient scheduler, but DCTCP was not included in that comparison. It remains to be seen if machine generated congestion control is practical or if it can lead to new and better understanding of congestion.

Dong, et al. [40] make the argument that even though Remy generates protocols, it searches a space of hardwired responses to packet level events, and its performance can degrade when the real network does not match its assumptions, just like most TCPs. They propose Performance-oriented Congestion Control (PCC), a sender-side modification to TCP that modifies its rate and packet pacing based on continuous experimental trials of rates differing 1-5%. PCC makes fewer assumptions than most congestion control algorithms, but one assumption it shares is that repeatedly trying higher rates is necessary even if those attempts always lower the measured utility. A PCC prototype is publicly available, but the current implementation of its packet pacing either consumes an entire core per flow or can be fragile depending on the operating system version or virtualization.

There has been a long line of TCP congestion control variants that try to keep congestion low and do not require extensive change or configuration of network equipment, but most are unable to compete for bandwidth with loss-based TCPs [22]. Some variants, such as CAIA Delay-Gradient (CDG) TCP [62], are able to coexist with loss-based TCP to some degree. Others have maturity or performance issues that prevent them from being practical. Performance-oriented Congestion Control (PCC) [40] has a prototype that has issues with its packet pacing implementation.

Lee, et al. propose DX [87], which shows that accurate queue delay measurements can be attained even for high speed networks by modifying drivers, adding TCP options, and modifying both senders and receivers. Their congestion response is driven by the ratio of the measured average queuing delay to an estimate of the number of competing flows, resulting in higher utilization and lower latency than DCTCP. The combination of these changes is almost impossible to implement in reality since broad support for new TCP header options is difficult to attain and diverse networks cannot be expected to have compliant hosts. TIMELY [105] uses hardware timestamps, delay gradients, and rate control to implement congestion control for RDMA traffic. While this paper does not include a direct comparison with DX or TIMELY, it is reasonable to expect that TCP Inigo would also benefit from improved timestamps. However, TCP Inigo can be used without any additional development effort and on any hardware.

Application

An application may be better suited to manage congestion than lower layers of the network stack because of special knowledge. That reasoning is part of why the Unreliable Datagram Protocol exists. If a programmer controls many end points, then they can take the most appropriate action to reduce network load.

Furthermore, it is difficult to evolve TCP without breaking it. New ideas cannot simply be an improvement—they also have to be deployable. With Google's experimental QUIC protocol, and now Transport Over UDP [35], it appears application developers may be willing to take on the responsibility of providing their own network stacks. This flexibility comes with many potential disadvantages: encapsulation is a performance overhead, it enables proprietary solutions, and it could fracture efforts to improve networking into even more efforts that must each independently be debugged.

Luo, et al. [96] report that one-sided paradigms suffer less from congestion than two-sided paradigms, but that they still suffer significantly from congestion. In fact, one-sided paradigms may be less effective than simple throttling.

> The behavior of RandomAccess illustrates an interesting performance inversion phenomenon: an implementation with blocking communication and congestion avoidance is able to attain better performance than an implementation hand optimized for communication overlap. Best performance is obtained by the implementation optimized for overlap and using congestion avoidance.

They propose throttling back on the number of cores used, as well as throttling the rate at which cores send messages, and show a 2× improvement for collective operations, 60% improvement for fine grained application benchmarks, and 17% improvement for the NAS Parallel Benchmarks.

Routing

The research in this dissertation is intended to address the congestion problems that arise on simple switched networks with full bisection bandwidth. As a network's topology becomes more complex, questions about how to best use multiple routes will naturally need to be addressed. For instance, admission control will need to be combined with a routing algorithm in order to maximize utilization of the network.

Google's B4 [74] is a global software defined WAN. They abstract each site to a single node on a graph with a single edge to each remote site. All links from a site to a remote site are treated as one link by using a custom variant of ECMP hashing. This simplification of the topology was done for scalability.

B4 operates on Flow Groups — source, destination, QoS tuples — rather than individual applications. This was also done in order to achieve scalability. B4 maps Flow Groups to a set of Tunnels which represent routes, preferring shortest paths first. That mapping, combined with corresponding weights, is called a Tunnel Group. QoS is specified with a Bandwidth Function, where the weight/priority is the slope of the function.

Google tried an optimal fair share algorithm called LP, but it was too slow. Instead, they created an algorithm that achieves similar fairness and utilization, but is faster. It iterates over the Flow Groups, looking at preferred tunnels first (min cost path with no contention). When multiple Flow Groups require more bandwidth and must use paths that share an edge, the algorithm iterates to find a fair share value that, when plugged into the Bandwidth Functions of the Flow Groups, produces their fair share ratios. Bandwidth is iteratively parceled out. Each Flow Group's demand is met, with the slack apportioned according to the Bandwidth Functions. Afterwards, the ratios need to go through a quantization so that they match the hardware capabilities.

As of Nov. 12th, B4 was managing 2700 Flow Groups and 240 Tunnel Groups. Most of their Traffic Engineering Operations were completed in 5 seconds. One half to one third of that time is taken by the Traffic Engineering Optimization algorithm.

Hedera [3] is intended for data center network flow scheduling, and uses simulated annealing to centrally schedule long lived flows while falling back to ECMP forwarding for short lived flows. It focuses on natural bandwidth demands rather than QoS specifications. To scale, they assign a core switch to each host rather than assigning it per flow. Their algorithm can schedule 27K hosts and 250K large flows in 100-200 milliseconds, with a demand measurement and reallocation frequency of 5 seconds. While Hedera generally performs much better than static ECMP hashing, it can be improved upon. The "optimal" non-blocking switch they compared to sometimes performed worse than Hedera, so it is difficult to say how close to optimal Hedera is. However, in their shuffle experiment, Hedera achieved 86% of the bisection bandwidth as the control network.

Multi-layer Approaches

Traditional traffic shaping, even combined with a global routing algorithm which ensures routes are not overloaded, could still allow a bottleneck queue to build up and drop packets. The bottleneck would have to be able to handle the worst case simultaneous burst from every flow on that route [86]. Furthermore, even lossless networks such as Infiniband suffer from congestion in the form of congestion trees and the Parking Lot Problem [55, 56, 160]. In practice, Infiniband's congestion control is not enabled because it must be tuned to the specific traffic patterns of the system. If the traffic changes, then overall throughput can be badly hampered.

Alizadeh, et al. proposed pFabric [8], in which flows are first class citizens, flow completion time is minimized, and rate control is decoupled from flow scheduling. End-hosts set packet priority independently, and adjust rate based on a minimal set of TCP mechanisms. Switches implement priority scheduling and dropping, and are capable of Earliest Deadline First (EDF) [93]. pFabric assumes 10Gbps host rates, a fixed retransmit timeout based on 3$\times$RTT, and a fixed chronic congestion threshold. Those improvements probably will not hold with heterogeneous and dynamic host rates which can be expected with incremental upgrades and energy-proportional networks. Simulations show pFabric delivers nearly optimal flow completion times, but its architecture allows starvation, and co-existing priority schemes require reconfiguration.

PriorityMeister [168] provides good end-to-end storage tail latency QoS by automatically configuring priorities and rates. The global controller greedily searches through a trimmed search space, requiring users to provide a Service Level Objective and a trace of access patterns. While PriorityMeister's autoconfiguration potentially makes it robust to a variety of scenarios, the approach was only shown for a few concurrent workloads. Its worst-case latency analysis focuses on a single workload, ignoring the interaction between workload arrival and service curves. And the controller must recalculate rates and priorities when workloads or service curves change. It meets 99.99% of latency objectives except for very bursty workloads because it treats the network as a black box.

PASE [111] combines normally independent strategies in order to minimize flow completion times better than any one approach. It coordinates DCTCP, distributed arbitration, and priority scheduling with modest end-host modification and available switch technology. PASE arbitrators have a overhead-accuracy trade-off inherent to pruning and delegation optimizations. They assume a tree topology and traffic similarity in both halves of the tree, adjusting if that is not true.

Dogar, et al. [39] focus on minimizing the completion time of tasks (i.e. sets of flows associated with waiting users) with Baraat, observing that Shortest Flow First scheduling shows little improvement over Fair Sharing when there are a large number of flows per task. They argue that users are best served in FIFO order when a task is small and by limited multiplexing when a task is heavy. Switches make consistent task-level scheduling decisions using a unique task priority assigned by a common entry point. The threshold governing when a task is considered heavy is determined using historical task size distribution of a data center.

QJUMP [57] combines traffic shaping with classification and priority queues. This approach can provide performance guarantees, but it is not a complete solution. The number of queues is significantly less than the possible types of traffic, and flows in the same class can still harm each other by filling up shared buffers. Furthermore, there is no guarantee that once a flow crosses into a new administrative domain that the owner will use the same traffic classification scheme or priorities.

2.2 Economic-based Scheduling

Google knows the benefit ordering for admission control [43].

> Ramaswamy was in charge of the many algorithms that make sure Google always orders the ads on its search results pages in the most profitable way. (Side note: For this, he deserves partial credit for the destruction of Yahoo, which used a simplistic straight auction to order its search ads. The difference in yield allowed Google to pay more for search distribution and eventually gain a practical monopoly.)

$$CoD = customer + business + criticality + risk + opportunity + \dots \tag{2.1}$$

$$WSJF = \frac{CoD}{Duration} \approx \frac{CoD}{JobSize} \tag{2.2}$$

Businesses have begun using the Weighted Shortest Job First (WSJF) algorithm developed by D. Reinertsen to minimize the cost of delaying work [68, 133]. From Reinertsen's economic viewpoint, FIFO is only appropriate when jobs are small and uniform, Least Slack Time First forces all jobs to share as much costly delay as possible, and schedules based on maximizing revenue ignore potentially high hidden costs. Similarly, Shortest Job First scheduling focuses exclusively on finishing what can be done quickly. And Earliest Deadline First maximizes slack in the system rather than allowing it to be leveraged to be more cost effective.

One of the key parameters of $WSJF$ is the Cost of Delay (CoD), which encompasses many attributes. Equation (2.1) defines a CoD appropriate for business that is the sum of various estimates, including but not limited to: preferences of the customer, business revenue, value decay, deadlines, dependencies between other jobs, and secondary benefits. After the CoD is summed, it is then divided by the expected job duration to produce $WSJF$, as seen in equation (2.2). Scheduling jobs according to the highest $WSJF$ ratio minimizes the sums of the products of the CoD and duration for each job.

Reinertsen gives guidance on estimating the economic benefits and risk of deferring work [134], and brings a more rigorous scientific approach than previous ill-defined business philosophies. Even so, applying WSJF to business often requires many estimations of costs and probabilities, whereas computer scheduling problems should be able to bring more live measurements to bear.

Job	CoD	Duration	WSJF		A,C,B	B,C,A	C,B,A	C,A,B
	($\$/s$)	(s)	$\$/s^2$		(\$)	(\$)	(\$)	(\$)
A	1.8	0.09	20.00		0.000	0.27	0.27	0.090
B	0.6	0.06	10.00		0.084	0.00	0.03	0.084
C	1.5	0.05	30.00	+	0.135	0.09	0.00	0.000
					0.219	0.38	0.30	0.174

Figure 2.1: Weighted Shortest Job First scheduling minimizes risk of not completing jobs occurring later in an ordering.

For example, consider a video website serving three types of jobs: *A* long advertisements, *B* short advertisements, and *C* primary content with costs and durations described in Table 2.1. The ordering A,C,B would be chosen by several different algorithms, including: Highest *CoD*, Highest $CoD \times Duration$. A Lowest *CoD* First algorithm would choose ordering B,C,A and Shortest Job First produces ordering C,B,A. Finally, WSJF results in ordering C,A,B . . . the only ordering that minimizes the risk and cost of not completing later jobs.

2.3 Real-time Scheduling

Scheduling algorithms like Earliest Deadline First (EDF) [93] would require all dispatchers contending for the same resource to know the release times of all jobs so that they can agree on the earliest deadline. Furthermore, the clocks of the dispatchers must be synchronized at a granularity corresponding to the differences between deadlines, and not just at the granularity of the periods. A different scheduling algorithm is clearly needed when a resource is scheduled by multiple dispatchers.

The Least Laxity First (LLF) [107] scheduling algorithm uses the notion of laxity, depicted in Figure 2.2. The laxity of a job $l_{i,j}$ is defined as the time remaining before the job

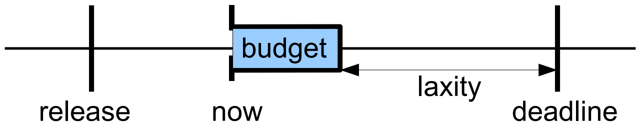

Figure 2.2: Laxity.

must be scheduled in order to meet its deadline, $l_{i,j} = d_{i,j} - t - e'_i$, where t is the current time and e'_i is the budget remaining in the period. In contrast with EDF, which schedules based on the deadline a job must be finished, LLF schedules based on the deadline a job must be started. LLF is optimal for scheduling a single resource in the same sense that EDF is, if a feasible schedule exists, then both will find one. Implementing LLF across multiple dispatchers would require just as much communication and synchronization as EDF, but it lends itself to an approximation suitable for distributed dispatchers because the measure of laxity is relative while deadlines are absolute.

Resource Allocating and Dispatching (RAD)

The Resource Allocation and Dispatching (RAD) scheduling model [19] has proven to be an effective way to provide a range of performance guarantees [92], first for CPU and later for disk [124] resources. Its success is due to the separation of *Resource Allocation*, which answers the question “How much?”, from *Dispatching*, which answers the question “When?” The *Resource Allocation* for a given task is specified using a reservation of some fraction of the resource at some granularity period, or more concisely, *Rate*. *Dispatching* schedules the work defined by the *Rate* such that it is finished by the *Deadline* at the end of each period.

Resource Allocation and Dispatching (RAD) reservations are $(rate, period)$ tuples that obsolete priority classes and previously defined rate-limit specifications. Prior non-realtime scheduling methods possess a limited number of relative, coarse-grained classes (priorities), require rates to be strictly satisfied for any measured interval (e.g. Token Bucket Filters), have common periods between all tasks, or have a fixed linear mapping between periods to priorities. RAD reservations enable arbitrarily fine-grained Quality of Service (QoS), possess meanings that stay consistent in a dynamic environment, and allow straightforward reasoning about composing end-to-end QoS.

The intention of adapting RAD to the network resource is to provide flexible, general, and fine-grained performance guarantees on standard commodity network hardware similar to what CPU and disk resources enjoy under the same general model. One important goal of applying the same scheduling model to every level of the operating system is the desire to compose guarantees system-wide.

Resource Allocation A task T_i's reservation (u_i, p_i), where u_i is network time utilization and p_i is the length of the period for which u_i is guaranteed.

Dispatching A task T_i has a budget $e_i = u_i \cdot p_i$, and is made up of a sequence of jobs $J_{i,j}$, each possessing a release time $r_{i,j}$ and a deadline $d_{i,j} = r_{i,j} + p_i$.

One of the ways the RAD model is flexible is that it allows the reservation to be specified in different types of units. Usually people prefer to specify I/O reservations as the amount of data per period, but that may not be the best way to implement the resource management. A data reservation is not appropriate if the usage pattern of the reservation has a large effect on the rate achieved. For example, disk I/O is greatly affected by sequentiality while network

I/O is greatly affected by the size of the individual jobs. If the achieved rates are dependent on the way in which the resource is used, then it is better to implement resource management with time reservations.

Perhaps the most important scheduler built on top of the RAD model is the Rate Based Earliest Deadline (RBED) scheduling algorithm. RBED is a generalization of EDF, enabling integrated scheduling of hard real-time, soft real-time (minimum rate with proportional sharing of slack), and best effort tasks [19]. The RAD model also clearly reveals how to apply various optimizing heuristics within the bounds of guarantees. This is exemplified in the Fahrrad disk scheduler's use of a horizon line to reorder requests [124]. It should also enable the RUN multiprocessor algorithm, described in § 2.3 to effectively handle more general task models.

RUN

The Reduction to UNiprocessor (RUN) [131, 132, 91] algorithm was first described by Regnier, et al. as a novel and elegant solution to real-time multiprocessor scheduling. The first practical implementation of RUN created by Compagnin, et al. [29], both verified the simulation results and showed that it can be efficiently implemented on top of standard operating system primitives. While RUN is now the proven best solution for scheduling fixed task sets with fixed rate on multiprocessors, further work remains to make it practical for common workloads and on resources other than CPUs.

The RUN algorithm takes advantage of two features of highly loaded systems. First, a busy system has little idle time, so it makes more sense to solve the dual schedule (i.e. when tasks are not running). The critical nature of idle time was first noticed by Levin, et al. when they created a theory explaining all previous optimal [1] multiprocessor algorithms [90, 89]. Second, a highly loaded system will generally have many small tasks that can be packed together and treated as one task. Both steps simplify the problem at hand, and when they are combined recursively they produce a reduction tree that partitions the scheduling problem amongst the packings and bounds the interactions between packings.

In a system with N processes and M processors where each process requires a fixed share of a processor, packing shrinks the size of N and taking the dual of the system reduces the size of M whenever $N < 2M$ By alternating packing and dual operations, Regnier, et al. [131] showed that they were able to reduce the difficult multiprocessor problem down to a simple uniprocessor problem. The approach is revolutionary because it is simple and provably more efficient in terms of context switches and migrations than any previous approach (e.g. the family of proportionate fairness and deadline partitioning scheduling algorithms).

Figure 2.3 shows the specification of 22 tasks randomly generated using the Randfixedsum algorithm [44], with hard real-time reservations of 6-98% utilizations and periods ranging 15-910 milliseconds. RUN was reimplemented in Python and used to generate their schedule on eight resources. The program also draws the first 100 milliseconds of the schedule in Figure 2.4 in order to aid in debugging and to help develop an intuition for scheduling patterns and densities of events. The tiny ticks scattered along the top of each row of the resource schedule are preemptions, and the thin black arrows leading from one row to the next indicate task migrations.

Considering that the 22 tasks keep eight resources 100% utilized, there are remarkably

[1]Optimal in the sense that an algorithm will produce a valid schedule for any task set that is feasible.

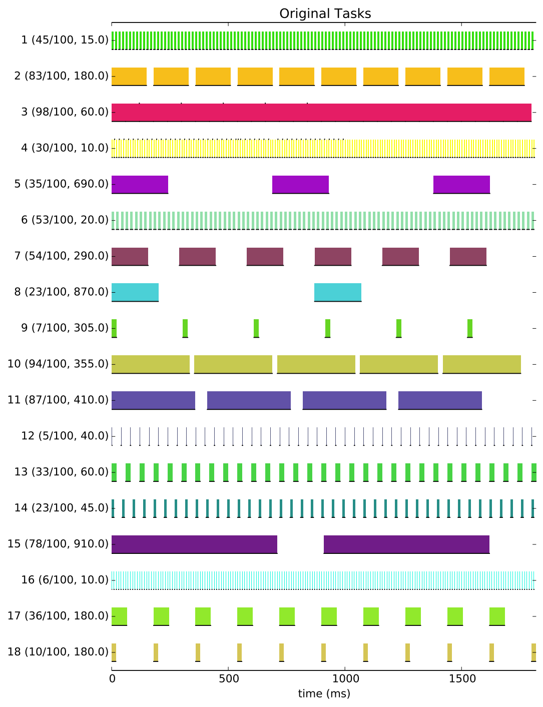

Figure 2.3: 18 Tasks with Hard Real-time reservations.

few migrations—an average of 3.205 migrations per task per second. In fact, most tasks of the task migrate little, if at all. Table 2.3 makes it clear that the bulk of the preemptions and migrations happen to the tasks with the both the highest utilization and longest periods: 10, 11, 15. Furthermore, this schedule was generated with a naive unordered First Fit policy. Other heuristics could minimize preemptions by packing harmonic or high frequency tasks together first.

See Appendix B for an example of how RUN is able to find proper subsets of tasks which can be scheduled in isolation from all the other tasks in a set. And see 5.3 for discussion about RUN heuristics and application to networking.

Table 2.1: Most tasks rarely incur migration penalties in a RUN-generated schedule!

Task ID	Avg. Preemptions/Job	Avg. Migrations/Job
1	4.507	0.000
2	145.400	0.000
3	60.375	0.938
4	2.770	0.000
5	157.000	0.000
6	7.980	0.000
7	158.000	0.000
8	133.000	0.000
9	18.333	0.000
10	354.500	4.000
11	390.000	9.500
12	2.720	1.520
13	22.471	1.471
14	15.682	2.591
15	827.000	36.000
16	1.200	1.680
17	61.000	0.000
18	15.000	0.000

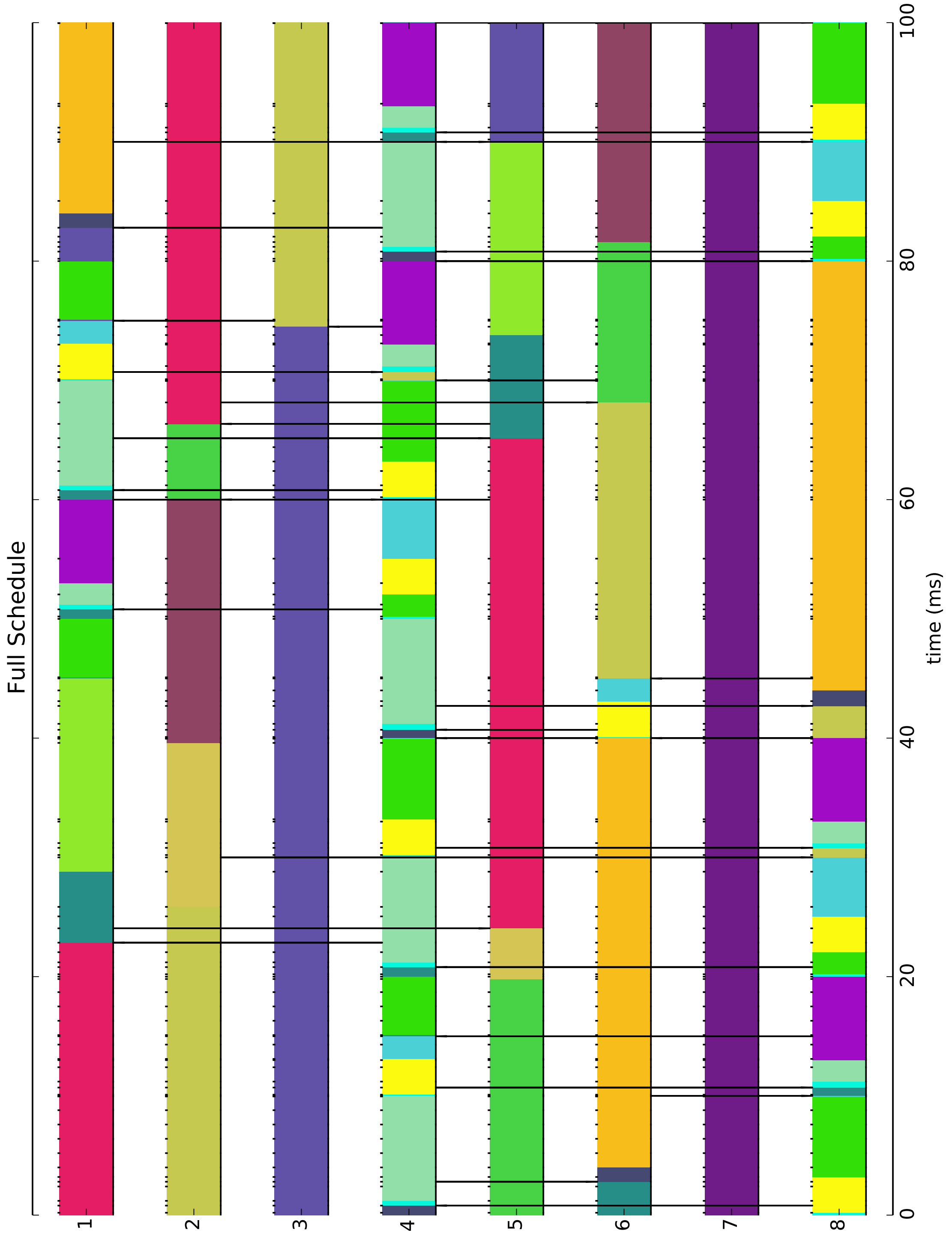

Figure 2.4: RUN - 18 Tasks on Eight Resources.

Chapter 3

Experimentation Strategies

After serving nine years on the SIGCOMM Program Committee, C. Partridge wrote "How to Increase the Chances Your Paper is Accepted at ACM SIGCOMM" in 1989 [120], and he had this to say about TCP papers:

> TCP performance is a well-trod ground and so the standards for getting a TCP paper accepted are now quite high.
>
> To be accepted at SIGCOMM, a TCP performance paper should demonstrate that the proposed performance improvements have been thoroughly tested. For instance, any changes to TCP flow control should be tested over heavily loaded multi-hop topologies with cross traffic. Furthermore, the analysis should show not only that the enhanced TCP performs better, but also show the effects of the enhanced TCP on non-enhanced traffic. Note that TCP performance papers are often measurement papers and so . . .
>
> Writing good measurement papers is very hard. It requires careful network monitoring, using good statistical techniques. A good monitoring paper should explain how the data was taken, why the data is believable (i.e., what statistical measures were taken to ensure the data was sound), and then analyze the data carefully with good charts and graphs and discussion that indicates the data was thoroughly analyzed and contemplated. Probably more than any other type of paper, measurement papers benefit from extra time for the author to refine the paper. Start writing early!

Writing TCP papers has not gotten easier in the last two decades. Experienced engi-

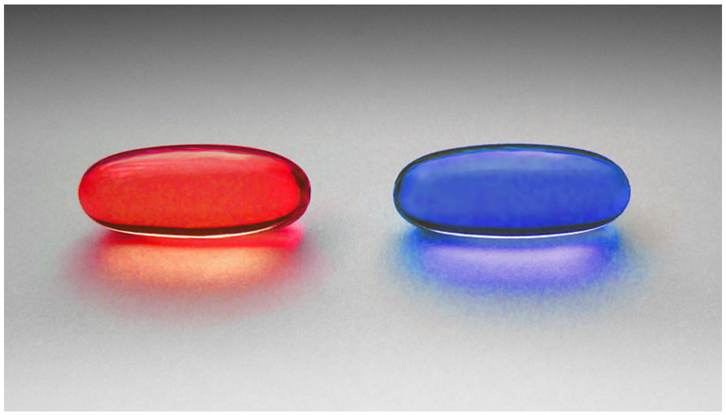

Figure 3.1: **Fortunately, we are not limited to two choices.**
Red and blue pill as in the Matrix. W. Carter. 2014. Creative Commons Attribution-Share Alike 4.0 International.

neers tend to discount the results in academic papers unless they can test working code. Best practices for evaluating congestion control, simulation, and testbed scenarios have been written by the IETF [48, 50]. However, there is no real consensus on how or what should be measured, except that researchers should *use as many methods and metrics to compare as is possible*.

The following sections describe some of the strengths and weaknesses of different strategies for demonstrating the effectiveness of network protocols. Although the sections are split up into a blue pill and red pill, as seen in Figure 3.1 [157], that is not to say that it is strictly an either/or choice.

3.1 Modeling

Mathematical modeling allows a researcher to set aside most implementation details in order to determine if an idea is otherwise feasible. A feasible congestion control or AQM should achieve high link utilization by not allowing queues to totally drain, while rarely, if ever, overflowing. Furthermore, the process of achieving good queue behavior should be globally stable even if it must naturally lead to local oscillations in queue sizes.

Another reason models are important is that they can correct misguided configuration. For instance, it became common over the decades for network hardware to gain bigger memories for buffers to hold large bandwidth-delay products. However, Raina [129] and others have shown through models and stability analysis that proper functioning of TCP requires small buffers. Eventually, that knowledge gained from modeling helped lead to a concerted effort to the de-bloat some network stacks in the real world, and it has made a significant positive difference in the behavior of the Internet, although it is an ongoing process.

Possible models include many simplifying assumptions, such as: the system is in a steady state, all flows have a single RTT, or all flows can be represented by a single average flow size. These assumptions are used to make equations more understandable and efficient to solve. And while the insights gained in this way can be invaluable for proving useful characteristics like average case bounds, there is a real danger they will not capture important real-world problems (e.g. the long tail of delay distributions), especially when the features are caused by relatively rare events, which assumptions can hide.

The simplest network models use renewal theory [101] to reason about the sawtooth pattern of TCP's window size. Assuming a steady state like Congestion Avoidance, it is straightforward to use algebraic equations to estimate a bottleneck queue's maximum size using the amplitude and periods of oscillations of a TCP window. This sort of analysis is adequate to provide basic insight for a small number of synchronized flows.

Variable	Description
A	Arrival process
S	Service time distribution
c	number of servers
K	capacity of queue
N	size of population of jobs
D	queueing discipline

Figure 3.2: Markov model variables.

The most common models in network analysis are based on Erlang's queueing theory [18, 46] combined with control theory [102]. The models are described by the notation, $A/S/c/K/N/D$, whose variable descriptions can be found in Figure 3.2. For example, consider the common Markovian / Markovian / 1 / ∞ / ∞ / FIFO model, often abbreviated as M/M/1. It is a stochastic process that describes the length of a queue in which jobs arrive according to a Poisson process, service times have an exponential distribution, with a single server, an infinite queue, an infinite number of jobs, and a First In First Out (FIFO) queueing discipline.

A Markov Chain is a memoryless stochastic process in which each state only depends on the previous state (or n previous states for an n-order Markov Chain) and a set of probabilities. Padhye, et al. proposed a model [115] for TCP Reno included a state space with window size, packet loss per round, timeout state, RTT, number of packets transmitted, and number of packets received. The evidence for the accuracy of the Poisson model for the Internet is well established [163]. However, it is likely inappropriate for data centers and HPC systems for two reasons. First, TCP requires segmentation offloading in order to reach high speeds and pacing is not always supported, so at the very least, a batch-Poisson would be a better fit [163]. Second, since the communication patterns in those settings can include collective operations such as reductions, the flows associated with those patterns can not be assumed to be stochastically independent.

Queueing disciplines (qdiscs) include, but are not limited to: FIFO, Shortest Job First (SJF), and variants of Round-robin (Processor Sharing [1, 80], Weighted RR, Deficit RR, etc.). Note the simplicity of these disciplines is partly driven by the desire to implement them in hardware. One of the key practical considerations regarding the implementation of queuing disciplines is the economic limit on the number of hardware queues that can be provided. This means qdiscs support a small number of priority classes. Of course, software queueing disciplines can be more intricate, but higher complexity decreases the line rates they can support.

Fluid models [94, 5] are continuous, deterministic alternatives to the discrete, stochastic queueing models described above. Instead of chains of states, a set of differential equations are used to describe the rate of change of variables of interest. These equations look like those used to describe physical systems, circuits, etc. Unlike Markov chains, the differential equations can depend on any previous state of the system. Thus, they are called time-delayed differential equations. In software used to solve the equations, the time parameter passed into functions cannot themselves be functions. This has to do with potentially endless recursion or the difficulty in solving such a system numerically.

All models must be analyzed for convergence or bounded stability [5, 6, 129, 128, 163]. More specifically, control theory dictates that closed loop systems (i.e. those utilizing feedback) need to be stable. Many control theoretic tools for studying stability have been developed, including Poincairé maps, and are used to find fixed stable points for edge cases or limit cycles more generally.

A researcher might wonder whether it is best to use stochastic queuing theory or fluid models, and it depends. Those who find it natural to reason in terms of state machines and probabilities may prefer queueing theory using Markov chains. Its semantics match TCP's policy of making decisions once every RTT, TCP implementations include many state machines, and the Poisson distribution has proven itself with regard to the Internet. However, the statistical nature of the model makes it difficult to increase the fidelity of a model. And as pointed out above, the Poisson distribution (while not required) may not be appropriate for some networks.

Fluid models, on the other hand, may be more intuitive in general since everyone has some experience with water and rates. And the faster a packet network is, the more it resembles fluid. In addition to being continuous, differential equations can also express dependencies at multiple time scales (e.g. per round) using time delays in the variable passed into its functions. And while fluid models are deterministic, they can be used in conjunction with probabilistic functions (e.g. packet dropping or marking). Adding detail to fluid models (e.g. preventing windows from decreasing below two packets or forcing a queue size to remain positive) can cause problems with solver software in which the numerical solution becomes unstable.

3.2 Implementation

When it comes to implementing an idea there is a huge spectrum in the level of detail and flexibility which is helpful for a given experiment. Increasingly, simulation, emulation, and the real-world can be mixed and matched, providing realism or flexibility where it is most needed. Simulators themselves may accurately represent hardware or they may represent everything at a much higher level and more closely resemble one of the mathematical models discussed previously. Emulators and emulated environments use code from real operating systems, but they fake characteristics of hardware or topologies. In the end, there is really only one way to know how well an idea works, and that is to try and measure it on an actual system.

Simulation

Many research papers include simulation results because they fill an important gap between modeling and a full blown implementation. Where modeling can be used to evaluate the fundamental feasibility and explore the mathematical limits of an idea, simulation's primary use case is validating the protocols and algorithms necessary to implement the mathematical model.

Simulations contain more detail than models, but are generally easier to program and more flexible than real-world implementations. The amount of detail can vary from cycle accurate hardware like BookSim [75] to the more common protocol level simulators. The latter include advantages such as the ability to configure almost any topology and allowing the use of floating point arithmetic.

The most venerable simulator is NS-2 [103], although there is an effort to replace it with something more modern NS-3 [64]. There are other simulators that have found favor in some subcommunities: Omnet++ [155], BookSim [75], and ROSS [26]. Omnet++ is best if Infiniband [97] is of primary interest. BookSim implements general credit-flow based fabrics and is cycle accurate, but high levels of detail such as cycle accuracy limit the size of a simulated experiment. ROSS is also general, but more high level, and can simulate larger networks (e.g.

a million node dragonfly fabric).

There are multiple downsides to using a simulator, including: the need to debug the simulator itself, lack of detail (especially of switch buffers), lack of performance, and the need to validate the simulator against real-world results. However, assuming the simulator is continually validated against real-world results [11, 49, 67, 109], its ability to help extrapolate beyond a simple laboratory network while not endangering any real traffic is valuable.

Emulation

Emulated environments [2, 60, 159, 167] leverage the code of real operating systems as well as virtualization, container, traffic shaping, and Software Defined Networking technology in order to fake larger numbers of hosts and more diverse topologies than a testbed actually possesses. The ability to develop and test distributed systems on a laptop is a great increase in productivity. However, emulation must sacrifice speed in order to scale out, and attempting to fake too many hosts or too much bandwidth decreases the fidelity of experiments.

One of the easiest to use and highest fidelity emulators is Mininet [60], which wraps a Python interface around Linux containers, queueing disciplines, and Open Virtual Switch [45]. The containers provide multiple hosts with the lightweight virtualization of namespaces and resource controllers. The characteristics of the network are defined by queueing disciplines—generally a Token Bucket Filter shapes bandwidth and netem injects delay, packet loss, packet duplication, and packet re-ordering.

Real-world

Some real-world considerations were previously discussed in § 1.3, but it was by no means exhaustive. Most network stacks are implemented in kernel-space in order to allow the network hardware to be pushed as hard as possible. Of course, this means algorithms must perform calculations without using floating-point arithmetic. Also, cross-layer information sharing such as providing TCP congestion control with transmission timestamps of packets, or giving RTTs of flows to qdiscs may be cumbersome or impossible.

Operating systems may provide an API for providing new functionality. In addition to TCP congestion control modules, Linux also provides an API for pluggable qdiscs and resource control groups (cgroups). Linux allots 64 bytes for custom congestion control per-connection variables in addition to many commonly used variables provided by the general socket structure. The hooks provided for congestion control are limited, but they have been extended—most recently to support DCTCP's use of ECN. A system-wide congestion control algorithm can be chosen, or more recently and also for DCTCP, it can be set per route. Congestion control behavior can be tuned by load-time module parameters, run-time procfs and sysfs tunables, and per socket options.

Link, socket, and qdisc statistics are made available through a handful of tools such as ip, netstat, ss, tc. If more detailed information is desired, packet traces can be recorded and analyzed with tcpdump, tcptrace [114], and wireshark [113]. Linux also provides a dynamic tcp_probe module that records per connection trace information such as the size of the congestion window and the current smoothed RTT. Trace events can be added to create an even more efficient and flexible capability than what tcp_probe provides.

Note well that modifying the Linux kernel involves a big learning curve, and that modifying the heart of its TCP stack is made difficult by the active work of professional developers. As an example, the original DCTCP code did not use the congestion control module API because it needed hooks that were not present. This made it extremely difficult to use as a basis for further work on more recent kernels that included many important enhancements like microsecond RTTs. Fortunately, the module API was enhanced and DCTCP became a clean, stable module in the upstream kernel.

Applications are needed to test network stacks, and in addition to microbenchmarks like iperf [152], more complex benchmarks resembling common industry use cases, such as the Yahoo Cloud Services Benchmark [30] have been created. For very specific TCP tests, one might create something at the packet level with Packetdrill [25]. Or if a higher level, reproducible test is required, then the coNCePTuaL [116, 117, 118] domain specific language can help.

The HPC community uses many microbenchmarks to compare high speed networks and validate performance. Among these microbenchmarks, System Confidence [139, 140, 141] stands out because it looks at the whole distribution of delay measurements instead of just a few, such as minimum, maximum, and average.

Part II

Toward Efficient Performance Guarantees

Chapter 4

Ambidextrous Congestion Control

In this chapter I present TCP Inigo, which effectively addresses all five network design goals stated in chapter 2 (latency, bandwidth, fairness, deployment, and administration), whereas other solutions neglect some goals or perform worse. In particular, Inigo out-performs the current widely-accepted solution, Data Center TCP [4, 13] (DCTCP), described in Section § 2.1.

Inigo includes two primary contributions to the state-of-the art. First, a sender-only modification inspired by DCTCP but uses Round-trip-times (RTTs) to mimic Explicit Congestion Notification (ECN). Second, a receiver-only modification that similarly mimics ECN with differences in One Way Delay (OWD). Inigo's worst-case performance better is than DCTCP's because Inigo's sender and receiver modifications are delay-based and can operate independently or together. The first row in Figure 4.1 represents a best-case administrative scenario, such as a data center, in which every component in the network is under a single authority and can be upgraded and configured coherently. The second and third rows illustrate worst-case

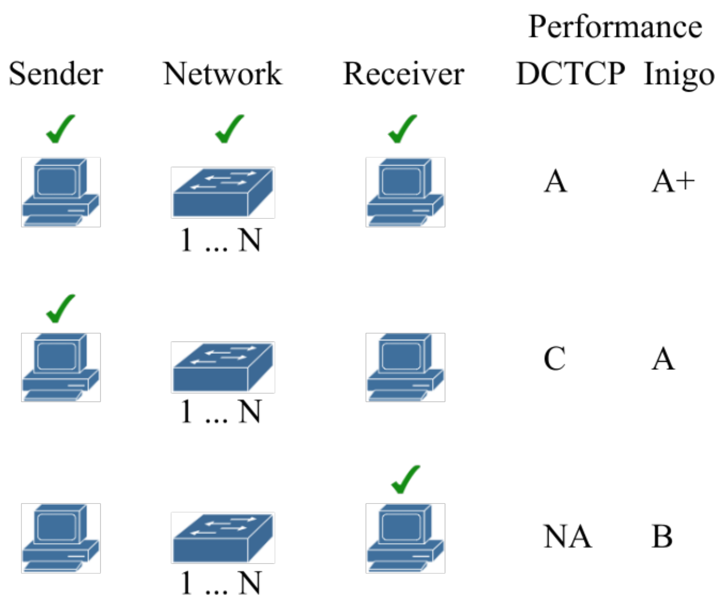

Figure 4.1: **Inigo's latencies are up to** 1.3× **better than DCTCP, the best deployable solution, when all components of a network are properly configured (green check). Inigo's sender-only mode is up** 42× **better than DCTCP's corresponding failure mode, according to fairness, bandwidth, and latency indices; and Inigo can also offer improvements when only the receiver is configured.** Letter grades are relative to a C for Reno-level performance.

scenarios in which every part of the network is owned by a different entity only one end-host can be modified.

In chapter 2 I overviewed many specific techniques that improve performance in controlled environments, such as data centers, where all components (i.e. end-hosts and middleboxes) can be modified. But the effectiveness of those same techniques significantly degrades in uncontrolled environments when interacting with unmodified components across network borders. Established systems with multiple owners and long histories, like most networks, favor incremental evolution over dramatic change because upgrades are costly and decisions to upgrade are made independently. Even data centers and supercomputers, which are prime examples of scenarios where new network technologies can be leveraged, must communicate frequently with external systems, and end-to-end arguments [14, 136] should be considered. There is considerable practical value in being able to improve network performance while minimizing the effort needed to deploy and maintain the changes. Figure 4.1 shows just a few examples of the many failure modes a solution to congestion should handle.

This chapter focuses on ways to improve network performance for applications built on TCP. While other protocols exist, TCP is the most widely used. Furthermore, the techniques Inigo uses can be re-applied to other protocols that track RTTs or use timestamps at end-hosts.

The rest of this chapter is organized as follows: First, the Inigo sender-side (§ 4.1) and receiver-side (§ 4.2) algorithms. I then evaluate Inigo (§ 4.4) and demonstrate the effectiveness of both techniques independently and combined. Finally, I describe the availability of TCP Inigo code and Mininet experiments (§ 4.5).

4.1 TCP Inigo Sender

TCP Inigo is composed of two independent techniques. The first is a sender-side only modification that uses TCP RTT measurements. The second uses differences in One Way Delays (OWDs) on the receiver-side. Both follow in the footsteps of DCTCP in using a congestion ratio, a measure of the extent of congestion, in order to proportionally adjust the congestion window.

The Inigo sender-side congestion control module uses the Linux kernel's pluggable congest control interface, and has been made possible by several developments in the Linux kernel since the original DCTCP paper [4]. Internal buffer bloat and delay variability were much improved with features such as Byte Queue Limits [33], TCP Small Queues [34], and TCP Segmentation Offload (TSO) sizing and pacing [36]. Importantly, the units of the sender's RTT measurement changed from milliseconds to microseconds in 2014 [42]. These changes improve the behavior of every TCP, but they are vital for Inigo's delay-based algorithms.

RTTs are readily available measurements, but TCP's timestamps are taken several layers and queues above the hardware. As such, they include the variability of the host operating systems and not just the network delay due to congestion. If the goal is to minimize the end-to-end delay variability observed by the application layer, then the fact that the TCP RTT includes delays due to Operating System (OS) buffers and network buffers may be an advantage. Also, RTT measurements do not combine independent signals in a way that might indicate more congestion than is actually present. In contrast, ECN marking at switches is done independently, so one could envision an unlikely scenario where a series of switches each experienced minor congestion at different times, causing the majority of a flow's packets to be marked. Since location and presence of congestion is combined in a marking, the end hosts cannot tell for certain how bad the congestion is.

RTT measurements are noisy, so reacting to individual measurements results in unpredictable behavior. That is why many algorithms use some sort of smoothing, but given the dynamic range of RTTs this can often prevent quick responses to changing conditions.

4.1.1 RTT Congestion Ratio

DCTCP's congestion ratio, $\alpha_{ECN} = \frac{bytes_{ECN}}{bytes_{total}}$, is driven by an ECN marking threshold designed to balance conflicting requirements—to fully utilize bandwidth while keeping latency low. In this section I will explain how I leverage the same reasoning that led to DCTCP's ECN marking threshold to justify the RTT threshold that drives Inigo's congestion ratio, $\alpha_{RTT} = \frac{RTTs_{late}}{RTTs_{observed}}$.

$$K \approx 0.17Cd \tag{4.1}$$

Alizadeh, et al. [5] derived equation (4.1), in which C and d denote the bottleneck capacity (packets/second) and RTT delay (seconds), giving a threshold of K (packets). This threshold is 2.7% larger than their original, and is based on a fluid model of DCTCP that is more accurate than their previous sawtooth model. Intuitively, this threshold says that the queue should absorb bursts of up to 17% of the bandwidth-delay product before it starts telling flows to slow down.

Inigo uses *late RTTs* in the same way that DCTCP uses ECN markings to calculate and respond to the extent of congestion. Increases in RTTs are generally due to congestion in current systems where the OS is not CPU bound and it keeps its internal bufferbloat under control.

$$d_{thresh} = K/C \tag{4.2}$$

$$d_{thresh} \approx 0.17Cd/C = 0.17d \tag{4.3}$$

Since the RTT signal arrives at the same frequency as ECN markings (i.e. every

ACK), and since TCP RTTs correspond to increased queuing, it can be assumed that the recommended DCTCP threshold is valid for defining what makes an RTT late. The corresponding queuing delay threshold, d_{thresh}, is simply K divided by the bottleneck capacity C. Substituting equation (4.1) into (4.2) gives (4.3). See Section 4.3 for more detail about the mathematics.

Algorithm 1 Inigo RTT Congestion Marking.

```
for each ACK do
    if $RTT_{min} = 0 \vee RTT < RTT_{min}$ then
        $RTT_{min} \leftarrow RTT$
    end if
    $RTTs_{observed} \leftarrow RTTs_{observed} + 1$
    if $RTT \geq RTT_{min} + d_{thresh}$ then
        $RTTs_{late} \leftarrow RTTs_{late} + 1$
    end if
end for
```

Algorithm 2 Inigo RTT Congestion Ratio.

```
for every window do
    $F \leftarrow RTTs_{late}/RTTs_{observed}$
    $\alpha_{RTT} \leftarrow (1-g) \times \alpha_{RTT} + g \times F$
    $RTTs_{observed} \leftarrow 0$
    $RTTs_{late} \leftarrow 0$
end for
```

Algorithm 2 calculates the congestion ratio α_{RTT} using the RTTs marked late by algorithm 1. It is nearly identical to the approach taken in DCTCP, where a fraction F of ECN-marked bytes is tracked during a window and used to update the exponential weighted moving average of the DCTCP congestion ratio α_{ECN}.

There are some subtle implications of using a congestion ratio driven by RTT observations instead of ECN marked bytes. Most importantly, RTTs allow a sender-only modification. Also, Inigo does not need to compensate for delayed ACKs since they only reduce the number of observations driving the congestion ratio and do not distort the magnitude of the congestion signal, as they do for ECN markings. Similarly, the number of RTT measurements can be further reduced when the lower layers of the network stack or hardware implement segmentation offloading, which aggregates TCP's segments into larger chunks before sending them out onto the network.

4.1.2 Slow Start

There are many variations of TCP Slow Start, in which the initial rate of transmission rapidly increases, usually via window doubling. This phase is necessary because TCP does not know the speed of the network. DCTCP shows that the standard method of doubling the window size every RTT can quickly achieve full throughput while keeping queue occupancy low with the help of ECN marking on switches.

Interestingly, the DCTCP Internet-Draft [13] does not specify Slow Start behavior, and the 4.4 Linux DCTCP implementation appears to overshoot the ideal congestion window and cause unnecessarily high RTTs. For this reason, when using ECN, Inigo exits Slow Start immediately upon seeing the first ACK in a window with a CE marking. Other than that, Inigo behaves the same as DCTCP when ECN is available. Matching the efficiency of an ECN-driven Slow Start exit using only delay is challenging, with HyStart [58] probably the most successful example of the technique.

Linux kernel implementations of HyStart in TCP CUBIC [59] and CDG [62] contain some experimental changes. Both variants detect congestion during Slow Start with ACK trains and when a late RTT is observed. They take the minimum of the first seven RTT samples and exit Slow Start immediately upon receiving one late RTT. The delay threshold, d_{thresh}, used by CUBIC in Linux 4.2.0 is one eighth the minimum RTT, bounded between 32 and 128 milliseconds. CDG's d_{thresh} is also one eighth the minimum RTT, but it is calculated with a microsecond clock and has an upper bound of 125 microseconds.

HyStart was designed to find an early, safe exit point to enter CUBIC's aggressive Congestion Avoidance phase. But the threshold was increased to one eighth in 2014 because one sixteenth was too sensitive. That over-sensitivity was one of the reasons Linux networking maintainer David Miller recommended disabling HyStart by default [104]. Interestingly, CUBIC's new threshold is within 1.8% of the initially recommended threshold for DCTCP [4].

Algorithm 3 Inigo Slow Start.

```
for every ACK do
    if $cwnd \leq ssthresh \wedge samples \geq 8$ then
        $F \leftarrow RTTs_{late}/RTTs_{observed}$
        $\alpha_{RTT} \leftarrow (1-g) \times \alpha_{RTT} + g \times F$
        if $\alpha_{RTT} > 0$ then
            $ssthresh \leftarrow cwnd - cwnd\_cnt \times \alpha/2$
        end if
    end if
end for
```

Inigo sets aside HyStart's ACK train heuristic, exiting Slow Start only upon seeing an RTT that exceeds $RTT_{min} + d_{thresh}$, as in algorithm 2. Similar to HyStart, Inigo requires a minimum number of observations to initialize RTT_{min}. But instead of simply exiting Slow Start by setting the Slow Start threshold *ssthresh* to the current congestion window *cwnd*, Inigo uses the congestion ratio to decrease the congestion window. Algorithm 3 uses $RTTs_{observed}$ and $RTTs_{late}$ from algorithm 2. If the congestion ratio is non-zero once eight RTTs are observed, then it reduces *cwnd* by the congestion ratio similarly to algorithm 5 in § 4.1.3. Finally it assigns

$ssthresh = cwnd$.

4.1.3 Congestion Avoidance and Response

Alizadeh, et al. proposed an RTT-fairness enhancement [5], in which DCTCP would respond to congestion every ACK. The improvement counter-acts the typical TCP behavior of flows with longer RTTs growing more slowly than flows with short RTTs by causing the latter to respond to congestion more rapidly. Conventional wisdom is for a congestion response to only occur once per RTT in order to see the effect of the response, but it is reasonable to use a quicker response when the sum of the adjustments are designed to approximate the once-per-RTT response.

As an analogy, passengers in a vehicle appreciate a driver who makes micro-adjustments instead of slamming on the breaks or the accelerator, even if the average speed is the same. While packets do not care about sudden changes in acceleration, a person feels it in the form of long tail latencies. Fortunately, a TCP that makes sub-window adjustment should be able to avoid over-steering.

This is about more than RTT-fairness. It affects convergence time for long lived flows with the same RTT starting at different times. Flows beginning earlier will have a larger window and respond more slowly than newer flows. And sub-window adjustments should allow short-lived flows to enter and leave with smaller perturbations to long-lived flows.

Unfortunately, DCTCP's RTT-fairness update algorithm needs to adjust the window by a fraction of a packet, and implementations of DCTCP in kernel-space require the use of whole integer variables. Alternatively, the sender's window could be tracked in bytes like the

Algorithm 4 DCTCP RTT-fairness Congestion Response.

```
for every ACK do
    $\alpha \leftarrow (1-g)\times\alpha + g\times ECN$
    $W \leftarrow W + \begin{cases} 1/W & \text{if ECN } = 0 \\ 1/W - \alpha/2 & \text{if ECN } = 1 \end{cases}$
end for
```

receiver's window, allowing fine-grained changes to accumulate. Or the window mechanism might be altered to allow sending at a different frequency, as has been proposed for scaling to small RTTs [23]. However, both would require either modifying the whole TCP stack or adding extra variables to the TCP congestion structure for private per-socket data. A sub-window approach to RTT-fairness is simpler to implement and requires fewer variables.

As an example of the problem, if $W = 200$ and $\alpha = 300/1024$, then upon seeing and ECN marking $W \leftarrow 100 + 1/200 - 150/1024 \approx 199.86$. Integer arithmetic would result in $W \leftarrow 200$, and if α remains relatively constant, then $W \leftarrow 200$ after a window of ACKs. On the other hand, the original once per RTT response would yield $W \leftarrow 171$.

Algorithm 5 Inigo RTT-fairness Congestion Response.

```
for every $W_{sub}$ ACKs, where $1 < W_{sub} < W$ do
    if $\alpha > 0$ then
        $W \leftarrow W - W_{sub}\alpha/2$
    end if
end for
for every window do
    $W \leftarrow W + 2$
end for
```

Linux implements Congestion Avoidance with a counter `snd_cwnd_cnt`, which is incremented by the number of ACKed packets until `snd_cwnd_cnt` reaches `snd_cwnd` and `snd_cwnd` is incremented by one. Similarly, `snd_cwnd` can be decremented by a fractional packet by responding every N ACKs as in algorithm 5. I observed (§ 4.4) that a sub-window response sometimes backs off a little too much, and I found that Congestion Avoidance of $W \leftarrow W + 2/W$ ensured better link utilization. It does not significantly alter the DCTCP fluid model analysis [5] since its impact on the average per-flow window size is small. At most, it causes flows to operate more often in the primary operating regime.

Also, since two is much smaller than the largest sensible window for an unloaded path, incrementing by two satisfies the additive increase policy stated by Jacobsen [70]. In appendix D Jacobson elaborated on the choice of the 1-packet increase, saying that the goal was to limit the average loss rate caused by gently probing for bandwidth to <1%. While the 1-packet increase was on the aggressive side for Arpanet with its 4-5 packet largest sensible window, later TCPs such as CUBIC showed the need for more aggressive probing on long, fat networks. Furthermore, Inigo can afford a more assertive probe for bandwidth since it uses delay to keep bottleneck buffers low and is unlikely to cause packet loss.

The frequency of response must be balanced with sensitivity to the congestion ratio, α_{min}, calculated with equation (4.4). For instance, a response interval $W_{sub} = 20$ and a maximum congestion ratio $\alpha_{max} = 1024$ would be able to adjust the window in response to $\alpha \geq 104$. The smallest sub-window that could make any adjustments below $\alpha = \alpha_{max}$ would be $W_{sub} = 3$, with $\alpha \geq 684$. Note that increasing the scaling factor of α does not significantly improve sensitivity.

$$\alpha_{min} \leftarrow \lceil 2\alpha_{max}/W_{sub} \rceil + \begin{cases} 1 & \text{if} \alpha_{min} \bmod 2 = 1 \\ 0 & \text{if} \alpha_{min} \bmod 2 = 0 \end{cases} \tag{4.4}$$

4.2 TCP Inigo Receiver

The second, separate congestion control that makes up TCP Inigo is a receiver-side only modification that detects congestion by monitoring the accumulation of differences in One Way Delays (OWDs). The receiver controls congestion in a style similar to DCTCP via the receive window. Theoretical benefits of receiver-side congestion control include: (1) switch configuration is unnecessary, (2) TCPs senders are forced to use a maximum window size calculated by a single algorithm instead of a variety of algorithms, and (3) mistaken maximum windows sizes due to ACK loss are corrected upon next ACK.

4.2.1 Relative Forward Delay

Relative Forward Delay (RFD) was defined as the difference of OWDs by Parsa, et al. [119] when they used it in the congestion control of TCP Santa Cruz. Example RFDs are shown in Figure 4.2, where S is the delta between send timestamps, R is the difference between receive timestamps, and RFD is D^F, the delta between any pair of S and R. Calculating RFD does not require clock synchronization, but it does require stable clocks of the same resolution.

The simulator implementation of TCP Santa Cruz required modifications to both the sender and receiver, and results showed promise, but it was never tested on real networks. This was evidently due in part to TCP Santa Cruz's reliance on an experimental TCP option, unlike this work.

Others have also used RFD to reason about bandwidth and congestion. Pathload [72] used packet trains to probe the available bandwidth of a network. HyStart [58] found Pathload's

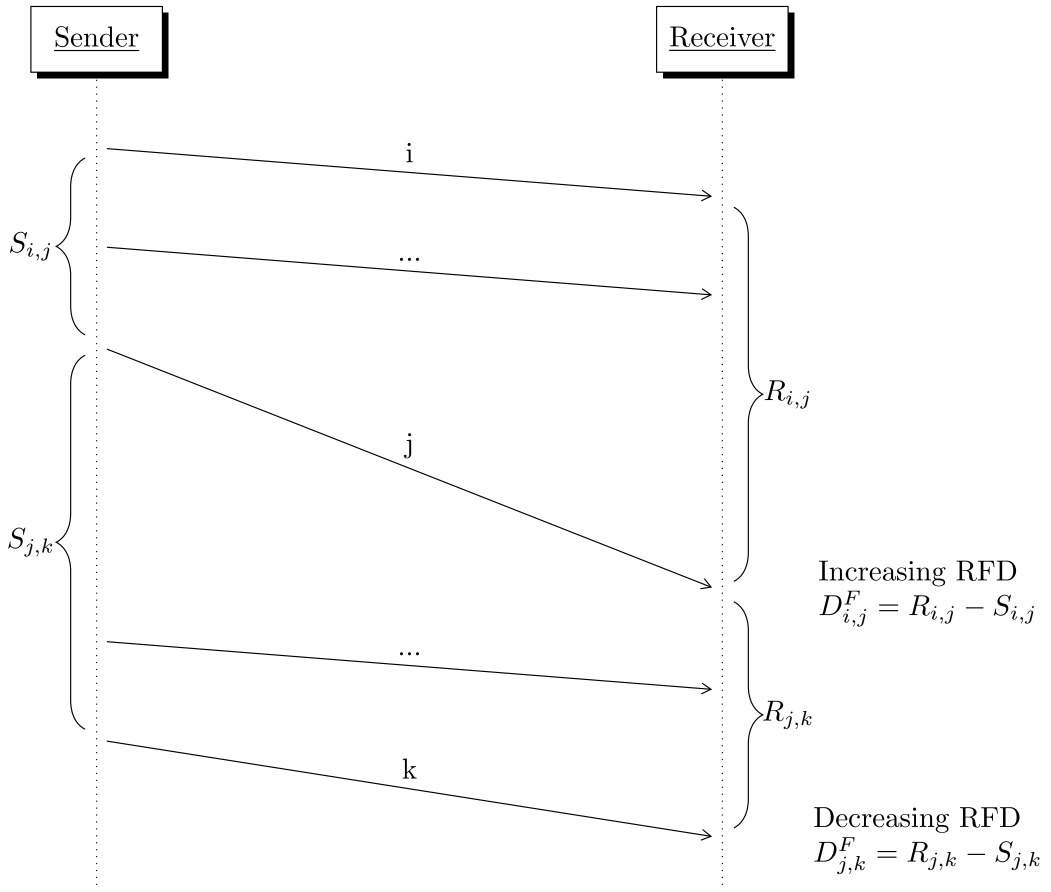

Figure 4.2: Examples of Increasing and Decreasing Relative Forward Delay Measurements.

techniques unsuitable for integration with TCP, but used them as inspiration for its ACK-train heuristic used as a signal to exit Slow Start.

The receiving side of TCP can use timestamps to calculate RFD, but unfortunately the existing TCP timestamps are too coarse-grained for data centers. RFC 1323 and the updated RFC 7323 [71, 15] both recommend a timestamp resolution between 1 millisecond and 1 second per tick, whereas data center RTTs are measured in microseconds. Similarly unfortunate, the receiver only has an estimate of the RTT in milliseconds, and it appears to be less than the actual RTT in our experiments. This will tend to magnify the measurement of congestion since the minimum RTT is used to define d_{thresh}.

In order to accommodate both Internet and data center latencies, TCP could keep track of minimum $S_{i,j}$ and $R_{i,j}$ for consecutive packets. If RTTs and timestamp deltas for both sender and receiver are less than or equal to one millisecond, then TCP could swap out the millisecond timestamp operations for microsecond versions. Relying on both sides being able to increase timestamp resolution would be the sort of change that would inhibit adoption. Also, a side effect of increasing the timestamp resolution would be to reduce opportunities for Generic Receive Offload [32].

Algorithm 6 shows how the running RFD total of D^F_{total} and d_{thresh} based on RTT_{min} can be used to mark bytes as late, similarly to DCTCP with ECN and Inigo's sender with RTTs. The receiver keeps track of the running sum of RFD using TCP timestamp value, ($tsval$), and timestamp echo reply ($tsecr$) fields from the header. Inigo keeps earlier timestamps until the clock resolution allows differences between sends and receives to be detected. If the total RFD becomes negative, then that means the measurements started taking place when congestion was

Algorithm 6 RFD Congestion Marking.

```
for each packet received do
    if $RTT_{min} = 0 \vee RTT < RTT_{min}$ then
        $RTT_{min} \leftarrow RTT$
    end if
    $bytes_{total} \leftarrow bytes_{pkt}$
    $S_{i,j} \leftarrow tsval_j - tsval_i$
    $R_{i,j} \leftarrow tsecr_j - tsecr_i$
    if $S_{i,j} = 0 \wedge R_{i,j} = 0$ then return
    end if                                              ▷ low clock resolution
    $D^F_{i,j} \leftarrow R_{i,j} - S_{i,j}$
    $D^F_{total} \leftarrow max(0, D^F_{total} + D^F_{i,j})$
    if $D^F_{total} \geq d_{thresh}$ then
        $bytes_{late} \leftarrow bytes_{pkt}$
    end if
    $tsval_i \leftarrow tsval_j$
    $tsecr_i \leftarrow tsecr_j$
end for
```

already in progress, and therefore the total RFD is zeroed as a new baseline. The receiver's congestion ratio is updated using $\frac{bytes_{late}}{bytes_{total}}$ similarly to Algorithm 2.

Whenever the total RFD exceeds d_{thresh} given by equation (4.3), the receiver marks Congestion Encountered (CE) bits on the next ACK. This is done just before the code that DCTCP added to accurately transmit the extent of congestion using ECN despite delayed ACKs.

Algorithm 7 Congestion Ratio with RFDs.

```
for every window do
    $F \leftarrow bytes_{late} / bytes_{total}$
    $\alpha_{RFD} \leftarrow (1 - g) \times \alpha_{RFD} + g \times F$
    $bytes_{total} \leftarrow 0$
    $bytes_{late} \leftarrow 0$
end for
```

The receiver tracks the congestion ratio and modifies the receive window in a fashion similar to algorithms 3 and 5, except in bytes and with an immediate ACK sent every congestion window change. Because the algorithm is so similar, it is not included in this paper. The receiver exits Slow Start immediately when D^F_{total} exceeds d_{thresh} instead of waiting for multiple measurements like the sender. This is partly because the magnitude of RFD between consecutive packets is naturally much smaller than that of RTTs, and partly because the millisecond granularity of TCP timestamps is not adequate for measuring RFD between consecutive packets sent at a high rate.

RFCs 793 and 1122 strongly discourage shrinking the receive window since in-flight packets would be dropped, but they also say that senders should be prepared for that case [123, 16]. However, Linux, at least, does not appear to be in danger of dropping packets due to a shrinking receive window. It keeps quadruple the amount copied to user space in the last RTT in order to handle packet losses, Slow Start, and three RTTs worth of data. Experiments in Section § 4.4 with Inigo show that shrinking the receive window carefully results in fewer retransmitted segments than would normally occur.

Essentially, because Linux receivers are already lying to the sender about having $4\times$ less buffer space than in reality, Inigo's small DCTCP-style adjustments, and frequent ACKs are in little to no danger of causing in-flight packets to be dropped. Other TCP stacks that wish to implement receiver-side congestion control like Inigo should ensure that their receive buffer is at least twice the size of the congestion window. This will prevent in-flight packets from being dropped during extreme congestion when the window is halved over the span of one RTT.

RFC 793 states:

> The mechanisms provided allow a TCP to advertise a large window and to subsequently advertise a much smaller window without having accepted that much data. This so called “shrinking the window,” is strongly discouraged. The robust-

ness principle dictates that TCPs will not shrink the window themselves, but will be prepared for such behavior on the part of other TCPs.

RFC 1122 states:

A TCP receiver SHOULD NOT shrink the window, i.e., move the right window edge to the left. However, a sending TCP MUST be robust against window shrinking, which may cause the "useable window" (see Section 4.2.3.4) to become negative.

If this happens, the sender SHOULD NOT send new data, but SHOULD retransmit normally the old unacknowledged data between SND.UNA and SND.UNA+SND.WND. The sender MAY also retransmit old data beyond SND.UNA+SND.WND, but SHOULD NOT time out the connection if data beyond the right window edge is not acknowledged. If the window shrinks to zero, the TCP MUST probe it in the standard way (see next Section).

DISCUSSION: Many TCP implementations become confused if the window shrinks from the right after data has been sent into a larger window. Note that TCP has a heuristic to select the latest window update despite possible datagram reordering; as a result, it may ignore a window update with a smaller window than previously offered if neither the sequence number nor the acknowledgment number is increased.

4.3 Fluid Model

As described in Section 4.1.1, the DCTCP and Inigo sender-side fluid models are composed of the following non-linear, delay-differential equations in which $W(t)$ describes the source's window size, $\alpha(t)$ is the congestion ratio, $q(t)$ is the queue size of the switch, $R(t)$ is the RTT of every flow, R^* is an approximate fixed RTT needed to solve the equations, $p_q(t)$ is the packet marking process on a switch queue, and $p_s(t)$ is the packet marking process on an Inigo source. DCTCP and Inigo use all of the same equations except for $p_q(t)$ and $p_s(t)$, which are mathematically equivalent. Those equations and their variables are described in Table 4.3.

Variable	Description
C	link capacity (packets per second)
K	marking threshold (1.5KB packets)
N	number of flows
d	propagation delay (seconds)
d_q	queueing delay (seconds)
d_{thresh}	delay threshold (seconds)
g	gain
t	time (seconds)
$R(t)$	RTT of every flow (seconds)
R^*	approximate fixed RTT (seconds)
$W(t)$	average of source window sizes (1.5KB packets)
$\alpha(t)$	perfect congestion ratio
$p(t)$	packet marking process
$p_q(t)$	packet marking process on a switch queue
$p_s(t)$	packet marking process on a source
$q(t)$	queue size of the switch (1.5KB packets)

Figure 4.3: Fluid model variables.

$$\frac{dW}{dt} = \frac{1}{R(t)} - \frac{W(t)\alpha(t)}{2R(t)} p(t - R^*) \tag{4.5}$$

$$\frac{d\alpha}{dt} = \frac{g}{R(t)} (p(t - R^*) - \alpha(t) \tag{4.6}$$

$$\frac{dq}{dt} = N \frac{W(t)}{R(t)} - C \tag{4.7}$$

$$p_q(t) = \mathbb{1}_{\{q(t) > K\}} \tag{4.8}$$

Equation (4.5) represents the Congestion Avoidance phase of TCP where the window size is incremented by one packet per RTT. It subtracts the traditional half window upon congestion except it is modifies the reduction according to the extent of congestion that was measured during the previous RTT. Equation (4.6) is a perfect continuous approximation of the ratio of packets marked with congestion to the total number of packets. It is perfect in the sense that all flows have a complete view of congestion, whereas in reality each flow would recieve a sampling of the actual ratio base on the position of its packets in the bottleneck queue. Equation (4.7) describe the change in the bottleneck queue, the difference in the aggregate number of packets sent by all flow minus the capacity of the link in packets per second.

$$\frac{5}{Cd+K} \lessapprox g \lessapprox \frac{1}{\sqrt{Cd+K}} \tag{4.9}$$

$$K \approx 0.17Cd \tag{4.10}$$

Alizadeh, et al. derived Equations (4.9) and (4.10) by analyzing the limit cycle solution of equations (4.5)-(4.8), and determining the miniimum normalized queue size during a period for many values of g. They bounded g by noting the frequency of the marking process, and by analyzing the convergence rate of a hybrid model.

$$d_q = \frac{q(t)}{C} \tag{4.11}$$

$$d_{thresh} \approx 0.17d \tag{4.12}$$

$$p_s(t) = \mathbb{1}_{\{d_q > d_{thresh}\}} \tag{4.13}$$

Equations (4.11)-(4.13) directly follow from Equations (4.8)-(4.10)

Figure 4.4 replicates Figure 2 from [5] using inigo. See Listing A for an example of the Mathematica code used to generate the plots. While Figure 4.4 initially appears much different, that is only because it uses the recommended threshold of $K \approx 0.17Cd$ instead of the $K \approx 0.78Cd = 65$ from the realworld 10Gbps experiments done in [4]. In those experiments, Alizadeh, et al. observed that DCTCP only approached full throughput utiliation when using a threshold $K \geq 65$ due to the inherent burstiness of the existing hardware and software stack. If $d_{thresh} = 0.78d$ in Listing A, or $K \approx 14.2$ in A, then DCTCP and Inigo plots would be identical.

Despite the caveats mentioned above, this simple fluid model gives us confidence that the mathematics behind a congestion ratio attenuating the window backoff is basically sound. Figure 4.4 plots the numerical solution of Inigo's equations, and like DCTCP it can theoretically keep queue sizes low even when the maximum possible number of flows are vying for the same link.

One important fact to keep in mind is that the fluid model is continuous, whereas TCP operates discretely at the packet level. Not only that, but TCP traditionally only responds to congestion once per RTT in order to observe the effects of its changes. The authors of DCTCP did propose a variant with a per packet response and simulations showed promise, but all current DCTCP implementations respond once per RTT.

The fluid model makes it easy to perform parameter sweeps. In particular, several

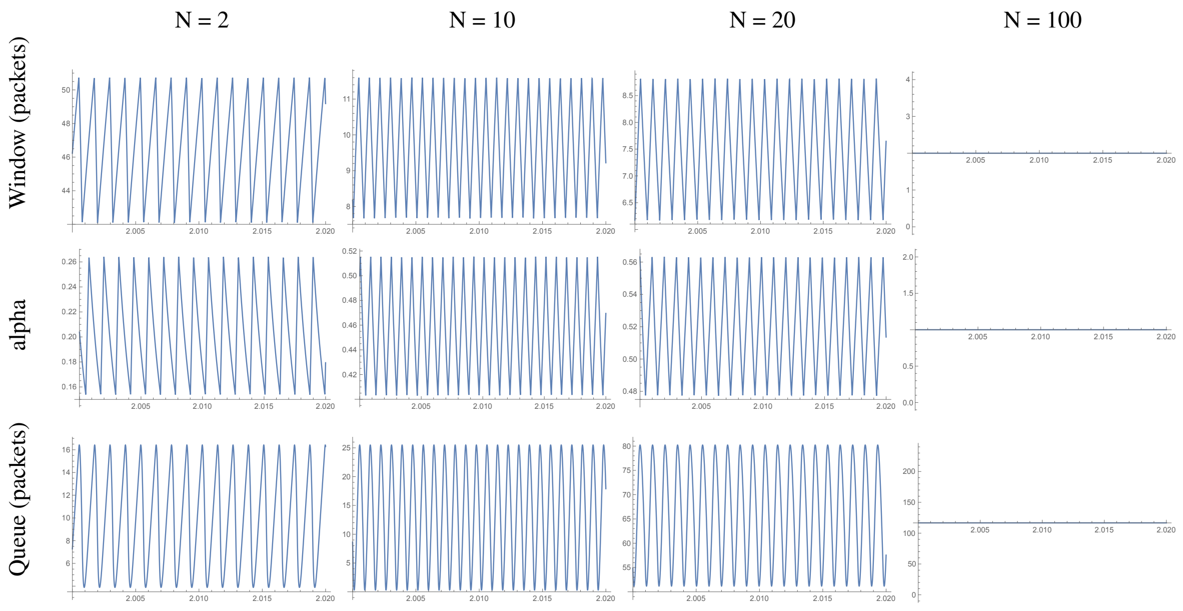

Figure 4.4: **Inigo Fluid Flow Model.**

TCPs have proposed decreasing the traditional backoff from $W/2$ to $W/3$. Note that the congestion ratio in Figure 4.4 is at most 0.56 up to 20 flows, where the backoff would be $0.28W$. See the Figure A.2 in the appendix for the full results, but Figure 4.5 shows the effects of a one third window reduction (modified by the congestion ratio) for 20 flows. Even though the range of window reductions are similar in magnitude, the $W/3$ flows drive closer to the ideal window size of 3.5 and they keep the queue size below half of the $W/2$ flows.

There is little effect of different reduction limits for fewer flows, but the story is not as good for $W/3$ reductions as the number of flows increase beyond. Its standing queue is much higher with 100 flows, as seen in Figure 4.6. In some sense, the congestion ration α is not simply a measure of congestion, but also a measure of contention. A larger number of flows will naturally drive the average α higher.

The simple fluid model can be extended to tease out whether or not there is any benefit

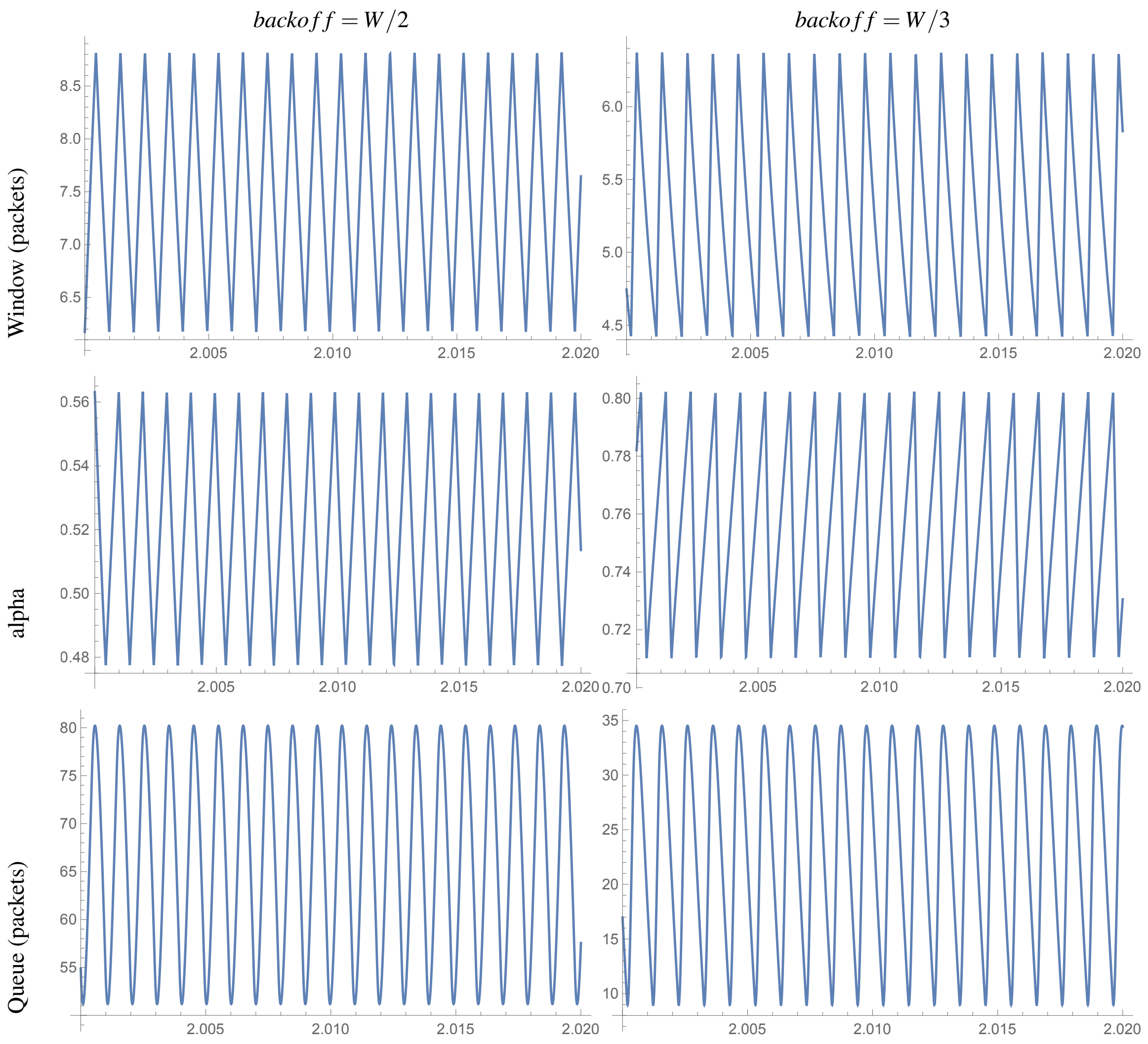

Figure 4.5: **One-third Window Reductions Can Keep Queue Sizes Lower Than One-half Window Reductions.**

to lowering the maximum window reduction. Instead of solving for a single average $\frac{dW}{dt}$, I solve for a system of $\frac{dW_i}{dt}$, one for each flow. I will retain a single, "perfect" α for now. A per-flow α could be calculated if I also tracked individual queue occupancies. However, tracking window sizes independently allows for different initial window sizes and run times. That allows us to see how well the equations converge to fairness in significantly more realistic scenarios than

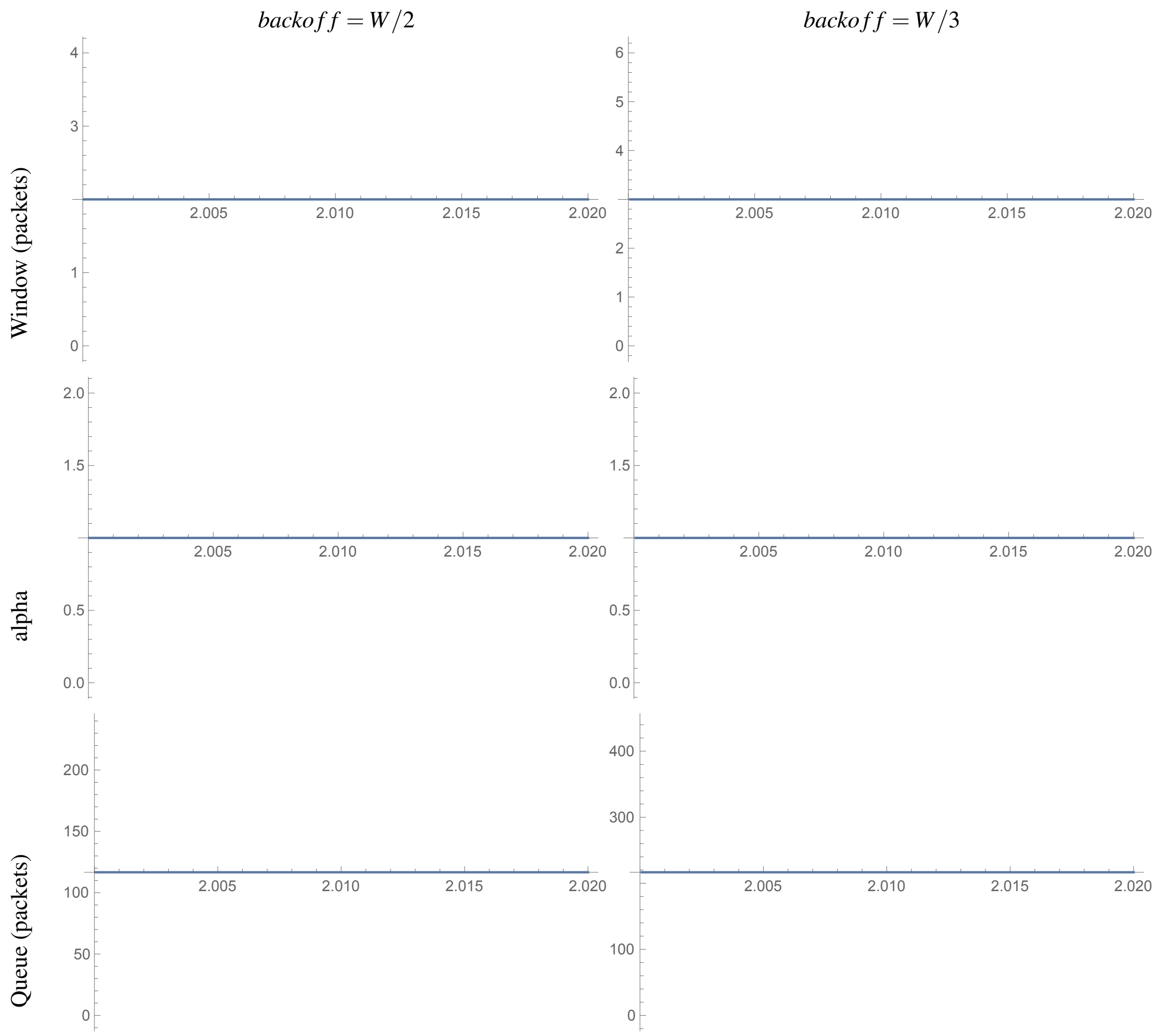

Figure 4.6: **One-third Window Reductions Do Not Back Off Enough Under High Contention.**

before. See code listing A, which also improves upon the simple model by ensuring that the queue size must be non-negative and window sizes never drop below two packets.

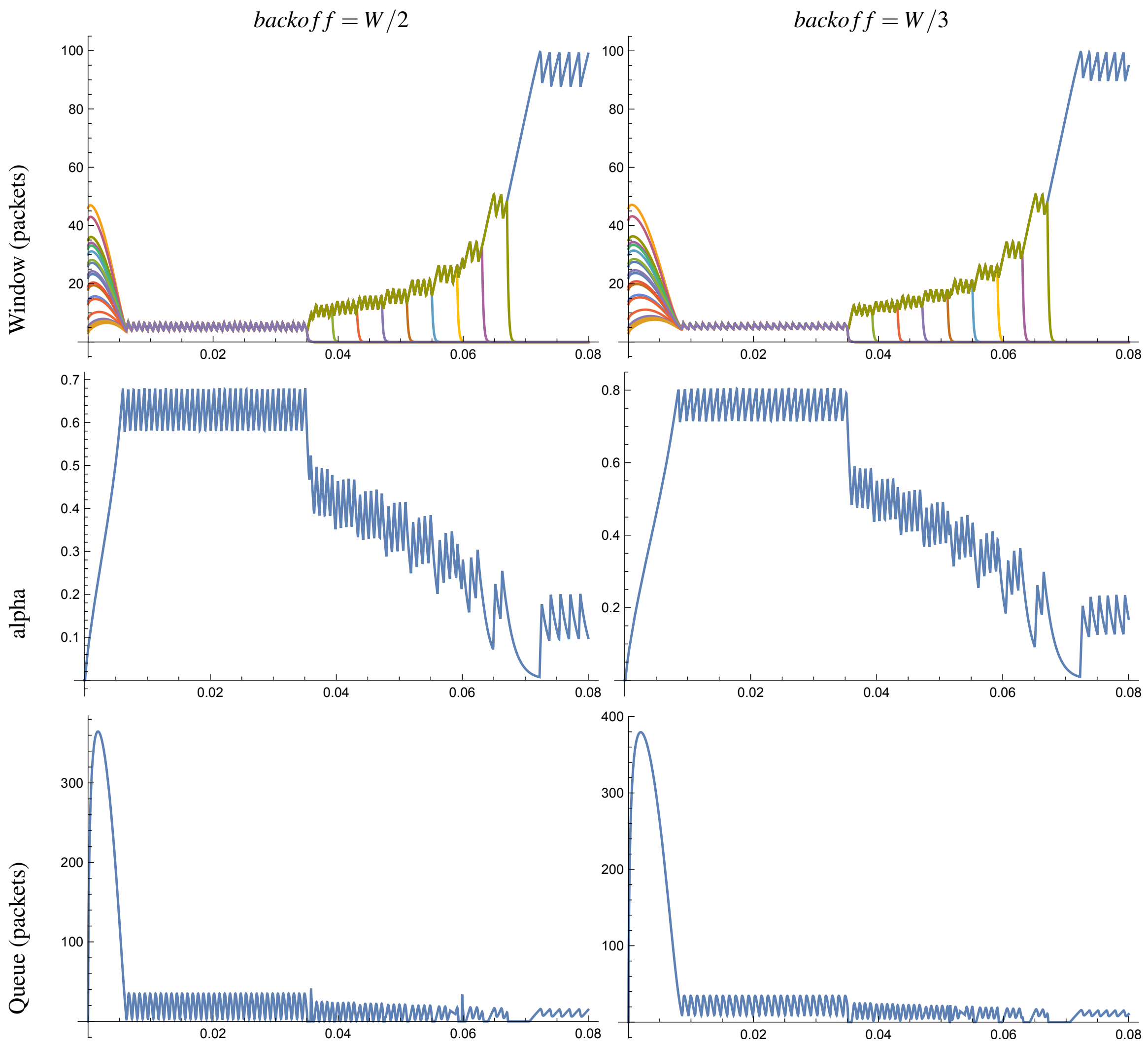

Figure 4.7: **One-third Window Reductions Do Not Back Off Enough Under High Contention (20 flows).**

4.4 Experiments

I have created TCP Inigo, which has two independent types of congestion control. The key feature of the first component of Inigo is an RTT-based congestion ratio on the sender, used as a fallback for DCTCP instead of TCP Reno when ECN is not supported. The key feature of the second component is an RFD-based congestion ratio used by the receiver to control the receive window.

Next I present convergence and incast experiments based on those commonly found in papers such as DCTCP [4]. Mininet [60] emulates a network using actual Linux code and enables rapid development and easy reproduction of experiments. Mininet is configured to provide a simple star topology with links running at 500Mbps and a 2ms one way delay between hosts. This is the highest speed network reasonable to emulate at high fidelity on a modern laptop, and while it is lower performance than one would find in a data center it is better than most connections between a data center and a home user, a case of particular interest. Iperf2 [152] clients generate the flows to one server.

Each experiment was run 30 times. The stacked bandwidth graphs show results from one representative run, and the probability distribution of results was analyzed for all results, although only a subset are shown, in violin plots due to space restrictions. Mininet does not currently allow link bandwidths above 1Gbps, and the fidelity of experiments can suffer long before that, depending on the system. However, Mininet does allow rapid development and easy reproduction of experiments.

The results of typical runs are shown using stacked bar graphs of iperf bandwidth vs. time. Each bar averages bandwidth over one second, and each graph has the same scale. In these graphs, the specific values are less important than the ability to see at a glance whether or not the expected pattern of flow bandwidths has been achieved. Below each graph are Jain's Fairness Index [73], an index defined by the fraction of aggregate application-level throughput achieved vs. theoretical rate ($\frac{rate_{achieved}}{rate_{theoretical}}$), and an index of the 99th percentile of the Smoothed

Roundtrip-time (SRTT) distribution ($\frac{RTT_{min}}{SRTT_{99th\ \%ile}}$). Note that the *latency index* inverts the theoretical:measured ratio compared to the bandwidth indices. Each index ranges from 0 to 1.0, and the combination of all three should be examined in order to evaluate the complete performance of a congestion protocol. The probability density curves showing the variation in results over 30 runs are shown after the stacked bandwidth graphs using violin plots.

I compare Inigo against CDG (the best delay-based sender variant of TCP available in Linux), CUBIC (the aggressive loss-based TCP that is default in Linux and in QUIC), and DCTCP, all described in chapter 2. Performance-oriented Congestion Control (PCC) results are not included because it was fragile, resulted in high latencies and packet loss, its experimental latency sensitive utility function did not fully utilize the link.

The first columns results are from a scenario where only the sender can be modified and ECN is typically not enabled or configured, such as in communication across networks and hosts owned by different entities. The second column corresponds to a data center-like environment in which the configuration of end-hosts and middle-boxes is coordinated. The third column is like the first except only the receiver can be modified.

4.4.1 Convergence

The first experiment demonstrates whether a technique can converge quickly to equal shares, maximize aggregate throughput, and minimize latencies while flows start and stop. Each iperf2 client precedes the next by five seconds and continues transmitting five seconds longer than the client that follows it. The bandwidth in each bar should be equally shared. Due to TCP overhead, 0.97 is the maximum Goodput index possible in this scenario.

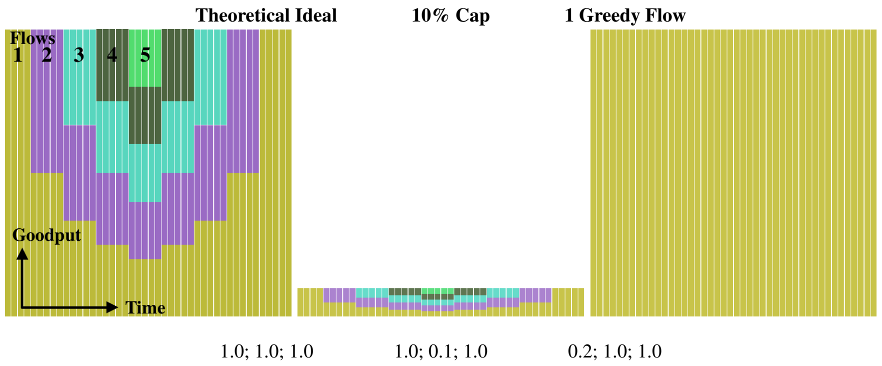

Figure 4.8: **Theoretical Ideal, 10% Capped, and 1 Greedy Flow examples.** Stacked Goodput vs. Time, five converging flows. Indices below: (1) Jain's Fairness Index; (2) Goodput Index; and (3) Latency Index of the Smoothed Round-trip-time distribution. Higher is better, and 1.0 is ideal. See beginning of Section § 4.4 for explanations.

The ideal theoretical graph for the five converging flows is shown in Figure 4.8 along with a situation where flows are fair with regard to each other, but are shaped to prevent them from fully utilizing the link's available bandwidth. Finally, one greedy flow is shown consuming all available bandwidth.

In Figure 4.9, DCTCP and Inigo are seen in two scenarios. Senders, receivers, and the network are all set up to cooperate in the first 1 administrative domain case, while only the sender is configured in the >1 administrative domain case. Inigo's worst case Latency Index is up to 42× better the competition while its Fairness and Goodput Indices are similar. This means that Inigo is good at fully utilizing links while encouraging good buffer behavior (i.e. low occupancy and draining).

Figure 4.10 shows additional possible combinations beyond Figure 4.9. Each TCP variant is tested without ECN support configured in the network, with ECN configured in the network, and without ECN but with an Inigo receiver. The Inigo sender (last row) and receiver (last column) have consistently better fairness, aggregate goodput, latencies compared to the

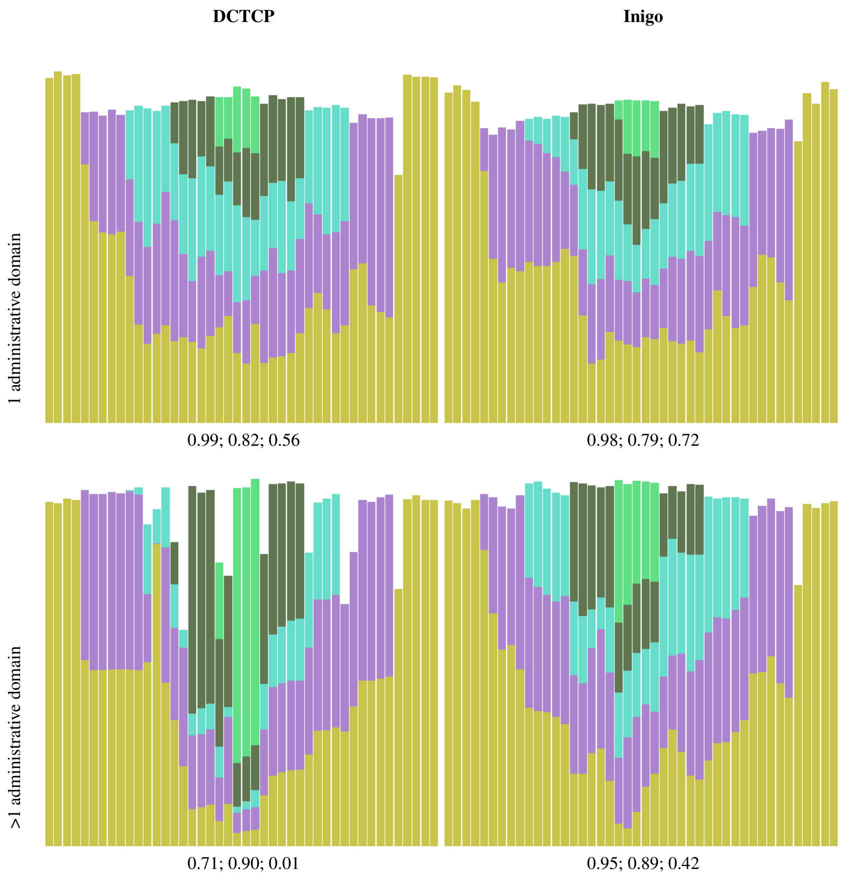

Figure 4.9: **Inigo improves upon DCTCP's Latency Index up to** $1.3\times$ **in simple environments and up to** $42\times$ **when not all components can be modified.**

other TCP variants in comparable situations.

Figure 4.12 shows how all modes of Inigo have comparable or better performance to that of DCTCP. Inigo$_{\text{ECN}}$ would have the exact same performance profile as DCTCP if it did not include a small Slow Start modification, where the first CE marked ACK in an observation window causes an immediate exit. Our latency index emphasizes the importance of tight distributions, and Inigo$_{\text{RTT}}$ does well without any help from ECN or the receiver. Other TCPs are not shown because their goodput and 99$_{\text{th}}$ percentile latency indices are not competitive.

Table 4.1: **Inigo Receiver eliminates most retransmissions and drops when paired with CUBIC and Reno.**
TCP Sender (TCP TX), Receiver-side Congestion Control (Inigo RX?), percent of retransmitted segments (%re-TX), percent of retransmitted segments that were lost (%Lost re-TX), and percent of packets dropped by the bottleneck switch (%Drop), for five converging flows. Fewer is better.

TCP TX	**Inigo RX?**	**%re-TX**	**%Lost re-TX**	**%Drop**
CDG		0.0000	0.00	0.0000
CDG	Y	0.0002	0.00	0.0000
CUBIC		1.5768	0.25	0.6660
CUBIC	Y	0.0001	0.00	0.0000
DCTCP		0.0009	0.00	0.0005
DCTCP	Y	0.0005	0.00	0.0005
DCTCP Reno fallback		0.8857	7.73	0.4293
DCTCP Reno fallback	Y	0.0002	0.00	0.0000
Inigo		0.0005	0.00	0.0006
Inigo	Y	0.0003	0.00	0.0009
Inigo RTT-based fallback		0.0004	0.00	0.0000
Inigo RTT-based fallback	Y	0.0002	0.00	0.0000

Of course, good utilization and fair bandwidth sharing is only part of the story. A link can be kept fully utilized if its buffer is kept filled to capacity, but the question is: how low a buffer can be kept without letting it drain completely too often? It can be seen in Figure 4.13 that the worst case mode for the Inigo sender approaches the low queue length of ECN-enabled TCPs, achieving greater aggregate goodput at a modest cost of increasing tail latencies. Inigo's receiver-side congestion control dramatically helps CUBIC in this case, preventing it from overflowing the bottleneck queue.

It is surprising that dropped packets or retransmitted segments were recorded in several cases (e.g. DCTCP with ECN configured and Inigo without receiver-side congestion control) since they never cause the switch's queue to overflow and the receiver window never shrinks. Inigo's and DCTCP's retransmissions were all fast retransmissions, and occurred both with and without recorded loss. Retransmits were reported even when fast retransmit [10] and early retransmit [9] were disabled, though the number of retransmits decreased. That decrease indicates reordered packets triggered some retransmits.

Analysis of packet traces using Wireshark [113] found no retransmissions occurring even while kernel stack traces confirmed that tcp_fastretrans_alert was being called. Future work is necessary to determine if Linux retransmission counts are being incremented erroneously, or if there is some other cause for these strange results

If Linux retransmit counts are taken at face value, then Table 4.1 shows that the danger of shrinking the receiver's window is low or zero. Inigo recorded fewer or equal retransmissions compared to DCTCP in every experiment. Furthermore, the Inigo receiver almost always reduces the number of retransmissions and drops. In the case of CUBIC and Reno, the Inigo receiver helps dramatically. CDG is the only TCP that sees an increase—3 fast retransmissions out of 1,721,616 sent segments.

Every TCP tested recorded segments being retransmitted—they occurred in over 86% of the experiments run, regardless of whether the switch queue dropped packets or whether receiver-side congestion control was used. In the cases where the need for retransmissions seemed improbable, analysis of packet traces with Wireshark [113] indicated none occurred. Figure 4.14 shows that Inigo's receiver-side congestion control (i.e. window shrinking) is not dangerous, and almost always reduces retransmissions for loss-based senders by $1000\times$.

4.4.2 Incast

Our incast experiment was performed using 33 servers connected by Gigabit Ethernet, with 32 iperf2 clients connecting to one server simultaneously and sending for 45 seconds. This type of scenario is common in parallel storage systems.

Figure 4.15 shows similar looking fairness and goodput. However, the underlying statistics gathered by the kernel indicate that CDG retransmitted 10000-13000 segments per flow, CUBIC retransmitted 7000-10000 segments per flow, and Inigo retransmitted 2800 to 3700 segments per flow.

Figure 4.16 shows that Inigo performs well in a modest incast experiment. And the Inigo receiver can do much to improve the fairness of DCTCP's Reno fallback. CDG and CUBIC results are not shown due to space restrictions and since their performance indices are in line with those seen in the convergence experiment.

Initial results from testing 32-flow incast on Gigabit hardware indicate that Inigo retransmits less than half the number of segments per flow compared to CDG and CUBIC. Full analysis of those results, as well as for 10 Gigabit Ethernet will be available for the final draft.

4.5 Availability

The kernel module implementing the RTT-based fallback for DCTCP, as well as the receiver-side congest control patch, together called TCP Inigo in this paper, the experimental results in this paper, and the Mininet experiment framework can be downloaded from GitHub.

`https://github.com/systemslab/tcp_inigo`

Inigo should be easily back ported to earlier kernels, although the effectiveness of the sender side will be strongly affected by the decreased RTT resolution before Linux 3.14, among other changes. The receiver-side modification is mostly contained in two functions, inserted before ECN processing and a seven line change to the receive window size selection code. Although the location of the new function calls will be slightly different prior to DCTCP's inclusion in Linux 3.17, the impact to Inigo with prior kernels should only effect DCTCP senders. Of course, DCTCP's receiver-side change could be backported too.

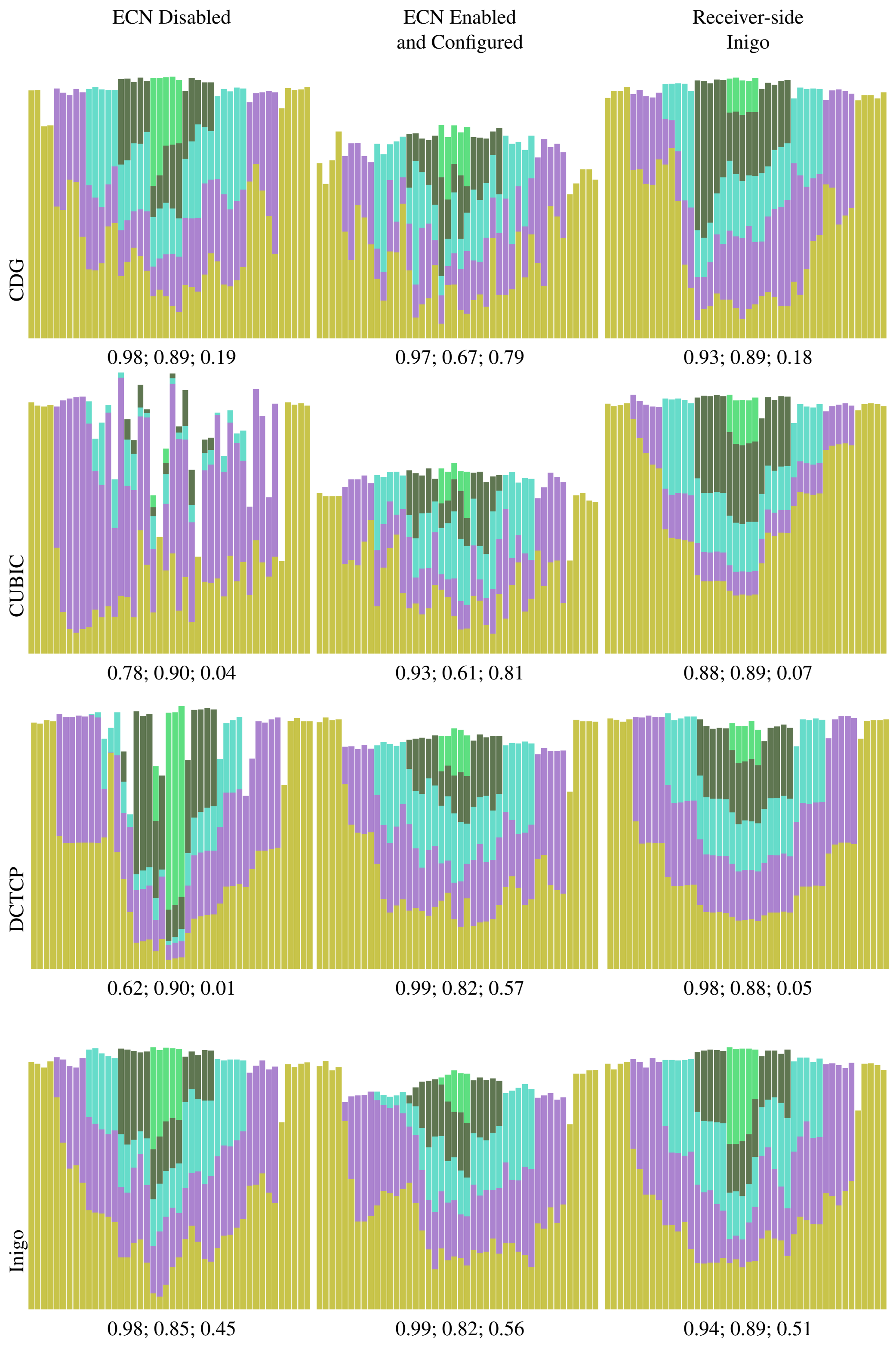

Figure 4.10: **Inigo's worst-case 99th percentile Latency Index is $> 2\times$ better than the nearest competitor, CDG, and is $> 40\times$ better than DCTCP's Reno fallback. Receiver-side Inigo improves fairness and latency of CUBIC, Inigo's RTT-based fallback, and Reno.**

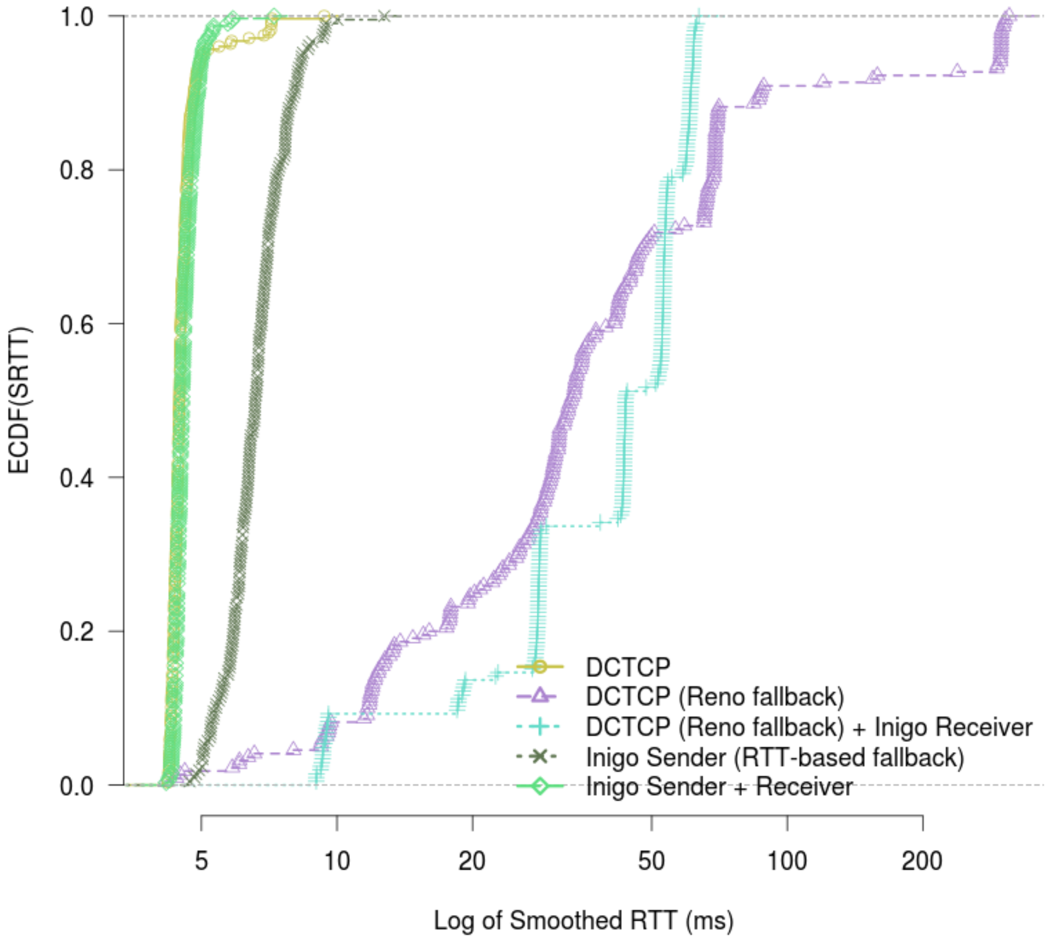

Figure 4.11: Five Flows Converge
Empirical CDF of TCP Smoothed RTT
Inigo's RTT-based fallback achieves latencies similar to DCTCP and $> 40\times$ better than DCTCP's Reno fallback.

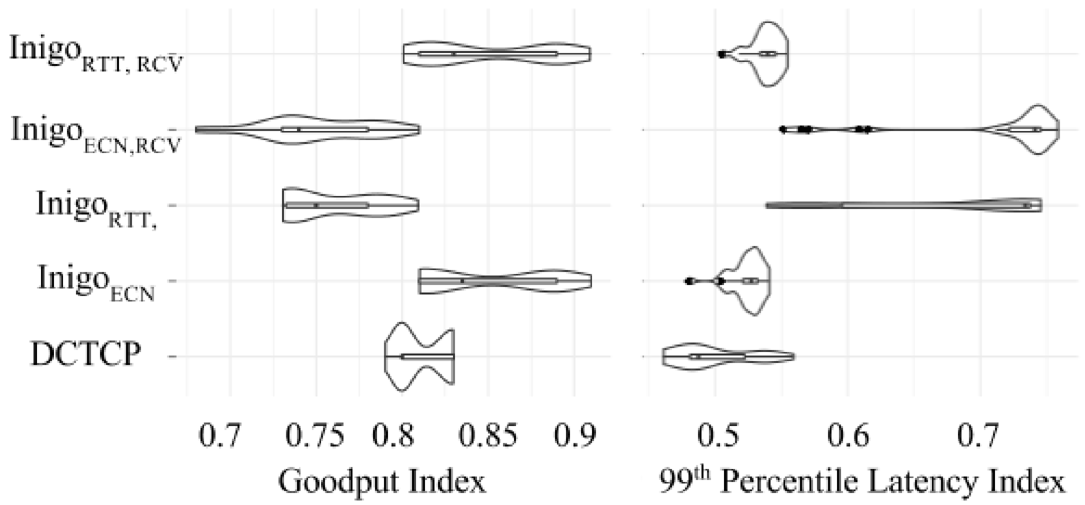

Figure 4.12: **Inigo's ECN mode optimizes for latency while its delay-based mode targets greater aggregate goodput while keeping tail latencies low. The Inigo receiver further decreases latencies while sacrificing little bandwidth.**
$\text{Inigo}_{\text{RTT,RCV}}$ (RTT-based sender with Inigo receiver), $\text{Inigo}_{\text{ECN,RCV}}$ (ECN-based sender with Inigo receiver), $\text{Inigo}_{\text{ECN}}$ (ECN-based Inigo sender), $\text{Inigo}_{\text{RTT}}$ (RTT-based Inigo sender), and DCTCP. Due to limited space, DCTCP's Reno fallback mode and other TCPs with smaller goodput and latency indices are not included. Probability density plot of goodput and latency indices. Right and thicker is better. Boxplot shows median, quartiles, and outliers.

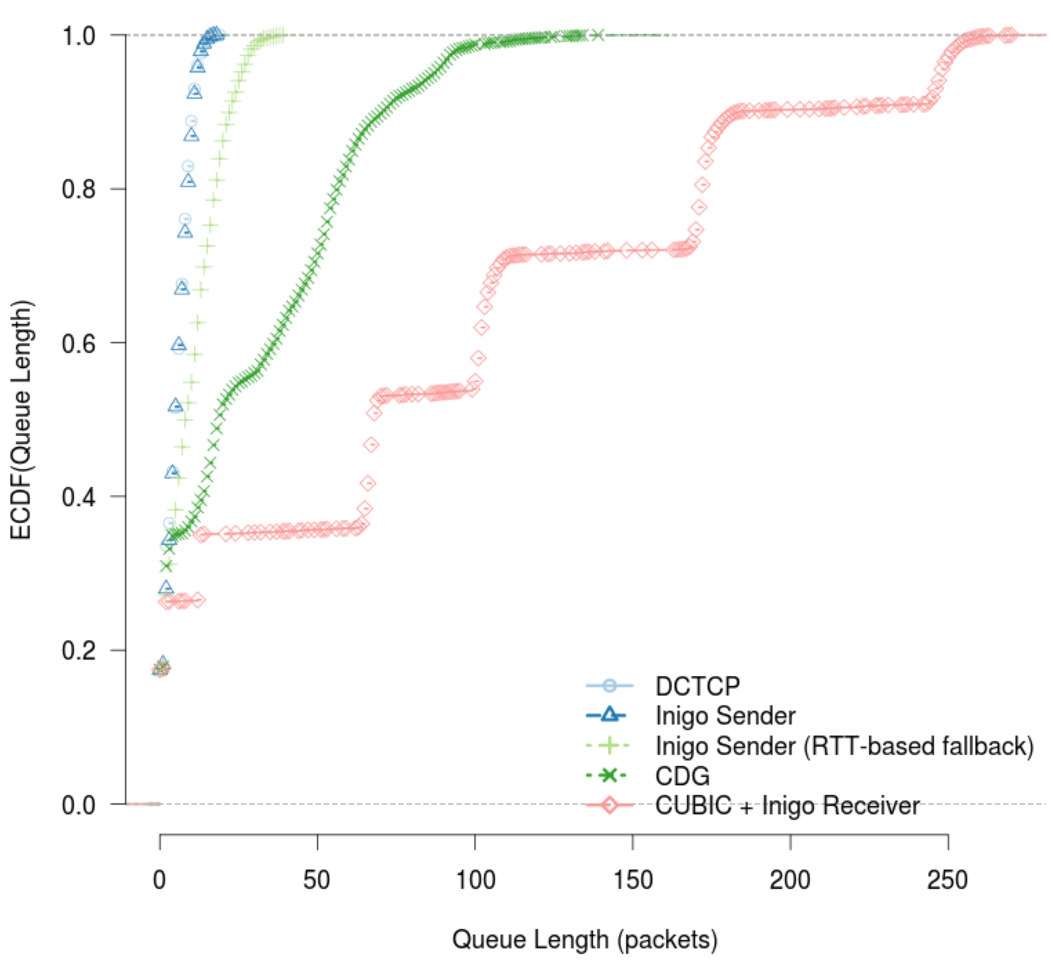

Figure 4.13: **Inigo's RTT-based worst-case sending mode approaches the low queue depth of the best-case scenario for DCTCP—Inigo's maximum queue is half CDG's.** Empirical CDF of Bottleneck Queue, five converging flows. Left and vertical is better. Results with worse aggregate goodput or greater queue lengths not shown.

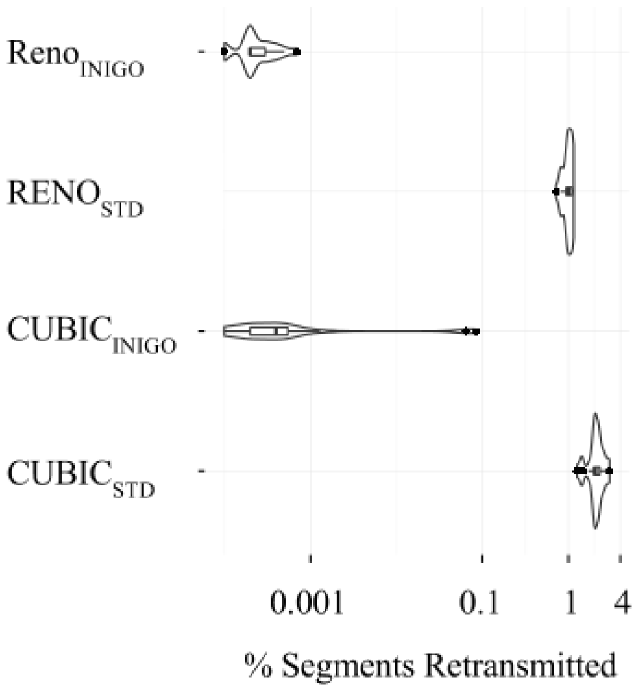

Figure 4.14: **Inigo's receiver-side congestion control (i.e. window shrinking) is not dangerous, and almost always reduces retransmissions for loss-based senders by** $1000\times$. $\text{Reno}_{\text{INIGO}}$ (DCTCP fallback with Inigo receiver), Reno_{STD} (DCTCP fallback with standard receiver), $\text{CUBIC}_{\text{INIGO}}$ (with Inigo receiver), and $\text{CUBIC}_{\text{STD}}$ (with standard receiver). Probability density plot of %retransmitted segments. Left and thicker is better.

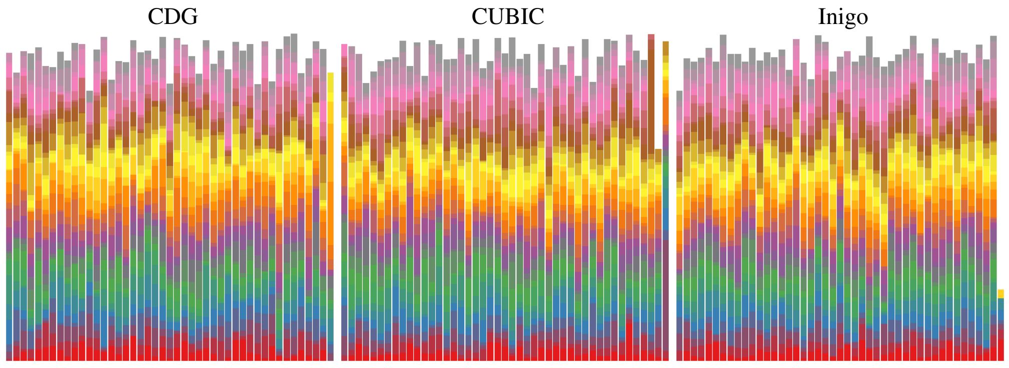

Figure 4.15: **Inigo's fairness, aggregate goodput, and latency indices are all significantly superior to DCTCP's Reno fallback.**

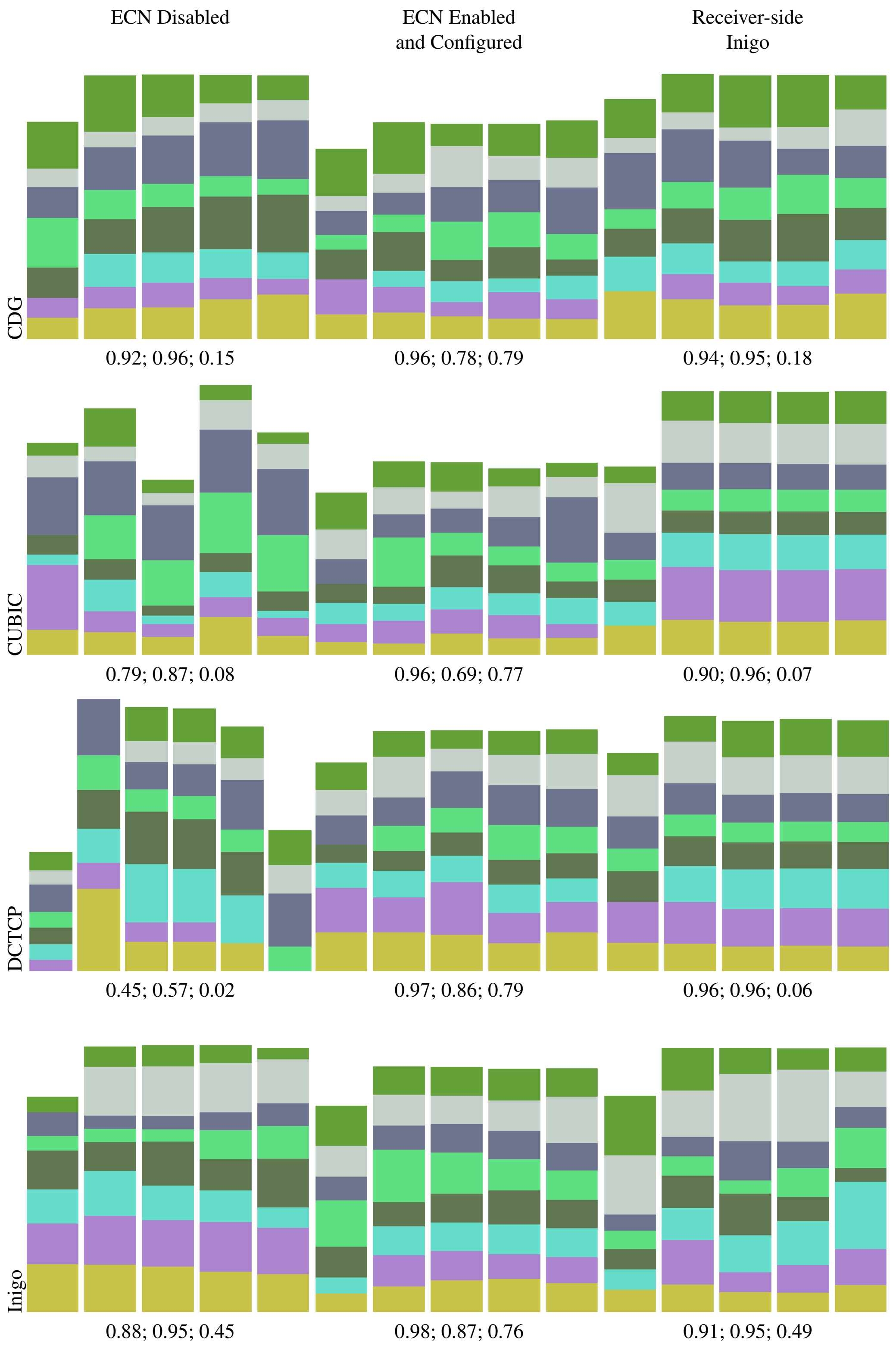

Figure 4.16: **Inigo's fairness, aggregate goodput, and latency indices are all significantly superior to DCTCP's Reno fallback.**

Stacked Goodput vs. Time, five converging flows. Indices below: (1) Jain's Fairness Index; (2) Normalized Goodput; and (3) Latency index of the Smoothed Round-trip-time distribution. Higher is better.

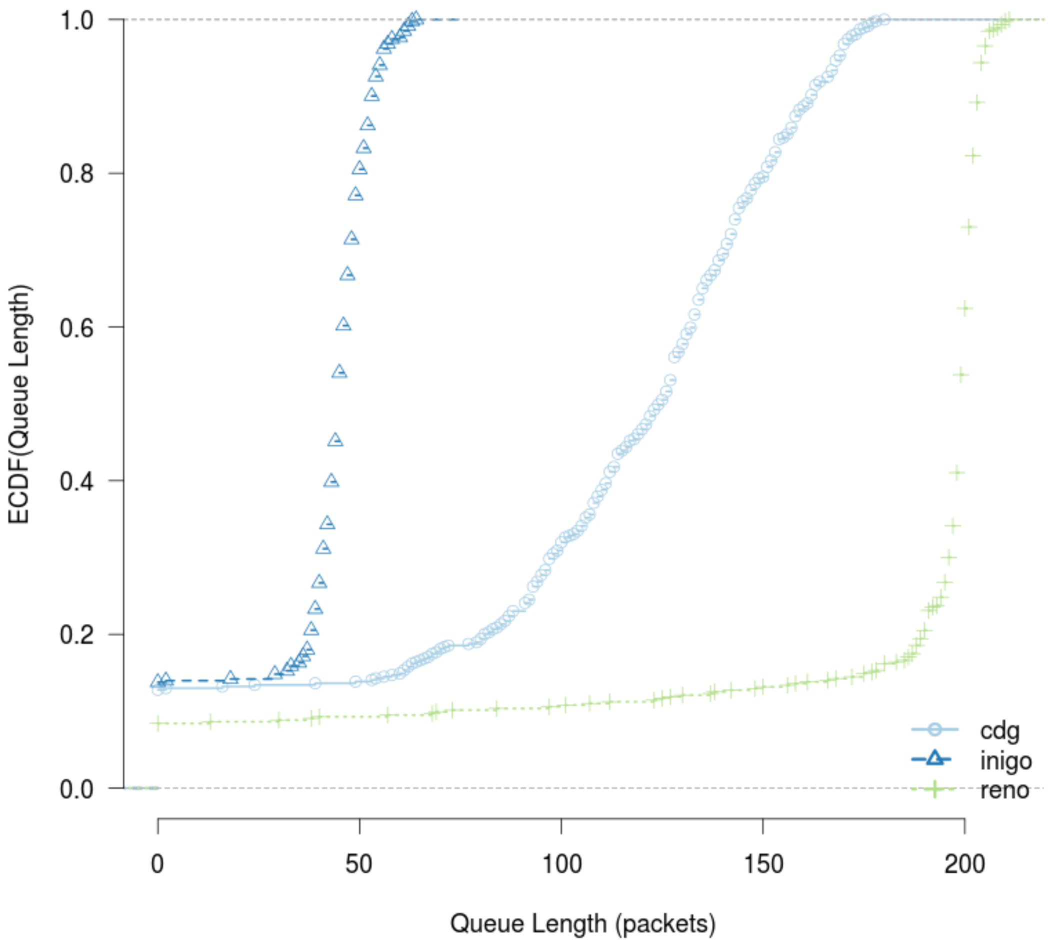

Figure 4.17: 9 Flow Incast - Receiver Congestion Control
Empirical CDF of Bottleneck Queue.
Inigo Receiver ...

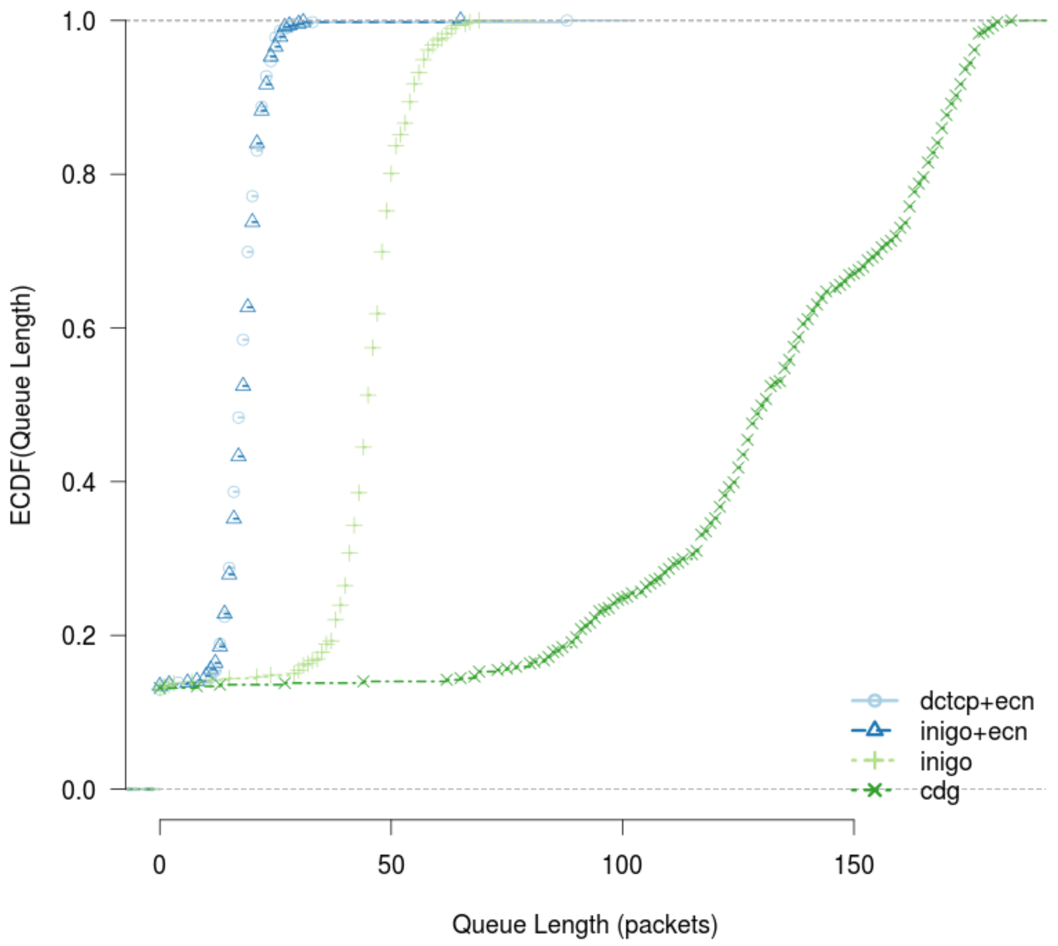

Figure 4.18: Empirical CDF of Bottleneck Queue, incast of eight flows. Left and vertical is better. Results with worse aggregate goodput or greater queue lengths not shown.
Inigo's RTT-based worst-case sending mode approaches the low queue depth of the best-case scenario for DCTCP—Inigo's maximum queue is a third of CDG's.

Chapter 5

Economic-Real-time Scheduling Theory Foundations For Queueing Disciplines

Systems should maximize the value of work being accomplished while minimizing the cost of completing it, rather than over-invest in the minimization of completion times. Admission policies like Shortest Flow First assume all flows of a given size are of equal value and finishing small flows quickly is more valuable than completing a larger flow earlier. But there are many cases where those assumptions are false. If the value of a given task is unknown, then a system should minimize completion times, but tasks should be allowed to ask for specific performance guarantees so that a system can use its resources effectively to maximize the value of work.

"What is the benefit (e.g. monetary, entertainment, scientific)?" is a question about what order a task should be admitted rather than a question about its place in a specific resource schedule. If it is always of less benefit, then it might suffer from starvation, but that means more resources need to be provided and not that the scheduler failed.

Take a social network company, for example. They must primarily serve their user's social needs, but they make money by serving ads. There are a set of core social functions that must work well, or else usage will decline and their ads will be worth less. Therefore, the benefit of those core features is basically infinite. There are extra features (e.g. auto-playing videos) that some users like, but are not essential. And then there are different types of ads that generate different amounts of revenue.

If the social network company finds that its extra features are not always getting resources, then they are still okay, but it is like a canary in a coal mine. If the advertisements cannot get reservations, then the business needs to calculate how much lost revenue there is and if it makes sense to buy more hardware. If the core functions cannot get reservations, then the system is unusable for that number of users. The business either has to kick people off to keep making enough money or take a loss while they bring up more resources.

“What are the economic costs and benefits?” is a separate question from “How much?” and "When?", and it can be used to extend RAD reservations.

$$(rate, period, benefit) \tag{5.1}$$

All three factors need to be used in order to determine admission order along the lines of WSJF, and then only rate and period are used in the actual resource schedule. The $WSJF$ equation (2.2) used in business possesses a term called Time Criticality that is not well defined, although it includes notions of value decay and deadlines.

$$urgency = T_c(t)/T_d(t) \tag{5.2}$$

$$T_c(t) = \begin{cases} period & \text{if } t \leq 0 \\ \\ period & \text{if } t \leq 0 \end{cases} \tag{5.3}$$

$$T_d(t) = t_{deadline} - t_{now} \tag{5.4}$$

D2TCP's [153] notion of urgency makes a good definition for Time Criticality, one that is especially suited for distributed deadline-aware congestion control and QoS. It is essentially the same as percent budget in my MS thesis [143], with the primary difference being that the latter used the metric to directly define pacing rather than attenuating window back-off.

RAD avoids wasted effort(where other approaches do and it enables a clean, separate decision for maximizing benefit at admission time (or periodically for long running RAD reservations). Welch, et al. partially separated out the benefit optimization from resource allocation [158].

5.1 Extending RAD With Benefit-based Admission Ordering

$$CoD = R_{direct} + R_{indirect} + R_{decay} \tag{5.5}$$

$$WSJF = \frac{CoD}{Duration} \tag{5.6}$$

Businesses sometimes use the Weighted Shortest Job First (WSJF) algorithm [68, 133] to optimize the return on investment of work being done. One of the key parameters of $WSJF$ is the Cost of Delay (CoD, which is an aggregate cost rate $\frac{price}{second}$ encompassing many attributes. Equation (5.5) defines a reasonable, although not comprehensive, CoD. For a given job the CoD is the sum of various estimates, including but not limited to: direct revenue (R_{direct}), indirect revenue ($R_{indirect}$) due to job dependencies, and revenue decay (R_{decay}).

R_{decay} might be a linearly increasing fraction of R_{direct}, or it may be more complex. For instance, it may have an absolute real-time quality in which the value of the job drops instantly to zero because the opportunity was lost. Determining the value of R_{decay} for a real-time job is somewhat difficult because equation (5.5) cannot simply be made to have multiple cases in which CoD goes to infinity if a deadline is not met. Instead, the job owner must choose an R_{decay} value that represents the cost of the job's urgency without inflating its cost beyond the revenue it can actually generate.

That sum of costs is then divided by the expected job duration, as in equation (5.6). In business, job size is often used as a proxy for duration since it can be difficult to know. In computer systems the amount of data involved in an operation and the amount of time required to complete an operation on it can often be predetermined or bounded. Ideally, CoD is calculated with absolute values. If relative values are used, then when a new job enters the system it may cause every other job's relative values to change. Highly dynamic systems should avoid the

Job	CoD	Duration	WSJF	Deadline		A,C,B	B,C,A	C,B,A	C,A,B
	($\$/s$)	(s)	$\$/s^2$	(s)		(\$)	(\$)	(\$)	(\$)
A	1.8	0.09	20.00	0.15		0.000	0.27	0.27	0.090
B	0.6	0.06	10.00	0.20		0.084	0.00	0.03	0.084
C	1.5	0.05	30.00	0.16	+	0.135	0.09	0.00	0.000
						0.219	0.38	0.30	0.174

Figure 5.1: Weighted Shortest Job First scheduling minimizes risk of not completing jobs occurring later in an ordering.

	A,C,B	B,C,A	C,B,A	C,A,B
	(s)	(s)	(s)	(s)
A	0.09	0.20	0.20	0.14
B	0.20	0.06	0.11	0.20
C	0.14	0.11	0.05	0.05

Figure 5.2: Completion times for each job under different orderings. Only A,C,B and C,A,B satisfy deadlines.

overhead of recalculations if possible.

Scheduling jobs according to the highest $WSJF$ ratio, which is in units of $\frac{price}{second^2}$, minimizes the sums of the product of the CoD and duration.

Then, the deadlines of each job should be validated by summing the durations of the current job and all previous jobs.

For example, consider a video website serving three types of jobs: A long advertisements, B short advertisements, and C primary content with costs and durations described in Table 2.1. The ordering A,C,B would be chosen by several different algorithms, including: Highest CoD, Highest $CoD \times Duration$, and Earliest Deadline First (EDF). A Lowest CoD First algorithm would choose ordering B,C,A and Shortest Job First produces ordering C,B,A. Finally, WSJB results in ordering C,A,B. Only A,C,B and C,A,B satisfy deadlines as can be seen by looking at the completion times in Table 5.2. And only C,A,B, chosen by WSJF, minimizes the risk of not completing later jobs.

$$B_{max} = \begin{cases} 2\left(\left\lceil \frac{p_c}{p_p} \right\rceil + 1\right) r_p p_p - r_p p_p & p_p \leq p_c \\ 2r_p p_p + \max\left(0, r_p p_p - \left(\left\lfloor \frac{p_p}{p_c} \right\rfloor - 1\right) r_c p_c\right) & p_p > p_c \end{cases} \tag{5.7}$$

If all the jobs complete, then the ordering may not matter. However, not all potential jobs can be admitted into a real system due to constrained resources, or they may not be able to complete before their deadlines due to the ordering. The WSJF admission ordering can be used to help ensure that user latency requirements are met while revenues are maximized.

RAD-based schedulers ensure that a task gets resources when it needs them instead of over-investing effort. However, RAD schedulers do not maximize the benefit (e.g. monetary, entertainment, scientific) of work. Fortunately, RAD can be easily extended with a notion of benefit or *CoD* such that admission control is ordered according to WSJF and then existing RAD scheduling algorithms ensure that admitted jobs complete on time.

5.2 Chained RAD Scheduler Analysis

If RAD schedulers are operating according to their design, then performance is guaranteed. However, the RAD model also enables sanity checking on the buffers between schedulers. A producer-consumer model, RAD-Flows [122], derives equations 5.7 and 5.8 describing the amount of buffer space B_{max} and time T_{max} is the amount of time it should take for the entire buffer to be rewritten for a well behaved producer/consumer pair of two interacting RAD $(rate, period)$ reservations (r_p, p_p) and (r_c, p_c).

$$T_{max} = \begin{cases} 2p_c & p_p \leq p_c \\ 3p_p & p_p > p_c \end{cases} \tag{5.8}$$

Given this knowledge, if an application suffers from overflow or underflow, RAD-Flow buffers always indicate the direction of the problem. A RAD scheduler can also guard against the unlikely situation where all RAD schedulers in a chain are misbehaving by producing and consuming too quickly.

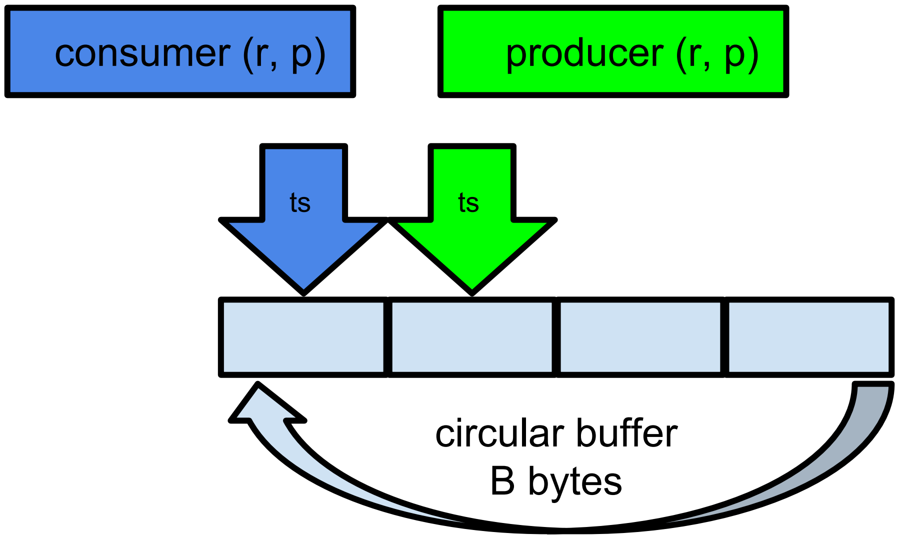

Figure 5.3: Circular RAD Buffer.

The following examples assume that a single circular buffer, as shown in Figure 5.3 can be efficiently accessed simultaneously by the producer and consumer, and a timestamp is recorded whenever there is an attempt to move a pointer. Since we know the amount of time it takes to rewrite a RAD-Flow buffer, the simple circular buffer is sufficient to illustrate the general approach for other buffer data structures.

The examples apply to both blocking and non-blocking producers. In the blocking case, overflow does not result in lost data and RAD allows us to determine whether the producer is blocking because it is attempting to write too quickly or whether the consumer caused the block by reading too slowly. Non-blocking producers will lose overflowing data and the same tests identify whether the producer or consumer bears responsibility.

If the producer pointer circles around to the consumer pointer (buffer overflow), then there are three possibilities:

1. the producer is sending faster than its reservation
2. the consumer is too slow
3. both 1 and 2

$$\text{producer} = \begin{cases} \text{fast} & ts_p - ts_c < T_{max} \\ \\ \text{slow} & ts_c - ts_p \geq T_{max} \end{cases} \tag{5.9}$$

Since the producer has overtaken the consumer, we know that it has rewritten the entire buffer from the consumer pointer on. It must have written to the consumer's location before the current value of the consumer timestamp. Because the buffer was sized according to the RAD reservations, we know the producer's pointer should not arrive at the consumer pointer's location before $ts_c + T_{max}$. Equation 5.9 uses that information to determine which party is to blame for overflow, and figures 5.4 and 5.5 give examples of both cases.

Similarly, a buffer is underflowing when the consumer pointer circles to the producer pointer. Equation 5.10 is a mirror to equation 5.9.

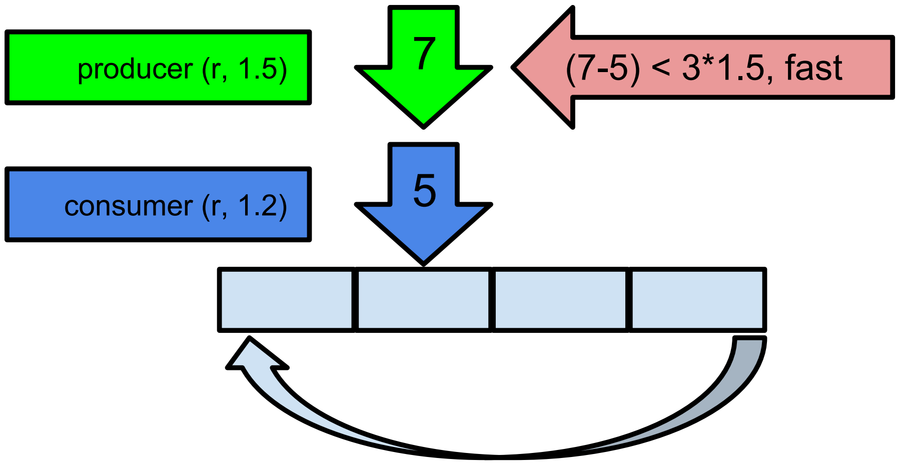

Figure 5.4: Overflowing RAD Buffer.

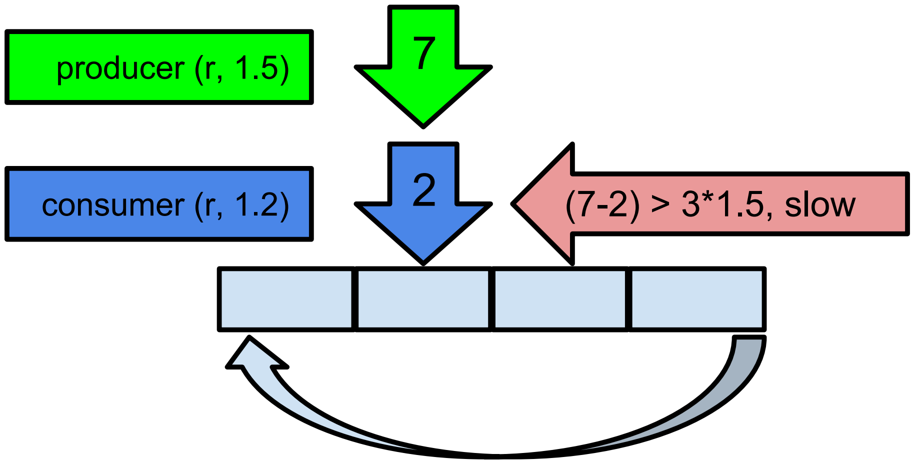

Figure 5.5: Underflowing RAD Buffer.

$$\text{consumer} = \begin{cases} \text{fast} & ts_c - ts_p < T_{max} \\ \text{slow} & ts_p - ts_c \geq T_{max} \end{cases} \tag{5.10}$$

If both the producer and consumer are misbehaving, then overflow will be blamed on the producer and underflow will be blamed on the consumer. Once their issues are fixed, the buffer will continue to overflow or underflow, but the remaining bad actor will be blamed.

With a chain of reservations, an overflowing upstream consumer might be the victim of a slow downstream consumer. So, if there are several overflowing buffers in a row connecting a chain of RAD reservations, then the blame falls on the furthest downstream consumer. Similarly, the blame for a chain of underflowing buffers percolates up to the furthest upstream producer.

There is another mode of misbehavior that is more difficult to detect. If a chain contains several RAD schedulers, including a producer and consumer which are speed-matched and both operating too fast or too slowly, then they will not overflow or underflow. However, as long as one RAD scheduler in a chain is behaving correctly, it will point in the direction of bad behavior. The only behavior dangerous to the system as a whole is when all producers and consumers in a chain are too fast. This can be guarded against with a pair of timestamps associated with the beginning of the buffer to track the last time it was produced or consumed (pick one). Whenever the beginning is accessed, the current time is compared to the last time and T_{max}, see equation 5.11.

$$\text{both fast} \;\; \text{if}\; ts_{now} - ts_{head} < T_{max} \tag{5.11}$$

In practice, comparisons will need to tolerate some small room for error to account for scheduling quanta and small indeterminate overheads in timekeeping, etc.

The final case of misbehavior is when every producer and consumer in the chain are too slow, but that would only happen when the ultimate producer is slow. In other words, it will only happen when an application is using a fraction of its reservation. This is not a danger to the system and best-effort applications can benefit from the dynamic slack.

5.3 Enhancing Reduction to UNi-resource Scheduling

Modern network interface hardware often possesses multiple queues, and Linux has supported them since the 2.6.27 kernel. That support currently allows an administrator several options, including pinning hardware queues to cores or NUMA nodes. While minimizing context switches and maximizing cache use, pinning suffers from head-of-line blocking. Alternatively, a round-robin scheduler can be used. While being fair and avoiding head-of-line blocking, round-robin necessarily hurts efficiency due to frequent migrations. Multiqueue scheduling can benefit from using RUN, which possesses provably low numbers of migrations and can prevent head-of-line blocking while enforcing QoS.

In addition to the multiqueue support already mentioned, a network classifier control group can tag packets to be handled by specific software queueing disciplines (qdiscs). Some of the existing qdiscs are Token Bucket Filters, Stochastic Fair Queueing, Fair Queuing Controlled Delay (FQ_codel), Random Early Detection, and Proportional Integral controller Enhanced. Each of these is an attempt to mitigate congestion or reduce buffer bloat in the network. Most of them concentrate on providing fairness, some provide coarse-grained QoS with priority classes.

Only one qdisc, the Hierarchical Fair Service Curve (HFSC) [149], claims to support real-time traffic, and it has fallen out of favor largely due to its complexity. Configuring a hierarchy of qdiscs to classify and shape traffic is not trivial, and HFSC separate link sharing and real-time service curves have led to contradictory information in tutorials.

FQ_codel is a "knobless" qdisc that combines Round-robin scheduling, packet pacing, and marking or dropping of packets based on the time spent in the queue. Even though a design goal was to reduce the amount of configuration, it still requires tuning based on expected and target RTTs. And CoDel requires a rate-limiting qdisc working in conjunction with it if a network does not supply its own back pressure. The CoDel part of FQ_codel is not intended for purely internal data center network traffic.

Since RUN can schedule flows according to QoS constraints across multiple hardware queues, it should be much simpler to configure. It would need to at least be able to distinguish packets according to flow, if not sub-flows multiplexed over a single connection. In that case, RUN would need to work in concert with a control group designed to tag packets with $(rate, period)$ information, in addition to the existing network priority control group.

By creating a RUN qdisc, not only will packets transmitted by Linux hosts be scheduled according to modern real-time theory, but it could lead to a RUN-based network fabric. The Open Virtual Switch module in the Linux kernel is intended to be used for both virtual machine networks as well as the operating system on hardware switches, and Open Virtual Switch uses the existing Linux qdiscs to enforce its QoS. Other Linux-based switches also exist. The efficacy of RUN can first be tested using Mininet [85], and then on real hardware.

RUN was designed to use Earliest Deadline First (EDF) scheduling within packings, but any optimal uniprocessor scheduling algorithm will work. In particular, since RUN already uses knowledge of rates and periods, it would make sense to use the Rate Based Earliest Deadline (RBED) generalization of EDF since it enables integrated scheduling of hard real-time, soft real-time (minimum rate with proportional sharing of slack), and best effort tasks [19].

The theoretical underpinnings of RBED are based on the concept of Resource Allocation and Dispatching (RAD) reservations, which are $(rate, period)$ tuples that obsolete priority classes and previously defined rate-limit specifications. Prior non-realtime scheduling methods possess a limited number of relative, coarse-grained classes (priorities), require rates to be strictly satisfied for any measured interval (e.g. Token Bucket Filters), have common periods between all tasks, or have a fixed linear mapping between periods to priorities. RAD reservations enable arbitrarily fine-grained Quality of Service (QoS), possess meanings that stay consistent in a dynamic environment, and allow straightforward reasoning about composing end-to-end QoS.

Implementing RUN with EDF is easier than with RBED since EDF only uses the tasks' deadline information. RBED flexibility requires performing an online calculation of a best effort task's rate from the ratio of its individual weight to the total weight of all best effort tasks.

If minimal preemptions with perfect proportional fairness between best-effort tasks is desired, then RUN can accommodate that goal. However, RUN could take advantage of best-effort tasks in a way similar to that of idle slack packing (see section 5.3.3). A per resource reserve would encourage both schedule partitioning and work conservation. Tasks with guaranteed rates are Best Fit packed to the best-effort reserve limit while best-effort tasks are left unassigned. RUN continues with idle slack packing, but when an idle task is executed or a guaranteed task has no work (generating dynamic slack), RBED chooses the next best-effort task that does have work.

Note that the LITMUS-RT implementation of RUN [29] does allow best effort tasks to use the processor when no real-time tasks are scheduled. While this is useful, RUN with RBED would be simpler than a hierarchical scheduler such as that. Also, RBED can support a broad range of quality of service. For instance, some tasks may only need best effort rates, but do require deadlines to be met.

The description of RUN, as well as its implementation in simulators and in practice, assumes the entire task set is known a priori and that it does not change throughout the life of the system. Certainly this does not reflect reality and should be addressed. Furthermore, the work required by the scheduler to adapt to these changes needs to be minimized.

If a task leaves the system, then it can be trivially swapped with an equal idle task without disturbing the schedule (i.e. online slack packing). However, the change in available utilization means that best effort tasks should recalculate their rates and be redistributed in order to maintain perfect proportional sharing. Perhaps it would be best to only do this work for the affected subsystem, even if that results in best effort tasks being treated somewhat unfairly across the entire system.

If a feasible task enters the system and has a rate less than or equal to an idle task, then it can be inserted with little effort. However, if the new task has a rate greater than any one idle task, then RUN must create a new reduction tree. Of course, it would best to maintain the previous reduction tree as much as possible in order to minimize one-time missed deadlines and migration penalties.

In fact, there is an additional constraint in online packing when compared to offline packing: in addition to task utilizations summing to less than or equal to one, the sums of their remaining budget must be less than the time left until the furthest deadline of the packed tasks.

This is always a constraint, but one that it is obviously satisfied when the system is offline. Best-effort tasks, like idle tasks, make RUN's job easier for dynamic task sets.

5.3.1 Sporadic Tasks

Defining all tasks as fixed-rate results in overprovisioning resources for a sporadic task. This unused utilization is called dynamic slack. RUN does not, as of yet, support the sporadic task model, so there is naturally interest in scheduling algorithms that do. It has been an open question whether or not RUN can be directly modified to support sporadic tasks—the original creators of it have, in fact, proposed the Quasi Partitioned Scheduling (QPS) algorithm as an alternative [99, 98].

However, if RUN supports both best effort tasks and dynamic task sets as described above, then it does not matter if a sporadic task is defined as a fixed-rate task. The dynamic slack can be used by best effort tasks packed with sporadic tasks.

5.3.2 Resource Assignment

The creators of RUN were primarily concerned with producing the set of tasks that should run at any given time. Their task-to processor assignment scheme is simple:

1. leave an executing task on its current processor
2. assign idle tasks to their last-used processor
3. assign remaining tasks arbitrarily

For systems expect zero slack or best effort tasks, it might also be worth the effort to keep track of the set of each tasks' m previously used resources. That way, if a task must migrate repeatedly, then it is more likely to migrate within a subset of the resources rather than amongst all of the resources within its subsystem. In other words, this heuristic should help on NUMA systems where the cost of migration becomes much larger between NUMA nodes.

5.3.3 Offline Bin Packing

The effectiveness of the pack step in RUN's offline reduction phase determines the performance of the algorithm in terms of the number of preemptions caused per job. The authors prove that each release event in a p-level reduction tree causes at most $\lceil (p+1)/2 \rceil$ preemptions, and so RUN suffers an average of no more than $\lceil (3p+1)/2 \rceil$ preemptions per job. Therefore, packing heuristics that minimize both the number of reduction levels and the number of release events are of interest.

The authors found that most packing heuristics achieved the same number of reduction levels as long as the task rates were sorted in descending order. Best Fit Descending achieves fewer preemptions and migrations than the others due to its packing of high rate tasks together first. Worst Fit Descending performs worse in this regard because it spreads out high rate tasks among bins, increasing the likelihood that one of those tasks must be preempted more often.

High rate tasks are at greater risk of suffering preemption than low rate tasks because they utilize a greater percentage of their period. If high rate tasks are packed together first, as in Best Fit Descending, then short period tasks are limited in the number of tasks they can preempt. Likewise, a long period task can only be preempted by a limited number of tasks. This vulnerability of high rate tasks also helps explain the humps in preemptions and migrations in Levin's figure 3.10 [91]. With a number of tasks, t, and resources m, the tasks are particularly vulnerable when $m < t < 2m$ on a fully utilized system.

Intuitively, one would expect tasks with similar periods (e.g. harmonic) to decrease the number of release events, and Levin did report testing that hypothesis with a Least Common Multiple (LCM) Fit heuristic in his dissertation. While LCM-fit sometimes results in trees with more reduction levels, it still achieves 4-5% fewer preemptions and migrations. Unfortunately, it comes at the cost of significantly greater algorithmic complexity.

By analyzing the number of additional events when a long period task is packed with successively shorter period tasks in figure 5.6, It can be seen that LCM-fit fails to minimize some events. When periods shorten, as in tasks 8 and 9, the least common multiple remains small with task 1, but the number of release events continues to increase. If several tasks with the same periods are encountered, then the packing benefits from overlapping events. However, it would be better to pack a few tasks that were within half a period length of a long task rather than a few beyond the 2nd harmonic.

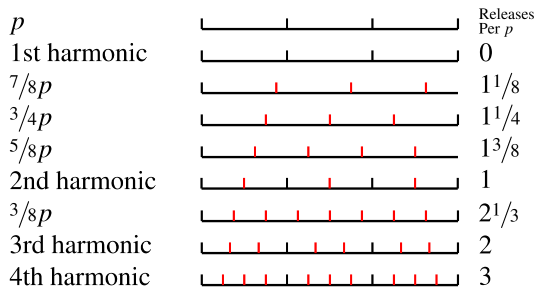

Figure 5.6: The Effect Of Period Length On Events.

In addition to measuring the preemptions caused per job, it would be illuminating to compare preemptions per time unit. Since small rate jobs are less likely to be preempted, the number of preemptions per job could look low even if the per time unit count is high.

There are considerations other than the number of preemptions that might make a heuristic attractive. Below are brief descriptions of some different packing heuristics and when they might be most useful.

Slack Packing As described in the original paper, Slack Packing increases the number of independent partitions (decreasing migrations) when the system utilization is less than 100%. By adding idle tasks at the end of the first packing (regardless of primary packing heuristic).

Worst Fit Decreasing Rates Optimizes spreading large tasks amongst resources, making it suitable for less highly loaded situations or when it is more important to use each resource rather than using fewer resources. For instance, this would benefit parallel applications.

Best Fit Decreasing Rates Minimizes the number of packings and reduction levels (minimizes preemptions and migrations), so it would be suitable for highly loaded situations or when it is more important to use fewer resources than using all resources.

These heuristics have already been evaluated in terms of performance (i.e. preemptions and migrations) by the original authors. Further work would be necessary to determine which would be appropriate for a power saving policy. It might be that it would be necessary to switch between Worst and Best Fit, as the former might allow all processors to finish more quickly where the latter could keep some cores off entirely.

Certainly other heuristics exist, but among these Slack Packing with Best Fit Decreasing Rates is both simple to implement and one of the best performing heuristics.

5.3.4 Affinity

Some work needs to be done to make RUN support resource affinity. This is common in the case of processes in a NUMA system that want to be as close to an attached device (e.g. network card or GPU) as possible. Affinity can be thought of as a partial pre-specification of the packings and placements. Pinning a task to a single resource should be easy to take into account, and regular recurring sets (i.e. a NUMA nodes) should also be straightforward to support. However, arbitrary affinity sets may be unworkable since they might over-constrain the packing problem. At any rate, affinity support in RUN requires further investigation.

In general, it is unlikely that RUN can be directly applied to global load balancing of routes. However, if the switches themselves use RUN as just described, then they should be able to provide valuable information to a global load balancing algorithm: the amount of unreserved rate (i.e. static slack), dynamic slack, number of best-effort flows, which flows are underflowing or overflowing and whether they blame upstream or downstream. This information, while simple, should be much more useful to a global scheduler than basic rate and drop information. The rest of this section discusses why it would be nice to use RUN for route management and why it is difficult to apply it.

The edges of networks present an interesting opportunity for RUN. Whether it is a global WAN gateway where bandwidth is extremely limited and precious or the high performance interconnect between a supercomputer and a parallel file system, flows should maximize the utilization of the available routes while preventing congestion and data loss. And in cases where there are multiple links or paths that immediately reconverge on the other side, as in figure 5.7, RUN can be applied.

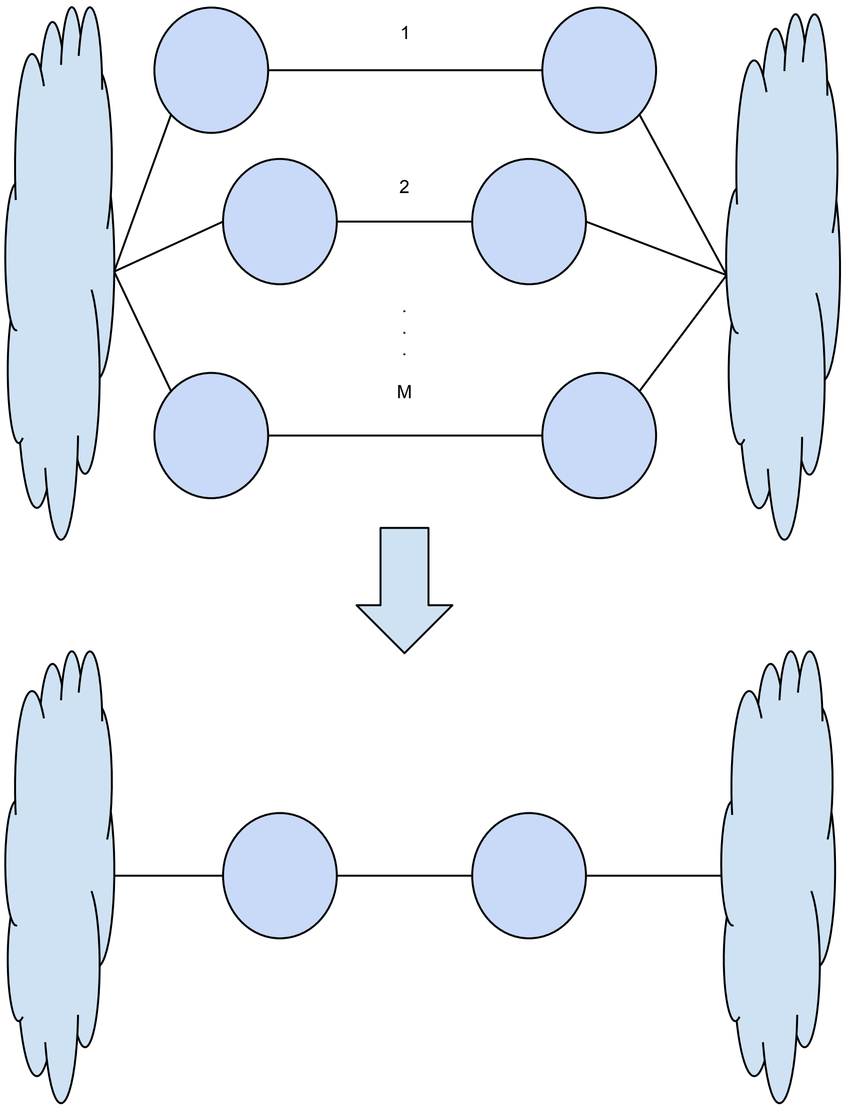

Figure 5.7: Reduce to Unipath.

As opposed to traditional distributed multipath routing approaches described by Hopps [66], RUN would be used by a SDN (Software Defined Network) Controller or a Subnet Manager (in the case of Infiniband) to assign routes to flows after they have passed the standard RAD admission control test: Would the new flow's rate cause the total flow rate to exceed capacity?

To be clear, RUN would not be discovering the topology of the network. It is scheduling the routes given to it to manage. Also, it would take further work to make RUN take considerations other than a path's cost (e.g. security) into account.

Can RUN be used in the case where the multiple routes are not immediately recombined as Jain, et al. addressed with B4 [74]? Perhaps, but there are two big concerns. First, how would one partition the graph so that at least the left hand side of the routes looks identical from RUN's perspective? In other words, if the graph is simply bisected, then RUN assumes that any route will work and could overload the left side of the graph. It appears RUN would have to execute recursively on the graph from the edges inward.

That brings up the second concern: when RUN schedules a task it may migrate between several of the resources. At that point, the single fixed-rate task becomes a sporadic task on each of those resources. Treating them each as fixed-rate will quickly exhaust the resources even though best effort flows could suck up the dynamic slack. So, even a hard real-time flow would be forced to become several best-effort flows, and thus comprise the original guarantee. An additional concern arises from the best effort tasks. Just because they can use up the dynamic slack at one hop does not mean the next hop can handle the burst.

Chapter 6

Future Work

In addition to testing Inigo in a much greater variety of conditions, I have several enhancements planned:

Linux now supports Kathleen Nichols' minimum RTT tracking algorithm [166], which might enable Inigo to adapt to situations with changing RTTs such as when a flow is forced to switch to a longer route or the underlying media toggles Forward Error Correction.

A variant of Inigo that uses a per-ACK congestion response was inspired by Bob Briscoe's mention of Relentless TCP [100] in his review of the DCTCP Internet Draft [21]. Preliminiary experiments show that it yields superior fairness, goodput, and latency indices compared to the sub-window response used in this paper. However, it appears to require modifications to Slow-start and Congestion Avoidance that need detailed analysis and a theoretical proof of correctness.

The effectiveness of the Inigo receiver would greatly benefit from microsecond timestamps, just as the sender did moving from millisecond to microsecond RTTs. An administrative configuration may be the only safe way of maintaining compatibility across domains. However, it may make sense to automatically toggle between resolutions depending on the RTT or the spacing between packets.

I also plan on adding deadline-awareness to Inigo similar to D2TCP [153], but with a probabilistic response instead of one adjusted by a fractional power (i.e. gamma) function. Using a "coin flip" technique where a pseudo-random number is compared to a scaled urgency should be more amenable to a kernel implementation than a fractional power function. Incorporating randomness in the window adjustment may also help desynchronize long-lived flows.

Chapter 7

Conclusion

Providing good performance guarantees on networks is impossible without first solving the problem of congestion, which negatively impacts even high speed networks like Infiniband. This dissertation presents both sender and receiver-side delay-based techniques for congestion control, as well as presenting a foundation for clean slate queue scheduling designs based on economic and real-time scheduling theory.

The difficulty inherent in deploying new technology on networks provided part of the motivation for the TCP congestion control variant, Inigo, described in this paper. Inigo does not require special hardware, driver development, or switch configurations.

Inigo's sender-side RTT-based congestion control integrates with DCTCP and provides a fallback that resembles DCTCP's ECN-based behavior. The receiver-side RFD-based congestion control, though less effective than the sender-side due to coarse-grained timestamps, is able to encourage fair bandwidth sharing and smaller buffer occupancy of TCP senders such as CUBIC and Reno. We refer to both of these modifications as TCP Inigo in this paper, even though each modification can be brought into service separately.

When Inigo's sender and receiver are used together, their latency, bandwidth, and fairness indices are up to 1.3$\times$ better than the best deployable solution. And only senders can be modified, Inigo performs up to 42$\times$ better than the competition.

Appendix A

Fluid Flow Model

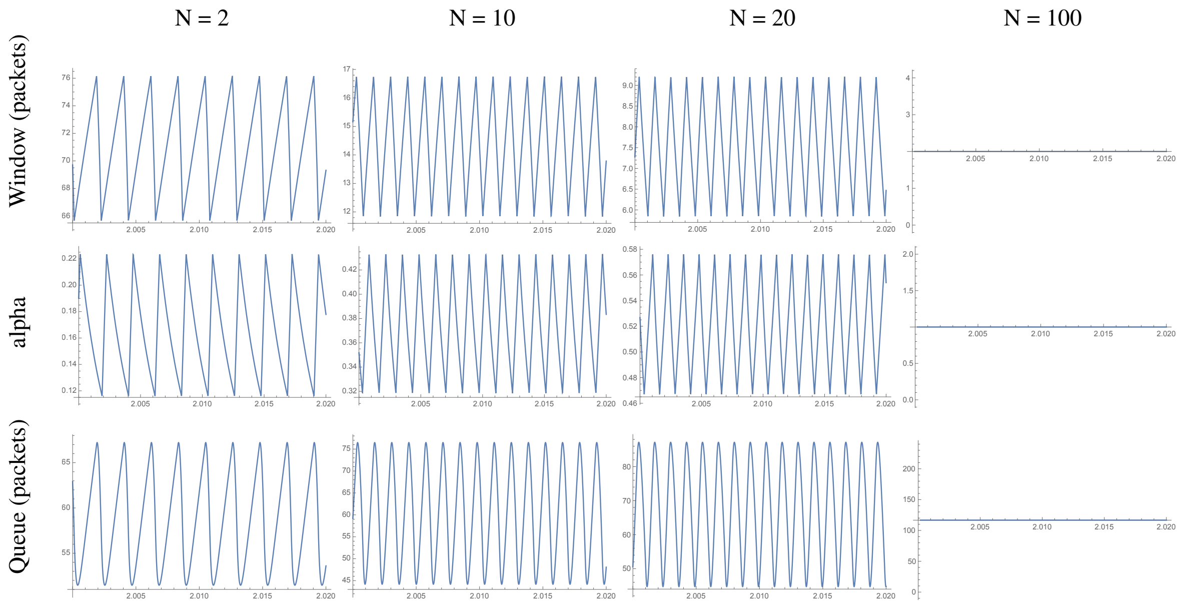

Figure A.1: **DCTCP Fluid Flow Model.**

```
Bpp = 8*1500;
Cap = 10 Power[10, 9]/Bpp;
K = 65;
d = 100 Power[10, -6];
g = 1/16;
Nf = 2;
p[t_] := Piecewise[{{1, q[t] > K}, {0, q[t] <= K}}]
```

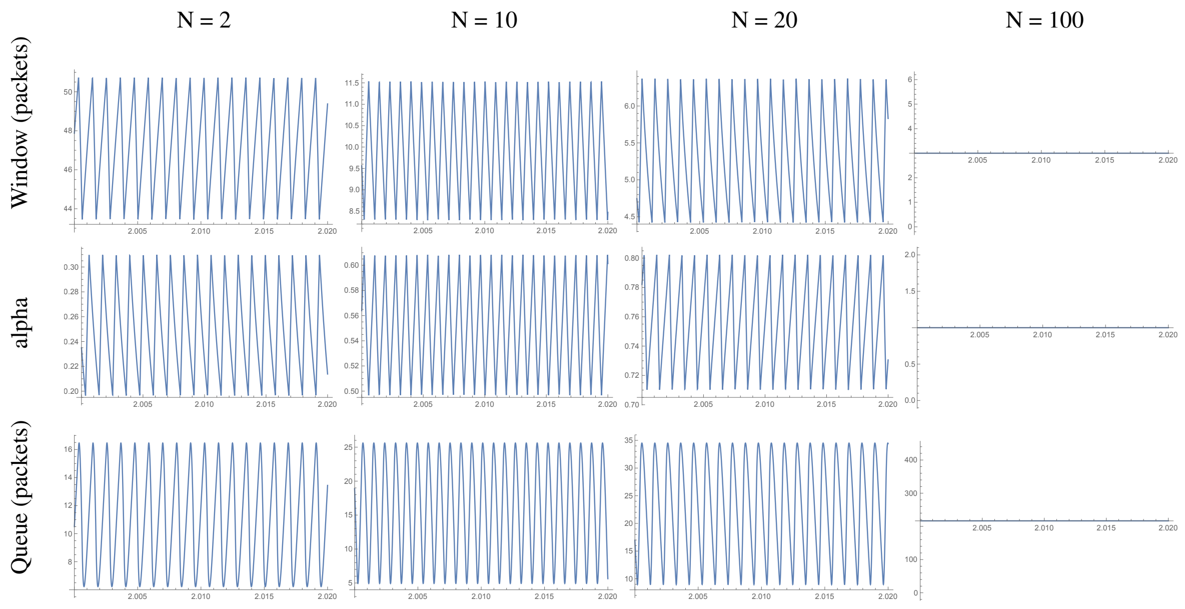

Figure A.2: **Inigo One-third Backoff Fluid Flow Model.**

```
nsol2 =
 NDSolve[
  { W'[t] == 1 / (d + q[t]/Cap) - W[t] \[Alpha][t] (p[t - (d + K / Cap)])/(2 (d + q[t] / Cap)),
    W[t /; t <= 0] == 10,
    \[Alpha]'[t] == g / (d + q[t] / Cap) (p[t - (d + K / Cap)] - \[Alpha][t]),
    \[Alpha][ t /; t <= 0] == 0,
    q'[t] == Nf W[t] / (d + q[t] / Cap) - Cap,
    q[t /; t <= 0] == 0},
  { W[t], \[Alpha][t], q[t] },
  { t, 2, 2.02 }
 ]
Plot[Evaluate[{W[t]} /. First[nsol2]], {t, 2, 2.02}, PlotRange -> All]
Plot[Evaluate[{\[Alpha][t]} /. First[nsol2]], {t, 2, 2.02}, PlotRange -> All]
Plot[Evaluate[{q[t]} /. First[nsol2]], {t, 2, 2.02}, PlotRange -> All]
```

code/dctcpfluidflow–fig2.txt

```
Bpp = 8 * 1500;
Cap = 10 Power[10, 9] / Bpp;
d = 100 Power[10, -6];
dthresh = 0.17 d;
g = 1/16;
```

```
Nf = 2;
p[t_] := Piecewise[{{1, q[t]/Cap > dthresh}, {0,  q[t]/Cap <= dthresh} }]

nsol2 =
 NDSolve[
  { W'[t] == 1 / (d + q[t] / Cap) - W[t] \[Alpha][t] (p[t - (d + dthresh)])/(2 (d + q[t] / Cap)),
    W[t /; t <= 0] == 10,
    \[Alpha]'[t] == g / (d + q[t] / Cap) (p[t - (d + dthresh)] - \[Alpha][t]),
    \[Alpha][t /; t <= 0] == 0,
    q'[t] == Nf W[t]/ (d + q[t] / Cap) - Cap,
    q[t /; t <= 0] == 0 },
  { W[t], \[Alpha][t], q[t] },
  { t, 2, 2.02 }
  ]
Plot[Evaluate[{W[t]} /. First[nsol2]], {t, 2, 2.02}, PlotRange -> All]
Plot[Evaluate[{\[Alpha][t]} /. First[nsol2]], {t, 2, 2.02}, PlotRange -> All]
Plot[Evaluate[{q[t]} /. First[nsol2]], {t, 2, 2.02}, PlotRange -> All]
```

code/inigofluidflow–fig2.txt

```
Nf = 40;
exportdir = "/tmp/";

SeedRandom[1];
Winit = RandomInteger[{2, 46}, 100];
tstop = 0.035;
tend = 0.08;
Wstop = {
   tend, tstop, tstop + 0.004, tstop + 0.008, tstop + 0.012,
   tstop + 0.016, tstop + 0.020, tstop + 0.024, tstop + 0.028,
   tstop + 0.032,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop
   };
```

```

bpp = 8*1500;(*bits per packet*)
Cap = 10 Power[10, 9]/bpp;(*pps link capacity*)
d = 100 Power[10, -6];(*Seconds RTT*)
dthresh = 0.17 d;(*Delay congestion threshold*)
g = 1/16;(*gain*)
b = 0.5; (*backoff factor*)

pqd[t_] := Piecewise[{{1, q[t]/Cap > dthresh}, {0, q[t]/Cap <= dthresh}}]
pq0[t_] := Piecewise[{{1, q[t] > 0}, {0, q[t] <= 0}}]
pw0[t_] :=
 Piecewise[{{0, t <= Wstop[[i]]},
           {(1 + 0.5 Subscript[W, i][t])/(d + q[t]/Cap), t > Wstop[[i]]}}]

Windows =
 Table[{Subscript[W, i]'[t] ==
    1/(d + q[t]/Cap) - Subscript[W, i][t] \[Alpha][t] b pqd[t - (d + dthresh)]/(d + q[t]/Cap) - pw0[t],
   Subscript[W, i][t /; t <= 0] == Winit[[i]]}, {i, Nf}]
Weqns = Table[{Subscript[W, i][t]}, {i, Nf}]

nsol =
 NDSolve[
  {Windows,
   \[Alpha]'[t] == g/(d + q[t]/Cap) (pqd[t - (d + dthresh)] - \[Alpha][t]),
   \[Alpha][t /; t <= 0] == 0,
   q'[t] == ( Sum[Subscript[W, i][t], {i, 1, Nf}])/(d + q[t]/Cap) - Cap pq0[t],
   q[t /; t <= 0] == 0},
  {Weqns, \[Alpha][t], q[t]},
  {t, 0, tend},
  MaxSteps -> 300000
  ]

Plot[Evaluate[Weqns /. First[nsol]] , {t, 0, tend},
 PlotRange -> Full, PlotLegends -> Flatten[ Weqns]]
LogLinearPlot[Evaluate[Weqns /. First[nsol]] , {t, 0.001, tstop},
 PlotRange -> Full, PlotLegends -> Flatten[ Weqns]]
Plot[Evaluate[{d + q[t]/Cap} /. First[nsol]] , {t, 0, tend},
 PlotRange -> Full, PlotLegends -> {"d"}]
Plot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
 PlotRange -> Full, PlotLegends -> {q[t]}]
LogPlot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
 PlotRange -> {{Automatic}, {1, Automatic}}, PlotLegends -> {q[t]},
 Ticks -> {{0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1,
     0.11, 0.12}, {0, 1, 10, 25, 50, 100, 350, 1000, 2000}}]
Plot[Evaluate[{\[Alpha][t]} /. First[nsol]], {t, 0, tend},
 PlotRange -> Automatic, PlotLegends -> {\[Alpha][t]}]

Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-window.pdf",
 Plot[Evaluate[Weqns /. First[nsol]] , {t, 0, tend},
  PlotRange -> Full] ]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-window-loglinear.pdf",
 LogLinearPlot[Evaluate[Weqns /. First[nsol]] , {t, 0.001, tstop},
  PlotRange -> Full, PlotLegends -> Flatten[ Weqns]] ]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-rtt.pdf",
 Plot[Evaluate[{d + q[t]/Cap} /. First[nsol]] , {t, 0, tend},
  PlotRange -> Full]]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-queue.pdf",
 Plot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
  PlotRange -> Full]]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-queue-logplot.pdf",
 LogPlot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
  PlotRange -> {{Automatic}, {1, Automatic}},
  Ticks -> {{0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08,
     0.09}, {0, 1, 10, 25, 50, 100, 350, 1000, 2000}}]]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-alpha.pdf",
 Plot[Evaluate[{\[Alpha][t]} /. First[nsol]], {t, 0, tend},
  PlotRange -> Automatic]]
```

code/inigofluidflow–table.txt

```
Nf = 40;
exportdir = "/tmp/";

SeedRandom[1];
Winit = RandomInteger[{2, 46}, 100];
tstop = 0.035;
tend = 0.08;
Wstop = {
   tend, tstop, tstop + 0.004, tstop + 0.008, tstop + 0.012,
   tstop + 0.016, tstop + 0.020, tstop + 0.024, tstop + 0.028,
   tstop + 0.032,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop,
   tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop, tstop
   };

bpp = 8*1500;(*bits per packet*)
Cap = 10 Power[10, 9]/bpp;(*pps link capacity*)
d = 100 Power[10, -6];(*Seconds RTT*)
dthresh = 0.17 d;(*Delay congestion threshold*)
g = 1/16;(*gain*)
b = 0.5; (*backoff factor*)

pqd[t_] := Piecewise[{{1, q[t]/Cap > dthresh}, {0, q[t]/Cap <= dthresh}}]
pq0[t_] := Piecewise[{{1, q[t] > 0}, {0, q[t] <= 0}}]
pw0[t_] :=
 Piecewise[{
  {0, t <= Wstop[[i]]},
  {(1 + 0.5 Subscript[W, i][t])/(d + q[t]/Cap), t > Wstop[[i]]}} ]

\[Alpha]thresh1 = 0.5;
\[Alpha]thresh2 = 0.19;
back[t_] :=
 Piecewise[{
  {\[Alpha][t]^2, \[Alpha][t] >= \[Alpha]thresh1},
  {0.5 \[Alpha][t], \[Alpha][t] >= \[Alpha]thresh2},
  {1.01 \[Alpha][t], \[Alpha][t] < \[Alpha]thresh2}} ]

Windows =
 Table[{Subscript[W, i]'[t] ==
     1/(d + q[t]/Cap) - Subscript[W, i][t] \[Alpha][t] back[t] pqd[t - (d + dthresh)]/(d + q[t]/Cap) - pw0[t],
   Subscript[W, i][t /; t <= 0] == Winit[[i]]}, {i, Nf}]
Weqns = Table[{Subscript[W, i][t]}, {i, Nf}]

nsol =
 NDSolve[
  {Windows,
   \[Alpha]'[t] == g/(d + q[t]/Cap) (pqd[t - (d + dthresh)] - \[Alpha][t]),
   \[Alpha][t /; t <= 0] == 0,
   q'[t] == ( Sum[Subscript[W, i][t], {i, 1, Nf}])/(d + q[t]/Cap) - Cap pq0[t],
   q[t /; t <= 0] == 0},
  {Weqns, \[Alpha][t], q[t]},
  {t, 0, tend},
  MaxSteps -> 300000
  ]

Plot[Evaluate[Weqns /. First[nsol]] , {t, 0, tend},
 PlotRange -> Full, PlotLegends -> Flatten[ Weqns]]
LogLinearPlot[Evaluate[Weqns /. First[nsol]] , {t, 0.001, tstop},
 PlotRange -> Full, PlotLegends -> Flatten[ Weqns]]
Plot[Evaluate[{d + q[t]/Cap} /. First[nsol]] , {t, 0, tend},
 PlotRange -> Full, PlotLegends -> {"d"}]
Plot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
 PlotRange -> Full, PlotLegends -> {q[t]}]
LogPlot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
 PlotRange -> {{Automatic}, {1, Automatic}}, PlotLegends -> {q[t]},
 Ticks -> {{0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1,
     0.11, 0.12}, {0, 1, 10, 25, 50, 100, 350, 1000, 2000}}]
Plot[Evaluate[{\[Alpha][t]} /. First[nsol]], {t, 0, tend},
 PlotRange -> Automatic, PlotLegends -> {\[Alpha][t]}]

Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-alpha2-backoff-window.pdf",
 Plot[Evaluate[Weqns /. First[nsol]] , {t, 0, tend},
  PlotRange -> Full] ]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-alpha2-backoff-window-loglinear.pdf",
 LogLinearPlot[Evaluate[Weqns /. First[nsol]] , {t, 0.001, tstop},
  PlotRange -> Full, PlotLegends -> Flatten[ Weqns]] ]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-alpha2-backoff-rtt.pdf",
 Plot[Evaluate[{d + q[t]/Cap} /. First[nsol]] , {t, 0, tend},
  PlotRange -> Full]]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-alpha2-backoff-queue.pdf",
 Plot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
  PlotRange -> Full]]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-alpha2-backoff-queue-logplot.pdf",
 LogPlot[Evaluate[{q[t]} /. First[nsol]] , {t, 0, tend},
  PlotRange -> {{Automatic}, {1, Automatic}},
  Ticks -> {{0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08,
     0.09}, {0, 1, 10, 25, 50, 100, 350, 1000, 2000}}]]
Export[exportdir <> "inigofluidflow-table-n" <> ToString[Nf] <> "-alpha2-backoff-alpha.pdf",
 Plot[Evaluate[{\[Alpha][t]} /. First[nsol]], {t, 0, tend},
PlotRange -> Automatic]]
```

code/inigofluidflow–table–alpha2.txt

Appendix B

RUN

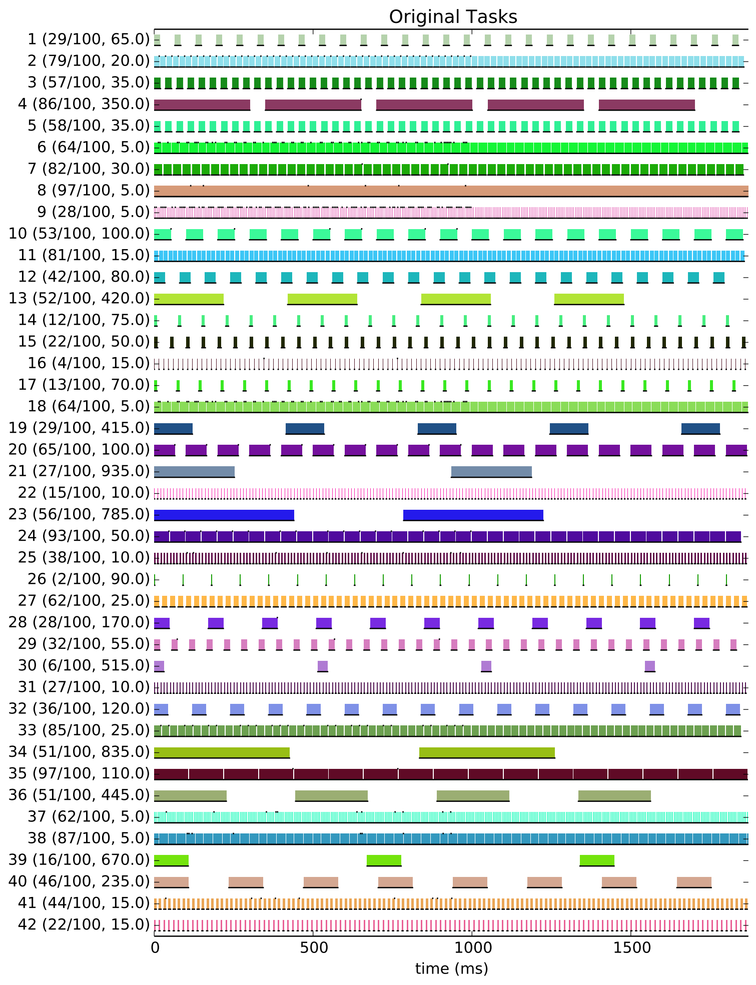

Figure B.1: RUN Scheduling 42 Tasks on 20 Resources.

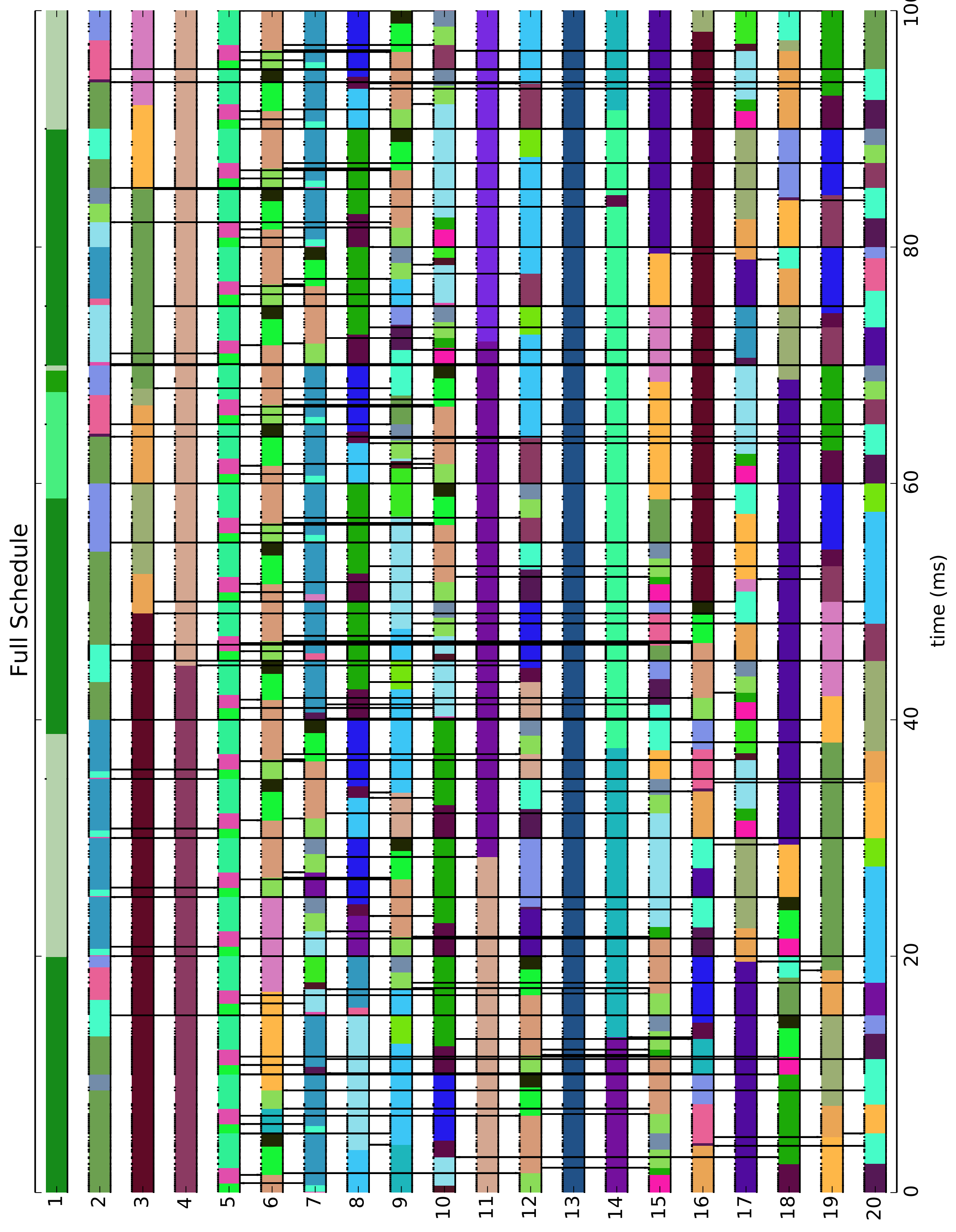

Figure B.2: RUN Scheduling 42 Tasks on 20 Resources.

Table B.1: Preemption and Migration Statistics for RUN Scheduling 11 Tasks on Four Resources.

Task ID	Avg. Preemptions/Job	Avg. Migrations/Job
1	1.605	0.000
2	85.980	0.000
3	0.746	0.000
4	1654.500	13.500
5	75.786	0.000
6	17.000	1.965
7	118.848	0.000
8	24.730	1.000
9	9.235	1.915
10	285.400	0.000
11	71.409	1.833
12	186.833	2.500
13	1334.500	0.000
14	0.824	0.000
15	32.050	0.000
16	3.657	0.940
17	30.857	0.000
18	19.100	1.855
19	656.667	3.667
20	360.800	4.300
21	882.000	0.000
22	11.500	0.800
23	2031.000	0.000
24	259.150	3.400
25	24.390	1.780
26	0.194	0.000
27	86.975	2.550
28	262.000	0.000
29	90.000	0.000
30	153.500	0.000
31	16.730	1.650
32	154.000	0.000
33	122.325	2.975
34	2303.000	0.000
35	583.778	3.222
36	993.500	0.000
37	14.435	0.000
38	20.225	0.000
39	271.000	0.000
40	574.750	3.250
41	42.343	1.448
42	19.134	0.000

Bibliography

[1] Samuli Aalto, Urtzi Ayesta, Sem Borst, Vishal Misra, and Rudesindo Núñez-Queija. Beyond processor sharing. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):36–43, 2007.

[2] Jeff Ahrenholz. Comparison of core network emulation platforms. In *2010-MILCOM 2010 MILITARY COMMUNICATIONS CONFERENCE*, 2010.

[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). *ACM SIGCOMM computer communication review*, 41(4):63–74, 2011.

[5] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 73–84. ACM, 2011.

[6] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability analysis of qcn: the averaging principle. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 49–60. ACM, 2011.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 19–19. USENIX Association, 2012.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[9] M. Allman, K Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. Tcp congestion control. `https://tools.ietf.org/html/draft-ietf-tcpm-early-rexmt-04`, 2009.

[10] M. Allman, V. Paxson, and E. Blanton. Tcp congestion control. https://tools.ietf.org/html/draft-ietf-tcpm-rfc2581bis-07, 2009.

[11] Mostafa Ammar. Why we still don't know how to simulate networks. In *38th Annual Simulation Symposium*, page 3. IEEE, 2005.

[12] Thomas Anderson, Andrew Collins, Arvind Krishnamurthy, and John Zahorjan. PCP: efficient endpoint congestion control. In *nsdi06*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[13] S. Bensley, L. Eggert, D. Thaler, P. Balasubramanian, and G. Judd. Datacenter tcp (dctcp): Tcp congestion control for datacenters. https://tools.ietf.org/html/draft-ietf-tcpm-dctcp-02, 2016.

[14] Marjory S Blumenthal and David D Clark. Rethinking the design of the internet: the end-to-end arguments vs. the brave new world. *ACM Transactions on Internet Technology (TOIT)*, 1(1):70–109, 2001.

[15] David Borman, Richard Scheffenegger, and Van Jacobson. Rfc 7323: Tcp extensions for high performance. https://tools.ietf.org/html/rfc7323, 2014.

[16] Robert Braden. Rfc 1122: Requirements for internet hosts. https://tools.ietf.org/html/rfc1122, 1989.

[17] Lawrence S. Brakmo and Larry L Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *Selected Areas in Communications, IEEE Journal on*, 13(8):1465–1480, 1995.

[18] Maury Bramson. A stable queueing network with unstable fluid model. *Annals of applied probability*, pages 818–853, 1999.

[19] Scott A Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 396–407. IEEE, 2003.

[20] Ron Brightwell. Communications. https://xstackwiki.modelado.org/Communications.

[21] B. Briscoe. [tcpm] review of draft-ietf-tcpm-dctcp-01. http://www.ietf.org/mail-archive/web/tcpm/current/msg10051.html, 2015.

[22] Bob Briscoe, Anna Brunstrom, Andreas Petlund, David Hayes, David Ros, Jyh Tsang, Stein Gjessing, Gorry Fairhurst, Carsten Griwodz, and Michael Welzl. Reducing internet latency: a survey of techniques and their merits. *IEEE Communications Surveys & Tutorials*, 2014.

[23] Bob Briscoe and Koen De Schepper. Scaling tcp's congestion window for small round trip times. Technical report, Technical report TR-TUB8-2015-002, BT, 2015.

[24] Bob Briscoe and Matt Mathis. Congestion exposure (conex) concepts, abstract mechanism and requirements. https://tools.ietf.org/html/draft-ietf-conex-abstract-mech-13, 2014.

[25] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 2013)*, pages 213–218, 2560 Ninth Street, Suite 215, Berkeley, CA, 94710 USA, 2013.

[26] Christopher D Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[27] Wen Chen, Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Ease the queue oscillation: Analysis and enhancement of dctcp. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 450–459. IEEE, 2013.

[28] Wu chun Feng and Peerapol Tinnakornsrisuphap. The adverse impact of the TCP congestion-control mechanism in heterogeneous computing systems. In *International Conference on Parallel Processing*, pages 299–306, 2000.

[29] Davide Compagnin, Enrico Mezzetti, and Tullio Vardanega. Putting run into practice: implementation and evaluation. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 75–84. IEEE, 2014.

[30] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[31] Jonathan Corbet. A damp discussion of network queuing. https://lwn.net/Articles/616241/.

[32] Jonathan Corbet. Jls2009: Generic receive offload. https://lwn.net/Articles/358910/.

[33] Jonathan Corbet. Network transmit queue limits. https://lwn.net/Articles/454390/.

[34] Jonathan Corbet. Tcp small queues. https://lwn.net/Articles/507065/.

[35] Jonathan Corbet. Transport-level protocols in user space. https://lwn.net/Articles/691887/.

[36] Jonathan Corbet. Tso sizing and the fq scheduler. http://lwn.net/Articles/564978/.

[37] Koen De Schepper, Olga Bondarenko, Jyh Tsang, and Bob Briscoe. Data centre to the home: Ultra-low latency for all (under submission). http://www.bobbriscoe.net/projects/latency/dctth_preprint.pdf, 2015.

[38] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[39] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 431–442. ACM, 2014.

[40] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, Oakland, CA, May 2015. USENIX Association.

[41] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. Processor sharing flows in the internet. In *International Workshop on Quality of Service*, pages 271–285. Springer, 2005.

[42] Eric Dumazet. tcp: switch rtt estimations to usec resolution. https://goo.gl/TtBZ3Z, 2014.

[43] Amir Efrati. The ascension of google's sridhar ramaswamy. https://www.theinformation.com/The-Ascension-of-Google-s-Sridhar-Ramaswamy.

[44] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[45] Paul Emmerich, Daniel Raumer, Florian Wohlfart, and Georg Carle. Performance characteristics of virtual switching. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 120–125. IEEE, 2014.

[46] Agner Krarup Erlang. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B*, 20(33-39):16, 1909.

[47] Norm Finn. Ieee standard for local and metropolitan area networks—virtual bridged local area networks - amendment: 10: Congestion notification. http://www.ieee802.org/1/pages/802.1au.html, 2010.

[48] Sally Floyd. Tools for the evaluation of simulation and testbed scenarios. https://tools.ietf.org/html/draft-irtf-tmrg-tools-05, 2005.

[49] Sally Floyd. Maintaining a critical attitude towards simulation results (invited talk). http://www.icir.org/floyd/talks/WNS2-Oct06.pdf, 2006.

[50] Sally Floyd. Metrics for the evaluation of congestion control mechanisms. https://tools.ietf.org/html/rfc5166, 2008.

[51] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, 1993.

[52] Editor Geoff Garner. Ieee 802.1 time-sensitive networking task group. http://www.ieee802.org/1/pages/tsn.html, 2016.

[53] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40, 2011.

[54] William Goldman. *The Princess Bride: S. Morgenstern's Classic Tale of True Love and High Adventure*. Houghton Mifflin Harcourt, 2007.

[55] Ernst Gunnar Gran, Magne Eimot, S-A Reinemo, Tor Skeie, Olav Lysne, Lars Paul Huse, and Gilad Shainer. First experiences with congestion control in infiniband hardware. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[56] Ernst Gunnar Gran, Eitan Zahavi, S-A Reinemo, Tor Skeie, Gilad Shainer, and Olav Lysne. On the relation between congestion control, switch arbitration and fairness. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 342–351. IEEE, 2011.

[57] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015. USENIX Association.

[58] Sangtae Ha and Injong Rhee. Taming the elephants: New tcp slow start. *Computer Networks*, 55(9):2092–2110, 2011.

[59] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[60] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[61] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems (TOCS)*, 21(2):207–233, 2003.

[62] David A Hayes and Grenville Armitage. Revisiting tcp congestion control using delay gradients. In *NETWORKING 2011*, pages 328–341. Springer, 2011.

[63] David Andrew Hayes and David Ros. Delay-based congestion control for low latency. In *ISOC Workshop on Reducing Internet Latency, Sep*, 2013.

[64] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and J Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 15:17, 2008.

[65] Urs Hengartner, Jürg Bolliger, and Thomas Gross. Tcp vegas revisited. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1546–1555. IEEE, 2000.

[66] Christian E Hopps and Dave Thaler. Multipath issues in unicast and multicast next-hop selection. 2000.

[67] Allison Hume. Re-thinking the tcp congestion control design in network simulators. 2016.

[68] Scaled Agile Inc. Weighted shortest job first, 2016.

[69] J. Iyengar and I. Swett. Quic loss recovery and congestion control. `https://tools.ietf.org/html/draft-tsvwg-quic-loss-recovery-01`, 2014.

[70] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[71] Van Jacobson, Robert Braden, and David Borman. Rfc 1323: Tcp extensions for high performance. `https://tools.ietf.org/html/rfc1323`, 1992.

[72] Manish Jain and Constantinos Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. In *In Proceedings of Passive and Active Measurements (PAM) Workshop*. Citeseer, 2002.

[73] Raj Jain, Dah-Ming Chiu, and William Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. 1998.

[74] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 3–14. ACM, 2013.

[75] Nan Jiang, James Balfour, Daniel U Becker, Brian Towles, William J Dally, George Michelogiannakis, and John Kim. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 86–96. IEEE, 2013.

[76] C. Jin, D. Wei, and S. Low. Fast tcp: Motivation, architecture, algorithms, performance, 2004.

[77] Abdul K. Kabbani. *Algorithms for congestion control and bandwidth-partitioning in data centers*. PhD thesis, Stanford University, 2011.

[78] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM computer communication review*, 32(4):89–102, 2002.

[79] Midori Kato. Improving transmission performance with one-sided datacenter tcp. Master's thesis, Keio University, 2014.

[80] Leonard Kleinrock. Time-shared systems: A theoretical treatment. *Journal of the ACM (JACM)*, 14(2):242–261, 1967.

[81] Elie Krevat, Vijay Vasudevan, Amar Phanishayee, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems. In *Proc. Petascale Data Storage Workshop at Supercomputing'07*, November 2007.

[82] Mirja Kühlewind, Richard Scheffenegger, and Bob Briscoe. Rfc 7560: Problem statement and requirements for increased accuracy in explicit congestion notification (ecn) feedback. https://tools.ietf.org/html/rfc7560, 2015.

[83] Mirja Kühlewind, David P Wagner, Juan Manuel Reyes Espinosa, and Bob Briscoe. Using data center tcp (dctcp) in the internet. In *Globecom Workshops (GC Wkshps), 2014*, pages 583–588. IEEE, 2014.

[84] Prabhakar Lakhera. Your app and next generation networks. http://goo.gl/UEHYtq, 2015. Apple.

[85] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[86] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[87] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *USENIX ATC*, volume 15, 2015.

[88] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, 1985.

[89] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 3–13. IEEE, 2010.

[90] Greg Levin, Caitlin Sadowski, Ian Pye, and Scott Brandt. Sns: a simple model for understanding optimal hard real-time multi-processor scheduling. *Univ. of California, Tech. Rep. UCSCSOE-11-09*, 2009.

[91] Greg M. Levin. *Old And New Approaches To Optimal Real-time Multiprocessor Scheduling*. PhD thesis, University of California Santa Cruz, 2013.

[92] Caixue Lin, Tim Kaldewey, Anna Povzner, and Scott A. Brandt. Diverse soft real-time processing in an integrated system. pages 369–378, Rio de Janeiro, Brazil, December 2006.

[93] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[94] Yong Liu, Francesco LoPresti, Vishal Misra, and Don Towsley. Fluid models and solutions for large-scale ip networks. In *Proceedings of ACM Sigmetrics*, San Diego, CA, 2003.

[95] J. Loeser and H. Haertig. Low-latency hard real-time communication over switched ethernet, 2004.

[96] Miao Luo, Dhabaleswar K Panda, Khaled Z Ibrahim, and Costin Iancu. Congestion avoidance on manycore high performance computing systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 121–132. ACM, 2012.

[97] German Maglione-Mathey, Pedro Yebenes, Jesus Escudero-Sahuquillo, Pedro J Garcia, and Francisco J Quiles. Combining openfabrics software and simulation tools for modeling infiniband-based interconnection networks. In *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pages 55–58. IEEE, 2016.

[98] Ernesto Massa, George Lima, and Paul Regnier. Revealing the secrets of run and qps: New trends for optimal real-time multiprocessor scheduling. In *2014 Brazilian Symposium on Computing Systems Engineering*, pages 150–155. IEEE, 2014.

[99] Ernesto Massa, George Lima, Paul Regnier, Greg Levin, and Scott Brandt. Optimal and adaptive multiprocessor real-time scheduling: the quasi-partitioning approach. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 291–300. IEEE, 2014.

[100] Matthew Mathis. Relentless congestion control. In *Proc. PFLDNeT*, 2009.

[101] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, 1997.

[102] J Clerk Maxwell. On governors. *Proceedings of the Royal Society of London*, 16:270–283, 1867.

[103] Steven McCanne, Sally Floyd, Kevin Fall, Kannan Varadhan, et al. Network simulator ns-2. http://www.isi.edu/nsnam/ns/, 1997.

[104] David Miller, Stephen Hemminger, et al. [patch] make cubic hystart more robust to rtt variations. http://thread.gmane.org/gmane.linux.network/188738/focus=188808, 2011.

[105] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 537–550. ACM, 2015.

[106] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15:217–252, 1997.

[107] Aloysius K. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-time Environment*. PhD thesis, Massachusetts Institute of Technology, May 1986.

[108] Jonathan Morton and David Taht. Cake - common applications kept enhanced. http://www.bufferbloat.net/projects/codel/wiki/CakeTechnical, 2016. http://www.bufferbloat.net/projects/codel/wiki/Cake.

[109] Paul R Muessig, Dennis R Laack, and John J Wrobleski. An integrated approach to evaluating simulation credibility. Technical report, DTIC Document, 2001.

[110] Achmad Munir, Ihsan Ayyub Qazi, Zartash Afzal Uzmi, Aleem Mushtaq, Saad N Ismail, M Safdar Iqbal, and Bilal Khan. Minimizing flow completion times in data centers. In *INFOCOM, 2013 Proceedings IEEE*, pages 2157–2165. IEEE, 2013.

[111] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 491–502. ACM, 2014.

[112] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[113] Angela Orebaugh, Gilbert Ramirez, Josh Burke, and Larry Pesce. *Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security)*. Syngress Publishing, 2006.

[114] Shawn Ostermann. Tcptrace official homepage, 2000.

[115] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling tcp throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 28(4):303–314, 1998.

[116] Scott Pakin. Conceptual: a network correctness and performance testing language. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 79. IEEE, 2004.

[117] Scott Pakin. Reproducible network benchmarks with conceptual. In *European Conference on Parallel Processing*, pages 64–71. Springer, 2004.

[118] Scott Pakin. The design and implementation of a domain-specific language for network performance testing. *IEEE Transactions on Parallel and Distributed Systems*, 18(10):1436–1449, 2007.

[119] Christina Parsa and J. J. Garcia-Luna-Aceves. Improving TCP congestion control over internets with heterogeneous transmission media. In *Proceedings of the 7th IEEE International Conference on Network Protocols (ICNP)*. IEEE, 1999.

[120] Craig Partridge. How to increase the chances your paper is accepted at acm sigcomm. *COMPUTER COMMUNICATION REVIEW*, 28:70–74, 1998.

[121] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[122] Roberto Pineiro, Kleoni Ioannidou, Scott A Brandt, and Carlos Maltzahn. Rad-flows: Buffering for predictable communication. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 23–33. IEEE, 2011.

[123] Jon Postel. Rfc 793: Transmission control protocol. usc. *Information Sciences Institute*, 27(793):123–150, 1981.

[124] Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 13–25, New York, NY, USA, 2008. ACM.

[125] Anna Povzner, Darren Sawyer, and Scott Brandt. Horizon: efficient deadline-driven disk i/o management for distributed storage systems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 1–12. ACM, 2010.

[126] Chromium Projects. Quic, a multiplexed stream transport over udp. https://www.chromium.org/quic.

[127] Chromium Projects. Quic faq for geeks. https://www.chromium.org/quic.

[128] Gaurav Raina, Don Towsley, and Damon Wischik. Part ii: Control theory for buffer sizing. *ACM SIGCOMM Computer Communication Review*, 35(3):79–82, 2005.

[129] Gaurav Raina and Damon Wischik. Buffer sizes for large multiplexers: Tcp queueing theory and instability analysis. In *Next Generation Internet Networks, 2005*, pages 173–180. IEEE, 2005.

[130] K Ramakrishnan, Sally Floyd, D Black, et al. Rfc 3168: The addition of explicit congestion notification (ecn) to ip. *Network Working Group, IETF*, (3168), 2001.

[131] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 104–115. IEEE, 2011.

[132] Paul D. E. Regnier. *Optimal Multiprocessor Real-time Scheduling Via Reduction To Uniprocessor*. PhD thesis, UFBA-UEFS-UNIFACS, 2012.

[133] Donald G Reinertsen. *The principles of product development flow: second generation lean product development*, volume 62. Celeritas Publishing, 2009.

[134] Donald G Reinertsen. Technical debt: Adding math to the metaphor, 2016.

[135] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference*, pages 18–18, Berkeley, CA, USA, 2001. USENIX Association.

[136] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[137] Linus E Schrage and Louis W Miller. The queue m/g/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[138] Pedro Yebenes Segura, Jesus Escudero-Sahuquillo, Crispin Gomez Requena, Pedro Javier Garcia, Francisco J Quiles, and Jose Duato. Bbq: a straightforward queuing scheme to reduce hol-blocking in high-performance hybrid networks. In *Euro-Par 2013 Parallel Processing*, pages 699–712. Springer, 2013.

[139] Bradley W Settlemyer, Stephen W Hodson, Jeffery A Kuehn, and Stephen W Poole. Confidence: Analyzing performance with empirical probabilities. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[140] Bradley W Settlemyer, Stephen W Hodson, Jeffery A Kuehn, and Stephen W Poole. Diagnosing anomalous network performance with confidence. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 612–613. IEEE Computer Society, 2011.

[141] Bradley W Settlemyer, Stephen W Hodson, Jeffery A Kuehn, and Stephen W Poole. Examining anomalous network performance with confidence. Technical report, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2011.

[142] Andrew Shewmaker, Carlos Maltzahn, Katia Obraczka, and Scott Brandt. Tcp inigo: Fighting congestion with both hands. Technical report, UC Santa Cruz, 2014.

[143] Andrew G Shewmaker. Investigating efficient real-time performance guarantees on storage networks. Master's thesis, UNIVERSITY OF CALIFORNIA SANTA CRUZ, 2009.

[144] Andrew G Shewmaker, Carlos Maltzahn, Scott Brandt, Katia Obraczka, and Ivo Jimenez. Run, fatboy, run: Applying the reduction to uniprocessor algorithm to other wide resources. Technical report, UC Santa Cruz, 2014.

[145] Andrew G Shewmaker, Carlos Maltzahn, Katia Obraczka, Scott Brandt, and John Bent. Tcp inigo: Ambidextrous congestion control. In *The 25th International Conference on Computer Communication and Networks (ICCCN 2016)*, 2016.

[146] Joel Sing and Ben Soh. Tcp new vegas: improving the performance of tcp vegas over high latency links. In *Fourth IEEE International Symposium on Network Computing and Applications*, pages 73–82. IEEE, 2005.

[147] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 479–490. ACM, 2014.

[148] Brent Stephens, Alan L Cox, Anubhav Singla, Jenny Carter, Colin Dixon, and Wes Felter. Practical dcb for improved data center networks. In *INFOCOM, 2014 Proceedings IEEE*, pages 1824–1832. IEEE, 2014.

[149] Ion Stoica, Hui Zhang, and TS Ng. *A hierarchical fair service curve algorithm for link-sharing, real-time and priority services*, volume 27. ACM, 1997.

[150] David Taht. Implementing comprehensive queue management on home routers. IETF, 2014.

[151] Kun Tan, Jingmin Song, Qian Zhang, and Murad Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings-IEEE INFOCOM*, 2006.

[152] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The tcp/udp bandwidth measurement tool. http://dast.nlanr.net/Projects, 2005.

[153] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[154] Iljitsch van Beijnum. Ipv6 celebrates its 20th birthday by reaching 10 percent deployment. http://arstechnica.com/business/2016/01/ipv6-celebrates-its-20th-birthday-by-reaching-10-percent-deployment/.

[155] András Varga et al. The omnet++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESM'2001)*, volume 9, page 65. sn, 2001.

[156] Jingyuan Wang, Yunjing Jiang, Yuanxin Ouyang, Chao Li, Zhang Xiong, and Junxia Cai. Tcp congestion control for wireless datacenters. *IEICE Electronics Express*, 10(12):20130349–20130349, 2013.

[157] W.carter. Red and blue pill as in the matrix. creative commons attribution-share alike 4.0 international. https://en.wikipedia.org/wiki/File:Red_and_blue_pill.jpg, 2014.

[158] Lonnie R Welch, David M Chelberg, and Barb Pfarr. Adaptive resource management technology for nasa computing systems.

[159] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[160] Philip Williams. Congestion in infiniband networks. 2007.

[161] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 50–61. ACM, 2011.

[162] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 123–134. ACM, 2013.

[163] Damon Wischik and Nick McKeown. Part i: Buffer sizes for core routers. *ACM SIGCOMM Computer Communication Review*, 35(3):75–78, 2005.

[164] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. Ictcp: Incast congestion control for tcp in data-center networks. *IEEE/ACM Transactions on Networking (TON)*, 21(2):345–358, 2013.

[165] Cheng Yuchung and Neal Cardwell. Rack: a time-based fast loss detection algorithm for tcp. https://tools.ietf.org/html/draft-cheng-tcpm-rack-01, 2016.

[166] Cheng Yuchung, Neal Cardwell, and Eric Dumazet. [patch net-next 2/7] tcp: track min rtt using windowed min-filter. http://thread.gmane.org/gmane.linux.network/383160/focus=383161, 2015.

[167] Marko Zec. Implementing a clonable network stack in the freebsd kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 137–150, 2003.

[168] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.