PUSHING TRANSPORT LAYER LATENCY DOWN TOWARDS ITS PHYSICAL LIMITS IN DATA CENTERS WITH PROGRAMMABLE ARCHITECTURES AND ALGORITHMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Serhat Arslan
May 2024

Abstract

Data center applications keep scaling horizontally across many machines to accommodate more users and data. This makes the communication performance requirements even more stringent, i.e., higher bandwidth and lower latency. The increasing link capacities address the bandwidth demands, but the latency requirements necessitate more sophisticated solutions.

In this thesis, I observe that **the transport layer is the only layer in the networking stack to impact latency both at the end-hosts and the network**. The way it handles packets sets the end-hosts processing delay. And its congestion control determines the queuing delay in the network. Hence, I study transport layer designs to push both latencies down to their physical limits.

First, I argue that **end-host latency can be minimized by offloading the transport layer to NIC hardware, but fixed-function chips prohibit custom solutions for diversified environments**. As a solution, I introduce nanoTransport, a programmable NIC architecture for message-based Remote Procedure Calls. It is programmed using the P4 language, making it easy to modify (or create) transport protocols while the packets are processed orders of magnitude faster than traditional software stacks. It identifies common events and primitive operations for a streamlined, modular, and programmable pipeline; including packetization, reassembly, timeouts, and packet generation, all expressed by the programmer.

Next, I argue that **network latency can only be minimized with quick and accurate congestion control decisions, which require precise congestion signals and the shortest control loop delay**. I present Bolt to address these requirements and push congestion control to its theoretical limits. Bolt is based on three core ideas, (*I*) Sub-RTT Control (SRC) reacts to congestion *faster* than one RTT, (*II*) Proactive Ramp-up (PRU) *foresees* flow completions to promptly occupy released bandwidth, and (*III*) Supply matching (SM) matches bandwidth demand with supply to maximize utilization. I show that these mechanisms reduce 99^{th}-p latency by 80% and improve 99^{th}-p flow completion time by up to 3× compared to Swift and HPCC even at 400Gb/s.

Acknowledgments

Writing a PhD thesis is a very long process that is full of ups and downs by its nature. Therefore, having people to support me throughout the entire journey was the absolute best thing I could hope for. As the famous Harvard Study of Adult Development[1] suggests, my warm relationships with these wonderful people will be more important than any technical work I can produce. It is now my honor to show how grateful I am to have them by my side as I write this thesis.

First of all, I would like to thank my advisors Nick McKeown and Sachin Katti. Despite things like the COVID-19 pandemic and their retirement from Stanford, I always felt that I could reach out and ask for any kind of help. There is certainly no limit to what one can learn from them.

It was also a privilege to have Balaji Prabhakar, John Ousterhout, and Chris Piech on my PhD committee. They have always intellectually inspired me to become a better researcher and engineer I am today.

Next, I was extremely lucky to have phenomenal lab mates who mentored me, eased the difficult times, and celebrated the good ones with me. I have shared so many unforgettable memories with Sundararajan Renganathan (my photo booth partner), Evgenya Pergament (my running partner), Bruce Spang (founder of our congestion control club), Stephen Ibanez, Alex Mallery, Theo Jepsen, Ali Abedi, Jenny Hong, and Catalin Voss. It has always been an amazing experience to watch them create groundbreaking research in our field, and I am confident that they will continue to do so in the future as well.

Creating an impact in the field would not be possible without my admirable collaborators Changhoon Kim, Muhammad Shahbaz, Nandita Dukkipati, Gautam Kumar, Yuliang Li, and Jeremias Blendin. The work you will read in this thesis exists thanks to their guidance, vision, and ability to answer my endless questions.

Another crucial piece of support was from our lab's admin Bisera Rakicevic who was always very

[1]https://www.lifespanresearch.org/harvard-study/

quick in responding to all sorts of logistical inquiries while bringing “börek” to the office every now and then to make us feel at home.

I cannot end this section without mentioning the people who defined my non-technical life during the PhD program. I am grateful for all the memories I had with Saffet Çakır, Bahadır Ünal, Koray Özdemir, Erkan Şen, Cem Aydın, Begüm Tuğlu, Alp Arıbal, Şebnem Özdemir Arıbal, Orçun Aysal, Fatma Yiğit Aysal, Mustafa Sezer Soysal, Saliha İspir Soysal, Serkan Genç, Merve Genç, Osman Soysal, Pınar Çelik Soysal, Armağan Öztürk, Ceren Küçükyurt, Bora Hamdullahpur, Eylül Bilgin, Arielle Anderer, Christian Kaps, Pia Ramchandani, Aneesh Rai, Katie Mehr, Tolga Dizdarer, Joseph Carlstein, Anna Helmke, Ilai Bistritz, Allen Zhao, Anton De Leon, Pedro Milani, Kaan Alp Yay, Berivan Işık, Erdem Bıyık, Beliz Günel, Meltem Tolunay, Melis Çakar, Cem Kesici, Anıl Kırcalıali, Anıl Kaplan, Mehmet Yalçın Aydın, Fatmanur Caygın Aydın, Utku Erol, Kemal Erol, and Sevil Erol. Without their companionship, I wouldn’t have the courage to move to another country and go through the rollercoaster of a PhD program.

Unquestionably, my parents Sinan Arslan and Gülten Arslan, along with my sister Sena Arslan have been immensely influential in how I think, interact with problems, and communicate with people. Without these skills, I wouldn’t even be admitted to any PhD program. Thanks to such an amazing family, I always felt safe and brave enough to explore new horizons in life.

Finally, the biggest acknowledgment belongs to my splendid wife Atiye Cansu Erol Arslan whom I had the honor of sharing my entire adult life so far. Sharing hobbies, friends, life challenges, and this PhD journey with her has always been a joy. I am grateful for her unwavering support for all the difficult times.

In the loving memory of my grandmother Fatma (Fatebe) Arslan who passed away while I was writing this thesis...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the creation of the Internet, continued advances in online technologies are shaping every aspect of our lives. Social media applications such as Facebook and Twitter have defined a new way we interact with each other. Video conferencing solutions like Zoom and Teams have enabled remote work arrangements. E-commerce sites such as Amazon or Alibaba have made it easier to consume goods and services. Digital streaming platforms like Netflix and HBO Max have changed the way we are entertained. Finally, artificial intelligence products like Siri or ChatGPT are helping us with our daily tasks or problems we face. The list can be extended much further.

Almost all such innovative applications have one thing in common: *Their software runs in data centers around the world.* This means they can scale much more easily compared to running on the personal computer of a person because data centers are comprised of millions of servers that are very powerful in computing. Developers simply need to design their application in a distributed way so that it can utilize multiple servers at a time. However, this is easier said than done.

The workloads in data centers constantly evolve towards larger-scale, highly parallel applications, and application developers expect the communication between distributed compute and storage nodes to keep up with the workloads. For example, machine learning models have grown so much that they can no longer fit in a single GPU server, and they need ever-increasing quantities of data for training [56]. As a consequence, various distributed training libraries have been developed. These libraries partition the model and/or training data across multiple servers and communicate intensively between those servers when necessary to aggregate and consolidate results [159].

As a consequence of distributed training, the demand for network capacity has increased immensely, with some arguing that it is now constrained by the communication data rate between servers [171]. There is therefore an enormous amount of investment taking place to increase line rates, as well as increasing the number of links, by increasing switch radix and thus the richness of interconnection. 100Gb/s links are already abundant, 200Gb/s is gaining adoption, and 400Gb/s as well as 800Gb/s Ethernet will be here soon [69].

In addition to high bandwidth, some applications need low latency communication [14]. In the context of networking, latency is the time it takes for a unit of data to travel from one point to another. For instance, consensus protocols require all participant nodes to receive and acknowledge the most recent message as quickly as possible in order to be able to reach a consensus [2]. Applications use such protocols for parallelism techniques such as memory coherency, efficient load-balancing, and in-order transaction processing.

In most cases, the performance of a low-latency application is determined by the *tail* in the system. Instead of the average or median, tail latency refers to the maximum or worst-case latency observed. In applications such as task scheduling [77], ML inference [28], and distributed sorting [75], it is important to ensure that even these outliers are within acceptable limits to provide a consistent and reliable user experience.

The tail latency becomes especially challenging for applications that typically send huge numbers of Remote Procedure Calls (RPCs) between large groups of servers [79, 154] and finish their processes only after they receive responses to all of the RPCs from the network. Accordingly, the Service Level Objectives (SLOs) for these applications tend to place stringent requirements on network performance.

In this dissertation, I introduce work that aims to minimize tail latency in data centers. Specifically, I present new primitives, mechanisms, and insights into different dynamics that generate delays in delivering data to remote nodes. To achieve this goal, I start by partitioning the communication latency into two parts and address each part individually.

The First part is the latency generated at the *end-host*. When data is emitted by an application for transmission, the underlying operating system processes this data along with any communication state. This process involves writing the data onto the network interface card, dividing it into small chunks with packet encapsulations, and keeping track of the state for the communication protocols (e.g., IP, TCP). Similarly, on the receive side, the end-host processes the incoming data to figure out the receiving application, writes the data into the memory of the correct core, and transmits control signals such as acknowledgments.

All this end-host processing takes a non-negligible time before the data can be serialized into the network or handed to the receiving application. For instance, sending 1KB of data, including packet headers, over a 100Gb/s network with a single non-congested switch between the end-hosts would incur the following latencies: The packets would take 80ns to be serialized onto the links at each hop – the sender and the switch – a total of 160ns. Assuming 150-foot-long fiber-optic cables, the propagation of the data over the links would take around 150ns per link, a total of 300ns. Then, with a rough estimate of 500ns packet processing latency in high-speed switches, the total networking latency in this case would be 960ns. Therefore, even a fast end-host kernel latency of 5μs [128] at the sender and the receiver would constitute over 91% of the one-way latency for the data transfer between application threads. Note that propagation latency is purely governed by the speed of light, and increasing data rate only helps serialization latency, but not the others.

The second part is the latency experienced in the *network*. This latency typically includes queuing delay at the congested switches in addition to serialization, propagation, and switch processing latencies. When the instantaneous demand for a link exceeds its capacity, switches store the incoming packets in buffers until they can forward them through this congested link, constituting the queuing delay. Depending on the size of the buffers and the instantaneous demand, this queuing time can sometimes be as high as tens of microseconds.

Clearly, minimizing overall RPC response time as the bandwidth demands increase depends on minimizing latency in both the end-host and the network. The transport layer of the data center networking stack has great potential for this goal as it can affect both the end-host packet processing latency and the in-network congestion. Hence, the work presented in this dissertation dissects the transport layer and explores ways in which the latency can be minimized. In particular, two strategies are pursued: (*I*) The Use of NICs at the end-host that run transport protocols in hardware, for minimal processing time; and (*II*) The use of congestion control algorithms that are smart, for minimal queuing delays as the packet traverses the network. Next, I discuss in detail how these strategies can minimize latency.

1.1 End-Host Latency and Hardware Offloading

To reduce processing time on the end-host, many proposals have focused on redesigning the network interface card (NIC) (commercial products [32, 36, 112, 113, 125, 132, 166] and research proposals [8, 45, 58, 62, 79, 83, 93, 98, 99, 104, 107, 154]). Indeed, latency can be minimized by avoiding OS-specific, mostly non-deterministic, software overheads and placing the transport layer in hardware. For example, eRPC [79] is a software design that combines many software techniques to reduce median RPC response times only down to $1 - 2\mu$s whereas NeBuLa [154] is a hardware design that reduces RPC response time below 100ns by integrating the high-speed NIC with the CPU, bypassing PCIe, and placing arriving RPC requests directly into the L1 cache.

The current fastest reported combination of a whole system — a low-latency NIC with a transport layer — is the nanoPU [62] designed by the McKeown Group at Stanford University which I am also a part of. Instead of placing incoming RPC data into the L1 cache, nanoPU introduces a direct message interface to the CPU register file, so that applications can access the incoming data even faster. To do so, it proposes a thread scheduler and a fixed transport layer on hardware that can process incoming packets before writing the correct piece of data onto the correct register files on the CPU cores. As a result, it achieves a 69ns median wire-to-wire RPC response time with a 7ns of one-way transport layer latency. This may be tempting to think that there is not much room to reduce latency further since the nanoPU processes packets entirely in hardware, right up until the RPC request starts processing in a thread. Yet, evolution in the network continues (i.e., increasing line rates and emerging applications), which means the end-hosts will likely need different networking protocols in the future for optimal performance. And, if the transport protocol is baked into fixed-function hardware, it will be an expensive and time-consuming task to modify it.

The design of the congestion control algorithm is the part of the transport layer that has the most significant consequences for network latency as the workloads and infrastructure evolve. The jury is still out as to which algorithm is the best, but there may not be a single *best* algorithm; rather, the notion of the best likely depends on the particular data center topology and the specific distributed application [142]. For example, §3.4.2 presents scenarios where NDP and Homa are each better than the other. Consequently, the minimum latency can be sustained only in environments where network owners can frequently change/update the transport protocol to minimize congestion delays of the evolving traffic patterns in their unique networks. Hence, a *programmable* hardware is the ultimate prerequisite for the task at hand.

1.2 Network Latency and Congestion Control

Once the packet processing latency at the end-hosts is minimized, the next task is to determine what it takes to minimize queuing delays in the network. For this purpose, many proposals designed new low-latency transport layer protocols [5, 6, 13, 29, 49, 57, 54, 119] and congestion control algorithms [1, 4, 91, 102, 117, 175] that utilize unique congestion signals and mitigation mechanisms. For example, NDP [54] mitigates incast congestion by trimming off a packet's data in congested switches, sending only the header to the receiver, and allowing the receiver to decide when the packet should be resent. In contrast, DCQCN [175] relies on Explicit Congestion Notifications (ECN) marked on packets to figure out congestion in the network. The DCQCN senders set their transmission rate based on the frequency of reflected congestion notifications.

One factor that determines how accurate the congestion can be mitigated is the granularity of the congestion signals used by the algorithm. Most congestion control algorithms rely on end-to-end signals to figure out how congested the network is. For instance, the only congestion signal for TCP used to be the absence of delivery acknowledgments, i.e., packet drop; a binary signal that kicks in only after the congestion is well underway. Plus, a packet drop may also capture events that are not related to congestion, e.g., link failures. These cases may lead to incomplete information at the senders about the state of congestion in the network.

More recently, the trend has been towards using Round Trip Time (RTT) as a congestion signal instead (e.g., Timely [117], Swift [91] and BBR [25]). But RTT is also a noisy surrogate for congestion; it contains a valuable signal about congestion at the bottleneck but also includes noise from the queuing delay at the non-bottleneck switches.

Taking a step back, it is worth asking: *Why don't the switches and routers simply tell the actual congestion they are experiencing?* After all, they must keep track of the precise occupancy of their queues anyway; they can directly tell the end-hosts. Conventional wisdom used to say that extracting queue occupancy information from the switches is too expensive in terms of additional bits in headers, or complexity and power consumption. Today, having switches report this precise value is quite feasible, with negligible increase in power or capacity loss, allowing the end-hosts to make more accurate decisions when minimizing switch buffering. Hence, utilizing such *precise congestion signals* for making higher quality decisions over emerging workloads is a prerequisite to satisfying SLOs of modern distributed applications.

For instance, HPCC [102] is designed to collect queue occupancy information from the switches using Inband Network Telemetry (INT) [88]. The collected information is used to calculate the link utilization over the last RTT. If the link is found to be underutilized, the sender increases its congestion window size (`cwnd`) to send more packets in the next RTT. Otherwise, `cwnd` is reduced to allow the congested switch drain its queue throughout the next RTT.

Unfortunately, collecting accurate congestion signals is not sufficient to minimize tail latency in the network. Collecting such signals as early as possible is also very important in making timely decisions to mitigate congestion. For example, in §4.2 it is shown that HPCC's RTT-scale signal collection and decision-making make the algorithm less stable during flow arrivals.

Moreover, most congestion control algorithms are designed to linearly increase their `cwnd` in the absence of a congestion signal to capture available bandwidth, if any. This feature is also used for fairness among senders in the network [176]. However, this routine `cwnd` increase implies that senders continue transmitting more and more data towards the congested bottleneck until the congestion notification is received. Therefore, the notification should be delivered to the senders as soon as possible to prevent larger amounts of congestion in the network.

The ever-increasing line rates in the data centers raise the bar for how early should the congestion notification be delivered to the senders. Higher line rates imply that more data can be sent before receiving any feedback about the state of congestion. When the earliest feedback is delivered after 1 RTT, the amount of data sent before the first feedback received is calculated as the Bandwidth-Delay-Product (BDP). Higher bandwidth increases the BDP, making more flows fit within a few BDPs. Such short flows tend to be extremely sensitive to congestion because any queuing in the network can delay their completion time significantly compared to a non-congested network case. Ironically, since these flows are short, they also leave less time for congestion control to kick in, inducing burstiness [164, 170].

For instance, current congestion control algorithms cannot even chime in for RPCs smaller than a BDP because the feedback is inherently delayed by one RTT, putting more pressure on sustaining SLOs. These SLOs are likely to evolve and become even more challenging for upcoming workloads such as disaggregated memory and ML. Hence, *sub-RTT feedback loops* are the ultimate requirement for controlling queuing and under-utilization at high line rates.

1.3 In This Dissertation

This thesis presents novel solutions to both fronts (§1.1 and §1.2) for reducing latencies in data centers. Before diving into these solutions, however, more background on transport layer protocols and congestion control algorithms – along with a discussion of related work – is provided in Chapter 2.

Next, Chapter 3 introduces NanoTransport, a hardware design for minimal transport layer latency without losing programmability. It is designed to run in an ASIC and programmed (in the field) using the P4 language, which can achieve ∼10× faster packet processing compared to FPGAs. P4 pipelines are already used in modern commercial NICs [112, 132, 166], and an industry group is creating a standard portable architecture for P4-programmable smart NICs [44]. Inspired by these architectures, nanoTransport lists a common set of triggering events (e.g., packet arrival, timeouts, duplicate ack) that are shared by a wide range of transport protocols. These events are exposed to developers via P4's simpler, widely accepted abstractions, which enable developers to trigger them in a programmable fashion. The resulting design processes incoming and outgoing packets in fast PISA pipelines with nanosecond-scale median and tail latencies, offloading work from the CPU.

Then, Chapter 4 introduces Bolt, a congestion control algorithm that harnesses the power of programmable data planes to design an extremely precise congestion control mechanism for ultra-low latency at very high line rates. It collects congestion feedback with absolute minimum (sub-RTT) delay and ramps up flows proactively to occupy available bandwidth promptly. To achieve this, it applies the "packet conservation" principle [70] to the traffic with accurate per-packet decisions in P4 [23]. Small, per-packet `cwnd` changes, combined with the fine-grained in-network telemetry, help limit the effects of noise in the instantaneous congestion signal. With Bolt, end-hosts do not make implicit estimations about the severity and exact location of the congestion or the number of competing flows, freeing them from manually tuned, hard-coded parameters, and inaccurate reactions.

Finally, a broad discussion of the future directions in low-latency transport layer research along with the concluding remarks is provided in Chapter 5.

Chapter 2

Background and Related Work

The transport layer is the interface that regulates the interactions between the applications and the network. However, little research focused on it until the infamous Internet congestion collapses reported in 1988 [70]. In fact, that same year, Clarke's retrospective on the Internet design [33] surprisingly did not even mention congestion. Instead, he highlighted the benefits of not assuming reliable delivery in the network. There was clearly a desire to keep the network simple and streamlined, and little attention was paid to congestion or packet loss.

It soon became clear that the demand for applications with reliable delivery was increasing, i.e., file transfer, and the unreliable datagram service of the Internet Protocol (IP) would not be enough for this demand. This led to the redesign of the transport layer so that it could handle higher traffic volumes while meeting higher performance standards at both end-hosts and the network.

In this chapter, I describe the purpose of the transport layer, various implementations, and techniques to achieve high performance. I also summarize the vast array of proposed congestion control algorithms and transport protocols. I compare and contrast many proposals, focusing on different approaches to signal congestion and methods to mitigate it in the network. I finish by summarizing the limitations and weaknesses of existing work, which led to the designs presented in Chapter 3 and Chapter 4 that drive transport layer latency down towards its physical lower bound.

2.1 The Transport Layer

The transport layer is one of the layers in the OSI (Open Systems Interconnection) [37] and the TCP/IP (Transmission Control Protocol/Internet Protocol) [26] models, conceptual frameworks that help us to understand and design computer networks. This layer plays a crucial role in distributed systems by providing end-to-end communication services for applications running across a network.

The primary functions of the transport layer are

1. *End-to-End Communication:* The transport layer establishes, maintains, and terminates connections between two devices; typically identified by IP addresses and port numbers. It can also create connectionless communication (e.g., via UDP - User Datagram Protocol [133]) in which data is sent as independent datagrams.

2. *Segmentation and Reassembly:* Large messages or data streams are divided into smaller segments at the sender for efficient transmission. Then, these segments are reassembled into the original message at the receiver before being delivered to the application thread.

3. *Reliability:* For applications that require reliable and accurate data delivery, protocols like TCP (Transmission Control Protocol) [19] ensure that data is delivered to the application correctly and in the correct order. To do this, the sender assigns monotonically increasing sequence numbers to each segment, allowing the receiver to identify missing segments or to deliver the segments to the application in the correct order.

4. *Flow Control:* The transport layer ensures that data is sent at a rate the receiver can handle. If more data arrives than the receiving application can consume, the receiver can get overwhelmed and drop data. The transport layer prevents this with a backpressure signal from the receiver to the sender in which it communicates how much space it has to hold new data, called the *receive window (`rwnd`)*.

5. *Congestion Control:* Similar to flow control, the transport layer also ensures that the amount of data sent does not overwhelm the packet buffers in the switches and routers along the network path. For this purpose, the *congestion window (`cwnd`)* identifies how much data can be outstanding; i.e., how much as-yet unacknowledged data can be sent. Based on the congestion signals collected from the network, the `cwnd` value is readjusted either to reduce the amount of outstanding data or to increase it. However, the actual amount of outstanding data is the smaller value among the `rwnd` and `cwnd`, to avoid overwhelming neither the network nor the destination host. Note that maintaining `rwnd` and `cwnd` values separately is just a design choice for protocols like TCP. Some protocols merge the two concepts into a single state variable whose value is updated both by the congestion signals and the receiver's overload feedback, e.g., Homa [119] and NDP [54].

6. *Error Detection and Correction:* The transport layer is also responsible for error detection and, in some cases, correction. It uses mechanisms such as checksums to verify the integrity of the data during transmission.

Although these functions were originally implemented in software, it is increasingly common for Network Interface Cards (NICs) to offload a subset of these functions into hardware. Running some functionality on the hardware usually helps gain performance benefits and free up precious CPU cycles. However, hardware implementations typically bake the protocols on the chip, making it hard and expensive to modify or update them when requirements change or bugs are discovered. Hence, the battle between software-based and hardware-based designs continues with middle-ground solutions, e.g., programmable SmartNICs [144].

2.1.1 Software-Based Designs

The default approach to running the packet processing and the transport layer has long been through kernel implementations in software, e.g., Linux kernel [109] or 4.2BSD [95]. However, it has often been noted that the default implementations are quite complex and slow. For example, the standard Linux kernel TCP implementation takes 50μs (median) to deliver an RPC to the application thread [174]. More recently, the latency demands of data centers have led to new, simpler transport protocols such as Homa, which can deliver an incoming message from the NIC to a Linux user thread in just 5μs [128]. But to put this in context, end-host processing would still be 90% of the latency incurred by a 1KB datagram sent over a 100Gb/s network (as discussed in Chapter 1). Therefore, there has been a great interest in building lower-latency networking stacks in data centers.

As latency requirements have become more stringent [80, 129], it has become popular to bypass the kernel altogether and run the transport layer in user space [16, 39, 73, 79, 84, 110, 127, 130, 134, 174]. In this approach, the packet processing latency is reduced with techniques such as lock-free [16] and zero-copy buffers [16, 97], polling and batching [16, 73], and cache planning [84] using NIC drivers like DPDK [11].

For example, eRPC [79] carefully optimizes for the common case and reports 850ns wire-to-wire latency for small 32-Byte RPCs with the Timely [117] congestion control algorithm. Although these optimizations help eRPC in the best-case scenario, the tail response time struggles during high-load scenarios.

Alternatively, eBPF – the extended Berkeley Packet Filter [157] – is a technology that allows the injection of custom code into the Linux kernel to programmatically analyze and modify kernel-level events and data. It allows users to attach small programs to certain hooks in the networking stack and filter or modify packets with low processing overheads [55, 78, 173].

Despite the extensive optimization efforts, all of the proposals above remain north of a microsecond latency threshold with MTU-size packets in reliable connections. This is mainly due to the overhead of processing the packets on general-purpose CPU cores [24]. In contrast, domain-specific accelerators (i.e., SmartNICs) can free up cycles from the main processor and reduce the time to deliver data to/from the application threads from/to the wire. The performance benefits of these hardware accelerators are vital for the stringent requirements of dynamic workloads.

2.1.2 Hardware-Based Designs

Offloading the transport layer onto hardware enables packet processing at the line rate while the main processor focuses only on the application processing. In addition, processing latency becomes fixed for each packet, significantly reducing the tail latency. The most common use case for such a high-performance system is the Remote Direct Memory Access (RDMA) [146]. The RDMA verbs read or write data to/from a remote device's memory without intermediate copy operations. To support this use case, modern NICs typically support RDMA over Converged Ethernet (RoCE) [111] at the transport layer, which implements DCQCN [175] as the only congestion control algorithm.

An alternative to RoCE is to offload the TCP stack onto NICs. In such designs, either a subset of the functionalities [48, 83, 118, 120] or the entire stack [40, 89, 148, 165] can be offloaded. For instance, AccelTCP [120] leverages programmable NICs to accelerate only the TCP connection setup and teardown operations. In contrast, Dagger [93] offloads the entire RPC stack while running only a UDP-based transport on FPGA.

Although hardware offloading solutions repeatedly report nanosecond-scale latencies, they usually pick a certain transport protocol and a congestion control algorithm. Yet, hundreds many algorithms and protocols have been proposed in the literature, and each one optimizes performance in different scenarios, i.e., incast with extremely short flows vs. distributed ML training with long-lived flows. Then, how can hardware architects decide which algorithm is the best to bake on the chip?

Making the NIC programmable allows network owners to decide which protocol and algorithm to implement on the hardware so that they can optimize performance specifically for their workload. To enable programmability, smartNICs typically utilize extra processors on board [104], but this solution creates a bump in the wire, adding extra cycles for the NIC interconnect.

In contrast, Tonic [8] and FlexTOE [148] propose partial programmability on the data path. However, these proposals require Verilog [64] and eBPF [157] programming, respectively, which brings extra difficulty in implementing congestion control policies. Instead, P4 [23] – a widely accepted domain-specific language – is designed specifically for expressing packet-processing behavior in a much simpler way. It empowers networking researchers and developers to experiment with new features and protocols both at the sender and receiver as quickly as possible. In addition, research on verifying packet processing using P4 [41, 106, 126, 152] is actively underway. Hence, leveraging this simple and widely accepted language promises rapid hardware offloading of custom algorithms or protocols in a flexible way while reducing transport latency down toward its physical limits. The work in Chapter 3 leverages this simplicity, and wide adoption of the P4 language to enable rapid hardware offloading of custom algorithms or protocols while being inspired by the primitives described in previous work on programmable transport layers.

2.2 Congestion Control Algorithms and Transport Protocols

The congestion control research community has been very active since the first paper on this topic was published in 1988 [70]. Every year, many new algorithms are proposed to improve the performance of modern workloads. It is fair to say that no clear winner has emerged, in large part because of the evolving requirements of workloads, including flow and message sizes, link bandwidths, latency between end-hosts, buffer sizes, fan-in/fan-out ratios, etc.

In addition, a network operator may have different performance objectives based on the type of application being prioritized over the network. For example, an application that runs many long-lived low-priority flows, such as backup traffic, may try to optimize the average throughput achieved. On the other hand, a lightweight application with many small messages such as financial trading would focus more on the median or tail latency. As the networks and workloads evolve, it would be reasonable to expect more algorithms and protocols to emerge, making it even more difficult to reach a global consensus.

At a high level, the existing congestion control algorithms can be classified based on the entity that drives them. Specifically, traditional congestion control algorithms rely on the senders to collect congestion signals from the network – usually reflected by the receivers – to decide on the `cwnd` size or the transmission rate. In contrast, there is a relatively newer approach for congestion control where the receiver selects senders for transmission based on the load at the receiver. Moreover, switches can decide on scheduling, congestion signaling, or pausing specific flows as well to manage congestion in the network. Below, I discuss the underlying mechanisms, assumptions, and weaknesses of these approaches.

2.2.1 Sender-Driven Algorithms

Sender-driven congestion control algorithms rely on senders to collect a congestion signal from the network. These signals are usually performance measurements or events that take place in the network, which typically accumulate at the receiver in the forward direction and are reflected to the sender. There are three popular signals in use today:

1. *Packet Loss*, i.e., the absence of an acknowledgment from the receiver. TCP Reno [52] attributes this signal to a packet drop at a switch due to congested buffers and cuts the `cwnd` size by half to allow the switch drain the congested queue. TCP CUBIC [137], the default algorithm in Linux environments, also uses this signal but differs from Reno in how `cwnd` is increased when no packet loss is observed.

2. *Explicit Congestion Notification (ECN)*, i.e., single bit on packet headers that are marked by switches whose buffer occupancy is above a threshold. Algorithms such as DCTCP [4] and Microsoft's DCQCN [175] calculate the frequency of incoming ECN markings to infer how congested switches are. Then, they reduce `cwnd` or transmission rate, respectively, until this frequency drops down to a certain level.

3. *Round Trip Time (RTT)*, i.e., the time it takes for the acknowledgment of a data packet to come back to the sender. Since this value increases when the network is congested, the senders of algorithms like Swift [91], BBR [25], Timely [117], and TCP Vegas [21] use the changes in RTT to infer the amount of congestion in the network.

All three signals are noisy, imprecise *surrogates*, and fail to accurately capture the amount and location of the congestion in the network. These shortcomings constitute a significant barrier to developing a high-throughput, low-latency congestion control algorithm for data centers. For instance, a packet drop is an event that takes place only after the congestion is well underway and the latency is at its maximum, which is too late for low-latency applications. In addition, packets may be dropped due to equipment failure, which adds noise to the congestion signaling. Although ECN can signal increasing queue occupancy before reaching the maximum, it is just a binary signal and, as such, does not convey how congested the switch is apart from being above the threshold.

RTT is very effective at capturing the queuing delay at the bottleneck accurately, but it also captures non-bottleneck delays experienced between the two end-hosts. To minimize this effect, Google uses extremely precise hardware timestamps on the end-hosts and avoids inadvertently measuring end-host processing latency [91, 110]. However, this technique is not sufficient to avoid measuring delays at the non-bottleneck switches in the network. Measuring both the bottleneck and non-bottleneck delays as the congestion signal prevents senders from accurately determining how much to slow down when high RTT is observed. As a result, operators heuristically tune algorithm parameters that regulate how much the `cwnd` is reduced [21, 91, 117]. This is a time-consuming process and needs to be repeated every time the algorithm is run in a different environment or with a different workload.

To overcome the limitations of surrogate signals, HPCC [102] proposed using INT (Inband Network Telemetry) [88] which can carry the most accurate congestion information possible. With INT, the instantaneous port state is reported precisely by every switch the data packet traverses. The INT header includes packet timestamps, some counter values, and queue occupancy, which is *the only direct measure of congestion* by definition. It can even be optimized to be collected only from the bottleneck to reduce the number of bytes reserved for congestion signaling on the packet header [17].

HPCC uses the INT information to calculate the utilization of each link every RTT so that congestion reaction simply becomes `cwnd` scaling to match the utilization target. Since it aims to set `cwnd` to its fair share at once, redundant signaling is prevented by calculating utilization only once every RTT over the RTT-wide aggregated telemetry data. However, using aggregated data as the signal can create a mismatch between the actual utilization and the measured one in a data center, where congestion-inducing events (i.e., flow arrivals or completions) happen almost every RTT.

Similarly, Poseidon [161] replaces RTT with the INT data for the congestion signal in Swift's algorithm. By extracting the bottleneck queue occupancy, Poseidon senders compare this accurate signal to a target occupancy and apply the Additive Increase, Multiplicative Decrease (AIMD) method appropriately on the `cwnd` until the target is matched. Although this is a great advancement in the collection of precise congestion signals, Poseidon collects these signals through the acknowledgments reflected by the receiver. Therefore the feedback cannot be collected in less than one RTT similar to Swift and HPCC, which makes it difficult to react to congestion promptly as discussed in Chapter 1.

Several other approaches have similar ideas about how to collect precise congestion signals and reduce feedback delay. For example, XCP [82] and RCP [42] also propose congestion feedback generated by the switch. Switches wait for an average RTT before calculating congestion control responses, which are stamped on the data packet and reflected on the ACK by receivers. However, this approach imposes a control loop delay that is at least one RTT as well. The Sub-RTT Control (SRC) mechanism motivated in §4.2 reduces this delay and helps senders make more frequent congestion control decisions with high granularity.

FastTune [172] attempts to shorten the feedback delay as well. Similar to HPCC, it uses programmable switches for precise congestion signals and calculates link utilization over an RTT to multiplicatively increase or decrease `cwnd`. For shorter feedback delay, it pads the INT header onto ACK packets in the reverse direction rather than the original data packet. ExpressPass [29] utilizes the control packets in the reverse direction too, similar to how Backward ECN (BECN) works [3]. However, forward and reverse paths for a flow are not always symmetrical due to ECMP-like load balancing or flow-reroutes in data centers. This makes approaches like ExpressPass and BECN less practical for real-world deployments.

FastLane [168] sends notifications from switches directly back to the senders with dedicated control packets. This prevents the congestion notification from being delayed by the congestion itself. However, notifications are generated only with packet drops, and the timing of this event is too late for low latency congestion control in data centers. In contrast, Annulus [140] uses standard QCN [65] packets from switches with quantized queue occupancy information as soon as the queue occupancy exceeds a threshold. Since these packets are not L3 routable, Annulus limits its scope to detecting bottlenecks only one hop away from senders. Exposing precise queue occupancy information like QCN, but from every switch in the network is an obvious next step.

In addition to efficient feedback delivery mechanisms, the content of the feedback also determines how well senders can mitigate congestion. For example, FCP [53] uses budgets and prices to balance the load and the traffic demand. Switches calculate the price of the link based on the demand, while senders declare flow arrivals or completions. However, the required time series averaging and floating-point arithmetic make the switch calculation infeasible for packet processing at a high line rate. Instead, §4.3.3 presents an instantaneous load measurement technique as a much more practical approach, using simple P4 primitives without a time series state and complex arithmetic.

Finally, switch feedback has been studied for wireless settings as well. For instance, ABC [50] access points (APs) mark packets for `cwnd` increments or decrements with an RTT-based control loop. In contrast, Zhuge [114] modifies the wireless APs to inject congestion signals onto packets in the reverse direction so that senders can detect congestion in less than one RTT. Adapting similar mechanisms feasibly for dynamic data center workloads is a promising direction for congestion control.

2.2.2 Receiver-Driven Algorithms

In addition to the congestion control algorithms discussed above, there are also receiver-driven approaches such as NDP [54], pHost [49], and Homa [119]. They differ from sender-driven approaches in that they require receivers to allocate/schedule credits based on the demand from senders.

For example, NDP [54] is a protocol that aims to reduce the tail latency of network messages by minimizing packet drops and having dropped packets retransmitted in a timely fashion. The NDP receiver explicitly sends PULL packets to allow a sender to transmit a data packet. Those PULL packets are paced such that the allowed packets would arrive at the exact rate of the bottleneck link bandwidth. In the case of multiple messages being transmitted at the same time, each arriving data packet would trigger a PULL for the corresponding message so that new data packets are allowed in a round-robin fashion among messages. This approach is based on the assumption that if a data packet leaves the network, a new one can be inserted without overwhelming it.

For retransmission acceleration, NDP uses the concept of "packet trimming" at the switches, in which data packets that do not fit in the bottleneck queue are trimmed and the headers are forwarded to the receiver with high priority. The receiver then quickly sends negative acknowledgments (NACK) to let the sender know about this loss. This mechanism prevents relying on long timeouts.

In contrast, Homa [119], published a year after NDP, argues that packet loss is not a major concern in modern data centers because of the large and dynamically shared buffer spaces in switches [128]. Instead, the authors highlight the need for smart message scheduling at the end hosts, i.e., SRPT [143], that optimizes the tail latency [123]. Therefore, instead of a round-robin pulling mechanism, the Homa receiver sends GRANT packets to the message with the smallest remaining size no matter which message the incoming data packet belongs to. When an RPC is fully granted, the Homa receiver starts sending grants for the next RPC before the current RPC is finished. This approach proactively utilizes the link.

Moreover, Homa authors claim that packet trimming is an impractical feature for commercial fixed-function switching ASICs. Instead of trimming, Homa uses priority queuing primitives that are already available in modern networks. Smaller messages thus have a higher priority in the network, allowing them to be completed sooner.

Schemes that use priority queues on switches have been proposed to improve the scheduling performance of the network by approximating SRPT-like behavior [6, 13, 57, 119]. Alternatively, R2P2 [90] approximates the join-bounded-shortest-queue (JBSQ) policy on switches to load-balance incoming RPCs to the most available server. These approaches work well for managing congestion at the last hop because receivers and Top-of-Rack (ToR) switches have good visibility into this link. Unfortunately, the last hop is not always the bottleneck for a flow, especially when the fabric is over-subscribed [149].

In addition to packet scheduling, there are also receiver-driven RPC scheduling algorithms, such as Breakwater [30], that avoid overloading in low latency services by issuing credits to RPCs based on receiver-side queuing delay. Such overload control mechanisms prevent senders from transmitting RPCs that will not get credit for the transmission of the entire RPC at once. However, they add extra network latency for short RPCs whereas these RPCs could have been immediately transmitted without waiting for one RTT to receive transmission credit.

2.2.3 Switch-Driven Algorithms

The entity that decides when to send packets or RPCs is not always the sender or the receiver. There are also per-hop flow control mechanisms such as BFC [51] and PFFC [160] that pause queues at the upstream switches when a bottleneck gets congested. This early notification mechanism prevents the bottleneck queue from overflowing, reducing the chances of packet loss and network congestion.

Queue pausing is also used in conjunction with Quality of Service (QoS) mechanisms to prioritize certain types of traffic over others. By pausing or slowing down lower-priority queues during times of congestion, higher-priority traffic (such as real-time applications or critical data) can be given precedence.

While queue pausing can be a useful mechanism in managing network congestion and optimizing performance, it has potential disadvantages. For example, in scenarios where congestion is short-lived and sporadic, queue pausing may not be effective. Pausing queues in response to brief congestion periods may lead to unnecessary delays and decreased throughput during normal network conditions.

More importantly, queue pausing may lead to head-of-line blocking or deadlock issues, which are well demonstrated for PFC [66]. If a switch link is paused by one of the downstream switches, all the packets using this link are paused even if they are destined for a different switch after the next hop. Such issues are resolved by keeping the per-flow state on switches in BFC and PFFC, but this can be overwhelming in terms of the memory requirements per switch in a data center.

2.3 Summary and Remarks

Internet congestion occurs when switch or end-host buffers are overrun. However, no provisions were made for the switches to signal congestion levels in the early days of the Internet because it was not originally designed with congestion in mind. And so when TCP congestion control was first deployed, it used packet loss as its signal because this did not require a change to the switches.

Obviously, packet loss is not a good enough signal for modern data center workloads. It is generally too coarse and arrives too late. Instead, there is a much more accurate and direct signal, i.e., *the current occupancy of the buffers where the congestion takes place.* Switches must keep track of this value for their bookkeeping anyway and the newer ones can expose it to end-hosts as well. The availability of this precise signal presents a new opportunity, which is exploited in the work presented in Chapter 4.

The next step is to deliver the signal to the decision-makers as quickly as possible. With traditional congestion control methods, the earliest a sender can react to congestion is one RTT after it occurs. Although prior work has attempted to generate sub-RTT feedback, it fails to extract this signal with low overhead and at high speeds. Moreover, a theoretical framework for understanding the benefits of sub-RTT feedback is lacking in the literature. This gap is addressed in Chapter 4 where delivering a precise congestion signal in less than one RTT is analyzed in depth.

As workloads evolve, there is no doubt that new, improved congestion control algorithms will be developed. However, it will be hard to deploy them if the networking equipment is fixed-function. Many NICs and switches have recently become more programmable and this trend is expected to continue [27]. Hence, there is an opportunity to create programmable transport layers, which can be programmed in the field, allowing new and improved algorithms to be rapidly deployed with the lowest processing latency possible. In Chapter 3, I describe novel primitives to program the transport layer in the NICs.

Chapter 3

Programmable NICs for Lower Transport Layer Latency

Transport protocols can be implemented in the Network Interface Card (NIC) to increase throughput, reduce latency, and free up CPU cycles. If the ideal transport protocol were known, the optimal implementation would be simple: bake it into fixed-function hardware. However, transport layer protocols and data center workloads continue to evolve, with innovative algorithms and more demanding applications introduced every year.

One way in which each transport protocol differs is in the relative importance given to throughput and latency (median or tail). For example, many distributed applications launch multiple RPC requests to different servers at the same time. If they must wait for all of the RPCs to return before making progress, their overall performance would be dictated by the slowest RPC response time. Even if an RPC request returns a result quickly *on average*, there are usually some outlier events that cause the RPCs to return slow, e.g., congestion in the network or contention in the end-host memory. These unusually long response times are called the tail latency of the system. As the number of cascaded RPCs increases, the likelihood of a long tail latency increases, defining the end-to-end performance of the entire system [38]. Hence, the transport protocols for such applications are primarily expected to minimize tail latency.

The best transport layer mechanism to limit tail latency is likely best determined by a cloud service provider that has the best view of all the performance demands in its unique data center. However, if the transport protocol is baked into fixed-function hardware, it is a costly and time-consuming task to modify it when the performance demands change or new mechanisms are proposed. This issue can be avoided by making the hardware programmable such that the service providers can quickly change the packet processing logic while running it on high-speed domain-specific hardware [100, 104].

However, to run the hardware at the line rate, the packet processing pipeline must be as streamlined as possible with a minimal instruction set. Therefore, the smallest set of necessary packet processing operations should be carefully identified for high performance and enough generality. This observation prompted the authors of Tonic [8] to propose a *programmable* hardware design for transport protocols, which "exploits the common patterns in transport logic to create reusable high-speed hardware modules". Their design assumes the transport layer will be implemented on an FPGA and that the programmer will use Verilog [64] – a hardware-design language – to implement a new algorithm. Since Verilog requires a steep learning curve, the authors also provide an NS3 [138] model to help users design new protocol layers.

In Tonic's design, the kernel offloads the connection state and packet processing to the FPGA-based NIC *after* the CPU establishes a transport connection. Its prototype utilizes ring buffers and bitmaps to keep track of the connection state on hardware while achieving 100Gb/s with 128-Byte packets and processing a packet in about 100ns.

However, while some NICs are implemented in FPGAs like Tonic, Application Specific Integrated Circuits (ASICs) are much more common because of their higher performance, lower power, and lower cost. In this chapter, I present my design of a programmable transport layer inspired by Tonic, prototyped on an FPGA, but optimized for implementation in an ASIC. It is called nanoTransport, and it extends Tonic in four ways.

First, nanoTransport is designed to run in an ASIC, as noted above, and programmed (in the field) using the P4 language [23]. ASICs can achieve $\sim 10\times$ faster clock frequencies, thus faster packet processing, compared to FPGAs. In addition, P4 pipelines are already used in modern commercial NIC ASICs [112, 132, 166], and an industry group is creating a standard portable architecture for P4-programmable smartNICs [44].

Second, a wide range of transport protocols share a common set of triggering events (e.g., packet arrival, timeouts, duplicate ack), and nanoTransport exploits P4's simple and widely accepted abstractions for them. It enables interfaces to trigger common events in a programmable fashion, unlike how Verilog implements each event individually for every protocol implementation. This is inspired by the event-driven P4 packet processing framework introduced in [60].

Third, Tonic is designed to offload only the sender-side protocol. In contrast, nanoTransport implements both the sender and the receiver clients of a transport protocol while not needing the main processor to establish a connection. This way, it provides an end-to-end solution for transport layer offloading.

Finally, nanoTransport's design is streamlined. It is designed to process packets with different transport protocols in less than 35 cycles roundtrip. This implies a packet processing latency of less than 11ns with a targeted clock frequency of 3.2GHz, an order of magnitude lower latency (tail and median) compared to Tonic. Moreover, its streamlined design allows issuing a new packet every 2.6ns and achieving a 200Gb/s line rate. §3.5.1 discusses more about synthesizing this design with an ASIC library.

The primary goal of this work is to reduce the transport-related processing latency with nanoTransport. The latency for delivering a message to the application thread after the transport layer involves mechanisms such as core selection and thread scheduling. These processes are beyond the scope of nanoTransport. Therefore, to demonstrate a complete programmable system, nanoTransport is prototyped on the open-source nanoPU design framework [62], which introduces a message interface directly to the CPU register file and a hardware thread scheduler on a RISC-V architecture [61]. Using the open-source nanoPU artifact for a complete system allows the research community to experiment with nanoTransport's design, try out new transport layer protocols, and improve upon this work. However, nanoTransport's programmable transport layer is not bound to the nanoPU; it could be used as a standalone, P4-programmable pipeline in any NIC that offloads the transport layer to hardware, e.g., the RDMA processing pipeline in a modern NIC.

In summary, the main contributions of this work are as follows:

1. It defines interfaces to a common set of events in transport protocols that can be used as primitives for a programmable solution.

2. It shows that transport layer processing can be efficiently expressed in the P4 programming language.

3. It presents and evaluates the first P4 programmable transport layer in hardware, nanoTransport, which could be added to the nanoPU system, or deployed standalone in an RDMA NIC pipeline.

4. It provides an open-source FPGA-based nanoTransport prototype based on Firesim [81], which runs at 200Gb/s, even for small packets, while maintaining less than 100 Bytes of state per

message.

5. NanoTransport can deterministically process small packets, i.e., 80 Bytes, in 11ns (median and tail latency, including both the ingress and egress paths), while running at 3.2GHz. This latency is three orders of magnitude lower than common software-based implementations and an order of magnitude lower than Tonic, which runs at 100MHz due to frequency limitations of the FPGA environment.

6. This work also provides a behavioral model of nanoTransport in NS3 to help designers evaluate new transport protocols and algorithms at scale before programming the hardware.

The remainder of this chapter describes nanoTransport's building blocks in §3.1, its design details in §3.2, its FPGA-based prototype implementation in §3.3, and evaluations of its prototype in §3.4. Further discussion about use-cases, feasibility, and limitations of programmable hardware transport layers is provided in §3.5.

3.1 Transport Layer Dissected

Despite their differences, most transport protocols share a large set of features. In this section, I explore and enumerate common features that are, later, used as the basis of the nanoTransport design.

3.1.1 Protocol Taxonomy

Broadly speaking there are two types of transport protocols: Wide Area Network (WAN) protocols such as TCP NewReno [52], CUBIC [137], and BBR [25]; and data center (DC) protocols, such as RoCE [111], DCQCN [175] and Timely [117].

WAN protocols are designed for long-lived, reliable bi-directional byte streams, and their primary performance metrics are usually throughput and fairness. Connections are established by a handshake that installs a per-connection state at both ends. This state is then maintained for the duration of the connection. Because WAN RTTs are typically 1-100ms, a microsecond level latency improvement in the end-host processing does not add much value. Therefore, nanoTransport is not designed for WAN transport protocols.

On the other hand, DC protocols are mostly used to exchange small messages between servers [10, 139]. RTTs are a few microseconds, and latency-sensitive applications can benefit greatly from further microsecond-level reductions in the end-host processing time [54, 57, 119]. Hence, nanoTransport focuses on *latency-sensitive, reliable, message-based transport protocols*, primarily for data centers.

Specifically, nanoTransport *is designed to allow a user to program a low-latency, reliable, one-way messaging service.*

A (beneficial) consequence of small message communication is that no persistent connection state is required. This reduces the amount of memory a NIC needs to track currently active messages, making faster and lower-power single-chip ASIC solutions possible.

3.1.2 Building Blocks

NanoTransport's programmable hardware transport layer has two service interfaces: Below, it exchanges Ethernet frames with the Ethernet MAC. Above, it exchanges complete, reassembled, reliable messages with a CPU core or RDMA engine. Regardless of the protocol specifics, a reliable message-based transport protocol on nanoTransport must:

1. Split an outgoing message into one or more packets. Packets are stored for retransmission until successfully delivered to the receiver.

2. Reassemble incoming packets back into messages. Packets arriving out of order are correctly resequenced during reassembly.

3. Maintain timers to trigger packet retransmissions or to cancel messages upon repeated failures.

4. Maintain state for each ongoing message that can, for example, allow congestion control logic to decide which packet to send next, and when.

5. Generate control packets to signal message state or congestion, for example, ACK, NACK, and GRANT.

A key observation of this work is that only the last two functions (maintaining per-message state and generating control packets) require programmability to support different congestion control algorithms. The other capabilities are fixed and common to all reliable message-oriented transport protocols I have encountered.

Tonic made a similar observation [8] and elected to use bitmaps, which keep track of the message state. Those bitmaps are, then, used to determine which packet to send next or retransmit.

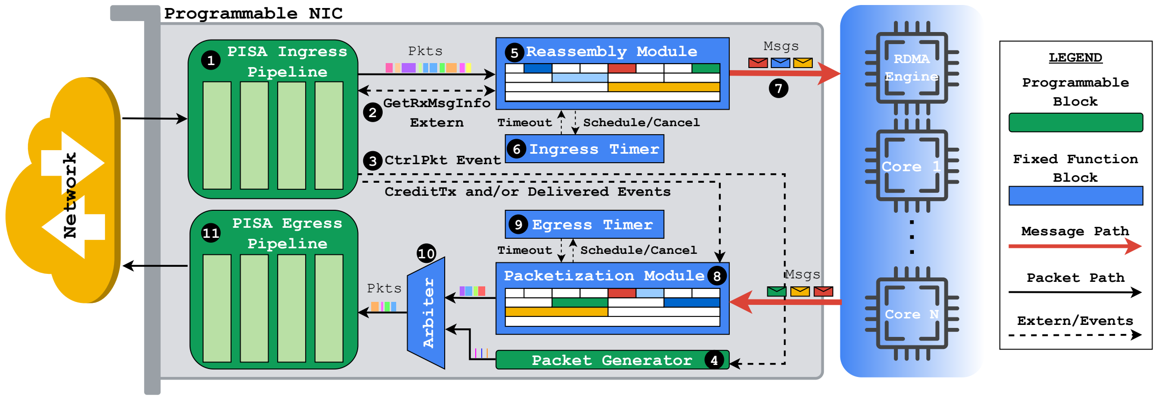


Figure 3.1: NanoTransport architecture design. Processing steps are numbered chronologically.

NanoTransport keeps these bitmaps in the reassembly and packetization modules, next to the associated packet data. Different protocols differ in how they modify the bitmaps when data or control packets are sent and received, how they detect a packet loss, and how they handle a lost packet. A nanoTransport programmer determines how events are triggered and processed (e.g., data packet arrival, packet loss detection, packet acknowledgment) via P4 `externs` [158] and by extending P4 metadata fields. §3.2 describes this design in more detail.

3.2 NanoTransport Architecture

Figure 3.1 shows the nanoTransport architecture. The pipeline sits between the external Ethernet packet interface (the MAC), with which it exchanges Ethernet frames, and the CPU core (or RDMA engine), with which it exchanges fully assembled, ready-to-use messages. The pipeline is self-contained and handles all aspects of the transport layer on behalf of the CPU. The CPU is needed to configure and initialize the pipeline, but the CPU is not involved in processing individual packets to minimize latency.

The design is deeply pipelined to process many packets in parallel with maximum throughput, but not too deep to keep latency low. The ingress and egress pipelines each contain a mix of fixed and programmable modules. The two pipelines also operate independently, other than when triggering a few well-defined events described in §3.2.4.

Let's start by walking through the high-level steps to process arriving and departing packets and then dive deeper into each stage: An arriving packet at the NIC ❶ is first processed by the programmable ingress pipeline, where protocol-specific logic determines how the packet will be processed. The `GetRxMsgInfo` extern is then called ❷. This extern uses flow identifiers such as the 5-tuple or unique message ID to fetch (or allocate) per-message state in the reassembly module. The per-message state is common to all protocols and is described in §3.2.3. Depending on the protocol, the ingress pipeline may then also choose to trigger a `CtrlPktEvent` ❸ that causes the packet generator to generate a control packet (acknowledgment, grant or NACK, etc. depending on the protocol) in response to the incoming packet ❹. The original data packet is passed to the reassembly module ❺, which stores it and checks if the message is complete. The reassembly module maintains and updates a per-message timer for incoming messages ❻. Should a timer expire (indicating message reception failure), all the state for the message is garbage collected. Once all of a message's packets have been received, they are reassembled in the correct order and forwarded as a full message to the CPU (or the RDMA Engine) ❼.

In the egress direction, when a message is sent from an application thread ❽, it is stored in the packetization module. The packetization module divides the message into MTU size segments and initializes per-message state variables. A per-message retransmission timer is set ❾; if it expires, some of the message's packets may be retransmitted. When the packetization module sends a packet, it is enqueued by the arbiter ❿, which schedules its departure alongside outgoing packets from the packet generator. Finally, packets pass through the programmable egress pipeline ⓫, where protocol-specific headers are added before the packet is sent to the network.

Next, I describe each block in detail and provide the API signatures for event handling.

3.2.1 Programmable Components

The nanoTransport architecture contains the following programmable modules: (*I*) the P4 programmable PISA pipelines and (*II*) the packet generator module.

PISA Pipelines

A PISA (Protocol Independent Switch Architecture) [20] pipeline provides a simple match-action abstraction. It allows fast and flexible packet processing by executing P4 programs [23]. The nanoTransport design dedicates separate PISA pipelines for ingress and egress. Each pipeline contains a standard P4 library (`core.p4`), as well as several custom externs to support nanoTransport-specific event handling logic. Developers program the pipelines to parse and emit protocol-specific headers and trigger the predefined event-handling logic in the fixed function blocks.

A typical ingress pipeline flow starts with a packet arriving at the parser, followed by the match tables. The tables are programmed to match protocol-specific events and this is where most protocol-specific functions are performed. For example, an ingress table may be programmed to match a flag field in the transport header; if it is a data packet, it is forwarded to the reassembly module while generating a control packet (e.g., an ACK) in response. If the incoming packet is a control packet (e.g., an ACK packet from the remote end), the ingress pipeline processes it and then discards it.

After ingress processing, data packets arrive at the reassembly module, carrying with them the metadata shown in Listing 3.1. The metadata includes the IP address and port number of the remote sender as well as a unique message ID (`tx_msg_id`) chosen by the sender. The three fields are used to map the message to a locally unique ID (`rx_msg_id`). The `pkt_offset` field indicates the offset of this packet within the message to which it belongs.

Listing 3.1: Metadata passed *from* the ingress pipeline *to* the reassembly module, along with the packet payload.

```
struct ingress_metadata_t {
    IPv4Addr_t remote_ip;
    PortNo_t   remote_port;
    bit<16>    msg_len;
    bit<8>     pkt_offset; // Similar to TCP seq no
    PortNo_t   local_port;
    MsgID_t    rx_msg_id;  // Set by the receiver
    MsgID_t    tx_msg_id;  // Set by the sender
    bool       is_last_pkt;
}
```

Listing 3.2: Metadata passed *to* the egress pipeline along with the packet payload.

```
struct egress_metadata_t {
    IPv4Addr_t remote_ip;
    PortNo_t   remote_port;
    bit<16>    msg_len;
    bit<8>     pkt_offset;
    PortNo_t   local_port;
    MsgID_t    tx_msg_id;
    bit<16>    credit; // Similar to TCP cwnd
    bit<8>     rank;   // Determines packet priority
    bit<8>     flags;
    bool       is_new_msg;
    bool       is_rtx;
}
```

The egress pipeline's main job is to create the correct packet header for an outgoing packet. The arbiter hands the raw packet payload to the egress pipeline, which constructs the correct packet headers using the accompanying metadata. The egress metadata is shown in Listing 3.2.

The `credit` value indicates the highest packet offset that is currently authorized to be sent for this message. The `rank` is the queueing priority of the outgoing packet. For example, the `credit` value in Homa signals which packets are granted by the receiver, and the `rank` value determines which in-network priority queue should be used by this packet. The first packet in a message has the `is_new_msg` flag set to initialize the message processing logic. The `is_rtx` flag identifies retransmitted packets, in case the protocol needs to process these packets differently.

In addition to header processing, transport protocols maintain protocol-specific state in the PISA pipelines. For example, NDP keeps some state in the ingress pipeline to identify which packet to request in a PULL control packet. While it is a common misconception that P4 cannot be used to implement stateful logic, read-modify-write (RMW) "register" operations are frequently exposed to the programmer for stateful data plane applications in a match-action pipeline. §3.2.2 describes the stateful primitives in the nanoTransport PISA pipelines and §3.4.3 discusses their feasibility.

Packet Generator

The developer programs the packet generator to send protocol-specific control packets, such as NACK packets in NDP [54], GRANT packets in Homa, and INT acknowledgments in HPCC [102]. The module is triggered by `CtrlPktEvent` extern call from the ingress pipeline, which is essentially a mirrored packet carrying the metadata shown in Listing 3.2. The metadata set by the ingress pipeline determines which control packet(s) to generate.

Different transport protocols generate control packets at different times and rates, and in different formats. For example, NDP paces its outgoing PULL control packets to tell the sender when to resend trimmed packets. The PULL packets must be sent at specific times. In contrast, HPCC piggybacks a template to outgoing packets in the reverse direction, to carry INT reports added by switches along the path. Fortunately, the range of operations is quite small.

3.2.2 Stateful Primitives

This section describes the stateful primitives that can be used by the programmer in the ingress and egress PISA pipelines to develop protocol-specific functionality. After a survey of low-latency transport protocols, I identified a list of primitives that would be required to implement a wide range of algorithms. These primitives implement various read-modify-write (RMW) operations and are exposed to the data-plane programmer as P4 externs. Sivaraman et. al [150] propose the following set of stateful primitives that are useful across many data plane applications:

- RW – Read or write a state variable.
- RAW – Add a value to OR overwrite a state variable.
- PRAW – Perform RAW on state variable only if the provided predicate evaluates to true, otherwise leave it unchanged.
- ifElseRAW – One RAW for true and one for false predicate.

Note that for some transport protocols (e.g., NDP), these operations are sufficient. However, other protocols (e.g., Homa) require more sophisticated stateful primitives, such as multi-ported memory, to share state variables across pipeline stages, and between ingress and egress pipelines. Hence, nanoTransport is designed with a multi-ported memory system.

In addition, nanoTransport also includes a priority scheduler, which is exposed to the programmer as a P4 extern. The scheduler can store and compare multiple stateful objects using a user-provided priority value and predicate function. The programmer can insert and remove objects, and update the priority of existing objects. When called, the scheduler will return the highest priority object for which the predicate evaluates to true. §3.3.4 describes how this priority scheduler and other primitives can be used to implement Homa's SRPT message-granting logic. This primitive will be useful for other protocols as well [6, 49, 57].

NDP and Homa are the two protocols that together require all the identified primitives. Therefore nanoTransport's performance is evaluated on these protocols. §3.5.2 further discusses implementing other protocols on nanoTransport.

3.2.3 Reassembly Module

The reassembly module is responsible for delivering message data in the correct order. If the packets within a message arrive out of order (e.g., because of packet-by-packet multipath routing, or retransmission), the reassembly module correctly resequences them before handing the message to the application thread. Since the packet reordering logic is protocol-agnostic, nanoTransport handles it with a fixed function block. The algorithm of the reassembly module is presented with a simplified pseudocode in Algorithm 1.

Algorithm 1: The processing logic for the Reassembly Buffer

```
1  Control ProcessNewPacket(Packet* pkt, ingress_metadata_t meta):
2      if m_rx_msg_id_table.match (meta.rx_msg_id).hit () then
               // Record pkt in buffer
3          m_buffers[meta.rx_msg_id][meta.pkt_offset] = pkt;
               // Mark the packet as received
4          m_receivedBitmap[meta.rxMsgId].set (meta.pktOffset);
               // Check if all pkts have been received
5          if m_receivedBitmap[meta.rxMsgId].all () then
                   // Reassemble packets into the message
6              Message* msg = this → ReassembleMsg (meta.rxMsgId);
                   // Prepend info for delivery to correct thread
7              msg.app_hdr.SetValid();
8              msg.app_hdr.remote_ip = meta.remote_ip;
9              msg.app_hdr.remote_port = meta.remote_port;
10             msg.app_hdr.local_port = meta.local_port;
11             msg.app_hdr.msg_len = meta.msg_len;
                   // Push the reassembled msg to the applications
12             this → NotifyApplication (msg);
13             this → ClearStateForMsg (meta.rxMsgId);
14         else
15             m_timer → ScheduleTimerEvent (meta.rxMsgId);
```

The reassembly module maintains a bitmap for every message, called `receivedBitmap`, where each bit corresponds to a packet in the message.[1] Each packet arriving at the reassembly module is stored in the corresponding buffer. If the `is_last_pkt` flag is set on the accompanying metadata, the module forwards the entire message to the cores. The `is_last_pkt` flag is calculated during the `GetRxMsgInfo` extern call, which is described next.

Listing 3.3: Metadata provided to the `GetRxMsgInfo` extern call

```
struct get_rx_msg_info_req_t {
  bool        mark_received; // Flag for read-only calls
  IPv4Addr_t  src_ip;        // Sender's IP address
  PortNo_t    src_port;      // Sender's port number
  MsgID_t     tx_msg_id;     // Unique ID set by the sender
  bit<16>     msg_len;       // Length of the message
  bit<8>      pkt_offset;    // Index of the current packet
}
```

[1]Every packet of a message, except the last one, is assumed to be MTU bytes long.

`GetRxMsgInfo` Extern

The `receivedBitmap` is maintained to allow for message reassembly and to determine which data or control packets to send next. The bitmap state is fetched by the ingress pipeline by calling the extern with the `get_rx_msg_info_req_t` metadata. The content of the request metadata for the `GetRxMsgInfo` extern is shown in Listing 3.3.

The `mark_received` flag in the input metadata signals whether or not the `receivedBitmap` should be updated by the extern call. If true, the value at index `pkt_offset` is set to 1 before the output metadata is generated.[2] The remaining `get_rx_msg_info_req_t` metadata is used as the match fields of the `rx_msg_id_table` in the reassembly module, a lookup table yielding the unique locally-assigned `rx_msg_id` for the arriving message. If no entry in the table is matched, a new ID is allocated from the list of free IDs.

The `GetRxMsgInfo` extern returns `get_rx_msg_info_resp_t` metadata, including the `rx_msg_id` for the message and the state corresponding to the message. The content of this response metadata is shown in Listing 3.4. The `fail` flag signals that the reassembly module was unable to allocate resources for this message, and the programmer decides how the ingress pipeline processes such messages. `is_new_msg` is used to initialize the packet processing logic for a new message. `is_new_pkt` helps prevent processing duplicate packets. `is_last_pkt` denotes that all the bits in the `receivedBitmap` are set to 1. This value is passed along with the packet to the reassembly module, to mark message completion.

Listing 3.4: Metadata returned from the `GetRxMsgInfo` extern call

```
struct get_rx_msg_info_resp_t {
  bool    fail;         // Extern return status
  MsgID_t rx_msg_id;    // Unique ID set by the receiver
  bool    is_new_msg;   // Msg not seen before
  bool    is_new_pkt;   // Packet not received before
  bool    is_last_pkt;  // Msg completely received
  bit<9>  ackNo;        // Smallest non-received pkt_offset
}
```

[2]This is useful when an arriving packet belongs to an incoming message, but it is not a data packet, e.g., trimmed packets in NDP.

3.2.4 Packetization Module

NanoTransport accepts complete messages from application threads and breaks them into Ethernet packets for network transmission. In addition to storing the message data, the packetization module maintains state variables for the message, similar to [8], so that the module can keep track of the communication between the sender and the receiver. The state variables and their roles are listed below:

- `deliveredBitmap`: Tracks which packets are delivered to the destination. The ingress pipeline can be programmed to trigger `DeliveredEvent` to update the values of this bitmap. Eventually, the packetization module clears the memory allocated to the message when all of its packets are delivered.

- `credit`: The largest `pkt_offset` value that is allowed to be transmitted. All the smaller `pkt_offset` values are allowed to be sent into the network. The protocol logic in the ingress pipeline uses `CreditTxEvent` to update this value. The default value for this variable is configured on compile time, similar to the initial window for TCP. [31] For the message to start at the line rate, this value should be configured to the Bandwidth Delay Product (BDP) of the network.

- `txBitmap`: Tracks packets that are to be (re)/transmitted. To emit a packet, the packetization module chooses the smallest index from this bitmap whose value is 1. Then, the corresponding value is reset to 0. `CreditTxEvent` can be triggered to set a value in this bitmap back to 1 for retransmission.

- `maxTxPktOffset`: Tracks the highest `pkt_offset` sent so far. Determines packets to be retransmitted upon a timeout.

- `timeoutCnt`: Tracks the number of timeouts the message has received without updating the `maxTxPktOffset` value. If this number is higher than the configured threshold, the packetization module gives up on the message and clears all memory allocated to it.

The packetization module also stores the message ID, the sender and receiver's port numbers, and the receiver's IP address. The egress metadata shown in Listing 3.2 is generated from these values whenever a packet is sent. The packetization module chooses a message from the memory which has packets that are allowed to be sent. The packet with the smallest allowed index is forwarded to the arbiter along with the metadata.

Algorithm 2: `DeliveredEvent` processing logic

```
  Inputs: $tx\_msg\_id$, and $ackBitmap$
1 deliveredBitmap = bitmap_table.lookup(tx_msg_id);
2 deliveredBitmap = deliveredBitmap or ackBitmap;
3 if deliveredBitmap.all() then
4     ClearStateForMsg (tx_msg_id);
```

The events that are used to update the packetization module's state variables are described next.

`DeliveredEvent`

Informs the packetization module that the packet has been successfully delivered to the remote host. The sender sets the corresponding bit in the `deliveredBitmap` so that the packet is not retransmitted in the future. Typically, a received acknowledgment packet triggers this event, as decided by the programmer. Algorithm 2 shows the main processing logic triggered by this event. `ackBitmap` is a bitmap created by the ingress pipeline to indicate which packets to mark as delivered.

`CreditTxEvent`

Signals that a message is allowed to send more packets (new packets or retransmissions) or the credit (e.g., `cwnd`) is reduced. `txBitmap` is modified to identify which packets can be sent next time there is sufficient credit to transmit one. For example, in Homa, an arriving Grant packet triggers this event. Algorithm 3 shows the main processing logic triggered by the event. `rtxBitmap` is the input argument indicating which packets are to be retransmitted. It is set by the ingress pipeline under the control of the programmer. For example, NDP sets the bit for NACK packets for trimmed packets. A protocol may require several packets to be retransmitted at the same time, e.g., selective NACK similar to SACK [71].

`TimeoutEvent`

Every message in the packetization module initiates a timer, along with metadata called `rtx_offset`, in the timer module. The metadata is the highest `pkt_offset` transmitted for the message as of the time the timer is scheduled. When a timer expires, the timer module triggers the packetization module's `TimeoutEvent` to compute packets for retransmission. All non-delivered packets which have a smaller offset than `rtx_offset` are retransmitted. Finally, a new timer is scheduled for the same message to account for future retransmissions. Algorithm 4 shows the processing logic triggered

Algorithm 3: `CreditTxEvent` processing logic

```
Inputs: tx_msg_id, rtxFlag, rtxBitmap, creditUpdateFlag, newCredit, allowTxFlag
1  txBitmap = bitmap_table.lookup(tx_msg_id);
2  if rtxFlag then
3  |   txBitmap = txBitmap or rtxBitmap
4  if creditUpdateFlag then
5  |   currentCredit = credit_table.lookup(tx_msg_id);
6  |   currentCredit = newCredit;
   // Determine which packets are allowed to be sent
7  txPkts = txBitmap & OnesUntil (currentCredit);
8  if txPkts.any() and allowTxFlag then
9  |   Emit (txPkts);
10 |   txBitmap = txBitmap and not txPkts;
```

Algorithm 4: `TimeoutEvent` processing logic

```
Inputs: tx_msg_id, and rtx_offset
1  maxTxPktOffset, timeoutCnt = state_table.lookup(tx_msg_id);
2  deliveredBitmap, txBitmap = bitmap_table.lookup(tx_msg_id);
3  if timeoutCnt >threshold then
4  |   ClearStateForMsg (tx_msg_id);
5  else
6  |   if maxTxPktOffset >rtx_offset then
7  |   |   timeoutCnt = 0
8  |   else
9  |   |   timeoutCnt = timeoutCnt + 1
10 |   rtxPkts = (not deliveredBitmap) and OnesUntil (rtx_offset);
11 |   if rtxPkts.any() then
12 |   |   Emit (rtxPkts);
13 |   |   txBitmap = txBitmap and not rtxPkts;
14 |   Timer → ScheduleEvent (tx_msg_id, maxTxPktOffset);
```

by this event. A detailed description of how the timer module works is provided in §3.2.5.

The fixed-function timeout event processing has been sufficient for a wide class of transport protocols so far. However, it is possible that some protocols will need to handle timeout events differently. For example, timer events might need to generate control packets, or periodically update protocol state in the ingress/egress pipelines. Therefore, a future version of the nanoTransport architecture may benefit from making the timeout event processing programmable as well.

3.2.5 Timer Module

Timers are required for two purposes: (1) identify packets that have not (yet) been acknowledged and need to be retransmitted, and (2) identify idle messages (have not sent or received packets) for a long time to clean up the per-message soft state.

Software implementations can maintain a timer per packet. In hardware, it is challenging to maintain a timer for every in-flight packet – potentially a large number depending on the network's BDP and the configured timeout duration. To reduce memory requirements, nanoTransport maintains a single timer *per message*.

When the applications write a new message to the packetization module, the egress timer module's `ScheduleEvent` is triggered. This event creates a new timer for the corresponding message, along with associated metadata. When this timer expires, the packetization module's `TimeoutEvent` is triggered. This event may or may not cause a new timer to be scheduled for the same message. When the message is successfully delivered to the remote client, the packetization module fires a `CancelEvent` within the timer module before deleting the state for the message. This event ensures that no timers are left behind which may timeout spuriously.

Similarly, when the first packet of a message arrives at the reassembly module, `ScheduleEvent` of the ingress timer module is triggered. This event creates a new timer for the corresponding message. Since there is no notion of retransmission in the ingress direction, this timer is only used to discard the state for the message from the reassembly module. Each arriving packet triggers `ReScheduleEvent` for the associated message, which mainly resets the timer and prevents timeouts. Finally, a completed message signals `CancelEvent` to invalidate the associated entry in the ingress timer module.

3.3 Building NanoTransport Hardware

The nanoTransport prototype extends the open source nanoPU design [62] by adding 2500 lines of Chisel [12] code and 1000 lines of P4 code. Large-scale, cycle-accurate simulations for the prototype are run on AWS FPGAs [145] using Firesim [81]. The FPGAs run at 90MHz, but the target clock rate is simulated as 3.2GHz, which mimics IceNIC [81] – the baseline comparison for nanoTransport.

The simulated NICs are connected by C++ switch models running on the AWS x86 host CPUs with 200Gb/s simulated line rates. Note that the exceptionally high link capacities and the clock rates are used to stress test the design and show how performant the nanoTransport design is for future networks.

In addition to the cycle-accurate hardware prototype, a behavioral model of the nanoTransport architecture is implemented in NS3 [138] for rapid functionality testing. This NS3 module has been mostly used to verify the protocol implementations.

Overall, the nanoTransport artifact enables the assessment of the end-to-end functionality and performance of the design. The following sections provide more details about the nanoTransport prototype.

3.3.1 Programmable Modules

The ingress and egress pipelines are implemented using P4 and Xilinx SDNet[3] [153]. The SDNet compiler generates a Verilog module with the required functionality, which is integrated into the nanoTransport prototype. The correct functionality of the design is verified using Synopsys VCS [155] cycle-accurate simulations. However, due to licensing restrictions, SDNet-generated modules on AWS FPGAs were not available. As a result, the P4 code was hand-compiled into Chisel so that the full system could be evaluated with Firesim on AWS FPGAs. The evaluation results described in §3.4 use the hand-compiled P4 code. Each P4 program is implemented as a custom pipeline, similar to how SDNet maps P4 programs to FPGAs. An ASIC prototype would instead have a fixed number of pipeline stages which all programs must be mapped to. This approach is planned to be explored in future work.

Recall that the Packet Generator in nanoTransport is a programmable module. When processing a `CtrlPktEvent` from the ingress Pipeline, the packet generator might generate one or more control packets while (optionally) pacing their transmission rate. These operations are not particularly well-suited for a P4 pipeline, which is typically used to transform individual packets. As a result, the packet generator is also programmed in Chisel for the nanoTransport prototype. Exploration of more convenient, higher-level abstractions for programmable packet generation remains as an opportunity for future work. One possibility is to use P4 along with new custom externs to fork (duplicate) and pace packets within the pipeline.

3.3.2 Reassembly and Packetization Modules

The reassembly module reassembles packets, which might arrive out-of-order, into contiguous messages for delivery to applications. The packetization module splits messages into segments, which might get retransmitted out-of-order due to packet loss in the network. In order for these tasks to be performed at line rate, simple data structures must be used, which require only constant time operations. One could choose from several different approaches; this section describes the buffer design in the final prototype.

[3]Xilinx SDNet is also known as Vitis Networking P4.

The message buffer is divided into buffers of several different fixed sizes, and a free list for each size class keeps track of which buffers are available. When a buffer is to be allocated, the smallest available one that is large enough to store the whole message is selected. For message reassembly, a buffer is allocated when the first packet of the message arrives from the network and is freed when the message is forwarded to the processing cores.[4] For message packetization, a buffer is allocated when the application writes the first word of the message and is freed when the entire message has been successfully delivered to the receiver. The design uses a table indexed by a message identifier to keep track of where each message is stored (the buffer pointer).

One of the benefits of using fixed-size buffers to store messages is that it simplifies out-of-order reassembly and retransmission: to find the position of a particular packet within the message, the hardware simply adds the appropriate offset to the message's buffer pointer. In addition, the logic that is required to manage memory buffers is very simple and can run at the line rate. Buffer allocation requires one dequeue from a free list, and freeing a buffer requires one enqueue to a free list; there is no need for complex partitioning and merging of variable-size buffers.

The primary drawback of using fixed-size buffers is that it leads to memory fragmentation and potentially poor utilization of the buffer space. It is therefore important to properly configure these message buffer modules. Configuration involves selecting how to carve the total buffer space into fixed-size buffers. If the message size distribution is known at configuration time, then it is often possible to achieve very high utilization of the buffer. For example, if a workload consists of 50% messages that are 100B and 50% messages that are 500B then the best option is to use two size classes, each with an equal number of buffers.

3.3.3 Timer Modules

The timer modules in the nanoTransport architecture maintain a single timer, along with associated metadata, for each message in the reassembly/packetization modules. The goal is to minimize memory and logic requirements while ensuring that timers can be scheduled or canceled in constant time. Furthermore, since timers are used to trigger packet retransmissions or for garbage collection in the background, there is neither need for the timers to expire exactly on time, nor for them to expire in the correct order. The main requirement is that they expire within a bounded amount of time.

These requirements lead to a very simple hardware design. The timers for each message are

[4]An arriving packet is dropped at the ingress of the reassembly module if it is unable to allocate a buffer for the message.

Algorithm 5: NDP P4 Pseudocode

```
Control Ingress(out ingress_metadata):
    state credit;
    if hdr.ndp.flags.DATA then
        msg_info = GetRxMsgInfo();
        if hdr.ndp.flags.TRIM then
            genNACK = true;
            pull_offset_diff = 0;
            Drop();
        else
            genACK = true;
            pull_offset_diff = 1
        if not msg_info.fail and msg_info.is_new_pkt then
            // ifElseRAW extern
            if msg_info.is_new_msg then
                credit[msg_info.id] = ... // initialize
            else
                credit[msg_info.id] += pull_offset_diff;
            pull_offset = credit[msg_info.id];
            CtrlPktEvent(genACK, genNACK, pull_offset);
    else
        if hdr.ndp.flags.ACK then
            DeliveredEvent()
        if hdr.ndp.flags.NACK or hdr.ndp.flags.PULL then
            CreditTxEvent()
        Drop();
Control Egress(in egress_metadata):
    hdr.ethernet.SetValid();
    hdr.ip.SetValid();
    hdr.ndp.SetValid();
    FillHeadersFromMetaData(egress_metadata);
```

stored in a single memory indexed by message ID. The entry contains the following fields: a single valid bit indicating whether or not the entry is valid, a 64-bit timeout value indicating the time at which the timer expires, and associated timer metadata. A background thread sequentially scans the entries and checks if the timer has expired. If so, it will extract the metadata and trigger a timeout event. Scheduling and canceling a timer simply involves writing a single entry to memory. In some cases, a timer may expire immediately after the background thread checks it, in which case the timeout event will not be triggered until the background thread loops back around to it. However, note that even in this case the timer will expire within a bounded amount of time, which is determined by the maximum number of timers/messages in the system. This simple design meets the requirements for a low-latency transport layer implementation in hardware.

3.3.4 Protocol Implementations

The evaluation of nanoTransporthas been conducted with two different protocol implementations. These protocols are chosen to represent a relatively wide range of features required by other protocols [5, 6, 13, 29, 49, 57].

NDP [54] is the first protocol programmed on the nanoTransport prototype. It is receiver-driven and aims to reduce the tail latency of network messages by ensuring that all dropped packets are retransmitted quickly. When congested, NDP-enabled switches trim data packets that would otherwise be dropped, forwarding only the packet headers to the receiver, at high priority. The receiver then quickly sends negative acknowledgments (NACKs) to inform the sender of the packet loss. This mechanism allows NDP to avoid relying on long timeouts. NDP senders initially send only up to one bandwidth-delay-product (BDP) worth of packets. The receiver, then, explicitly pulls the remaining ones while pacing them to ensure that the arrival rate of the pulled packets does not exceed the capacity of the bottleneck link. New data packets are pulled round-robin among messages with the assumption that if a data packet leaves the network, a new one can be inserted without overwhelming it.

Algorithm 5 provides pseudocode for the NDP implementation in P4. The protocol uses a stateful operation to read the previous credit for the message and increment it if needed. This operation is represented with the `IfElseRaw` extern, described in §3.2.2.

Homa [119] is the second protocol programmed on the nanoTransport prototype. It is also a receiver-driven protocol, but unlike NDP, it is designed with the assumption that packet loss is extremely rare in modern networks. Thus, it simply relies on timeouts to detect dropped packets rather than utilizing packet trimming within the network. However, it does require switches to support at least a few strict priority queues.

Additionally, rather than using a round-robin "pull" mechanism, Homa aims to minimize message completion time by approximating SRPT [143] scheduling at the receiver. The associated SRPT message granting logic is implemented via the priority scheduler extern described in §3.2.2. The scheduler maintains metadata about all active messages, and the rank (i.e., priority) is assigned as the remaining size of the message (a lower value is higher priority). The scheduler returns the highest priority "grantable" message, where a grantable message is the one that has fewer than one BDP of data outstanding. Messages are removed from the scheduler after they have been fully granted.

The implementation of the priority scheduler takes advantage of the fact that most messages are small (less than 1 BDP) and hence do not need to be scheduled; only a few messages need to be

Algorithm 6: Homa P4 Pseudocode

```
state msgPrio;
Control Ingress(out ingress_metadata):
    state msgState;
    priorityScheduler grantScheduler;
    if hdr.homa.flags.DATA then
        msg_info = GetRxMsgInfo();
        if not msg_info.fail and msg_info.is_new_pkt then
            // ifElseRAW extern
            if msg_info.is_new_msg then
                msgState[msg_info.id] = ... // initialize
            else
                msgState[msg_info.id].remaining_size -= 1;
            sched_msg = grantScheduler.apply(...);
            if sched_msg then
                // RAW extern
                msgState[sched_msg.id].grantedIdx = sched_msg.grant_offset;
                CtrlPktEvent(msgState[sched_msg.id]);
    else
        DeliveredEvent();
        if hdr.homa.flags.GRANT then
            msgPrio[hdr.homa.tx_msg_id] = hdr.homa.prio;
            CreditTxEvent();
        Drop();
Control Egress(in egress_metadata):
    hdr.ethernet.SetValid();
    hdr.ip.SetValid();
    // RW extern
    hdr.ip.tos = msgPrio[egress_metadata.tx_msg_id];
    hdr.homa.SetValid();
    FillHeadersFromMetaData(egress_metadata);
```

scheduled at any given time. Therefore, the scheduler extern maintains the message state in registers so that it can compare them all simultaneously. The prototype scheduler extern supports up to 16 scheduled messages for simultaneous comparison whereas the remaining scheduled messages, if any, are stored in a FIFO queue until a register space opens.

In addition to the scheduler, Homa uses two dual-ported memory primitives (§3.2.2) as shown in Algorithm 6. One of those dual-ported memories is used to maintain information about messages – it is accessed/updated by data packets as they arrive and then updated further down the pipeline after deciding which message to grant. The other dual-ported memory is used to track the priority of messages being transmitted. Incoming GRANT packets update the memory and outgoing data packets read it. Hence, this state is shared between the ingress and egress pipelines.

To evaluate the programmability of the prototype, a new low-latency, reliable message transport protocol called ***Homa-Tr*** was created. Homa-Tr combines features from NDP and Homa, in the manner a programmer might pick and choose features from different protocols. It includes NDP's ability to quickly recover from packet loss by trimming packets in the switches and sending negative acknowledgments (NACKs). In the meantime, it adopts Homa's ability to reduce message completion time by GRANT'ing messages in SRPT order. Overall, it proved relatively quick and easy to implement Homa-Tr, incorporating NDP's packet trimming and NACK mechanism into Homa code. Evaluation details are provided in §3.4.2.

P4 source code for the protocols is available in nanoTransport's open source artifact [61]. The NDP and Homa implementations required 376 and 520 lines of P4 code, respectively, which is an order of magnitude less code than available software-based implementations.

3.4 Evaluating NanoTransport

The nanoTransport architecture is evaluated on the performance, correctness, and feasibility of the transport protocol implementations. Microbenchmarks and end-to-end experiments with cycle-accurate simulations on AWS FPGAs [145] with Firesim [81] are utilized to evaluate performance and correctness. The FPGAs run at 90MHz, but a target CPU and NIC clock rate of 3.2GHz is simulated. All of the results reported in this section are based on the target 3.2GHz clock rate. To evaluate the feasibility of deploying the nanoTransport design in hardware, the FPGA resource utilization is compared to a more traditional, open-source NIC, called IceNIC [81], which does not implement the transport layer in hardware. More discussion about how the nanoTransport design perform on an ASIC is provided in §3.5.1.

The design, implementation, and testing cycle for hardware prototyping is slow and expensive (even on FPGAs). Yet, transport protocol designers generally need to conduct large-scale experiments to verify a protocol's functionality and usefulness. To ease the development process, a C++ based behavioral model for the nanoTransport architecture was developed in NS3 [138]. A protocol is first tested at scale using NS3, before programming the hardware. Since the performance results are the same for the NS3 model and the hardware prototype, the NS3 results are omitted here. The source code for the NS3 behavioral model is provided as a part of the open-source artifact along with the hardware prototype [61].

Table 3.1: RX and TX latency (from first byte in until first byte out) on the nanoTransport architecture for the NDP and Homa implementations when processing a single 16-Byte message (80-Byte packet).

	RX Latency (ns)			TX Latency (ns)			Grand
	Ingress	**Reassembly**	**Total**	**Packetize**	**Egress**	**Total**	**Total (ns)**
NDP	5	0.94	5.94	2.81	0.31	3.12	9.06
Homa	6.25	0.94	7.19	2.81	0.94	3.75	10.94

3.4.1 Latency and Throughput Microbenchmarks

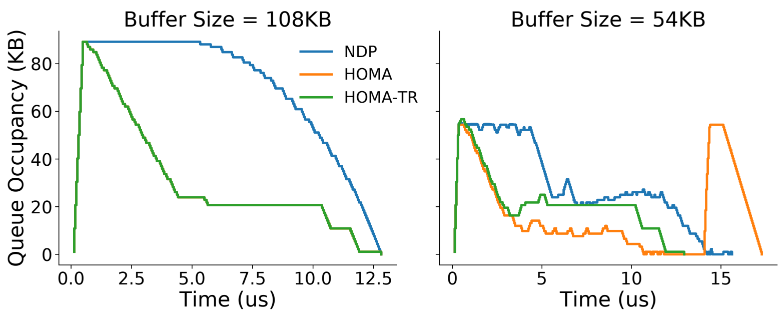
NanoTransport is targeted to process packets at 200Gb/s. For 1088-Byte packets, 200Gb/s means that a new packet can be transmitted or received every 44ns. The incast experiment described in §3.4.2 verifies that this is the case.

In addition, the maximum throughput for the worst-case traffic pattern is evaluated with nanoTransport. To measure the RX throughput, small 65-Byte packets are sent to a nanoTransport receiver at 200Gb/s (380Mrps). Each packet is a separate message and carries 1 Byte of payload as well as 64 Bytes of packet header. The measurements at the application layer verify that the minimum-sized incoming messages are forwarded to the cores at line rate. On the TX side, the same workload is generated on cores. The measurements from the network verify that the outgoing messages are transmitted at line rate onto the wire. Overall, the prototype is verified to support the target throughput of 200Gb/s in the worst case for both NDP and Homa implementations.

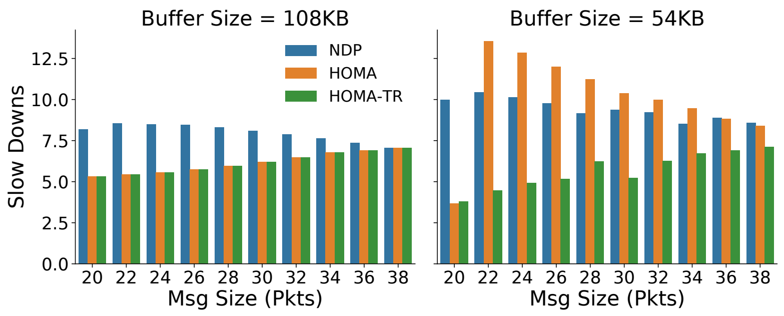
Table 3.1 shows the RX and TX latency breakdown for our NDP and Homa implementations. Homa's ingress and egress pipelines utilize five and two stages respectively and have a slightly higher latency due to its central message scheduling decisions, whereas NDP utilizes only three and one stage. The number of stages is determined by sequential dependencies between extern calls/memory accesses. Nevertheless, the transport processing requires at most 7.2ns in the ingress path, and 3.8ns in the egress path, resulting in a maximum transport layer round-trip time of 11ns.

NanoTransport's latency is three orders of magnitude lower than the 4.8μs reported for the Homa Linux Kernel Module [128]. The latency through the Linux network stack is very sensitive to interrupt processing overheads and OS thread scheduling decisions. Ousterhout [128] reports *tail* round-trip latency of 15.1μs, 23.4μs, and 24.1μs for Homa, TCP and DCTCP respectively.

eRPC [79] is a state-of-the-art, low-latency software network stack and reports a wire-to-wire latency of 850ns. It is difficult to compare nanoTransport directly to eRPC's transport layer. However, the paper does report measurements that suggest that the congestion control logic adds an average of 17.8ns of per-packet software latency, which is comparable to nanoTransport's latency.



(a) Bottleneck queue occupancy



(b) Message completion slowdowns.

Figure 3.2: Ten incast messages to the same receiver with different transport protocols and bottleneck buffer sizes while sender and receiver NICs are all running the nanoTransport prototype.

That being said, this measurement is reported under the best-case conditions in which the network is not congested and thus most of the congestion control logic is bypassed for almost all packets. On the other hand, the nanoTransport latency values reported in Table 3.1 are deterministic. Furthermore, the eRPC measurement does not include other aspects of the transport protocol such as message packetization/reassembly or retransmission logic. Finally, as a result of running in software, a single eRPC core can only process up to about 10Mrps, which is about 38× lower throughput than the pipelined nanoTransport design.

3.4.2 End-to-end Evaluation

To evaluate the end-to-end performance, the functionality of the architecture, and protocol implementations (i.e., NDP, Homa, and Homa-Tr), incast experiments were run using Firesim. In these experiments, ten senders each transmit one message to the same receiver at the same time. Each message has a distinct size, ranging from 20 to 38 MTU-sized (1088B) packets. This experiment is run on a simple dumbbell topology where the bottleneck link is the receiver's downlink. The RTT between the sender and the receiver is 525ns, and all the links run at 200Gb/s. Two experiments are conducted in this setup; one in which the bottleneck buffer size is large enough to absorb the incast, and one in which the bottleneck buffer size is too small to absorb the incast, resulting in packet loss and/or trimming.

The correctness of the protocol programs is verified by examining the packet traces of the incast experiment. Figure 3.2a shows the bottleneck queue occupancy in each experiment. As expected, the NDP client PULLs a data packet every time it receives one, so that the total number of packets in flight, and hence the queue occupancy, stays high until some messages are complete. On the other hand, the Homa client sends GRANTs for only a few messages,[5] which allows the queue occupancy to stabilize at a low level after the first RTT of the incast.

Figure 3.2b shows the message completion time slowdown for each message in each of the two experiments. The slowdown is defined as the ratio of the actual message completion time to the ideal completion time without any congestion in the network (smaller is better).

When the buffer is large, packets are not lost, enabling both NDP and Homa to smoothly PULL/GRANT new packets from the senders. However, Homa achieves lower slowdowns because messages are GRANT'ed in SRPT order – a policy designed to minimize message completion time. Since larger messages wait until the smaller ones are complete, the slowdown for Homa increases with the message size. On the other hand, NDP pulls messages in a round-robin fashion, causing similarly high slowdowns across all messages.

When the buffer size is too small to absorb the incast, the relative performance of the protocols completely changes. In this case, NDP can achieve lower slowdowns because it enables senders to quickly retransmit lost data using packet trimming and NACKs. Homa, on the other hand, relies on timeouts to detect packet loss. Therefore, NDP still achieves similar slowdowns for all the messages, whereas it takes longer for Homa to complete the messages.

The nanoTransport prototype is also programmed to implement a new protocol called Homa-Tr (§3.3.4), which combines features of Homa and NDP. Homa-Tr incorporates NDP's packet trimming and NACK'ing mechanism into Homa so that messages are granted in SRPT order while enabling quick recovery from packet loss. Figure 3.2b shows that Homa-Tr performs exactly the same as Homa when the buffer size is sufficiently large. However, when the buffer size is reduced by half (54KB), Homa-Tr is able to quickly recover from losses, and achieve $\sim 2\times$ better slowdown compared

[5]Homa sends GRANTs to multiple messages, called overcommitment, to account for cases where some senders are busy with sending other messages.

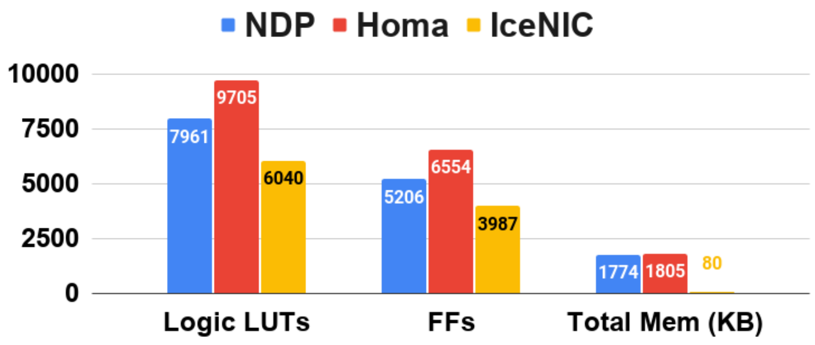


Figure 3.3: FPGA resource utilization of nanoTransport when running NDP and Homa (39KB max message size and 16 concurrent messages) compared to traditional IceNIC, which does not implement any transport processing.

Table 3.2: The resource utilization of the nanoTransport NDP prototype when configured to support both 16 and 128 concurrent 32KB messages. The percentage in each entry indicates the % utilization of the corresponding resource available on the Virtex Ultrascale+ FPGA.

# Msgs	Logic LUTs	Flip Flops	Total Mem (MB)
16	6999 (0.59%)	5043 (0.21%)	1.2 (11.9%)
128	23578 (1.99%)	8941 (0.38%)	8.4 (85.7%)

to Homa and ~1.5× better slowdown compared to NDP.

Experimental results suggest that the nanoTransport architecture can be programmed to run different low-latency protocols and that the protocol implementations behave as expected.

3.4.3 Feasibility

Next, the cost of implementing a programmable transport layer in hardware is evaluated. Figure 3.3 shows the FPGA resource utilization for the NDP and Homa implementations. To gauge the cost of putting the transport layer in hardware, we compare the resources used by nanoTransport against a baseline, called IceNIC [81], which does not implement any transport processing.

A basic NIC, like IceNIC, is very simple: It contains Ethernet header parsing, some staging memory, and the DMA logic to transfer packets to and from host memory. Relative to IceNIC, nanoTransport adds all the transport logic described above, and Figure 3.3 shows that the logic and flip-flop utilization grows by about 30% for NDP and 60% for Homa. This is as much a reflection of the simplicity of the simple IceNIC as the additional complexity of nanoTransport. Table 3.2 shows that nanoTransport consumes less than 2% of the logic and flip-flops of a Virtex Ultrascale+ FPGA [167]. It would require a much smaller fraction of an ASIC.

NanoTransport also requires memory for packetization and reassembly as opposed to IceNIC. The amount of memory depends on the number of concurrent messages. When designed to support up to 16 concurrent 39kB messages, nanoTransport needs about 1.2MB of on-chip SRAM[6] (Table 3.2). If instead 128 concurrent 39kB messages are supported, it consumes 8.4MB, which occupies less than 2mm^2 on a modern 7nm ASIC. The memory requirement increases linearly with the number of concurrent messages supported.

In conclusion, nanoTransport can easily be added to a modern NIC. Modern NIC ASICs already include tens of MB of onboard SRAM [166]; adding the logic and memory for a programmable, low-latency, reliable messaging transport layer appears to be a relatively small additional cost.

3.5 Discussion

3.5.1 FPGA versus ASIC

The nanoTransport prototype was built to run on an FPGA as a proof-of-concept and evaluation platform. Yet, its design is not necessarily the right choice for an FPGA-based NIC, where the FPGA itself can be re-optimized for a new transport protocol using Verilog.

In contrast, ASICs mostly run faster, consume less power, and cost less in volume than FPGAs [34]. For example, the Tonic [8] prototype uses a clock frequency of 100MHz whereas nanoTransport is targeted to run at 3.2GHz. This frequency is a design choice that follows the NanoPU, and IceNIC artifacts used to develop the nanoTransport design. NanoPU introduces a direct interface between the NIC and the CPU register arrays. The width of the data path for this interface is 64 bits to fit into the RISC-V register arrays. Hence, 200Gb/s at this interface can only be achieved with 3.2GHz clock frequency, which is not far from being realistic for a typical CPU architecture.

On the other hand, commercial PISA pipelines typically run with 1 to 1.5GHz clock frequencies to meet the timing requirements of stateless and stateful packet processing atoms when synthesized into actual ASICs [150]. For example, Tofino [35] runs at 1GHz with a 7nm ASIC library to achieve 12.8Tb/s for 32$\times$400Gb/s ports. To achieve its exceptional speed – despite a clock frequency lower than 3.2GHz used for nanoTransport – it uses a much wider data path than 64 bits used for nanoTransport. By widening the data path, nanoTransport can also afford to run at lower clock frequencies that are more feasible for today's manufacturing technologies and still maintain 200Gb/s.

Synthesizing the nanoTransport design and developing an actual ASIC implementation, possibly with a RISC-V CPU core, remains an attractive future work. The processes used to synthesize

[6]Including message payload and the associated state, as described in §3.2.4.

commercial PISA pipeline solutions, e.g., Tofino, give confidence that nanoTransport can also be taped out on high-performance chips. Even if the clock frequency is halved to succeed at this task, nanoTransport would still be able to process packets almost an order of magnitude faster than the state-of-the-art programmable architectures. After all, the clock frequency determines the computation time for each cycle in the system while the number of cycles required to process a packet stays constant with the given logic of protocols.

For example, Tonic uses only 10 cycles to process a single packet in the egress direction, but it adds up to 100ns in total. In contrast, nanoTransport requires 23 cycles at most for the evaluated protocols in one direction. However, the targeted – 32× faster – clock rate enables 93% lower end-to-end latency, i.e., 7ns, which implies only 14ns when the clock frequency is halved.

Note that NanoTransport's higher number of cycles to process a packet is the cost of easier programmability, i.e., relatively higher-level P4 language instead of Verilog. Nevertheless, this cost is easily paid off with the capability to run P4 pipelines in ASICs, which run more cycles per unit of time. NanoTransport's contribution here is that it replaces the programmable modules with PISA pipelines, which can run at higher frequencies while still being programmable.

One downside of using PISA pipelines on ASICs is the limited number of compute operations. For example, P4 does not allow floating point operations whereas it is possible – but very expensive in latency and resources – to implement such operations in Verilog. These operations can theoretically enable the developer to implement a much wider range of algorithms. Even then, Tonic does not allow for such complex arithmetic to make sure every module completes in 1 cycle only.

Fortunately, both Tonic and nanoTransport show that a significant fraction of transport protocols and algorithms can be implemented without floating point arithmetic. For the cases where such arithmetic is required, §3.5.2 lists alternative strategies to enable or approximate the corresponding computations. Hence, nanoTransport avoids this unnecessary flexibility to obtain faster clock frequencies with ASICs in return.

In conclusion, while tested on FPGA, nanoTransport is designed to be implemented in a custom NIC ASIC for faster packet processing with a lower energy footprint and enough programmability.

3.5.2 Programming New Protocols

So far, nanoTransport is programmed and evaluated when running low-latency receiver-driven protocols, NDP, Homa, and Homa-Tr. For comparison, it was also evaluated for what it would take to program nanoTransport to run HPCC [102], which is sender-driven (rather than receiver-driven).

An HPCC sender examines the stack of INT reports [88] in every packet, determines the bottleneck link, and calculates the new window size. The PISA pipelines in nanoTransport can be used to process INT reports given switches are capable of generating them. If needed, P4 programmable switches can leverage the optimizations proposed in PINT [17] to reduce the amount of processing, thus pipeline stages, in the NIC.[7] The sender nanoTransport client can then use simple lookup tables in the P4 pipeline to calculate the congestion window size.

Implementing DCQCN [175] and Swift [91] were also considered for nanoTransport. Both of these algorithms involve floating point computation at the NIC to calculate transmission rates and congestion windows. Although the nanoTransport prototype does not support floating point operations, there are three alternative design choices to enable them: (1) Add floating point to the P4 pipeline in hardware; assuming we need only about 200 million floating point operations per second, this is relatively straightforward in a modern ASIC, (2) Use higher precision fixed point arithmetic, which is already supported in switch ASICs [35], or (3) Use lookup tables in the P4 pipeline. I anticipate that future ASIC implementations will utilize all three techniques.

3.5.3 Multiple Concurrent Protocols

The CPU might host multiple applications, each requiring high-performance transport protocols. Hence, the NIC may need to support several protocols at the same time. For example, it might offer a tail latency-optimized protocol for RPCs, while running a throughput-optimized protocol for the same application's bulk transfer traffic.

NanoTransport can do this, provided it has sufficient resources. Essentially, the programmable parser branches depending on the transport protocol identifier in the packet header, and the corresponding control logic is applied.

Care would need to be taken by the protocol designer to avoid undesirable interactions between the different transport protocols in the network. This is not specific to nanoTransport. It is a problem that all cloud service providers need to solve, whether the transport layer is in hardware or software. For example, Homa and NDP both assume that its receiver is the only entity allocating bottleneck bandwidth to the incoming messages. The PULL/GRANT mechanism of Homa and NDP may over/under-utilize the bottleneck link if the link is shared with non-GRANTed/PULLed traffic.

[7] The switches compute the link utilization along the path instead of the end host.

3.5.4 Encryption and Compression

Network operators may choose to use encrypted traffic in their network for security reasons. Modern NICs commonly include dedicated hardware modules for end-to-end encryption, and to compress data to and from storage [112, 125, 132]. Although such modules are not included in the prototype, an ASIC implementation of nanoTransport could easily include them in its processing pipeline.

3.5.5 Serializing RPC Data

Low-latency reliable message protocols frequently carry RPC requests, which need to be serialized and deserialized at each end. It was recently observed that this process can add quite a lot of latency to RPC requests [163]. Zerializer shows how marshaling and unmarshaling can be done in hardware. While beyond the scope of this work, I anticipate ASIC implementations of nanoTransport to add such capabilities to the hardware P4 pipeline.

3.5.6 Scalability

A key design choice when designing a nanoTransport ASIC will be the size of the SRAM. Once picked at design time, all programmed protocols will need to live within the constraint. This means the ASIC designer needs to decide, upfront, how many messages can be supported, and the size of the largest message. The nanoTransport prototype supported up to 128 concurrent 32kB messages, which is reasonable for Homa and NDP. However, a more careful study of other transport protocols is needed before committing the size to an ASIC.

Careful consideration is also required when choosing the number of P4 pipeline stages, which in turn determines how many serially-dependent operations can be performed on each packet header. The prototype NDP and Homa programs require significantly fewer stages than the 10–20 stages supported in some commercial P4 pipelines today [35]. However, more protocols should be evaluated before committing to an ASIC design.

3.5.7 Other Use-Cases

In addition to RPCs, small messages are frequently sent for RDMA operations as well. Typically, an RDMA-enabled NIC terminates transport logic with a fixed protocol, i.e., RoCE or Infiniband, and directly accesses host memory without bothering the host CPU. NanoTransport can do the same by sending reassembled messages directly to the DMA engine. I anticipate that this approach would be commonly supported on ASIC implementations of nanoTransport.

Moreover, an ASIC design would likely be configurable to bypass the packetization and reassembly module, for transport layers that the application developer prefers to process in software. This would be especially useful for applications that implement complex transport features that are not available on hardware.

The available PISA pipelines also enable running data plane programs that are not transport layer related, such as NetCache [76], SwitchML [141], and PPS [74] as long as enough TCAM, SRAM, and pipeline stages are available. The exploration of other services that can be offloaded onto nanoTransport is left as future work.

Chapter 4

Sub-RTT Congestion Control for Lower Network Latency

Network operators are inclined towards increasing line rates and MTU sizes as much as possible to satisfy the performance requirements of modern applications such as NVMe and distributed ML [156, 110]. However, with larger Bandwidth Delay Products (BDPs), an increasing number of transfers fit within a few BDPs, which entails transfer times that are predominantly a function of queuing and propagation delays. While propagation delays are static, congestion control (CC) manages queuing in the network.

Additionally, the transfer of data that fits within a few packets brings more challenges to CC because it leaves little time for CC to make the correct decisions before the flow completes. The only time CC can kick in is when a congestion signal arrives after each transmitted packet. However, An RPC fitting inside fewer packets means that CC kicks in less frequently. Therefore, CC is under more pressure than ever before to achieve minimal queuing and high link utilization, leaving no room for imperfect control decisions.

Figure 4.1 illustrates how the rising BDPs make the data center workloads *burstier*, necessitating CC to make decisions with *higher quality and timeliness.* The data for this figure was collected in a recent analysis of RPC sizes in Google data centers for BDP sizes at 100Gb/s and 400Gb/s (calculated using a typical base delay/RTT in data centers).

According to the distribution presented in Figure 4.1, the fraction of RPCs that fit within 1 and 4 BDP increases from 62% and 80% at 100Gb/s to 80% and 89% at 400Gb/s. Due to the heavy-tailed

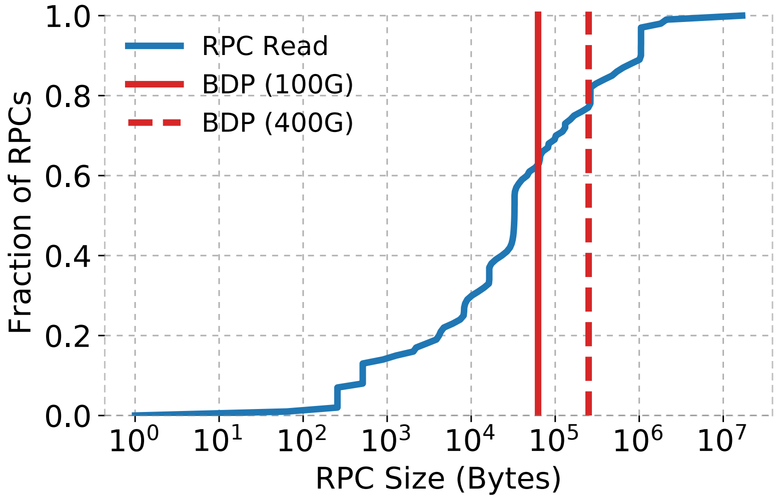


Figure 4.1: RPC size distribution for READ operations in Google's data centers as of 2022

nature of the distribution, this 18% increase is equivalent to 3× more RPC bytes that fit within a single BDP at 400Gb/s. These RPCs are performance-sensitive to queuing and under-utilization, and even a single incorrect or slow CC decision may end up creating tens of microseconds of tail queuing [38], or cause under-utilization [151] in this regime. These events can prolong the flow completion time by a few RTTs. Therefore, an increasing fraction of such short and bursty RPCs raises the bar for the quality and timeliness of CC in minimizing network latencies.

In addition, Figure 4.1 reveals that a 400Gb/s link with just 40% load sees an RPC arrival or completion roughly every RTT! Unfortunately, this is too frequent for CC when the control feedback is inherently delayed by an RTT for each transmitted packet. In other words, when an RPC fits within one BDP, there is no time for CC to correct for congestion before the RPC completes. A sender can only send the entire *small* message at once and hope the network is capable of delivering it with zero queuing for an ideal flow completion time (FCT). In the meantime, throughput is the main contributor to FCT for *long* flows, so the senders must also attempt to utilize the links fully, casually creating queuing.

The work presented in this chapter identifies two key aspects of CC that are imperative to address the challenges of achieving higher CC quality and timeliness on burstier workloads:

First, the collection of **granular feedback** about the location and severity of congestion helps mitigate over/under-reaction by the end-hosts. A precise CC algorithm should receive the exact state of the bottleneck to correctly ramp down during congestion and ramp up during under-utilization.

This congestion signal should involve switch telemetry, such as the current queue occupancy and link utilization because it is the *only* accurate measure of congestion. This way, end-hosts can calculate the *exact* number of packets they can inject into the network without creating congestion. §4.1 argues more about the effectiveness and the feasibility of this accurate signal.

Second, the **control loop delay** is a significant determinant of how sensitive a control algorithm can be. This delay is the time between a congestion event and the reaction from the senders arriving at the bottleneck. The smaller the control loop delay, the more accurate and simpler decisions a control system can make [116]. The state-of-the-art congestion control algorithms in production are reported to work well to the extent their control loop delay allows [91, 102, 175]. However, even a delay of one RTT will be too long for future networks to tolerate because of the increasing BDPs [170]. Moreover, the ever-growing dynamism of workloads makes it inefficient to take multiple RTTs when converging to the fair share, making it necessary to explore mechanisms that reduce the control loop delay to sub-RTT levels. §4.2 demonstrates how sub-RTT feedback mechanisms can be designed as well as their promises.

Fortunately, the flexibility and precision provided by programmable switches [35, 67, 18] allow the design of new mechanisms that can reduce the control loop delay and increase the granularity of control algorithms. These state-of-the-art switches can generate custom control signals to report fine-grained telemetry so that flows don't need to rely on end-to-end measurements for detecting congestion at the bottleneck link.

In this chapter, I present Bolt, an extremely precise CC design with minimal tail latency at very high line rates. It is based on two main mechanisms: First, it harnesses the power of programmable data planes to collect the most precise congestion feedback ever with absolute minimum (sub-RTT) delay. Second, it ramps up flows proactively to occupy available bandwidth promptly. It is therefore both as accurate and responsive as possible. To achieve its accuracy and responsiveness, it applies the "packet conservation" principle [70] to the traffic with accurate per-packet decisions in P4 [23]. The small per-packet `cwnd` changes, combined with Bolt's fine-grained in-network telemetry, limit the effects of noise in the instantaneous congestion signal. Moreover, with Bolt, end-hosts do not make implicit estimations about the severity and exact location of the congestion or the number of competing flows, freeing them from manually tuned hard-coded parameters and inaccurate reactions.

The main contributions of this work are as follows:

1. A discussion of the fundamental limits for an optimal CC algorithm with minimal control loop delay.

2. Description of 3 mechanisms that collectively form the design of Bolt – an extremely precise congestion control algorithm with the shortest control loop possible.

3. Implementation and evaluation of Bolt on P4 switches which achieves 86% and 81% lower RTTs compared to Swift [91] for median and tail respectively.

4. NS3 [138] implementation for large scale scenarios where Bolt achieves up to 3× better 99^{th}-p flow completion times compared to Swift and HPCC [102].

The remainder of this chapter describes the rationale behind the design of Bolt in §4.1 and §4.2. It then provides the design details of Bolt in §4.3 and implementation insights in §4.4. Finally, further evaluations and benchmarks are provided in §4.5, followed by a discussion of practical considerations in §4.6.

4.1 Finding Precise Congestion Signals

4.1.1 Handicap of Surrogate Signals

Loss, ECN, and RTT are widely used signals, and this popularity mainly stems from the fact that they are easily obtained from the network. Given the number of well-defined standards [46, 47, 21], congestion control has evolved around these signals for the last four decades. But all three signals are by-products of the congestion and are not an exact measure of the congestion itself. Next, I consider the quality of each signal in turn.

Loss, of course, is the most common congestion signal and indicates that at least one buffer is full along the forward or reverse path,[1] which means the news of congestion arrives late after the congestion is well underway. This is typically too late for latency-sensitive high-throughput applications.

RTT is the only continuous, high-resolution signal that combines the constant propagation delay and the dynamic queuing delay of switches along the path. Timely [117] uses RTT gradient to infer congestion in the network. The idea is that a change in RTT indicates that the queue occupancy of at least one switch is changing. The end-host should change its `cwnd` only if the change is at the bottleneck link. One could argue that the primary purpose of a congestion control algorithm is to adjust the sending rate of the end-host to match the fair share rate determined by the bottleneck link. But, of course, RTT does not necessarily tell about changes at the bottleneck; an increase in

[1]The non-congestion related loss is ignored here, assuming that it is rare in a modern wireline network

RTT does not mean the bottleneck is more congested and therefore does not mean the end-host should reduce its sending rate. Additionally, RTT is the sum of all the delays encountered along the path, so some queue occupancy values may decrease while others increase making RTT a noisy estimate of congestion. If an end-host reacts to changes in the aggregate queue occupancy, it will react to irrelevant changes at non-bottleneck links, even when its fair share rate at the bottleneck has not changed.

ECN [47] is a single-bit marking on packets by switches in the network. Unlike RTT, the ECN value does not change depending on the distance to the destination. If the queue occupancy on a link is above a threshold, the packet's ECN field is set with a particular probability distribution. Consequently, congestion is completely ignored until the queue occupancy reaches the threshold value which requires relatively larger buffer sizes. If the ECN bit is set, the current bottleneck queue occupancy is guaranteed to be above the threshold. However, the binary ECN value does not indicate how bad the congestion is. Instead, modern schemes such as [4, 175, 122] use ECN marks from consecutive packets to infer the corresponding distribution and estimate the congestion themselves. Some studies suggest that multiple bits of ECN marking improve these algorithms [135, 147], motivating a more precise signal.

4.1.2 A Non-Surrogate Signal - Stamping Queue Occupancy

At the risk of stating the obvious, in a packet-switched wireline network there is a direct relationship between queue occupancy and congestion. They are *equivalent*; by controlling one the other is inherently controlled. The exact measure of current congestion is determined *precisely* by the current queue occupancy (or queuing delay, which is queue occupancy divided by link rate) at links along the path. There is no other direct measure of congestion in a packet-switched wireline network. An end-host armed with an up-to-date measure of queue occupancy at the bottleneck has the best possible signal of congestion.

If the bottleneck buffer goes empty, the capacity is wasted and hence the long-lived flows are needlessly prolonged. Then the end-host should increase its sending rate or `cwnd`. If buffers are too full, packets for short flows are unnecessarily delayed, which requires end-hosts to back off. Therefore, it is reasonable to provide queue occupancy as an explicit congestion signal from the network.

Another way to think about this precise signal, as opposed to RTT measurements, is to consider what happens when a cross-flow causes a non-bottleneck queue to temporarily go non-empty. In such cases, the sender does not need to adjust its `cwnd` to keep the bottleneck link busy because the congestion conditions have not changed. As a consequence, end-hosts need to be able to ignore irrelevant delay information to make the best decision on the `cwnd` or the sending rate.

Obtaining the queue occupancy information from a switch is now easier than ever before. In fact, it was never technically difficult or expensive [65, 96]. Nowadays, any new switch ASIC is required to support INT [88] and hence is already capable of placing queue occupancy information into IP or custom higher layer headers as they pass through.

There are erroneous claims that stamping packets with queue occupancy consumes significant additional power. A modern high-end switch chip consumes very little power reading a queue occupancy, increasing the packet size to hold it, and then placing the value into the header. In one estimate, if every packet was stamped with its queue occupancy, it would add less than 0.05% to the overall chip power. This is in part because the majority of the chip power is expended on fixed overhead (leakage current, serial I/O) and per-packet (not per-bit) processing in a pipeline [9].

In fact, queue occupancy is a simple and free piece of information for the following reasons:

1. **Switches always know the current queue occupancy value.** To maintain one or more FIFO queues, the switch must do internal bookkeeping to keep track of their occupancy.

2. **Switches read the queue occupancy value anyway.** As soon as a packet arrives, a switch needs to read the current occupancy value to decide whether to queue the packet or drop it. Similarly, the queue occupancy value is updated at the time of departure. There is no need to read the value again.

3. **ECN is already a form of queue occupancy stamping.** ECN requires switches to read the current queue occupancy value, compare it to a threshold, and (in the case of RED AQM) toss a coin to decide whether to mark the packet. Queue occupancy stamping would be a very similar procedure except using a larger field in the packet header.

With modern programmable switches [35, 20, 67, 18] and programming languages [23], network owners can redefine how their switches process packets. A 700-line P4 program shows how metadata can be stamped to support INT, showing it is trivial to add queue occupancy stamping to deployed, programmable switches. Furthermore, there is no need to wait for new standards to be defined: the network operator only picks a header location that works in their network and is agreed upon by the end hosts. In a data center, this can be proprietary and chosen to work with existing protocols and boxes. RFC 8592 [22] offers guidelines to stamp packets with metadata. Once such precise information is extracted from the switches, all that is left is to deliver it to the CC decision-makers as soon as possible.

4.2 Towards Minimal Control Loop Delay

Timely feedback and reaction to congestion are well understood to be valuable for CC [117]. The goal of Bolt is to push the limits on minimizing the control loop delay that is composed of two elements: (*I*) *Feedback Delay* (§4.2.1) – the time to receive any feedback for a packet sent, and (*II*) *Observation Period* (§4.2.2) – the time interval over which feedback is collected before `cwnd` is adjusted. Most CC algorithms send a window of packets, observe the feedback reflected by the receiver over another window, and finally adjust the `cwnd`, having a total control loop delay that is even longer than an RTT [4, 29, 54, 91, 102, 175]. This section describes both *Feedback Delay* and *Observation Period* in detail and discusses how these elements can be reduced to their absolute minimum motivating Bolt's design in §4.3.

4.2.1 Feedback Delay

There are two main types of feedback to collect for congestion control purposes: (*I*) *Congestion Notification* and (*II*) *Under-utilization Feedback*.

Congestion Notification

The earliest time a CC algorithm can react to drain a queue is when it first receives the notification about it. Traditionally, congestion notifications are reflected by the receivers with acknowledgments [4, 19, 91, 102, 117, 137, 175]. We call this the RTT-based feedback loop since the delay is exactly one RTT.

An experiment where the congestion notification is delivered to the sender after a constant, configured (i.e., artificial) delay (and not via acknowledgments) demonstrates how notification delay affects performance. Setting this artificial delay to the queuing delay plus the propagation time in the experiment is equivalent to RTT-based control loops described above. The experiment runs two flows with Swift CC [91] on a dumbbell topology[2] where the second flow joins while the first one is at a steady state. The congestion signal is the RTT the packet will observe with current congestion. Figure 4.2 (left) shows the time to drain the congested queue for different notification

[2]RTT is 8 μs and all the links are 100Gb/s.

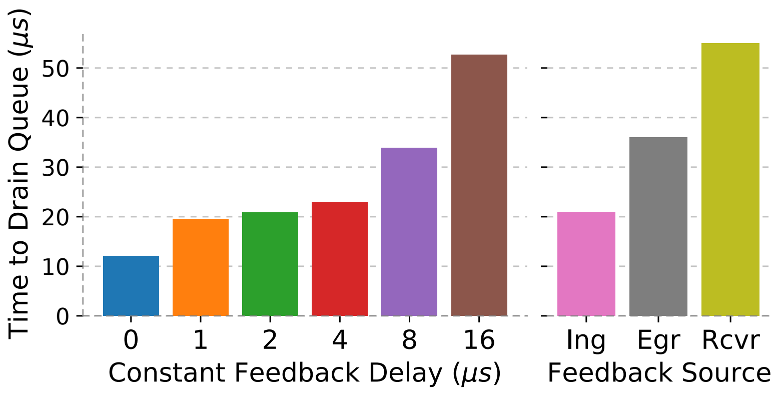


Figure 4.2: Effect of notification delay on queue draining time.

delays. Clearly, smaller notification delay helps mitigate congestion faster as senders react sooner to it.

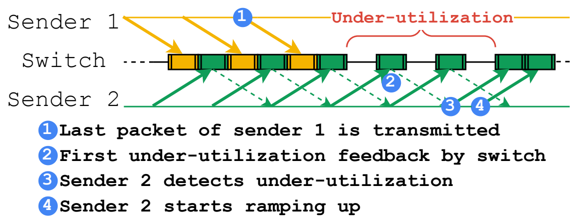
More importantly, in addition to traveling unnecessary links, traditional RTT-based feedback loops suffer from the congestion itself because the notification waits in the congested queue before it is emitted. Adding the queuing delay to the notification delay hinders tackling congestion even more. During severe congestion events, this extra delay can add multiples of the base RTT to the feedback delay [91].

To understand this more, the congestion mitigation time of scenarios where the notification is generated at different locations in the network is also measured in Figure 4.2 (right). "Rcvr" represents the RTT-based feedback loop where the congestion notification is piggybacked by the receiver. "Egr" is when the switch sends a notification directly to the sender from the egress pipeline, after the packet waits in the congested queue. "Ing" is when the notification is generated at the ingress pipeline, as soon as a packet arrives at the switch. As expected, generating the congestion notification as soon as possible improves performance by more than 2×.

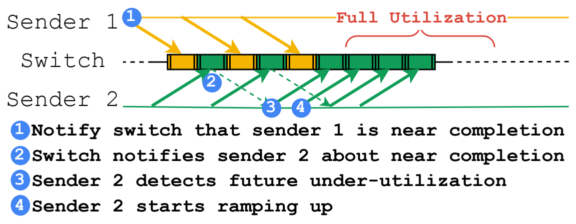
Correspondingly, I stress that in order to reduce the notification delay to its absolute minimum, the congestion notification should travel directly from the bottleneck back to the sender without waiting in the congested queue.

Under-utilization Feedback

While flow arrival events add to congestion in the network, flow completion events open up capacity to be used by other flows. Note that these events play an equally significant role in the network performance because the number of flow completion events in a network is always equal to the number


Sender 1
Switch
Sender 2
Under-utilization
1 Last packet of sender 1 is transmitted
2 First under-utilization feedback by switch
3 Sender 2 detects under-utilization
4 Sender 2 starts ramping up

(a) Reactive Ramp-Up


Sender 1
Switch
Sender 2
Full Utilization
1 Notify switch that sender 1 is near completion
2 Switch notifies sender 2 about near completion
3 Sender 2 detects future under-utilization
4 Sender 2 starts ramping up

(b) Proactive Ramp-Up

Figure 4.3: Under-utilization feedback

of flow arrival events. When a flow completes on a fully utilized link with zero queuing, the packets of the completing flow leave the network and the link will suddenly become under-utilized until the remaining flows ramp up (Figure 4.3a). As traffic gets more dynamic, such under-utilization events become more frequent, reducing the total network utilization. Therefore, in addition to detecting congestion, a good control algorithm should also be able to detect any under-utilization in order to capture the available bandwidth quickly and efficiently [121].

In practice, CC schemes deliberately maintain a standing queue under a steady state, so that when a flow completes, the packets in the queue can occupy the bandwidth released by the finished flow until the remaining flows ramp up [101, 115]. For example, while HPCC was designed to keep near-zero standing queue, the authors followed up that in practice, HPCC target utilization should be set to 150% to improve network utilization [103], which implies half a BDP worth of standing queue. Other CC schemes used in practice also maintain standing queues by filling up the buffers to a certain level before generating any congestion signal [4, 91, 175].

Figure 4.4 demonstrates how Swift behaves upon a flow completion when a long enough standing queue is not maintained. There are two flows in the network[3] and one of them completes at $t = 200\mu$s. The remaining flow's `cwnd` takes about 25 RTTs to occupy the released bandwidth as per the additive increase mechanism in Swift. During this time interval, under-utilization happens despite the non-zero queuing at a steady state. This under-utilization can also be observed when there are a larger number of flows if the standing queue size is not adjusted appropriately [151].

Ideally, any remaining flow should immediately capture the `cwnd` of the completing flow without under-utilizing the link. Therefore, an optimal congestion control algorithm is to detect flow completions early enough, **proactively**, to ramp up as soon as the spare capacity becomes available (Figure 4.3b).

[3]The dumbbell topology from Figure 4.2 (RTT: 8 μs, 100Gb/s links).

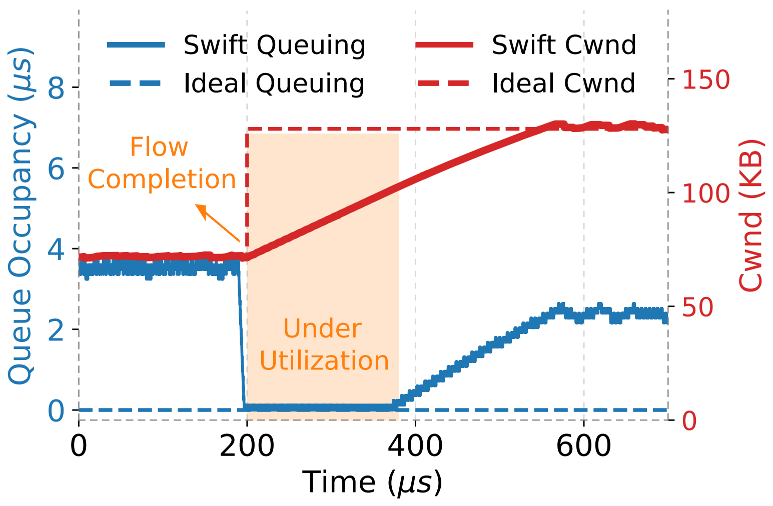


Figure 4.4: `cwnd` of the remaining Swift flow and queue occupancy after a flow completion.

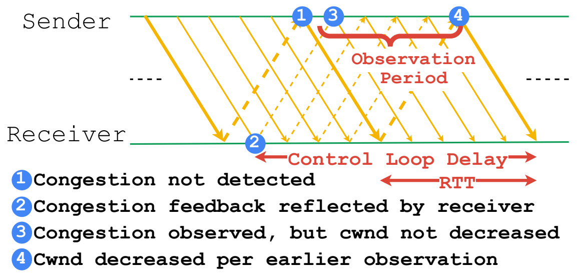

Sender
Receiver
1
2
3
4
Observation Period
Control Loop Delay
RTT
1 Congestion not detected
2 Congestion feedback reflected by receiver
3 Congestion observed, but cwnd not decreased
4 Cwnd decreased per earlier observation

Figure 4.5: Observation period adding up to an RTT to the control loop delay.

4.2.2 Observation Period

In addition to the feedback delay, the total control loop delay is usually one RTT longer for window-based data center CC schemes. Namely, once the sender adjusts its `cwnd`, the next adjustment happens only after an RTT to prevent reacting to the same congestion event multiple times. I call this extra delay the *observation period* and illustrate it in Figure 4.5.

Once-per-window semantics are very common among CC schemes where the per-packet feedback is aggregated into per-window observation. For example, DCTCP [4] counts the number of ECN markings over a window and adjusts `cwnd` based on these statistics once every RTT. Swift compares RTT against the target every time it receives an ACK but decreases `cwnd` only if it has not done so in the last RTT. Finally, HPCC picks the link utilization observed by the first packet of a window to calculate the reference `cwnd` which is updated once per window. As a consequence, flows stick

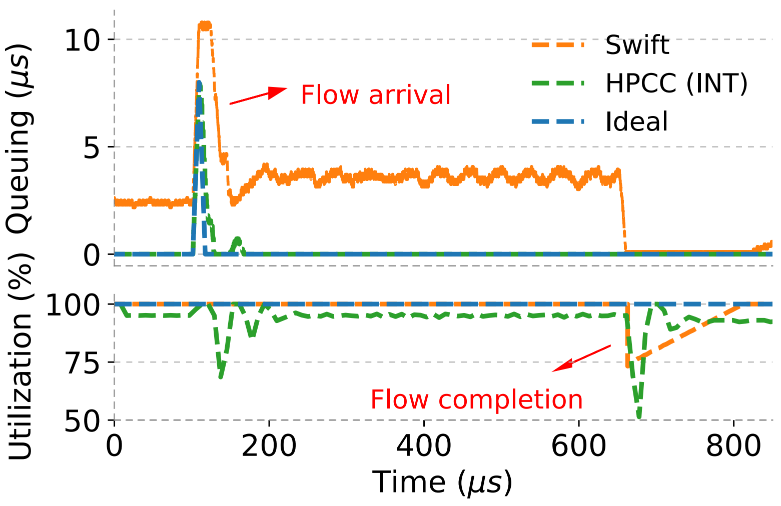


Figure 4.6: HPCC and Swift's reaction to flow arrival and completion versus the ideal behavior.

to their `cwnd` decision for an RTT even if the feedback for a higher degree of congestion arrives immediately after the decision.

Updating `cwnd` only once per window removes information about how dynamic the instantaneous load was at any time within the window. This effect, naturally, results in late and/or incorrect congestion control decisions, causing oscillations between under and over-utilized (or congested) links when flows arrive and depart. Consider the scenario[3] in Figure 4.6 where a new flow joins the network at $t = 100\mu s$ while another flow is at its steady state. HPCC drains the initial queue built up in a couple of RTTs, but immediately oscillates between under-utilization and queuing for a few iterations. Moreover, the completion of a flow at $t = 650\mu s$ again causes oscillations. Under highly dynamic traffic, such oscillations may increase tail latency and reduce network utilization.

An alternative way to avoid oscillations would be to react conservatively similar to Swift. It also reduces `cwnd` only once in an RTT during congestion but uses manually tuned parameters (i.e., ai and β) to make sure reactions are not impulsive. Although oscillations are prevented this way, Figure 4.6 shows that Swift takes a relatively long time to stabilize.

In conclusion, once per RTT decisions can lead to either non-robust oscillations or relatively slow convergence. This is especially problematic in high-speed networks where flow arrivals and completions are extremely frequent. Ideally, the shortest observation period would be a packet's serialization time because it is the most granular decision unit for packet-switched networks. Yet, the per-packet CC decisions should only be incremental to deal with the noise from observations

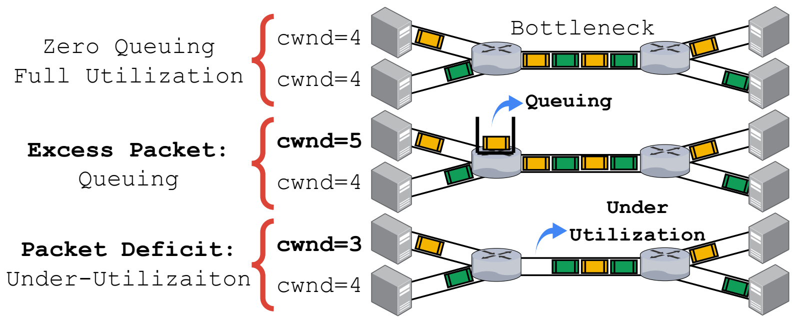


Figure 4.7: Pipe model of Packet Conservation Principle

over such a short time interval.

4.3 Designing Precise and Sub-RTT Congestion Control

Bolt is designed for ultra-low-latency even at very high line rates by striving to achieve the ideal behavior shown in Figures 4.4 and 4.6. The design aims to reduce the control loop delay to its absolute minimum as described in §4.2.1. *First*, the congestion notification delay is minimized by generating notifications at the switches and reflecting them directly to the senders (§4.3.1). *Second*, the flow completion events are signaled by the senders in advance to hide the latency of ramp-up and avoid under-utilization (§4.3.2). *Third*, `cwnd` is updated after each feedback for quick stabilization where the update is at most one per packet to be resilient to noise. Together, these three ideas allow for a precise CC that operates on a per-packet basis minimizing incorrect CC decisions.

Prior works have separately proposed sub-RTT feedback [51, 140, 168], flow completion signaling [53], and per-packet cwnd adjustments [50, 82] which are discussed in §2.2. Bolt's main innovation is weaving these pieces into a harmonious and precise sub-RTT congestion control that is feasible for modern high-performance data centers. The key is to address congestion based on the *packet conservation principle* [70] visualized in Figure 4.7 where a network path is modeled as a pipe with a certain capacity of packets in-flight at a time. When the total `cwnd` is larger than the capacity by 1, there is an excess packet in the pipe which is queued. If the total `cwnd` is smaller than the capacity by 1, the bottleneck link will be under-utilized by 1 packet per RTT. Therefore, as soon as a packet worth queuing or under-utilization is observed, one of the senders should *immediately* decrement or increment the `cwnd`, without a long observation period.

Bolt's fundamental way of minimizing feedback delay and the observation period while generating precise feedback for per-packet decisions is materialized with 3 main mechanisms:

1. **SRC (Sub-RTT Control)** reduces congestion notification delay to its absolute minimum. (§4.3.1)
2. **PRU (Proactive Ramp Up)** hides any feedback delay for foreseen under-utilization events. (§4.3.2)
3. **SM (Supply Matching)** quickly recovers from unavoidable under-utilization events. (§4.3.3)

To realize these 3 mechanisms, Bolt uses 9 Bytes of transport-layer header detailed in listing 4.1. The purpose of each field is explained as the design of Bolt is described in this section and the switching logic for Bolt is summarized in Algorithm 7.

Listing 4.1: Bolt header structure

```
header bolt_h:
  bit<24> q_size;    // Occupancy at the switch
  bit<8>  link_rate; // Rate of congested link
  bit<1>  data;      // Flags data packets
  bit<1>  ack;       // Flags acknowledgements
  bit<1>  src;       // Flags switch feedback
  bit<1>  last;      // Flags last wnd of flow
  bit<1>  first;     // Flags first wnd of flow
  bit<1>  inc;       // Signals cwnd increment
  bit<1>  dec;       // Signals cwnd decrement
  bit<1>  reserved;  // Reserved
  bit<32> t_data_tx; // TX timestamp for data pkt
```

4.3.1 SRC - Sub-RTT Control

As discussed in §4.2.1, a smaller feedback delay improves the performance of CC. Therefore, Bolt minimizes the delay of the feedback by generating control packets at the *ingress* pipeline of the switches – before the data packet waits in the congested queue – and sending them directly *back to the sender*, a mechanism available in programmable switches such as Intel-Tofino2 [94]. While in spirit, this is similar to ICMP Source Quench messages [124] that have been deprecated due to feasibility issues in the Internet [96], Bolt's SRC mechanism exploits precise telemetry in a highly controlled data center environment.

Figure 4.8 depicts the difference in the paths traversed by the traditional ACK-based feedback versus the SRC-based feedback mechanism. As SRC packets are generated at ingress, they establish the absolute minimum feedback loop possible by traveling through the shortest path between a

Algorithm 7: Bolt logic at the switch

```
BoltIngress ($pkt$):
    if not $pkt.data$ then ForwardAndReturn($pkt$)
    CalculateSupplyToken($pkt$)                              ▷ see Algorithm 9
    if $cur\_q\_size \geq CC_{THRESH}$ then                  ▷ Congested
        if not $pkt.dec$ then
            $pkt_{src}.queue\_size \leftarrow switch.q\_size$
            $pkt_{src}.link\_rate \leftarrow switch.link\_rate$
            $pkt_{src}.t\_data\_tx \leftarrow pkt.tx\_time$
            SendSRC($pkt_{src}$)
        $pkt.dec, pkt.inc \leftarrow 1, 0$
    else if $pkt.last$ then                                  ▷ Near flow completion
        if not $pkt.first$ then $pru\_token{+}{+}$
    else if $pkt.inc$ then                                   ▷ Pkt demands a token
        if $pru\_token > 0$ then
            $pru\_token \leftarrow pru\_token - 1$
        else if $sm\_token \geq MTU$ then
            $sm\_token \leftarrow sm\_token - MTU$
        else
            $pkt.inc \leftarrow 0$                           ▷ No token for cwnd inc.
    ForwardAndReturn($pkt$);
```

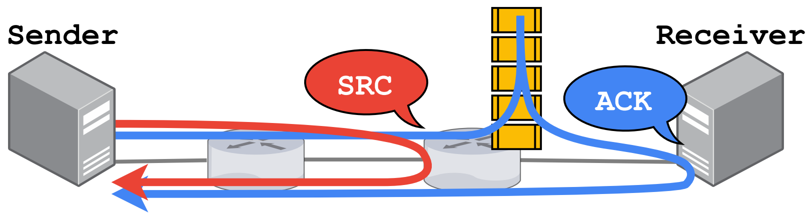


Figure 4.8: Path of ACK-based vs. SRC-based (Sub-RTT) feedback

congested switch and the sender. Moreover, to further minimize the feedback delay, Bolt prioritizes ACK and SRC packets over data packets at the switches.

Bolt generates SRC packets for every data packet that arrives when the queue occupancy is greater than or equal to the CC_{THRESH} which is trivially set to a single MTU for minimal queuing. Yet, if there are multiple congested switches along the path of a flow, generating an SRC at each one of them for the same data would flood the network with an excessive amount of control packets. To prevent flooding switches mark the DEC flag of the original data packet upon generation of an SRC packet, such that no further SRC packets at other hops can be generated due to this packet (lines 5 and 10 in Algorithm 7). This implies that the number of SRC packets is bounded by the number of data packets in the network at any given time. In practice, however, the actual load of

SRC packets is found to be extremely lower (§4.5.2). An approximation for the additional load of SRC packets is presented in §4.6.3.

When there are multiple congested hops, and the flow receives SRC packets only from the first one, the `cwnd` decrement still helps mitigate congestion at all of them. Consequently, even if congestion at the first hop is not as severe as the others, Bolt would drain the queue at the first hop and quickly start working towards the subsequent hops.

Bolt stamps two vital pieces of information on the SRC packets – the current queue occupancy and the capacity of the link. In addition, it reflects the TX timestamp of the original data packet (lines 6-8 in Algorithm 7). As the sender receives this packet, it runs the decision logic shown in Algorithm 8. First, rtt_{src} is calculated as the time between transmitting the corresponding data packet and receiving an SRC packet for it. This is the congestion notification delay for Bolt, which is always shorter than RTT and enables sub-RTT control. The reflection of the TX timestamp enables this computation without any state at the sender. Next, $reaction\_factor$ is calculated as a measure of this flow's contribution to congestion. Multiplying this value with the reported queue occupancy gives the amount of queuing this flow should aim to drain. All the flows aiming to drain only what they are responsible for organically help for a fair allocation.

Finally, $\frac{rtt_{src}}{target_q}$ gives the shortest time interval between two consecutive `cwnd` decrements. This interval prevents over-reaction because switches keep sending congestion notifications until the effect of the sender's `cwnd` change propagates to them. For example, if the target queue has a single packet, the sender decrements its `cwnd` only if rtt_{src} has elapsed since the last decrement. However, if the queue is larger, Bolt allows more frequent decrements to equalize the total `cwnd` change to the target queue size in exactly one rtt_{src}. As the required `cwnd` adjustments are scattered over rtt_{src}, Bolt becomes more resilient to noise from any single congestion notification.

Events such as losses and timeouts do not happen in Bolt as it starts reacting to congestion way in advance. However, due to the possibility of such events occurring, say due to misconfiguration or packet corruption, handling retransmission timeouts, selective acknowledgments, and loss recovery are kept the same as in Swift [91] for completeness.

4.3.2 PRU - Proactive Ramp Up

Bolt explicitly tracks flow completions to facilitate Proactive Ramp Up (PRU). When a flow is nearing completion, it marks outgoing packets to notify switches, which plan ahead on distributing the bandwidth freed up by the flow to the remaining ones competing on the link. This helps

Algorithm 8: Bolt logic at the sender host

```
1  HandleSrc ($pkt_{src}$):
2      $rtt_{src} \leftarrow now - pkt.t\_tx\_data$
3      $reaction\_factor \leftarrow flow.rate/pkt_{src}.link\_rate$
4      $target_q \leftarrow pkt_{src}.queue\_size \times reaction\_factor$          ▷ in number of packets
5      if $\frac{rtt_{src}}{target_q} \leq now - last\_dec\_time$ then
6          $cwnd \leftarrow cwnd - 1$
7          $last\_dec\_time \leftarrow now$
8  HandleAck ($pkt_{ack}$):
9      if $pkt_{ack}.inc$ then          ▷ Capacity available
10         $cwnd \leftarrow cwnd + 1$
11     if $pkt_{ack}.seq\_no \geq seq\_no\_at\_last\_ai$ then
12         $cwnd \leftarrow cwnd + 1$          ▷ per-RTT add. inc.
13         $seq\_no\_at\_last\_ai \leftarrow snd\_next$
```

remaining Bolt flows to *proactively ramp up* and eliminate the under-utilization period after a flow completion (see Figure 4.3b).

When flows larger than one BDP are sending their last `cwnd` worth of data, they set the *LAST* flag on packets to mark that they will not have packets in the next RTT. Note that this does not require knowing the application-level flow size. In a typical transport like TCP, the application injects a known amount of data to the connection at each `send` API call, denoted by the `len` argument [87]. Therefore, the amount of data waiting to be sent is calculable. *LAST* is marked only when the remaining amount of data in the connection is within `cwnd` size. More detailed implementation is described in §4.4.2.

A switch receiving the *LAST* flag, if it is not congested, increments the *PRU token* value for the associated egress port. This value represents the amount of bandwidth that will be freed in the next RTT. The switch distributes these tokens to packets without the *LAST* flag, i.e., flows that have packets to send in the next RTT, so that senders can ramp up proactively.

However, only flows that are not bottlenecked at other hops should ramp up. To identify such flows, Bolt uses a greedy approach. When transmitting a packet, senders mark the *INC* flag on the packet. If a switch has PRU tokens (line 14 in Algorithm 7) or has free bandwidth (line 16 in Algorithm 7, explained in §4.3.3), it keeps the flag on the packet and consumes a token (line 15 and 17, respectively). Else, the switch resets the *INC* flag (line 19), preventing future switches on the path to consume a token for this packet. Then, if no switch resets the *INC* flag along the path, it is guaranteed that all the links on the flow's path have enough bandwidth to accommodate an extra packet. The receiver reflects this flag in the ACK so that the sender simply increments the `cwnd` upon receiving it (lines 9-10 in Algorithm 8). There are cases where the greedy approach can result in wasted tokens. The fallback mechanisms for these cases are discussed in §4.3.3.

Flows shorter than one BDP are not accounted for in PRU calculations. When a new flow starts, its first `cwnd` worth of packets are not expected by the network and contribute to the extra load. Therefore, the switch shouldn't replace these with packets from other flows once they leave the network. Bolt prevents this by setting the *FIRST* flag on packets that are in the first `cwnd` of the flow. Switches check against the *FIRST* flag on packets before they increment the *PRU token* value (line 12 of Algorithm 7).

Note that PRU doesn't need reduced feedback delay via SRC packets, because it accounts for a flow completion in the *next* RTT by design. A sender shouldn't start ramping up earlier as it can cause extra congestion before the flow completes. Therefore, the traditional RTT-based feedback loop is the right choice for correct PRU accounting.

4.3.3 SM - Supply Matching

Events like link and device failures or route changes can result in under-utilized links without proactive signaling. In addition, *PRU tokens* may be wasted if assigned to a flow that can not ramp up due to being already at line rate or bottlenecked by downstream switches. For such events, conventional CC approaches rely on gradual *additive* increase to slowly probe for the available bandwidth which can take several tens of RTTs [4, 91, 117, 175]. Instead, Bolt is able to probe *multiplicatively* by explicitly matching utilization demand to supply through *Supply Matching* (SM) described below.

Bolt leverages stateful operations in programmable switches to measure the instantaneous utilization of a link. Each switch keeps track of the mismatch between the supply and demand for the link capacity for each port, where the number of bytes the switch can serialize in unit time is the supply amount for the link; and the number of bytes that arrive in the same time interval is the demand for the link. Naturally, the link is under-utilized when the supply is larger than the demand, otherwise, the link is congested. Note the similarity to HPCC [102] that also calculates link utilization, albeit from an end-to-end point of view which restricts it to make once per RTT calculations. Bolt offloads this calculation to the switch data plane so that it can capture the precise instantaneous utilization instead of a coarse-grained measurement.

When a data packet arrives, the switch runs the logic in Algorithm 9 to calculate the *supply token* value (sm_token in the algorithms) associated with the egress port. The token accumulates the mismatch between the supply and demand in bytes on every packet arrival for a port. A negative

Algorithm 9: Supply Token calculation at the ingress pipeline for each egress port

```
1 CalculateSupplyToken (pkt):
2     $inter\_arrival\_time \leftarrow now - last\_sm\_time$
3     $last\_sm\_time \leftarrow now$
4     $supply \leftarrow BW \times inter\_arrival\_time$
5     $demand \leftarrow pkt.size$
6     $sm\_token \leftarrow sm\_token + supply - demand$
7     $sm\_token \leftarrow \min(sm\_token, MTU)$
```

value of the token indicates queuing whereas a positive value means under-utilization. When the token value exceeds one MTU, Bolt keeps the *INC* flag on the packet and permits the sender to inject an additional packet into the network (lines 16-17 in Algorithm 7). The *supply token* value is then decremented by an MTU to account for the inflicted future demand.

If a switch port doesn't receive a packet for a long time, the *supply token* value can get arbitrarily large, which prohibits capturing the instantaneous utilization if a burst of packets arrives after an idle period. To account for this, Bolt caps the *supply token* value at a maximum of one MTU. Details on how this feature is implemented in P4 are provided in §4.4.

As noted earlier, there are cases where there can be wasted tokens, i.e., a switch consumes a token (either PRU or SM) to keep *INC* bit but is reset by downstream switches. In such cases, SM will find the available bandwidth in the next RTT. In the worst case, this happens for consecutive RTTs and Bolt falls back to additive increase similar to Swift [91] (lines 12-14 in Algorithm 8). Namely, `cwnd` is incremented once every RTT to allow flows to probe for more bandwidth and achieve fairness even if they do not receive any precise feedback as a fail-safe mechanism.

4.4 Implementing Bolt Congestion Control

Bolt is implemented through Host (transport layer and NIC) and Switch modifications in the lab. Snap [110] is used as the foundational user-space transport layer and Bolt is added in 1340 LOC in addition to the existing Swift implementation. Plus, the switch-side implementation consists of a P4 program – `bolt.p4` – in 1120 LOC. Figure 4.9 shows the overview of the lab prototype as a whole and details are provided below.

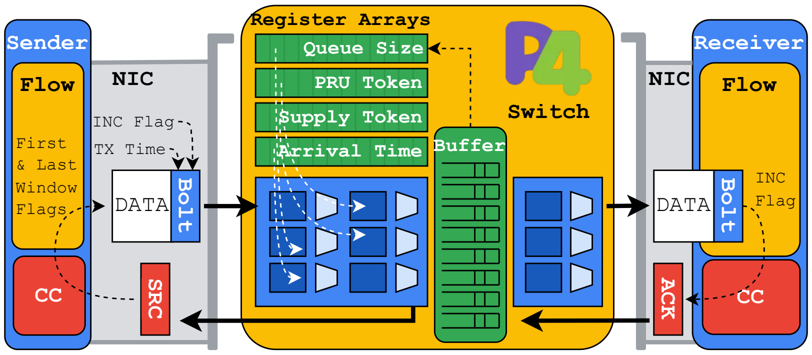


Figure 4.9: Bolt system overview

4.4.1 Switch Prototype

The implementation is based on the programmable data plane of Intel Tofino2 [35] switches as they can provide the queue occupancy of the egress ports in the ingress pipelines and generate SRC packets [94]. This is crucial for Bolt to minimize the feedback delay incurred by SRC packets as they are not subject to queuing delay at congested hops.

When congestion is detected in the ingress pipeline, the switch mirrors this packet to the input port while forwarding the original one along its path. The mirroring configuration is determined with a lookup table that matches the ingress port of the packet and selects the associated mirroring session.

The mirrored packet is then trimmed to remove the payload and the flow identifiers (i.e., source/destination addresses and ports) are swapped. Finally, *SRC* flag is set on this packet to complete its conversion into an SRC packet.

The entire `bolt.p4` consists mainly of register array declarations and simple if-else logic as shown in Algorithm 7. There are 4 register arrays for storing queue occupancy, token values, and the last packet arrival time. All of the register arrays are as large as the number of queues on the switch because the state is maintained per queue. In total, only 3.6% and 0.6% of available SRAM and TCAM, respectively, are used for the register arrays, tables, and counters.

The switch keeps the last packet arrival time for every egress port to calculate the supply for the link. On each data packet arrival, the difference between the current timestamp and the last packet arrival time is calculated as the inter-arrival time. This value should ideally be multiplied with the link capacity (line 4 of Algorithm 9) to find the supply amount. However, since floating point arithmetic is not available in PISA pipelines, a lookup table indexed on inter-arrival times is used to determine the supply amount. The size of this lookup table is set as 65536 where each entry is for a different inter-arrival time with a granularity of a nanosecond. Consequently, if the inter-arrival time is larger than 65 microseconds, the *supply token* value is directly set to its maximum value of 1 MTU which triggers *INC* flag to be set. We find that, at a reasonably high load, 65 microseconds of inter-arrival time is rare enough for links greater than 100Gb/s such that any longer value can be safely interpreted as under-utilization.

The Bolt prototype is based on a single HW pipeline. Therefore, the Bolt logic is implemented entirely at the ingress pipeline to make it easier to understand and debug its logic. However, since PRU and SM maintain state per egress port, they could also be implemented at the egress pipeline with minor modifications. This way, the state for packets from multiple ingress pipelines would naturally be aggregated.

4.4.2 Host Prototype

The Snap transport layer uses the NIC hardware timestamps to calculate rtt_{src} as described in Algorithm 8. When a sender is emitting data, the TX timestamp is stamped onto the packet. The switch reflects this value back to the sender, so that rtt_{src} is the difference between the NIC time when the SRC packet is received (RX timestamp) and the reflected TX timestamp. This precisely measures the network delay to the bottleneck without any non-deterministic software processing delays.

The transport layer also multiplexes RPCs meant for the same server onto the same network connection. Then, the first `cwnd` bytes of a new RPC aren't necessarily detected as the *first* window of the connection. To mitigate this issue, the Bolt prototype keeps track of idle periods of connections and resets the *bytes-sent counter* when a new RPC is sent after such a period. Therefore the *FIRST* flag is set on a packet when the counter value is smaller than `cwnd`.

Finally, the *last* window marking for PRU requires determining the size of the remaining data for each connection. In the prototype, the connection increments *pending bytes counter* by the size of data in each `send` API call from the application. Every time the connection transmits a packet into the network, the counter value is decremented by the size of the packet. Therefore the *LAST* flag is set on a packet when this counter value is smaller than `cwnd`. On a standard Linux kernel implementation, equivalent signaling can be achieved by setting the *LAST* flag whenever the send buffer occupancy is smaller than `cwnd`.

Alternatively, some modern applications know the size of the flows they will create in advance, e.g., distributed ML training [136]. If such applications could reveal this information to the transport layer, Bolt could use it to start PRU signaling as well. However, developing this interface between the applications and the transport layer would be disruptive to existing implementations of networking stacks and applications. Hence, Bolt's design focuses on self-contained transport layer solutions and leaves inter-layer collaboration as an interesting future work.

4.4.3 Security and Authentication

Getting Bolt to work for encrypted and authenticated connections was a key challenge in the lab. The prototype uses a custom version of IPsec ESP [68, 85] for encryption atop the IP Layer. However, switches need to read and modify CC information at the transport header without breaking end-to-end security. The *crypt_offset* of the protocol allows packets to be encrypted only beyond this offset. It is set such that the transport header is not encrypted, but is still authenticated.

In addition, switches cannot generate encrypted packets due to the lack of encryption and decryption capabilities. To remedy this, SRC packets are generated on switches as unreliable datagrams per RoCEv2 standard by adding IB BTH and DETH headers while removing the encryption header.

The RoCEv2 packets have the invariant CRC calculated over the packet and appended as a trailer. Fortunately, Tofino2 provides a CRC extern that is capable of this calculation over small, constant-size packets [92]. As a result, NICs are able to forward the SRC packets correctly to the upper layers based on the queue pair numbers (QPN) on the datagrams.

4.5 Evaluating Bolt

Bolt is evaluated on NS3 [138] micro-benchmarks to demonstrate its fundamental capabilities in §4.5.1 followed by sensitivity and fairness analysis in §4.5.2 and §4.5.3. Then, large-scale experiments are run in §4.5.4 to measure the end-to-end performance of the algorithm, i.e., flow completion time slow-downs. Finally, the lab prototype is evaluated in §4.5.5.

4.5.1 Micro-Benchmarks

Significance of SRC

The only way for Bolt to decrease `cwnd` is through SRC whose effectiveness is best observed during congestion. Therefore, the same flow arrival scenario described in Figure 4.6 is repeated with Bolt.[4]

[4]The dumbbell topology with two flows (8 μs RTT at 100Gb/s).

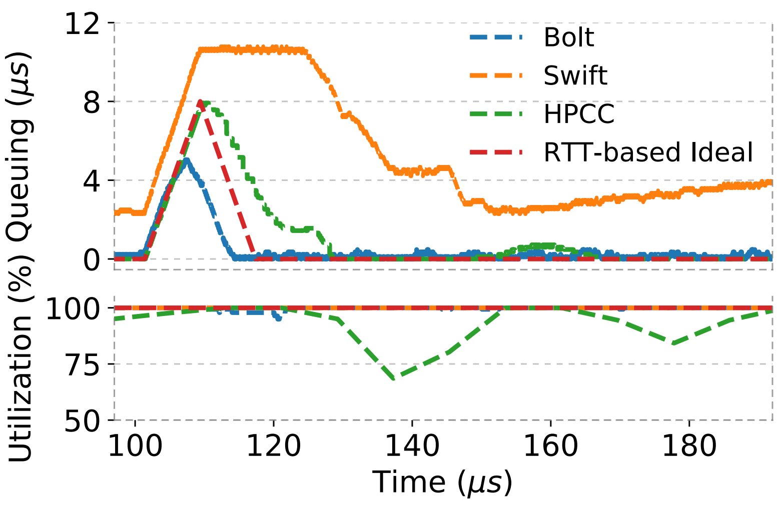


Figure 4.10: Bolt's reaction to flow arrival versus the ideal behavior.

Typically, with conventional RTT-based congestion control algorithms, a new flow starting at line rate emits BDP worth of packets until it receives the first congestion feedback after an RTT. If the network is already fully utilized before this flow, all emitted packets end up creating a BDP worth of queuing even for an RTT-based ideal scheme. Then, the ideal scheme would stop sending any new packets to allow draining the queue quickly which would take another RTT. This behavior is depicted as red in Figure 4.10 where a new flow joins at 100μs.

HPCC's behavior in Figure 4.6 is close to the ideal given that it is an RTT-based scheme with a high precision congestion signal. As the new flow arrives, the queue occupancy rises to 1 BDP. However, the queue is drained at a rate slower than the link capacity because flows continue to occasionally send new packets while the queue is not completely drained.

Bolt, on the other hand, detects congestion earlier than an RTT. Therefore it starts decrementing `cwnd` before the queue occupancy reaches BDP and completely drains it in less than 2 RTTs, even shorter than the RTT-based ideal scheme.

In addition, HPCC's link utilization drops to as low as 75% after draining the queue and oscillates for some time, which is due to the RTT-long observation period (§4.2.2). Bolt's per-packet decision avoids this under-utilization.

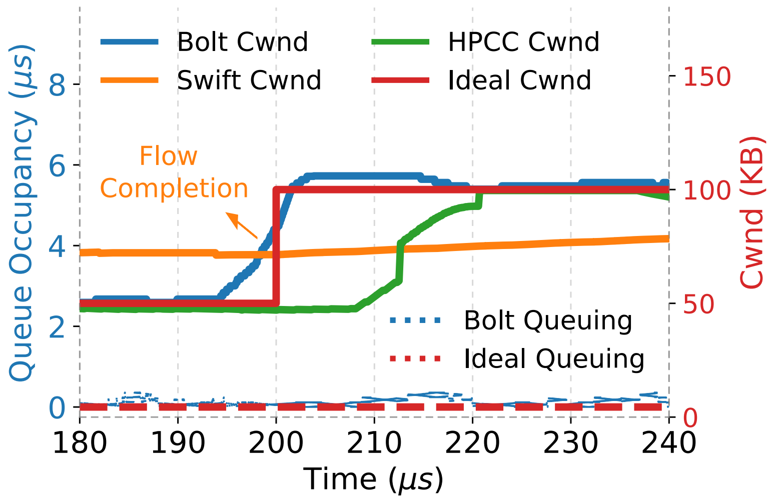


Figure 4.11: Queuing and `cwnd` of the remaining flow after a flow completes. See Figure 4.4 for the complete ramp-up of Swift.

Significance of PRU

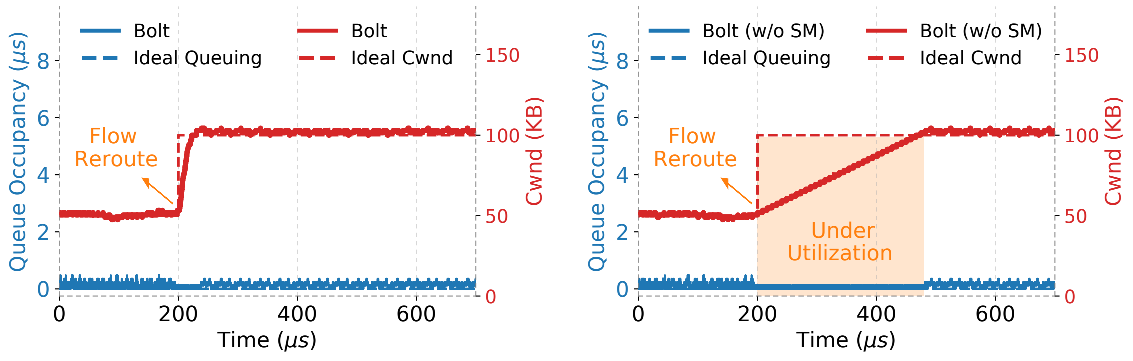
Flow completions cause under-utilization without proactive ramp-up or standing queues because conventional congestion control algorithms take at least an RTT to react to them (§4.2.1). Moreover, as shown in Figure 4.4 for Swift, a standing queue might not be enough to keep the link busy if the `cwnd` of the completing flow is larger than the queue size.

The same scenario is repeated with Bolt to test how effective proactive ramp-up can be upon flow completions against Swift and HPCC. Figure 4.11 shows the `cwnd` of the remaining flow and the queue occupancy at the bottleneck link. When a Bolt flow completes at t=200μs, the remaining one is able to capture the available bandwidth in 1μs because it starts increasing `cwnd` (by collecting PRU tokens) one RTT earlier than the flow completion. Moreover, neither queuing nor under-utilization is observed. HPCC, on the other hand, takes 20μs ($> 2\times$RTT) to ramp up for full utilization because it needs one RTT to detect under-utilization and another RTT of observation period before ramping up. Finally, Swift takes more than 370μs to reach the stable value due to the slow additive increase approach which doesn't fit into Figure 4.11. The complete ramp-up of Swift is shown in Figure 4.4.

Although PRU and SM seem to overlap in the way they quickly capture available bandwidth, PRU is a faster mechanism compared to SM because it detects under-utilization proactively. To demonstrate that, a star topology is created with 100Gb/s links and a base RTT of 5μs, where 5 senders send 500KB to the same receiver. Flows start 15μs apart from each other to complete at

Utilization (%)		**PRU OFF**	**PRU ON**
SM	**OFF**	90.46	97.38
	ON	92.41	98.54

Table 4.1: Effectiveness of Bolt's PRU and SM on the bottleneck utilization.



(a) Bolt

(b) Bolt (without SM)

Figure 4.12: `cwnd` of the remaining Bolt flow and queue occupancy after a flow is rerouted.

different times so that PRU and SM can kick in. The experiment is repeated while disabling PRU or SM and measuring the bottleneck utilization to observe how each mechanism is effective at achieving high throughput.

Table 4.1 shows the link utilization between the first flow completion and the last one. When only PRU is disabled, the utilization drops by 6% despite having SM. On the other hand, disabling SM alone causes only a 1% decrease. This indicates that PRU is a more powerful mechanism compared to SM when under-utilization is mainly due to flow completions in the network. Together, they increase utilization by 8%.

Significance of SM

Unlike flow completions, events such as link failure or rerouting are not hinted in advance. Then, PRU doesn't kick in, making Bolt completely reliant on SM for high utilization. To demonstrate how SM quickly captures available bandwidth, the same setup from Figures 4.4 and 4.11 are used, but the second flow is rerouted instead of letting it complete.

Figure 4.12 shows the `cwnd` of the remaining flow after the other one leaves the bottleneck. Thanks to SM, `cwnd` quickly ramps up to utilize the link in 23μs (Figure 4.12a). When SM is disabled, the only way for Bolt to ramp up is through traditional additive increase which increases

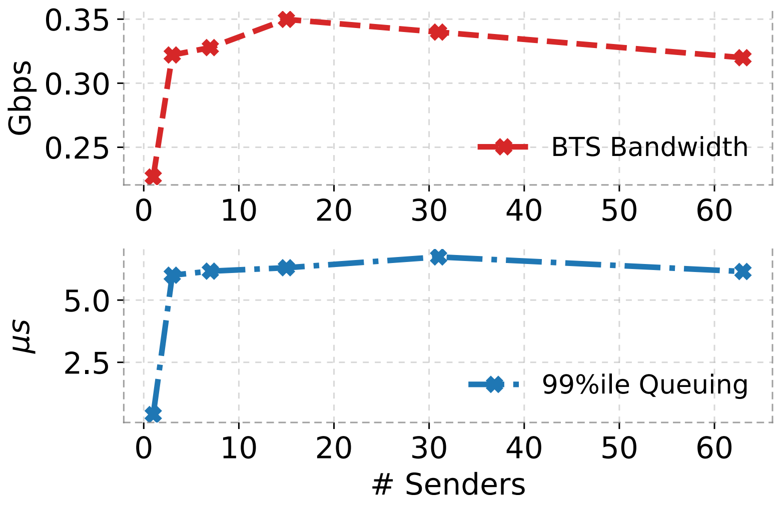


Figure 4.13: SRC overhead and sensitivity for different levels of burstiness

`cwnd` by 1 every RTT (Figure 4.12b). Therefore it takes more than 33 RTTs to fully utilize the link.

4.5.2 Sensitivity Analysis

Overhead of SRC

To mitigate congestion, Bolt generates SRC packets in an already-loaded network. In order to understand the extra load created by SRC, the bandwidth occupied by SRC packets at different burstiness levels is measured. For this purpose, the same star topology from §4.5.1 is used. The number of senders changes between 1 and 63 to emulate different levels of burstiness towards a single receiver at 80% load. The traffic is based on the READ RPC workload from Figure 4.1.

Figure 4.13 shows the bandwidth occupied by the SRC packets (top) and the 99^{th}-p queue occupancy at the bottleneck (bottom) with a different number of senders. When there are multiple senders, the SRC bandwidth is stable at 0.33Gb/s (0.33% of the capacity). Similarly, the tail queuing is also bounded below 6.4μs for all the experiments. Therefore, one can conclude that Bolt is able to limit congestion with a negligible amount of extra load in the network. In §4.5.5, it is shown that the overhead is negligible for the lab prototype as well.

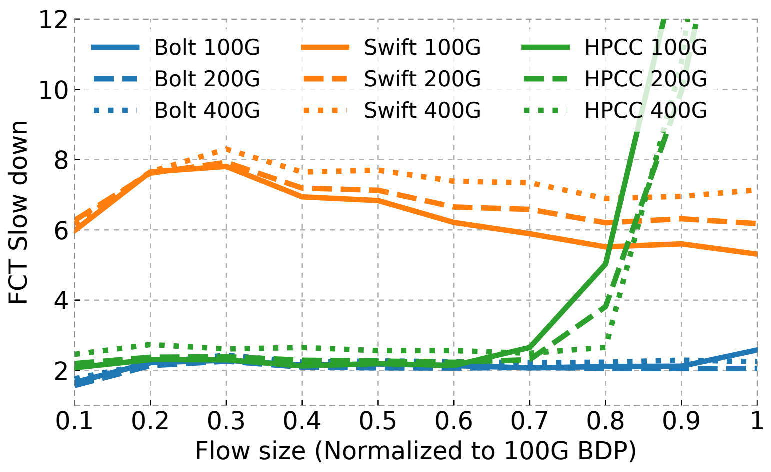


Figure 4.14: 99^{th}-p Slowdown for messages smaller than BDP at line rates higher than 100Gb/s

Robustness Against Higher Line Rates

One of the goals of Bolt is to be robust against ever-increasing line rates in data centers. To evaluate the performance at different line rates, the previous simulations is repeated with 63 senders where the link capacity is increased from 100Gb/s to 200Gb/s and 400Gb/s. This way, the burstiness of the senders increases, making it difficult to maintain small queuing at the switches. Therefore, flow completion time (FCT) slowdown[5] of small flows are affected the most, whereas throughput oriented large flows would trivially be better off with higher line rates [43].

Accordingly, only the 99^{th}-p FCT slowdown for flows that are smaller than BDP (at 100 Gb/s) are plotted in Figure 4.14. Swift's performance monotonically decays with higher link rates due to the increasing burstiness. Similarly, HPCC at 400Gb/s achieves 25% worse performance compared to the 100Gb/s scenario for flow sizes up to 0.7 BDP. For the rest of the workload, HPCC makes a leap such that it performs worse than other algorithms irrespective of the line rates. Bolt on the other hand is able to maintain small and steady tail slowdowns for all the small flows despite the increasing line rates.

4.5.3 Fairness Analysis

The fairness of Bolt was tested with an experiment on a dumbbell topology with 100Gb/s links. In this experiment, a new flow is added or an existing one is removed every 10 milliseconds. Then, the

[5]FCT slowdown is flow's actual FCT normalized by its ideal FCT when the flow sends at line-rate, e.g., when it was the only flow in the network.

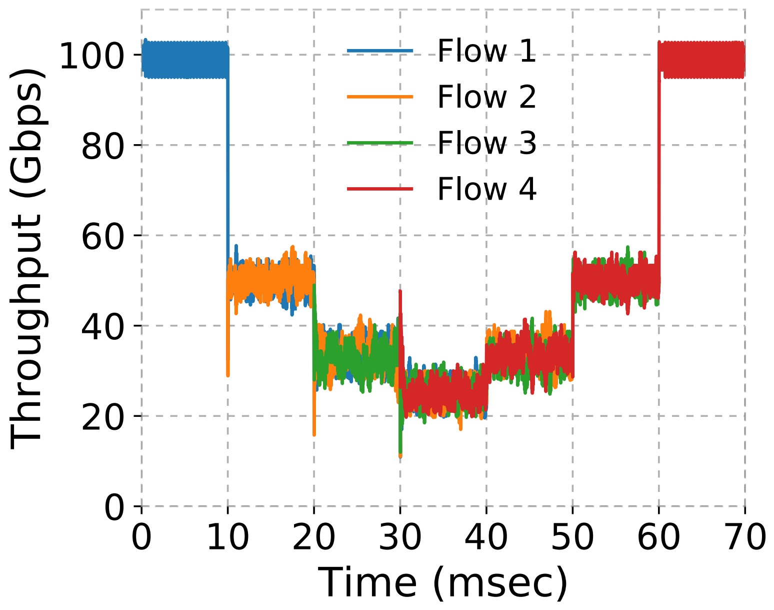


Figure 4.15: Fair share allocation by Bolt

Metric	**Swift**	**HPCC**	**Bolt**
99th-p Queuing (msec)	23.543	23.066	13.720
99th-p FCT Slowdown	7017	5037	5000

Table 4.2: Tail queuing, and FCT slowdown for Bolt, HPCC, and Swift in a 5000-to-1 incast.

throughput of each flow is measured which is shown in Figure 4.15. The results indicate that Bolt flows converge to the new fair share quickly when the state of the network changes.

4.5.4 Large Scale Simulations

One of the most challenging cases for CC is a large-scale incast. To evaluate Bolt's performance in such a scenario, a 5000-to-1 incast is set up on the star topology described earlier where each one of 50 senders starts 100 same-size flows at the same time. Table 4.2 presents the 99th-p queue occupancy and FCT slowdown for the incast. Since Bolt detects congestion as early as possible, it bounds tail queuing to a 41% lower level compared to Swift and HPCC. In addition, the tail FCT slowdown for Bolt is 5000, indicating full link utilization. Moreover, the bandwidth occupied by the SRC packets is as low as 0.77Gb/s throughout the incast. This is only twice the overhead for 80% load in §4.5.2, despite the extreme bursty arrival pattern of the incast.

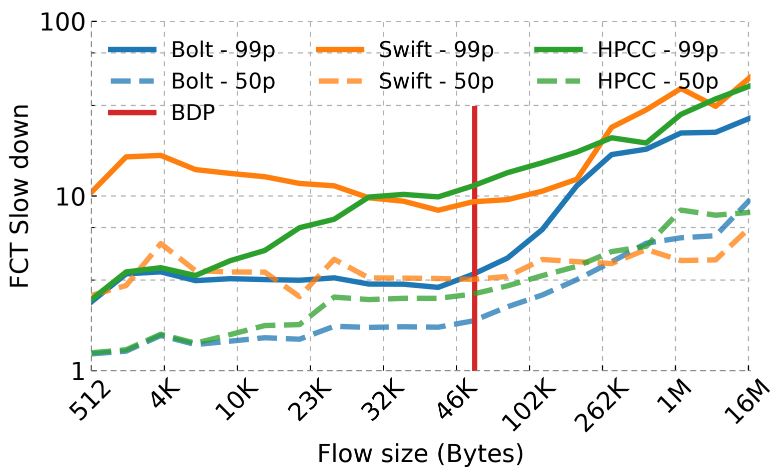


Figure 4.16: FCT slowdown for READ RPC Workload from Figure 4.1

The performance of Bolt on a cluster-scale network is evaluated as well. In this simulation, 64 servers are connected with 100Gb/s links to a fully subscribed fat-tree topology with 8 ToR switches. All the other links in the simulation are 400Gb/s and the maximum unloaded RTT is 5μs. The servers run traffic based on two workloads at 80% load: (i) the READ RPC workload described in Figure 4.1 represents traffic from Google data centers, (ii) the Facebook Hadoop workload [139]. Figure 4.16 and 4.17 show the median and 99^{th}-p FCT slowdown for the workloads. Note that the Hadoop workload is relatively more bursty where 82% of the flows/RPCs fit within a BDP in the given topology. Hence a large fraction of the curves in Figure 4.17 is flat where all the RPCs in this region are extremely small (i.e., single packet).

For both of the workloads, Bolt performs well across all flow sizes. Specifically, Bolt and HPCC achieve very low FCT for short flows (<7KB) because of a few design choices: First, they maintain zero standing queues. Plus, Bolt's SRC reduces the height of queue spikes after flow arrivals. HPCC, on the other hand, tends to under-utilize the network upon flow completions (§4.2.1), statistically reducing queue sizes.

FCT of median-size flows (a few BDPs) starts to degrade for HPCC due to under-utilization described in §4.2.1 and §4.2.2. Bolt performs up to 3× better in this regime by avoiding under-utilization thanks to PRU and SM. Swift's standing queues prevent under-utilization, but FCTs are high because median-size flows are also affected by the queuing delay.

The impact of queuing diminishes and utilization becomes the dominant factor for long flows. Therefore Bolt and Swift perform better than HPCC. In addition, Bolt is slightly better at the tail

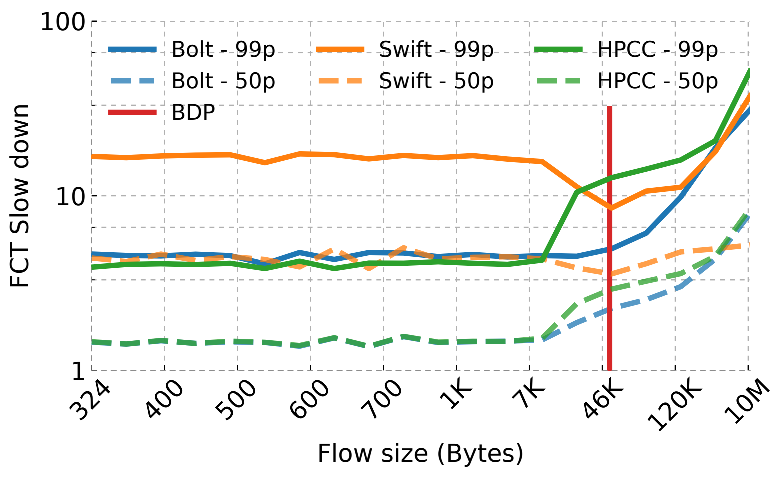


Figure 4.17: FCT slowdown for Facebook Hadoop Workload

compared to Swift, while Swift is slightly better at the median, suggesting that Bolt is fairer.

4.5.5 Bolt in the Lab

Bolt's lab testbed consists of 2 servers and 2 Intel Tofino2 [35] switches. Each server runs 4 packet processing engines running Snap [110] that provide the transport layer with the Bolt algorithm. Each engine is scheduled on a CPU core that independently processes packets so that a large number of connections can be created between the servers. Links from the servers to the switches are 100Gb/s and the switches are connected with a 25Gb/s link to guarantee that congestion takes place within the network. The base-RTT in this network is 14μs and flows between the servers are generated based on the READ RPC workload.

Bolt is evaluated on two scenarios. First, a 100% (of 25Gb/s) load is run to see if the prototype can saturate the bottleneck. Then, a 80% load is run to compare the congestion mitigation performance of Bolt against Swift in a more realistic scenario. Finally, the results from the lab and the simulations are compared to verify the simulator implementations.

The median and the 99^{th}-p RTT at 100% load for Swift are 189μs and 208μs respectively. These numbers are high because Swift maintains a standing queue based on the configured base delay to fully utilize the link even after flow completions. Bolt, on the other hand, attains 27μs and 40μs of median and tail RTT, 86% and 81% shorter than Swift. In the meantime, it achieves 24.7Gb/s which is only 0.8% lower compared to Swift despite the lack of a standing queue.

The same experiment is repeated with 80% load. The results show that both Swift and Bolt

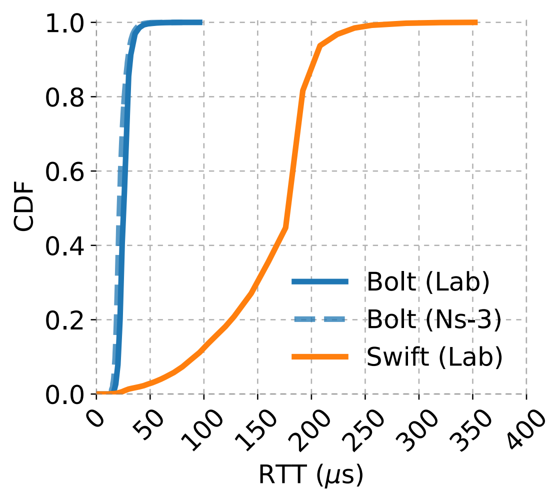


Figure 4.18: Bolts's lab prototype matches its simulator

can sustain 80% (20Gb/s) average link utilization. Figure 4.18 shows the CDF of measured RTTs throughout the experiment. Similar to the 100% load case, the median and tail RTTs for Bolt are 25μs and 40μs, 86% and 83% lower compared to Swift respectively.

For Swift in this experiment, the base target delay is set as 50μs and the flow scaling range as 200μs, which are the specified values in the paper [91]. Swift's average RTT in Figure 4.18 is higher than Swift paper's value (~50μs), because of two reasons. First, this workload is burstier than the ones in Swift paper. Second, the 25Gb/s bottleneck implies a higher level of flow scaling than with 100Gb/s links.

Moreover, the bandwidth occupied by the SRC packets is measured in the lab as 0.13Gb/s, 0.536% of the bottleneck capacity. This is consistent with the observation in §4.5.2 despite the larger SRC packets with custom encapsulations.

Finally, the 80% load experiment is simulated in NS3 [138] with the same settings to verify that the simulator matches the observations in the lab. Figure 4.18 also shows the CDF of RTTs measured throughout the simulation. The median and tail RTTs from the simulations are 21μs and 39μs, within 15% and 0.025% of the lab results respectively. These results give confidence when interpreting the results of larger-scale simulations of Bolt.

4.6 Discussion

4.6.1 Practical Considerations

Typically, new products are deployed incrementally in data centers due to availability, security, or financial concerns. As a consequence, the new product (i.e., the CC algorithm) lives together with the old one for some time called brownfield deployment. There are four potential issues that Bolt could face during this phase, which are addressed below.

First, some switches in the network may not be capable of generating SRC packets while new programmable switches are being deployed. Unfortunately, the vanilla Bolt design cannot control the congestion at these switches. This can be addressed by running an end-to-end algorithm on top of Bolt. For example, imagine the Swift algorithm calculates a fabric `cwnd` as usual in parallel with Bolt's calculation of `cwnd` using SRC packets. Then, the minimum of the two is selected as the effective `cwnd` for the flow. When an older generation switch is congested, SRC packets are not generated, but Swift adjusts the `cwnd`. Consequently, flows benefit from ultra-low queuing at the compatible switches while falling back to Swift when a non-programmable switch becomes the bottleneck.

Moreover, while queue occupancy stamping is quite promising, the use of other signals might still be beneficial to end-hosts. For example, a congestion control algorithm might measure RTT in addition to queue occupancy stamps, similar to combining RTT and ECN as in [169]. While bottleneck queue occupancy helps the end-host to converge to max-min fair rate allocation, RTT value could be used to determine proportionally fair resource allocation across the network. A flow with high RTT but low bottleneck queue occupancy could still be throttled to prevent over-utilization of other network resources.

The second possible issue with Bolt's deployment would be the incremental migration of end-hosts to Bolt. Then, Bolt would need to coexist with the prior algorithm. Studying the friendliness of algorithms with Bolt through frameworks such as [72] and [162] remains a future work. For example, TCP CUBIC would not coexist well with Bolt as it tries to fill the queues until a packet is dropped while Bolt continuously decrements its `cwnd` due to queuing. Instead, the use of QoS (Quality of Service) queues can be useful to isolate Bolt traffic from the rest. §4.6.2 describes a baseline approach for such deployment.

Third, the transport offloading on modern smart NICs uses batching to sustain high line rates. Scenarios where packet transmissions are batched even when the `cwnd` is smaller than BDP can still trigger SRC generation, inhibiting flows to increase `cwnd` to the right value. Bolt can alleviate such bursts with a higher CC_{THRESH} that tolerates batch size worth of queuing at the switches.

Finally, Bolt is designed to run on Tofino programmable switch architecture, but Intel has discontinued the development of the future generations for Tofino. This should not be a game-stopper for Bolt because programmability was just a utility while designing the algorithm. Now that it is designed and evaluated, its logic can also be implemented on fixed-function ASICs to assist network owners if needed. Essentially, most switches that are capable of exposing switch buffer occupancy, maintaining counters (i.e., token values), and stamping packets based on those counter values can be configured to run the Bolt logic. Validating this hypothesis on an actual commercial switch remains as an interesting industry challenge.

4.6.2 Bolt with QoS

The relationship between congestion control algorithms and QoS has always been contradictory. An ideal congestion control algorithm aims to mitigate any queuing at the switch, whereas a QoS mechanism always needs enough queuing to be able to differentiate packet priorities and serve one before the other. Put another way, QoS only takes effect when the arrival rate at a link is greater than the capacity such that it causes queue build-up. Yet, QoS is vital for commercial networks to be able to differentiate applications or tenants for business-related reasons [15]. This is particularly true for unavoidable transient congestion events, e.g., incast.

Fortunately, the way Bolt reports queue occupancy is QoS-agnostic such that it can generate SRC packets with the occupancy of the queue assigned by the QoS mechanism. Consequently, it would try to minimize queuing at that particular queue. Similarly, the way *PRU tokens* are calculated would be queue-specific instead of being egress port-specific. For example, if there are P ports on a switch and n QoS levels per port, the size of the register array that maintains the token values would be of $P \times n$ and flows would only be able to proactively ramp up if another flow with the same QoS level is about to finish.

On the other hand, accounting for the *supply token* requires the service rate for the associated queue (§4.3.3) which would be a dynamic value depending on the current demand for different QoS levels. There are at least two approaches for maintaining *supply tokens* correctly and implementing a QoS-aware version of Bolt on programmable switches.

Algorithm 10: Supply Token calculated for QoS queue i at the switch with n QoS levels serving the same egress port

```
1  Function CalculateSupplyToken(pkt):
2      $inter\_arrival\_time \leftarrow now - last\_sm\_time$
3      $last\_sm\_time \leftarrow now$
4      $w_{effective} \leftarrow 0$
5      for $j \leftarrow 0$ to $n$ do
6          if $i = j$ or $q\_size_j \neq 0$ then
7              $w_{effective} \leftarrow w_{effective} + w_j$
8      $supply \leftarrow BW \times inter\_arrival\_time \times (\frac{w_i}{w_{effective}})$
9      $demand \leftarrow pkt.size$
10     $sm\_token \leftarrow sm\_token + supply - demand$
11     $sm\_token \leftarrow \min(sm\_token, MTU)$
```

Ideal Approach

Imagine a scenario where weighted fair queuing [131] is applied for QoS purposes. Then, Bolt would need to be able to increment the *supply token* value based on the weight associated with the QoS level (w_i) and the link capacity (C) as well as the demand for each QoS level. For example, when all QoS levels have at least 1 packet in their queue, a packet arriving at QoS level i should increment the token value by $C \times w_i \times t_{inter-arr}$. Where $\sum_{\forall i} w_i = 1$ and $t_{inter-arr}$ is the time between the arrival of the most recent packet and the previous one on the associated QoS queue.

If a QoS queue is empty, its weight is distributed to other QoS levels in proportion to each level's own weight. Therefore, Bolt should adjust the *supply token* value of QoS level i based on the logic presented in Algorithm 10.

Note that in order to be able to determine the service rate of each queue, queue occupancy of other queues would be required. This requirement creates a challenge for P4 switches since only one queue's occupancy can be read at a time. A workaround to this would be to create shadow register arrays for each priority queue where they get updated whenever a value is not being read from them. Moreover, the calculation at line 8 of Algorithm 10 requires floating point arithmetic which could be addressed via lookup tables.

Heuristic Approach

A simpler mechanism to enable QoS on Bolt switches would be to introduce probabilistic SRC generation where higher priority traffic has a lower probability of generating an SRC packet. This would naturally keep the rates of high-priority flows high while throttling others. Yet, an extensive empirical study would be required to determine the probabilities such that the queuing for all the QoS levels is bounded to some extent. Given that Bolt aims to minimize tail queuing in the network, introducing probabilistic behavior into its design may not be a preferable strategy. Any trade-off between deterministic congestion mitigation and QoS is left to network operators' discretion if such a heuristic approach is pursued.

4.6.3 Approximating SRC Overhead

Bolt switches generate SRC packets for every data packet they receive as long as there is queuing, given that the data packet is not marked with the *DEC* flag. Then the number of SRC packets in the network depends on how long queuing persisted and how many packets are received in this time interval.

At a steady state where no new RPCs join the network, one can estimate the fraction of time queuing persists on a bottleneck. Congestion at this regime happens only due to the once-per-RTT additive increase of 1 by each flow.

As described in §4.3.1, senders pace `cwnd` decrements such that the total number of decrements equals the queue occupancy after 1 rtt_{src}. This implies that any queuing will persist for 1 rtt_{src}, but will be completely drained after. Since new congestion is not inflicted until the next RTT, the fraction of time that the switch has non-zero queuing is governed by the following golden ratio:

$$\text{fraction of time switch is congested} = \frac{rtt_{src}}{rtt} \tag{4.1}$$

which is always less than 1.

Note that equation 4.1 is an approximation for congestion interval since it doesn't incorporate traffic load, new RPC arrivals, or multi-bottleneck scenarios. Nonetheless, one can calculate the number of SRC packets generated at a bottleneck with it.

$$\text{\# of SRC pkts} = \text{\# of DATA pkts} \times \frac{rtt_{src}}{rtt} \tag{4.2}$$

Finally, equation 4.2 can be mapped to the bandwidth occupied by the SRC packets by incorporating the link capacity and the packet sizes:

$$\text{SRC Bandwidth} = C \times \frac{p_{src}}{p_{data}} \times \frac{rtt_{src}}{rtt} \tag{4.3}$$

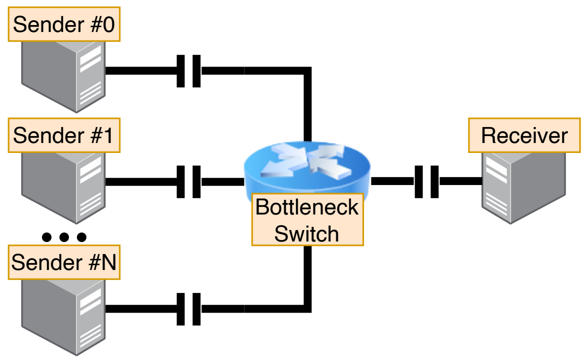


Figure 4.19: Simplified network topology for the theoretical analysis.

Where C is the rate at which the traffic is flowing through the bottleneck link, p_{src} is the size of SRC packets, and p_{data} is the size of data packets, i.e., MTU.

The bandwidth of SRC packets calculated according to equation 4.3 for the simulation in §4.5.2 is 0.37Gb/s which is within 12% of the simulation result of 0.33Gb/s. Moreover, equation 4.3 gives 0.10Gb/s for the lab setup in §4.5.5 which is within 23% of the measured value of 0.13Gb/s.

4.6.4 Buffer Sizing Analysis for Sub-RTT Control

Figure 4.2 shows that a reduced signaling delay such as the sub-RTT feedback primarily helps to minimize queuing in the network as it would allow senders to detect and react to congestion much earlier, i.e., before a large queue builds up. Therefore, I ask *What is the minimum buffer size a switch should have given a feedback delay such that it can tolerate a congestion event without dropping any packets?* To explore a theoretical boundary to this question, I will assume optimal flow control and congestion control algorithms that set the transmission rate or `cwnd` to the fair share immediately after receiving the first Congestion Notification (CN) after pausing an ideal amount of time to drain the existing congestion.

Imagine the topology in Figure 4.19 where N+1 flows are running through the same bottleneck switch towards the receiver. Note that this topology is not necessarily a two-hop star topology. Plus, the bottleneck switch does not need to be the last hop for this analysis. The links shown in the figure abstract away all the other networking elements and model them as a single link since they do not mandate flow's transmission rate or `cwnd`.

Define the following:

- C: Bottleneck link capacity (i.e., bits per second)

- t_{pre}: Propagation delay between the senders and the bottleneck switch. Every flow may have a different delay to the bottleneck switch, but only the largest among senders is considered for the worst-case analysis.
- t_{post}: Propagation delay between the bottleneck switch and the receiver.
- t_s: The serialization delay of an MTU size packet. (i.e., $t_s = MTU/C$) The control packets are assumed to have zero serialization delay.
- $rtt = 2t_{pre} + 2t_{post} + 2t_s$.
- $rtt_{src} = 2t_{pre} + t_s$.
- k: The minimum number of packets in a buffer to trigger a CN (i.e., $k = CC_{THRESH}/MTU$).

Let's start with a simple scenario and progressively generalize the case.

2 Flow Congestion

Suppose Sender#0 is at a steady state, transmitting at the line rate without queuing at the bottleneck. At $t = 0$, Sender#1 starts at the line rate.

The first CN will be generated when the k^{th} packet of Sender#1 arrives at the switch. Without BTS, the CN is reflected by the receiver with ACK packets after waiting in the queue first, i.e., one-way-delay (OWD) for OnRamp [108]. Therefore the first CN will be observed by a sender at $t = 2t_{pre} + 2k \times t_s + 2t_{post}$. Even if the senders immediately decrease their rate or `cwnd` to the new fair share, all the packets Sender#1 has sent so far are going to create queuing at the bottleneck switch. The size of the buffer required to accommodate all these packets, B_{ack}, can be calculated as:

$$B_{ack} = C \times (2t_{pre} + 2k \times t_s + 2t_{post}) \tag{4.4}$$

In the case of Sub-RTT control, the sender will receive the first CN at $t = 2t_{pre} + k \times t_s$. Therefore, the required amount of buffer for accommodating the congestion becomes

$$B_{src} = C \times (2t_{pre} + k \times t_s) \tag{4.5}$$

Note that $B_{src} < B_{ack}$ and the saving for buffer space is $C \times (2t_{post} + k \times t_s)$. For a scenario where $C = 100Gb/s$, $t_{pre} = 4\mu$s, $t_{post} = 1\mu s$, $k = 1$, and $MTU = 4000$B, this saving is 29KB or 2.32μs of unnecessary queuing which corresponds to 21.8% reduction in the maximum congestion. Note that this saving increases when a larger value of k is used. For example, as discussed in §4.6.1, network owners may choose to use a higher CC_{THRESH} for tolerating the batching behavior of sender NICs. When k is set to 16 packets, the buffer utilization savings increase to 89KB or 7.12μs.

N Flow Incast

This time, suppose $N > k$ additional flows start at the line rate at $t = 0$ while Sender#0 is at a steady state. Then the first CN would be emitted by the switch when the first packets of each sender arrive at the bottleneck switch which will be observed by the senders at $t = 2t_{pre} + t_s$. Then,

$$B_{incast} = N \times C \times (2t_{pre} + t_s) = N \times C \times rtt_{src} \tag{4.6}$$

Without Sub-RTT control, the first CN would be delivered to one of the senders at $t = rtt$ whereas other senders would receive a CN later depending on the order in which their data packets arrive at the congested queue, which implies that $B_{ack} > N \times C \times rtt$. Therefore, the buffer space required without Sub-RTT control would be more than $\frac{rtt}{rtt_{src}}$ times higher compared to the use of SRC packets. Note that this is the reciprocal of the golden ratio introduced in §4.6.3, and it is calculated as 1.28 for the numerical example given previously. This implies 21.8% buffer savings with Sub-RTT control compared to traditional RTT-based control loops.

Chapter 5

Conclusions

5.1 Dissertation Takeaways

Cloud and data center applications are gaining more adoption in daily tasks. To accommodate this increasing demand, they are developed in a more distributed fashion for scalability and with more stringent requirements from the infrastructure to keep up. These requirements include high bandwidth, which is addressed by deploying higher-capacity links in the network. However, there is mostly ultra-low latency requirement as well. This requirement necessitates more sophisticated solutions than just deploying more infrastructure. In particular, the tail latency has become the determining factor of performance for many applications and larger scales of applications inherently make it harder to achieve lower tail latencies.

This dissertation focuses on two locations in data centers that can impact the tail latency for the applications: (I) *The end-host;* How the incoming and outgoing packets are processed determines how fast the data can be emitted or delivered from/to the applications. (II) *The network;* How the network resources are allocated determines how much queuing the packets experience while traveling over the network. The first takeaway is that **the latency must be minimized both at the end-host and the network to reduce overall tail latency**.

In Chapter 2, I laid out the existing literature on low-latency architectures and algorithms. On the end-host side, off-loading the networking stack onto hardware is the best alternative to eliminate non-deterministic latencies and support the ever-increasing line rates. Yet, despite decades of research, the community has not identified a single transport protocol that performs the best for every edge case – there is no one-size-fits-all. The innovation continues with newer congestion signals and control algorithms for further lowering the network latencies.

Then, I asked *"Is there a physical limit to how much we can reduce the tail latency in a data center?"* and more importantly, *"How can we reach there?"* My answers to these questions are presented in Chapter 3 and Chapter 4.

In Chapter 3, I presented nanoTransport, which offloads the transport layer into pipelined NIC hardware to achieve the lowest possible packet processing latency while running at the line rate. It adopts a one-way reliable message delivery interface that supplies ready-to-use messages to CPU cores or an RDMA engine with orders of magnitude lower tail latencies. At the same time, nanoTransport utilizes a P4 programmable NIC architecture that allows data plane programmers to implement their choice of transport protocols on hardware. The key takeaway from this work is that **it is possible to build a very high throughput (200Gb/s) NIC, with a transport layer that is very low latency (11ns round-trip), yet is programmable to ease innovation**.

Finally, in Chapter 4, I presented Bolt, which studies how to minimize tail latencies in the network with a faster and more accurate congestion signal. First, it uses the most granular congestion signal, i.e., precise queue occupancy, for a per-packet decision logic. Second, it minimizes the control loop delay to its absolute minimum by generating feedback at the congested switches and sending them directly back to the senders. Third, it hides the control loop delay by making proactive decisions about foreseeable flow completions. As a result, `cwnd` is calculated as accurately and quickly as possible, achieving more than 80% reduction in tail latency and $3\times$ improvement in tail FCT compared to the production algorithms of Google and Alibaba. The key takeaway from this work is that **sub-RTT and precise congestion feedback – along with proactive decision-making – is the key to fast and accurate congestion control in highly dynamic data center environments**. Fortunately, the flexibility provided by programmable switches enables easier prototyping for such algorithms, hopefully inspiring fixed-function ASIC designers to support such timely and accurate signals in their products in the future.

5.2 Future Directions

Given that nanoTransport reduces packet processing latency to hardware limits at the end-hosts and Bolt reduces feedback loop delay to its absolute minimum in the network, can we say we are done with the transport layer latency problem in data centers? The answer to this question is a big no.

The network topologies, link capacities, hardware technologies, available compute resources, and the scale of data centers are changing rapidly and are likely to continue changing in the future. Each of these factors potentially affects the optimal transport protocol and congestion control algorithm for a data center.

For example, as **the available memory size and the speed to read/write to/from this memory increases** in a switch, keeping per-flow state might become feasible even for core data center switches. Then, revisiting solutions like BFC [51] for large-scale deployments would be more promising.

Another emerging trend is the **use of optical circuit switching** in data centers [105]. It essentially forms a slow-moving circuit switch where the mirrors in the network's core are reconfigured rather seldom to provision direct optic links for the duration of a job, or to provision capacity to a whole cluster. This type of switching avoids many dynamics of packet switching networks such as switch buffering and in-network packet losses, achieving close to zero fiber-to-fiber latency. Moreover, it has the benefit of consuming much less energy than an electronic packet switch. Hence, the optimal congestion control, flow control, or scheduling logic will need to be reexamined for such environments in the future.

In addition to changing networking and hardware production technologies, **the applications and their networking requirements also keep evolving**. For example, distributed machine learning training and inference has recently become one of the most popular applications in modern data centers. The communication pattern and the user behavior for such machine learning applications are unique and require special protocols and algorithms to perform well in constrained environments. In particular, high bandwidth and low packet loss rate are the most prominent networking requirements, instead of low tail latency, for the training tasks.

To comply with the unique requirements of ML models, the community has been training models mainly on Infiniband-based networks. Even then, networking is considered the bottleneck for such tasks, and community effort for transport protocols with linear scale-out is needed [171]. However, the proprietary nature of Infiniband makes infrastructure scaling slower while also making it more difficult to innovate on the existing infrastructure. Hence, there is likely going to be a desire to transition to Ethernet environments, which will open up many opportunities to explore custom transport protocols that can satisfy the specific requirements of various ML tasks while being friendly to other low-latency applications in the data center.

Obviously, optimizing the data center transport layer only for ML applications would not be wise. Many other applications are also mission-critical for many data centers, e.g., distributed storage or financial trading. Since the networking requirements of these applications are likely different from each other, **being able to adapt to the dynamic requirements of any application in real-time** plays a significant role in making sure the performance is optimized.

For example, an application that does not require its RPC request to return until some other computation finishes locally could inform the transport protocol, so that the transport protocol would prioritize other flows for improved overall user experience. Such a capability will likely require a smarter interface between the applications and the transport layer. Designing this interface for wide adoption remains to be an interesting future work.

On the other hand, the latency requirements of algorithmic trading have pushed the financial industry toward a completely different approach. Even the nanoseconds spent for packet processing at the NICs and switches can not be accepted when competing against other traders. Hence, switching and routing are handled in the physical layer [86], which makes the transport layer invisible to the network. Then, one would need to reexamine how congestion is signaled while scaling up the trading system with such a limited budget for packet processing times.

Finally, the energy consumption of data centers increases in parallel with the rising popularity of online applications. The electricity usage of these facilities is estimated to be approximately 3% of the global consumption [59], a figure projected to rise significantly in the future [7, 63]. Therefore, any **improvement in the energy efficiency of communications** will be beneficial – both environmentally, and financially for network owners. In this regard, I value exploring how congestion control, in particular, affects the energy footprint of data centers as a starting point. In a preliminary work, I showed that energy efficiency can be increased when congestion control algorithms approximate the Shortest Remaining Processing Time First paradigm as opposed to fair resource allocation per-flow [9]. This result suggests that we as a community should rethink our current approach to congestion control and potentially more for substantial savings and environmental impact.

5.3 Concluding Remarks

We are still in the early days of cloud computing. The slowing of Moore's Law and Dennard Scaling means single-core performance is leveling off. New applications must be distributed across an ever-increasing number of cores. Cloud service providers and their customers are still learning how to develop large, and fast distributed applications that perform well on a shared infrastructure. This trend is helped by steadily increasing network speeds. Server NICs have transitioned quickly from 10Gb/s to 25Gb/s, 100Gb/s, and now 400Gb/s. However, the benefits of "many cores and a fast network" are often lost because of an inefficient NIC design, or software in the network stack.

It is therefore natural to consider offloading the transport layer into pipelined NIC hardware which runs at the line rate with very low latency. Despite the performance benefits of such hardware – i.e., higher line rates, lower latencies, and lower energy footprint – baking a particular logic on the hardware fixes it for many years in production. Yet, the network topologies, traffic patterns, flow sizes, performance metrics, and performance requirements constantly evolve in data centers. **One must keep up with this evolution to extract the best performance out of data centers.**

The evolution of data centers is not the enemy of us, the networking and systems people. Instead, it is an exciting challenge for us to solve interesting and impactful problems. We should rapidly adapt our tools and techniques to attack this challenge. This thesis demonstrates that this is possible, without compromising throughput or latency.

Naturally, keeping up with the evolving data centers depends on having the flexibility to program the stack and deploy tailored protocols in a relatively short period. However, some data center owners still use fixed-function ASICs. This slows them down to try out new protocols and algorithms mainly because it takes years to add features to the next generations of hardware and deploy them. On the other hand, **programmable networking equipment, i.e., NICs and switches, can be deployed once; while the production logic can be modified as frequently as needed**.

Given that modern programmable networking equipment is comparable to fixed-function ASICs in terms of speed, power, and cost, the capability to innovate with rapid prototyping essentially comes for free. Such programmable equipment is already commercially off-the-shelf for packet processing at high line rates. And some large data center owners have already transitioned to such equipment, e.g., Mount Evans IPUs at Google, Azure SmartNICs at Microsoft, and Tofino switches at Alibaba. Hence, it is likely to have all of the networking equipment programmable in the foreseeable future. Even if this prophecy does not come true in the future, **programmability will continue to be the primary enabler of rapid prototyping and research, especially in determining the best network behavior for a new workload**. Then fixed-function ASIC vendors will be able to make better proven decisions about which features to support in their products.

Bolt is an excellent example to show how easy innovative prototyping can be with programmable networking equipment. It is designed by a network owner and an academic researcher but can advise equipment vendors about what kind of congestion mitigation features should be implemented in future products. In addition, it suggests that **there is great value in enabling network owners to design and implement their custom designs with programmable architectures**.

NanoTransport is just a step in the direction of enabling programmability in dynamic data centers. Similar SmartNICs are gaining adoption among large data center owners such as Google and Microsoft. Therefore it is very timely to think about how we can innovate with them to reach the physical lower bounds of packet processing latency while optimizing the network resource allocations. It will certainly be very exciting to see how much further such innovations can impact our digital experience in the future.

Bibliography

[1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51–70, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/addanki.

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/aguilera.

[3] F. Akujobi, I. Lambadaris, R. Makkar, N. Seddigh, and B. Nandy. BECN for congestion control in TCP/IP networks: study and comparative evaluation. In *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE*, volume 3, pages 2588–2593 vol.3, 2002. doi:10.1109/GLOCOM.2002.1189098.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.*, 40(4):63–74, August 2010. URL: https://doi-org.stanford.idm.oclc.org/10.1145/1851275.1851192, doi:10.1145/1851275.1851192.

[5] 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*, NSDI'12, page 19, USA, 2012. USENIX Association.

[6] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. PFabric: Minimal near-Optimal Datacenter Transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, August 2013. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/2534169.2486031`, `doi:10.1145/2534169.2486031`.

[7] Anders S. G. Andrae and Tomas Edler. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges*, 6(1):117–157, 2015. URL: `https://www.mdpi.com/2078-1547/6/1/117`, `doi:10.3390/challe6010117`.

[8] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, Santa Clara, CA, February 2020. USENIX Association. URL: `https://www.usenix.org/conference/nsdi20/presentation/arashloo`.

[9] Serhat Arslan, Sundararajan Renganathan, and Bruce Spang. Green With Envy: Unfair Congestion Control Algorithms Can Be More Energy Efficient. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 220–228, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3626111.3628200`.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/2254756.2254766`, `doi:10.1145/2254756.2254766`.

[11] AvidThink and Converge. Myth-busting DPDK in 2020, 2020 [Online]. URL: `https://www.dpdk.org`.

[12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, New York, NY, USA, 2012. IEEE, IEEE Press. `doi:10.1145/2228360.2228584`.

[13] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. PIAS: Practical Information-Agnostic Flow Scheduling for Commodity Data Centers. *IEEE/ACM Trans.*

Netw., 25(4):1954–1967, August 2017. URL: `https://doi-org.stanford.idm.oclc.org/10.1109/TNET.2017.2669216`, `doi:10.1109/TNET.2017.2669216`.

[14] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Commun. ACM*, 60(4):48–54, March 2017. `doi:10.1145/3015146`.

[15] Luiz Andrè Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool Publishers, San Rafael, CA, USA, 3rd edition, October 2018. URL: `https://doi-org.stanford.idm.oclc.org/10.2200/S00874ED3V01Y201809CAC046`.

[16] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay`.

[17] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3387514.3405894`.

[18] Tanya Bhatia. UADP - The Powerhouse of Catalyst 9000 Family. Cisco Systems Inc., December 2018. URL: `https://community.cisco.com/t5/networking-blogs/uadp-the-powerhouse-of-catalyst-9000-family/ba-p/3764605`.

[19] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. RFC 5681, September 2009. URL: `https://rfc-editor.org/rfc/rfc5681.txt`, `doi:10.17487/RFC5681`.

[20] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/2486001.2486011`, `doi:10.1145/2486001.2486011`.

[21] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, page 24–35, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/190314.190317`.

[22] Rory Browne, Andrey Chilikin, and Tal Mizrahi. Key Performance Indicator (KPI) Stamping for the Network Service Header (NSH). RFC 8592, May 2019. URL: `https://rfc-editor.org/rfc/rfc8592.txt`, `doi:10.17487/RFC8592`.

[23] Mihai Budiu and Chris Dodd. The P4-16 Programming Language. *SIGOPS Oper. Syst. Rev.*, 51(1):5–14, September 2017. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3139645.3139648`, `doi:10.1145/3139645.3139648`.

[24] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3452296.3472888`.

[25] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control. *Commun. ACM*, 60(2):58–66, January 2017. URL: `http://doi.acm.org/10.1145/3009824`, `doi:10.1145/3009824`.

[26] V. Cerf and R. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, 1974. `doi:10.1109/TCOM.1974.1092259`.

[27] The Networking Channel. Network Programmability: The Road Ahead, Jul 2023. URL: `https://www.youtube.com/watch?v=CtxfmES4T7E`.

[28] Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. Optimizing in-memory database engine for AI-powered on-line decision augmentation using persistent memory. *Proc. VLDB Endow.*, 14(5):799–812, jan 2021. `doi:10.14778/3446095.3446102`.

[29] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 239–252, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3098822.3098840`, `doi:10.1145/3098822.3098840`.

[30] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314. USENIX Association, November 2020. URL: `https://www.usenix.org/conference/osdi20/presentation/cho`.

[31] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. Increasing TCP's Initial Window. RFC 6928, April 2013. URL: `https://rfc-editor.org/rfc/rfc6928.txt`, `doi:10.17487/RFC6928`.

[32] Cisco. Nexus SmartNIC, May 2021 [Online]. URL: `https://www.cisco.com/c/en/us/products/interfaces-modules/nexus-smartnic/index.html`.

[33] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 106–114, New York, NY, USA, 1988. ACM. URL: `http://doi.acm.org/10.1145/52324.52336`, `doi:10.1145/52324.52336`.

[34] Intel Corporation. Comparing FPGAs, Structured ASICs, and Cell-Based ASICs. `https://www.intel.com/content/www/us/en/products/programmable/fpga-vs-structured-asic.html`, 2021. Accessed on 2021-05-09.

[35] Intel Corporation. Tofino 2: Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise, May 2021. URL: `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html`.

[36] Alpha Data. ADM-PCIE-9V3, April 2019 [Online]. URL: `https://www.alpha-data.com/pdfs/adm-pcie-9v3.pdf`.

[37] J.D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983. `doi:10.1109/PROC.1983.12775`.

[38] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013. URL: `http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext`.

[39] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483571.

[40] Li Ding, Ping Kang, Wenbo Yin, and Linli Wang. Hardware TCP Offload Engine based on 10-Gbps Ethernet for low-latency network communication. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 269–272, 2016. doi:10.1109/FPT.2016.7929550.

[41] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: formal foundations for p4 data planes. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi:10.1145/3434322.

[42] Nandita Dukkipati. *Rate Control Protocol (Rcp): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3292347. URL: https://dl-acm-org.stanford.idm.oclc.org/doi/10.5555/1368746.

[43] Nandita Dukkipati and Nick McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, January 2006. doi:10.1145/1111322.1111336.

[44] Andy Fingerhut, Radostin Stoyanov, and Nate Foster. Portable NIC Architecture, May 2021 [Online]. URL: https://github.com/p4lang/pna/blob/main/generated-html/PNA-v0.5.0.pdf.

[45] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018.

USENIX Association. URL: https://www.usenix.org/conference/nsdi18/presentation/firestone.

[46] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, 1999. URL: https://rfc-editor.org/rfc/rfc2582.txt, doi:10.17487/RFC2582.

[47] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001. URL: https://rfc-editor.org/rfc/rfc3168.txt, doi:10.17487/RFC3168.

[48] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, Prashant Pradhan, and John Tracey. Server Network Scalability and TCP Offload. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/server-network-scalability-and-tcp-offload.

[49] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery. URL: https://doi-org.stanford.idm.oclc.org/10.1145/2716281.2836086, doi:10.1145/2716281.2836086.

[50] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. ABC: A Simple Explicit Congestion Controller for Wireless Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, Santa Clara, CA, February 2020. USENIX Association. URL: https://www.usenix.org/conference/nsdi20/presentation/goyal.

[51] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/goyal.

[52] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, April 2012. URL: `https://rfc-editor.org/rfc/rfc6582.txt`, `doi:10.17487/RFC6582`.

[53] Dongsu Han, Robert Grandl, Aditya Akella, and Srinivasan Seshan. FCP: A Flexible Transport Framework for Accommodating Diversity. *SIGCOMM Comput. Commun. Rev.*, 43(4):135–146, August 2013. `doi:10.1145/2534169.2486004`.

[54] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3098822.3098825`, `doi:10.1145/3098822.3098825`.

[55] Jörn-Thorben Hinz, Vamsi Addanki, Csaba Györgyi, Theo Jepsen, and Stefan Schmid. TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 1–7, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3609021.3609295`.

[56] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3458817.3476223`.

[57] Shuihai Hu, Wei Bai, Baochen Qiao, Kai Chen, and Kun Tan. Augmenting Proactive Congestion Control with Aeolus. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, APNet '18, page 22–28, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3232565.3232567`, `doi:10.1145/3232565.3232567`.

[58] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 60–68, New York, NY, USA, 2019. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/`

3365609.3365856, doi:10.1145/3365609.3365856.

[59] AFL Hyperscale. What Makes Hyperscale, Hyperscale?, March 2023. URL: https://www.aflhyperscale.com/articles/what-makes-hyperscale-hyperscale/.

[60] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-Driven Packet Processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 133–140, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi-org.stanford.idm.oclc.org/10.1145/3365609.3365848, doi:10.1145/3365609.3365848.

[61] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, and Muhammad Shahbaz. nanoPU GitHub, August 2020 [Online]. URL: https://github.com/l-nic.

[62] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Nick McKeown, and Changhoon Kim. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, Boston, MA, July 2021. USENIX Association. URL: https://www.usenix.org/conference/osdi21/presentation/ibanez.

[63] IEA. Data Centres and Data Transmission Networks, Fetched June 25th, 2023. https://www.iea.org/reports/data-centres-and-data-transmission-networks.

[64] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006. doi:10.1109/IEEESTD.2006.99495.

[65] IEEE. IEEE Standard for Local and Metropolitan Area Networks– Virtual Bridged Local Area Networks Amendment 13: Congestion Notification. *IEEE Std 802.1Qau-2010 (Amendment to IEEE Std 802.1Q-2005)*, pages 1–135, 2010. doi:10.1109/IEEESTD.2010.5454063.

[66] IEEE. IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011*, (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011):1–40, 2011. doi:10.1109/IEEESTD.2011.6032693.

[67] Broadcom Inc. High-Capacity StrataXGS Trident4 Ethernet Switch Series, May 2021. URL: `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series`.

[68] Google Inc. PSP, March 2022. URL: `https://github.com/google/psp`.

[69] Versa Technology Inc. 400G Ethernet: It's Here, and It's Huge, December 2021. URL: `www.versatek.com/400g-ethernet-its-here-and-its-huge/`.

[70] Van Jacobson. Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 314–329, New York, NY, USA, 1988. Association for Computing Machinery. `doi:10.1145/52324.52356`.

[71] Van Jacobson and Robert Braden. TCP extensions for long-delay paths. RFC 1072, October 1988. URL: `https://rfc-editor.org/rfc/rfc1072.txt`, `doi:10.17487/RFC1072`.

[72] Raj Jain, Dah-Ming Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR*, cs.NI/9809099, January 1998. URL: `https://arxiv.org/abs/cs/9809099`.

[73] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association. URL: `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong`.

[74] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast String Searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 21–28, New York, NY, USA, 2019. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3314148.3314356`, `doi:10.1145/3314148.3314356`.

[75] Theo Jepsen, Stephen Ibanez, Gregory Valiant, and Nick McKeown. From Sand to Flour: The Next Leap in Granular Computing with NanoSort, 2022. `arXiv:2204.12615`.

[76] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136,

New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3132747.3132764`, `doi:10.1145/3132747.3132764`.

[77] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for microsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association. URL: `https://www.usenix.org/conference/nsdi19/presentation/kaffes`.

[78] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3477132.3483548`.

[79] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association. URL: `https://www.usenix.org/conference/nsdi19/presentation/kalia`.

[80] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, jun 2015. `doi:10.1145/2872887.2750392`.

[81] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, New York, NY, USA, 2018. IEEE, IEEE Press.

[82] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 89–102, New York, NY, USA, 2002. Association for Computing Machinery. `doi:10.1145/633025.633035`.

[83] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/2872362.2872367`, `doi:10.1145/2872362.2872367`.

[84] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3302424.3303985`.

[85] Stephen Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005. URL: `https://www.rfc-editor.org/info/rfc4303`, `doi:10.17487/RFC4303`.

[86] Zeus Kerravala. Arista Launches Low Latency Switches Aimed at Financial Sector. TechnologyAdvice, June 2022 [Online]. URL: `https://www.eweek.com/cloud/arista-low-latency-switches/`.

[87] Michael Kerrisk. send(2) — Linux manual page, March 2021. URL: `https://man7.org/linux/man-pages/man2/send.2.html`.

[88] Changhoon Kim, Parag Bhide, Ed Doe, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. Inband Network Telemetry (INT), 2016. URL: `https://p4.org/assets/INT-current-spec.pdf`.

[89] Hyong-youb Kim and Scott Rixner. Connection handoff policies for TCP offload network interfaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, page 21, USA, 2006. USENIX Association.

[90] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association. URL: `https://www.usenix.org/conference/atc19/presentation/kogias-r2p2`.

[91] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for*

Computer Communication, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3387514.3406591`, `doi:10.1145/3387514.3406591`.

[92] Shiv Kumar, Pravein Govindan Kannan, Ran Ben Basat, Rachel Everman, Amedeo Sapio, Tom Barbette, and Joeri de Ruiter. Open Tofino, July 2021. URL: `https://github.com/barefootnetworks/Open-Tofino`.

[93] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3445814.3446696`.

[94] Jeongkeun Lee, Jeremias Blendin, Yanfang Le, Grzegorz Jereczek, Ashutosh Agrawal, and Rong Pan. Source Priority Flow Control (SPFC) towards Source Flow Control (SFC), November 2021. URL: `https://datatracker.ietf.org/meeting/112/materials/slides-112-iccrg-source-priority-flow-control-in-data-centers-00`.

[95] Samuel J. Leffler, Robert S. Fabry, and William N. Joy. A 4.2BSD Interprocess Communication Primer. Technical report, University of California at Berkeley, USA, 1983.

[96] Konstantin Lepikhov. Source Quench. Atlassian Corporation Pty Ltd., April 2018. URL: `https://wiki.geant.org/display/public/EK/Source+Quench`.

[97] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 90–103, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3341302.3342071`.

[98] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3132747.3132756`, `doi:10.1145/3132747.3132756`.

[99] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/2934872.2934897`, `doi:10.1145/2934872.2934897`.

[100] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming Network Stack for Middleboxes with Rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 551–570, Berkeley, CA, April 2021. USENIX Association. URL: `https://www.usenix.org/conference/nsdi21/presentation/li`.

[101] Yuliang Li. *Hardware-Software Codesign for High-Performance Cloud Networks*. PhD thesis, Harvard University, 2020. URL: `https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37368976`.

[102] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3341302.3342085`, `doi:10.1145/3341302.3342085`.

[103] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control, December 2021 [Online]. URL: `https://hpcc-group.github.io/results.html`.

[104] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259, Boston, MA, November 2020. USENIX Association. URL: `https://www.usenix.org/conference/osdi20/presentation/lin`.

[105] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, Erji Mao, Daniel Nelson, George Papen, Mukarram Tariq, and Amin Vahdat. Lightwave Fabrics: At-Scale Optical Circuit Switching for Datacenter and Machine Learning Systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 499–515, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3603269.3604836`.

[106] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 490–503, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3230543.3230582`.

[107] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3341302.3342079`, `doi:10.1145/3341302.3342079`.

[108] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 47–63, Berkeley, CA, USA, April 2021. USENIX Association. URL: `https://www.usenix.org/conference/nsdi21/presentation/liu`.

[109] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. URL: `https://books.google.com/books?id=5BwdBAAAQBAJ`.

[110] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3341301.3359657`.

[111] Mellanox. RoCE in the Data Center, October 2014. URL: `https://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf`.

[112] Mellanox. BlueField SmartNIC for Ethernet - High Performance Ethernet Network Adapter Cards, May 2021 [Online]. URL: `https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf`.

[113] Mellanox. Mellanox Innova-2 Flex Open Programmable SmartNIC, May 2021 [Online]. URL: `https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf`.

[114] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving Consistent Low Latency for Wireless Real-Time Communications with the Shortest Control Loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 193–206, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3544216.3544225`.

[115] Rui Miao, Li Bo, Hongqiang Harry Liu, and Ming Zhang. Buffer sizing with HPCC. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, pages 1–2, New York, NY, USA, 2019. Association for Computing Machinery. URL: `http://buffer-workshop.stanford.edu/papers/paper5.pdf`.

[116] Leonid Mirkin and Zalman J. Palmor. Control Issues in Systems with Loop Delays. In Dimitrios Hristu-Varsakelis and William S. Levine, editors, *Handbook of Networked and Embedded Control Systems*, pages 627–648. Birkhäuser Boston, Boston, MA, 2005. `doi:10.1007/0-8176-4404-0_27`.

[117] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/2785956.2787510`, `doi:10.1145/2785956.2787510`.

[118] Jeffrey C. Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, May 2003. USENIX Association. URL: `https://www.usenix.org/conference/hotos-ix/tcp-offload-dumb-idea-whose-time-has-come`.

[119] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3230543.3230564`, `doi:10.1145/3230543.3230564`.

[120] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, Santa Clara, CA, February 2020. USENIX Association. URL: `https://www.usenix.org/conference/nsdi20/presentation/moon`.

[121] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting TCP for Reconfigurable Datacenter Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, Santa Clara, CA, February 2020. USENIX Association. URL: `https://www.usenix.org/conference/nsdi20/presentation/mukerjee`.

[122] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan. Minimizing flow completion times in data centers. In *2013 Proceedings IEEE INFOCOM*, pages 2157–2165, April 2013. `doi:10.1109/INFCOM.2013.6567018`.

[123] Aisha Mushtaq, Radhika Mittal, James McCauley, Mohammad Alizadeh, Sylvia Ratnasamy, and Scott Shenker. Datacenter Congestion Control: Identifying What is Essential and Making It Practical. *SIGCOMM Comput. Commun. Rev.*, 49(3):32–38, November 2019. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3371927.3371932`, `doi:10.1145/3371927.3371932`.

[124] John Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984. URL: `https://www.rfc-editor.org/info/rfc896`, `doi:10.17487/RFC0896`.

[125] Netronome. About Agilio SmartNICs, May 2021 [Online]. URL: `https://www.netronome.com/products/smartnic/overview/`.

[126] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. Verification of P4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference*

on Emerging Networking EXperiments and Technologies, CoNEXT '18, page 73–85, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3281411.3281421`.

[127] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association. URL: `https://www.usenix.org/conference/nsdi19/presentation/ousterhout`.

[128] John Ousterhout. A Linux Kernel Implementation of the Homa Transport Protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 99–115, Boston, MA, July 2021. USENIX Association. URL: `https://www.usenix.org/conference/atc21/presentation/ousterhout`.

[129] John Ousterhout. It's Time to Replace TCP in the Datacenter, 2023. `arXiv:2210.00714`.

[130] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3), aug 2015. `doi:10.1145/2806887`.

[131] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993. `doi:10.1109/90.234856`.

[132] Pensando. DSC-100 Distributed Services Card, May 2021 [Online]. URL: `https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf`.

[133] Jon Postel. User Datagram Protocol. RFC 768, August 1980. URL: `https://www.ietf.org/rfc/rfc768.txt`, `doi:10.17487/RFC0768`.

[134] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3132747.3132780`.

[135] Pierre-Francois Quet, Sriram Chellappan, A. Durresi, M. Sridharan, Hitay Ozbay, and Raj Jain. Guidelines for optimizing Multi-Level ECN, using fluid flow based TCP model. In

Proceedings of ITCOMM 2002 Quality of Service over Next Generation Internet, Boston, MA, USA, August 2002.

[136] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, Santa Clara, CA, April 2024. USENIX Association. URL: `https://www.usenix.org/conference/nsdi24/presentation/rajasekaran`.

[137] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, February 2018. URL: `https://rfc-editor.org/rfc/rfc8312.txt`, `doi:10.17487/RFC8312`.

[138] George F. Riley and Thomas R. Henderson. The ns-3 Network Simulator. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. `doi:10.1007/978-3-642-12331-3_2`.

[139] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 123–137, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2785956.2787472`.

[140] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 735–749, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3387514.3405899`.

[141] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808, Boston, MA, April 2021. USENIX Association. URL: `https://www.usenix.org/conference/nsdi21/presentation/sapio`.

[142] Michael Schapira and Keith Winstein. Congestion-Control Throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 122–128, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3152434.3152446`, `doi:10.1145/3152434.3152446`.

[143] 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. `arXiv:https://doi.org/10.1287/opre.14.4.670`, `doi:10.1287/opre.14.4.670`.

[144] Henrik Schuh. *Accelerating Networked Systems With Programmable and Tightly-Coupled NICs*. PhD thesis, University of Washington, 2023. URL: `https://www.proquest.com/dissertations-theses/accelerating-networked-systems-with-programmable/docview/2915818227/se-2`.

[145] Amazon Web Services. Amazon EC2 F1 Instances. `https://aws.amazon.com/ec2/instance-types/f1/`, 2006. Accessed on 2020-08-10.

[146] Hemal Shah, Felix Marti, Wael Noureddine, Asgeir Eiriksson, and Robert Sharp. Remote Direct Memory Access (RDMA) Protocol Extensions. RFC 7306, June 2014. URL: `https://www.rfc-editor.org/info/rfc7306`, `doi:10.17487/RFC7306`.

[147] D. Shan and F. Ren. Improving ECN marking scheme with micro-burst traffic in data center networks. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017. `doi:10.1109/INFOCOM.2017.8057181`.

[148] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association. URL: `https://www.usenix.org/conference/nsdi22/presentation/shashidhara`.

[149] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data*

Communication, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2785956.2787508.

[150] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi-org.stanford.idm.oclc.org/10.1145/2934872.2934900, doi:10.1145/2934872.2934900.

[151] Bruce Spang, Serhat Arslan, and Nick McKeown. Updating the theory of buffer sizing. *Performance Evaluation*, 151:102232, 2021. doi:https://doi.org/10.1016/j.peva.2021.102232.

[152] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 518–532, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230543.3230548.

[153] Henning Stubbe. P4 compiler & interpreter: A survey. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, 7:47–52, May 2017. doi:10.2313/NET-2017-05-1_07.

[154] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The NeBuLa RPC-Optimized Architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 199–212, New York, NY, USA, 2020. IEEE Press. URL: https://doi-org.stanford.idm.oclc.org/10.1109/ISCA45697.2020.00027, doi:10.1109/ISCA45697.2020.00027.

[155] Synopsys. VCS: Industry's Highest Performance Simulation Solution, May 2021 [Online]. URL: https://www.synopsys.com/verification/simulation/vcs.html.

[156] Hitek Systems. 800G Ethernet FPGA IP Core Solution, Mar 2023 [Online]. URL: https://hiteksys.com/fpga-ip-cores/800g-ethernet-ip-core.

[157] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast Packet Processing with eBPF and

XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.*, 53(1), feb 2020. doi:10.1145/3371038.

[158] Han Wang, Andy Fingerhut, Santiago Bautista, Nate Foster, and Chris Dodd. Portable Switch Architecture, April 2021 [Online]. URL: https://p4lang.github.io/p4-spec/docs/PSA.pdf.

[159] Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. CocktailSGD: fine-tuning foundation models over 500mbps networks. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[160] Shie-Yuan Wang, Yo-Ru Chen, Hsien-Chueh Hsieh, Ruei-Syun Lai, and Yi-Bing Lin. A Flow Control Scheme Based on Per Hop and Per Flow in Commodity Switches for Lossless Networks. *IEEE Access*, 9:156013–156029, 2021. doi:10.1109/ACCESS.2021.3129595.

[161] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 255–274, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/wang-weitao.

[162] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 17–24, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3365609.3365855.

[163] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards Zero-Copy Serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 1–6, New York, NY, USA, 2021. Association for Computing Machinery. URL: https://doi-org.org/10.1145/3458336.3465283, doi:10.1145/3458336.3465283.

[164] Jackson Woodruff, Andrew W Moore, and Noa Zilberman. Measuring Burstiness in Data Center Applications. In *Proceedings of the 2019 Workshop on Buffer Sizing*, BS '19, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi-org.stanford.idm.oclc.org/10.1145/3375235.3375240, doi:10.1145/3375235.3375240.

[165] Zhong-zhen Wu and Han-chiang Chen. Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet. In *Proceedings of 15th International Conference on Computer Communications and Networks*, pages 245–250, 2006. `doi:10.1109/ICCCN.2006.286280`.

[166] Xilinx. Alveo Adaptable Accelerator Cards for Data Center Workloads, May 2021 [Online]. URL: `https://www.xilinx.com/products/boards-and-kits/alveo.html`.

[167] Xilinx. Virtex Ultrascale+ FPGA, May 2021 [Online]. URL: `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html`.

[168] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. FastLane: Making Short Flows Shorter with Agile Drop Notification. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 84–96, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2806777.2806852`.

[169] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, and Yibo Zhu. Combining ECN and RTT for Datacenter Transport. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet '17, page 36–42, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3106989.3107002`.

[170] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, page 78–85, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/3131365.3131375`, `doi:10.1145/3131365.3131375`.

[171] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is Network the Bottleneck of Distributed Training? In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI '20, page 8–13, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3405671.3405810`.

[172] Renjie Zhou, Dezun Dong, Shan Huang, and Yang Bai. FastTune: Timely and Precise Congestion Control in Data Center Network. In *2021 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages

238–245, New York City, NY, USA, 2021. IEEE. `doi:10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00043`.

[173] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, Boston, MA, April 2023. USENIX Association. URL: `https://www.usenix.org/conference/nsdi23/presentation/zhou`.

[174] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu. Deploying User-space TCP at Cloud Scale with LUNA. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 673–687, Boston, MA, July 2023. USENIX Association. URL: `https://www.usenix.org/conference/atc23/presentation/zhu-lingjun`.

[175] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery. URL: `https://doi-org.stanford.idm.oclc.org/10.1145/2785956.2787484`, `doi:10.1145/2785956.2787484`.

[176] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 313–327, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2999572.2999593`.