Hardware-Software Codesign for High-Performance Cloud Networks

A DISSERTATION PRESENTED
BY
YULIANG LI
TO
SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
AUGUST 2020

Dissertation advisor: Professor Minlan Yu Yuliang Li

Hardware-Software Codesign for High-Performance Cloud Networks

Abstract

The cloud is part of the daily life of billions of people and is carrying most of the computation happening on the planet. To deliver the hyper computing power, the network plays a core role in connecting hundreds of thousands of machines inside datacenters. However, as many new cloud applications (e.g., large-scale deep learning, high-performance computing) and new architectures (e.g, resource disaggregation, more heterogeneous hardware accelerators) are demanding ever-increasing high performance, the network starts to become the bottleneck, and it is very difficult to troubleshoot performance problems. This boils down to the insufficiency in the two essential networking tasks: control and telemetry.

- Congestion control and clock synchronization are the two control tasks critical for application performance. However, they have to sacrifice the normal-case performance for worse cases in production, because they are not robust to the high dynamics of traffic and failures.
- We also need precise and fine-grained telemetry for performance troubleshooting. However, we often miss important information and cannot pinpoint the exact culprits, because existing telemetry systems supported by switches and hosts are either imprecise or coarse-grained.

To tackle the challenges, we set qualitatively better objectives than existing approaches: introducing robustness to the performance-critical control tasks, and design telemetry systems that are both precise and fine-grained. Achieving the new objectives is challenging due to the resource and observation limitations of the network devices. Fortunately, new programmable switches and NICs make it possible to codesign different devices, leveraging the advantage of different devices to collaboratively achieve breakthroughs and realize the new objectives.

Based on the new opportunities for codesign, we have three key design principles: (1) closing the gap between observation and control to make control precise and timely, (2) designing new algorithms and data structures to make effective use of different devices' capabilities, and (3) rethinking the division of labor among switches, hosts, and the controller with a paradigm shift away from the self-contained design model. Guided by the principles, we design novel network control and telemetry schemes that achieve the new objectives.

To robustly provide high performance under dynamics, we design novel control schemes that close the gap between observation and control. We design HPCC, a congestion control scheme, which uses a novel metric for both observation and control, and use new programmability to deliver switch states to hosts to help calculate the new observation metric. We also design Sundial, a clock synchronization scheme, which uses a backup plan precomputed by the centralized controller to enable fast failure recovery based on device-local observation.

We also design precise and fine-grained telemetry systems. For switches, we design FlowRadar and LossRadar to expose precise flow and loss information, by dividing the maintenance of hash tables into simple per-packet updates in switches and small amounts of complex computation in the controller. For the host TCP stack, we design DETER, in which hosts only record 0.03% of traffic and the controller can replay per-packet, per-line-of-code information.

Our systems have very wide impacts. Since we designed HPCC in 2019, it is not only deployed in Alibaba cloud, but also supported by many switch and NIC vendors (Intel, Mellanox, Broadcom, Cisco, Innovium, Marvell, etc.); in addition, Alibaba, Intel, and Mellanox are actively writing an IETF draft of HPCC, with the latest version in September 2020. We have built a prototype of Sundial at Google and deployed it in a test cluster with >500 servers in mid 2020, and we show performance improvements in Spanner and in Swift brought by Sundial. FlowRadar and LossRadar also result in a joint patent with Barefoot/Intel, and Alibaba is very interested in using the technique for loss detection. Finally, DETER's Linux kernel-based implementation is open-sourced, and we reveal several TCP problems when running large Spark and RPC systems in the dissertation.

Contents

List of figures

List of Tables

To my parents, who persistently pray for my pursuit of truth, and my grandparents, who cultivated my curiosity.

Acknowledgments

It has been my privilege to work with my PhD advisor, Minlan Yu. She patiently guided me through all the obstacles during my research, taught me how to define research problems, how to conduct solid research, and how to effectively communicate the research findings with various audiences. I also learned a lot from her how to appreciate different people's thoughts. Beyond that, she cares deeply about my career—she tried everything to help me when I faced challenges with my internship, introduced me to various job opportunities, and spent lots of effort helping me practice my interviews. She also connected me with plenty of opportunities for presentations and collaborations. I am very grateful to her.

I would like to thank my dissertation committee members. I learned a lot about TCP and congestion control from Mohammad Alizadeh, who provided me lots of insightful guidance. I want to thank Nandita Dukkipati for her strong support in the research of clock synchronization, and James Mickens for the exciting discussion about advancing replay. Thank you all for the invaluable comments for this dissertation.

I was lucky to collaborate with many great researchers and engineers. For Chapter 2, I thank Rui Miao and Harry Liu for the discussion that finally shaped the design, Frank Kelly for the contribution on the theory, Yan Zhuang for the FPGA implementation, and Fei Feng for setting up the testbed. For Chapter 3, I thank Gautam Kumar, Hassan Wassel, and Peter Hochschild for the help on motivating the design, Peter Cuy for the guidance of the firmware implementation, and Hema Hariharan for the help on hardware. For Chapters 4 and 5, I thank Changhoon Kim for the help on motivating the design and sharing the information of switch architecture. For Chapter 6, I thank Wei Bai for sharing his experience in TCP.

I also thank Ming Zhang, Lingbo Tang, Zheng Cao, Lorenzo Alvisi, Amin Vahdat, Prashant Chandra, Dave Platt, Arjun Singh, Jakov Seizovic, David Wetherall, David Dillow, Joe Zbiciak, Sujata Banerjee, Marco Canini, Sylvia Ratnasamy, George Varghese, Ori Rottenstreich, Alex C. Snoeren, Danyang Zhuo, Yurong Jiang, Sivaramakrishnan Ramanathan, and Bradley McDanel for the constructive feedback; Shin Mao, Nanfang Li, and Ioannis Giannakopoulos for the help on experimentation; Alex Iriza, Bryant Chang, Antonin Bas for the implementation discussion.

I was lucky to meet the research group members: Masoud Moshref, Rui Miao, Omid Alipourfard, Jiaqi Gao, Sivaram Ramanathan, Ran Ben Bassat, Xuemei Liu, Mingyang Zhang, Junzhi Gong, Yang Zhou, Muhammad Tirmazi, and Zhiying Xu. The lively discussions at group meetings and the sharing of the fun of life makes research so enjoyable.

Thanks to my friends in LA. Special thanks to my Uncle TC and Auntie Anna, for helping me settle down and get the driver's license when I first arrived in the US, and the cares about my life throughout the years. I am so blessed to have Yu Feng, Yue Tao, Weili Wang, and Ying Xu as my housemates in LA, without whom I could not have the great meals, fun time, and tidy home everyday. Thank Robbie for spending many days taking me around to select my lovely car.

Thanks to my friends in Boston. I am so blessed to have my spiritual family here praying for each other everyday. Thank Sherry, Xuezhi, Xinyi, Alex, Zheng, and Hao for the help in my daily life. Thank Sai Qian Zhang for being a housemate that can discuss research. Thank Mingzhu Zeng for the accommodation, Junshang Duan for the often encouraging and inspiring discussion, and Jie Mao for the help on moving.

Thanks to my Uncle Tianzhong and Auntie Lien, for setting up an enjoyable environment so I can focus on the writing of this dissertation.

Thank Jesus for igniting my desire for truth, and the amazing plan for my career, and life.

1 Introduction

The cloud is part of the daily life of billions of users and is carrying most of the computation happening on the planet: web search, social media, video services, online shopping, smart home, IoT, scientific computation, etc. For example, Google already serves 3.5 billion search queries per day; yet, not only the number of search queries is continuously growing, the type of search is also diversifying (e.g., from texts to audio, image, video). In addition, these user-facing services drive even larger and ever-growing volumes of backend computations, such as big-data analytics, video analytics, machine learning, distributed database, and in-memory cache.

The huge amount of cloud computing takes place in large-scale datacenters that each consists of 100,000s of hosts inter-connected by 10,000s of switches. The network plays a core role in making all resources available to applications with high performance.

However, the network is becoming the performance bottleneck under the ever-increasing requirements for high performance. Many applications faced by billions of users, such as translation, recommendation, search, and virtual assistant, are driven by continuous, large-scale machine learning training running on high-speed devices such as TPU or GPU, which periodically transfers large volumes of data, and their performance bottleneck is usually in the network. In addition, many HPC applications with profound impact, such as medical research, genomics, weather forecasting, and seismic analysis, are moving into the cloud[44], bringing their low latency and high throughput requirements[170].

Besides the applications, new architecture trends are raising the bar of good performance even further. Resource disaggregation is one trend: resources like disks (HDD, SSD, NVMe) are not limited to their hosting server, but shared across the entire datacenter; even memory disaggregation is on the way. So the previous in-server disk or memory accesses are or will go through the network, demanding ultra-low latency and adding tremendous load to the network. Moreover, cloud providers are developing more and more heterogeneous, modularized hardware accelerators, and rely on the network to compose them to support various applications[51].

1.1 Challenges and Objectives of Network Control and Telemetry

While the performance requirements have grown dramatically, the networking techniques used today have not kept up. Consequently, the network starts to become the bottleneck in next-generation cloud computing, and it is very difficult to troubleshoot performance problems.

This boils down to the insufficiency in the two essential networking tasks: control and telemetry. We present the challenges and find that the design objectives of existing solutions are unsuitable under the rapidly-increasing performance requirements. Therefore, we propose qualitatively better objectives to tackle the problems.

1.1.1 Control is not Robust to Dynamics

Applications need high-speed data transfer to freely share data and scale out, and need precisely synchronized clocks for fine-grained coordination across servers. Thus to achieve high performance, we need careful control of the traffic rate to fully utilize the bandwidth while avoiding congestion, and synchronization of clocks over the network. However, these performance-critical control tasks have to sacrifice the normal-case performance for worse cases in production, because they are not robust to the high dynamics in production, where new traffic starts and finishes frequently at microsecond scale, and failures are very common and exhibit high diversity (connectivity, cooling, power, etc.).

Congestion control (CC). CC is critical to providing high-throughput and low-latency data transfer to applications. However, existing CC schemes, although may perform well in lab settings, have never achieved the raw potential allowed by the hardware. For example, my study of Alibaba's RDMA network* shows that under dynamic traffic (which exhibits long-tail flow size distribution

*A high-speed network accelerated by hardware-offloaded networking stack.

and includes incast), even only utilizing 30% of network bandwidth on average, the 95-th percentile latency can go above 1ms[131] (around 200x the base RTT)!

The reason is that the CC schemes in use today are not robust to the traffic dynamics in production, where flows start and finish frequently. When multiple flows start (e.g., incast), CC cannot robustly slow down to the right level, causing buffer overflow, so it's hard to run the network at high utilization. When some flows finish and release bandwidth, CC is also not robust enough to ramp up other flows immediately, so to avoid wasting bandwidth, CC keeps standing queues all the time, which inflates normal-case latency.

The severity is increasing. As the bandwidth continues growing exponentially, rate mismatch fills up the buffer much faster. As the bare-metal latency continues decreasing, queuing delay will be more dominant.

Clock synchronization. Synchronized clocks are critical for many distributed transactional systems and databases to achieve low latency and high throughput[77,159,18]. Several recent solutions target submicrosecond-level precision, but they can only achieve it in lab settings where several important types of failures are ignored. In real production, only millisecond-level precision is available at the datacenter scale.

The reason is that these schemes are not robust to failures, which is highly dynamic in datacenters. For example, they fail to notice large clock drifts during cooling failures, causing the risk of application errors (e.g., inconsistent transactions)! Moreover, they take a long time to recover from synchronization disruptions caused by switch or link failures, leading to even higher risks. Such risks are unacceptable for many applications that serve millions or billions of people, so they have to still use millisecond-level precision bound today.

This problem is more urgent today. With recent software and hardware improvements such as RDMA, NVMe, and in-memory storage, network and storage latencies are going towards the microsecond level, so the coordination time is becoming the bottleneck, which is on the order of the precision bound of clocks.

New objective: adding robustness to these performance-critical control tasks, so they can quickly adapt to the dynamics with high performance. Although robustness is neglected by existing schemes, it is actually important for high performance in highly dynamic production environments.

1.1.2 Telemetry is either Imprecise or Coarse-grained

The performance can be affected by many network factors, such as congestion, packet losses, failures, software bugs, misconfigurations, network updates, etc. Network telemetry is necessary for troubleshooting performance problems. However, we often miss important information and cannot pinpoint the exact culprits, because existing telemetry systems are either imprecise (sampled or approximated) or coarse-grained (e.g., device or port-level counters). Specifically, as the bandwidth continues growing exponentially, problems (e.g., buffer overflow) can be caused by just a small percentage (e.g., <10%) overload, which necessitates precise telemetry. Moreover, we should know not only the existence of problems, but also the culprits, victims, and locations, which needs fine-grained telemetry.

We take an example to illustrate the importance of precise and fine-grained telemetry from switches. When a TCP connection reports a packet loss, we first need to locate the loss. However, existing switches can only report total loss counters, which is too coarse-grained; we need header information to locate the loss. After knowing the location, we then want to know what other flows cause the congestion, and by how much. Unfortunately, NetFlow is based on sampling, and sketches are approximated. While they can detect heavy hitters (large flows), the volumes are imprecise; moreover, congestion can happen without heavy hitters (e.g., during incast), in which case they miss most culprits. We need precise, flow-level information to answer the question.

On the host, the networking stack is very complex (e.g., Linux 4.4 TCP has 63 parameters and

>10K lines) and under continuous error-prone development (16 bugs identified in Linux TCP in two months [128]), so every line in the code can be the source of bug. Moreover, subtle differences in parameters and features may lead to completely different performances. These problems are extremely difficult to diagnose at large scales, but no logging tools can provide all information at large scales.

New objective: design telemetry systems that are both precise and fine-grained. This is in contrast to existing works that focus on precision or granularity independently, but not both.

1.2 Limitations of Individual Devices and Opportunities for Codesign

Achieving these new objectives—adding robustness to performance-critical control tasks and designing precise and fine-grained telemetry—are not easy, because there are fundamental limitations in the network devices that carry out these control and telemetry tasks. Fortunately, there are new opportunities in these devices. Although these new opportunities cannot directly break the fundamental limitations, they open doors for codesign that leverages the advantages of different devices to collaboratively achieve breakthroughs and realize the new objectives (e.g., coordinate hardware's high-speed processing and software's ability of complex computation to achieve tasks that are impossible in either hardware or software separately.).

1.2.1 Resource and Observation Limitations of Individual Devices

Limited observation of other devices. Each device has very limited observation of other devices, so it is difficult to make robust decisions. For example, hosts only observe drops, end-to-end delay, or ECN marks of packets; such vague observation is hard for CC to make agile control decision robustly. Also devices only observe local failures, so they cannot take robust reactions to failures that need global information, such as failures that disrupt clock synchronization.

Limited computing capability and memory. To achieve a high speed, the per-packet processing time is very limited. For example, at a 100Gbps link, a 64B packet only have 6ns. Consequently, hosts can perform a very limited number of operations per-packet; switches have more strict constraints because of its higher throughput—besides the limited operations, they also cannot support complex operations that take too many cycles, such as multiplication and division[160].

Limited per-packet time also means the usable memory size is limited. Switch ASIC only uses SRAM and TCAM, which are only 10s of MB in size. Recent studies[102,133,169] also show that, despite the large DRAM, hosts should fit the memory usage in the CPU cache to sustain line-rate packet processing, which is also only 10s of MB in size.

Therefore, these devices support very limited functionalities. For example, switches cannot even maintain a hash table that is important for keeping precise and fine-grained information, because resolving hash collisions takes multiple, non-constant operations, which is beyond the switch capability. Hosts also do not have the cycle and memory budget for copying and storing fine-grained runtime states.

1.2.2 New Opportunities for Codesign

Within the fundamental limitations, however, the previously fixed-function hardware is becoming flexible and programmable (i.e., we can redefine the usage of the limited memory and reprogram the functions within the limitation of number and type of operations). It provides new opportunities for codesign: we can customize the function of each device to leverage its own strength and to take over the job that faces constraints on other devices, and we can customize the interactions among devices.

Taking over simple per-packet operations from other devices, and maintaining states for complex processing by other devices. Compared to traditional switches, new programmable switch ASIC's match-action tables are programmable—they can be configured to match any bits in the packet or metadata instead of pre-defined fields, and perform more flexible actions instead of fixed ones. Therefore, we can do more flexible operations at line rate, such as arithmetic operations (+ and -), comparison (>, >=, <, <=, ==, and !=), bit operations (AND, OR, XOR, NOT, and bit shift), and modifying packet with any values. Branching is also achievable by using the match to check bits in the packet or metadata. Moreover, programmable switch ASICs also support more diverse states, such as queue length, timestamp, etc. We can also customize states such as counters (e.g., packet counters, byte counters), moving average, and state machines, as long as the state update can be done in one cycle.

On the host, NICs are also becoming more programmable. With diverse types of SmartNICs (e.g., multicore SoC-based, FPAG-based) quickly taking up the market, we can have customized logic running in the NIC. SmartNICs have fewer constraints in flexibility than switch ASICs. For example, some complex operations can be done in the SmartNIC, such as multiplications and divisions. The state updates also don't have the one-cycle constraint.

Although the fundamental limitation remains—the number and/or type of per-packet operations and memory size are still limited—the programmable and stateful processing at line rate lays the foundation for codesigning switches or NICs with other devices. We can leverage switch ASIC or NIC's high speed to take over simple but per-packet operations from software, and use them to maintain states that can be used by other devices for complex processing.

Flexible and fine-grained interaction with other devices. Switch ASICs and NICs can interact with other switches and hosts by writing states to the packet header. A feature based on this, called In-band Network Telemetry (INT)[33], is widely supported recently[131,65], which commonly writes metadata such as queue length and timestamp to the packet header but can also write other customized states. Other switches and hosts get the information as the packet passes through them, and possibly trigger processing there. This enables very fine-grained cross-device collaborations.

Switch ASICs and NICs can also interact with the local CPU and the network controller. The switch ASIC can trigger local CPU processing by interrupting the CPU (e.g., by sending a packet to CPU), and the CPU may further invoke the network controller processing. The trigger can be customized, based on periodic timers, timeout[†], or certain packets. The NIC can interact with the CPU or the controller more flexibly.

1.3 Key Design Principles

Based on the new opportunities for codesign, we have three key design principles.

Closing the gap between observation and control to make the control precise and timely. Control schemes are normally in an observation-control loop (e.g., take reactions after observing congestion or failures). The problem is, however, that in today's performance-critical control schemes, what devices observe is far from what is needed for making the right control decisions. So the control schemes either take multiple rounds of trial and error (e.g., TCP halves the window each round, and based on the next-round observation decides whether to continue slowing down or not), or reach out to remote devices which takes a long time (e.g., the device that observes a failure has to notify the controller to recover the synchronization of clocks), thus lacking robustness. We close the gap between observation and control, by redesigning the control loops, both observation and control, so that they can make robust control decision immediately based on the observation.

Designing new algorithms and data structures to make effective use of different devices' capabilities. Because of the limited computing capability and memory, we often have to sacrifice func-

[†]Switch ASIC can generate packets with configured timer interval. We can use this feature to create customized periodic routines, by programming match-action tables to match these timer-generated packets; we can also setup customized timeout, by using the periodic routines to check and maintain states.

tionality, precision, or granularity. So we design customized algorithms by extending techniques from graph theory, data structure, and distributed systems to best leverage the device capability. For example, based on graph theory, we extract global topology features, and encode them into a small piece of data for each device to make decisions locally. We design special data structures to decompose complex operations into simple ones that are supported by switches. We also extend the replay technique to networking, making extremely efficient use of the packet processing resource.

Rethinking the division of labor among switches, hosts, and the controller with a paradigm shift away from the self-contained design model. To unleash the true power of codesign, we need a paradigm shift: a single device does not necessarily need to provide self-contained functions. Traditionally, in the Internet, various devices (NICs and switches of different vendors and versions controlled by different entities) coexist in the network, so they have to be self-contained to function well regardless of other devices. For example, traditionally TCP works without assumptions on the switch, so it has to use non-switch-specific data (packet drops or delay[‡]); switches must provide readily readable or usable monitoring data. Such a self-contained design model significantly constrains the achievable functionality. Fortunately, the cloud can program all devices, so individual devices no longer need to provide self-contained functions. Instead, we can rethink the division of labor among them to design much more balanced systems across the entire cloud ecosystem.

1.4 Control and Telemetry for High-performance Cloud Networks

Based on the design principles, we redesign the network control and telemetry schemes that achieve the new objectives.

[‡]ECN is rarely turned on in the Internet.

1.4.1 Control Schemes that Robustly Provide High Performance

To robustly provide high performance under dynamics, we design novel schemes that close the gap between observation and control. The new schemes are made possible by dividing the labor of observation between hosts and switches in congestion control, and dividing the labor of control between devices and the controller in clock synchronization.

HPCC[131]: a novel congestion control (CC) scheme that robustly provides high performance under dynamic traffic. CC schemes used today converge slowly compared to the highly dynamic DC traffic, so they have to sacrifice normal-case performance to prepare for cases that they cannot handle timely. The slow convergence is because of the gap between the observed states (e.g., drop, ECN, delay) and the control (rate adjustment): CC doesn't know the best control after observing states, and how much the states will change after control. To close the gap, we propose to observe and to control both based on *inflight bytes*, so the observed state is unambiguously mapped to the right control decision and vice versa. The number of inflight bytes going through a bottleneck is not directly observable, so we identify the right set of switch states for hosts to calculate the number of inflight bytes, and uses INT to convey information from switches to hosts. HPCC is proved to converge to high throughput and 0 queueing delay in just 1 RTT in most cases. With the quick convergence, HPCC robustly delivers high performance even under highly dynamic traffic.

Sundial[127]: a fault-tolerant clock synchronization scheme that robustly bounds the precision at submicrosecond-level under diverse failures. Several schemes work in the sub-μs region, but they don't benefit many applications because many applications need *bounded* precision. We are the first to introduce the *precision bound* to sub-μs clock synchronization. We find in existing schemes, we have to sacrifice the bounds by 2-3 orders of magnitudes to prepare for failures, which is slow to handle (during which clocks drift away). The key challenge is the gap between observation and control: devices can only observe local failures, but handling failure needs the global information in the centralized controller. We close the gap with a novel backup-plan-based control: the controller uses its global information to precompute and assign a backup plan to each device, based on which the devices can take local actions with its local observation. To make effective use of the limited device resource, we use graph theory to encode the global information into an extremely simple backup plan that is generic to different types of failures (link, switch, domain). By properly dividing the labor between devices and the controller, Sundial enables fast failure handling, thus robustly bounding the precision at sub-μs level.

1.4.2 Telemetry Systems that are Both Precise and Fine-grained

We design precise and fine-grained telemetry systems for both switches and hosts. By carefully rethinking the division of labor, we design new algorithms that run per-packet simple operations customized to the capabilities of switches or host packet processing, and run small amounts of complex computation on the controller.

FlowRadar[129] and LossRadar[130]: precise flow/loss telemetry from switches. Existing switch telemetry systems are either imprecise (based on sampling or approximation) or coarse-grained (e.g., device or port-level packet counters), but effective diagnosis needs precise flow/loss information. The key challenge is that handling hash collisions takes multiple, non-constant memory accesses, which is impractical due to the limited capability of switch ASIC. Instead of handling collisions, we design data structures that embrace collisions by XORing colliding items in the same entry, which is readily supported by switches. To extract items, the controller collects the data structures from multiple switches, and leverages the redundancy (e.g., flows traverse multiple switches) to reduce the number of items mixed in each entry. With this technique, we design FlowRadar which provides precise per-flow counters, and LossRadar which provides precise per-loss information. In this way, we rethink the previously self-contained functions (e.g., NetFlow) in switches, and divide the labor among switches and the controller.

DETER[128]: precise per-packet, per-line-of-code information from host TCP stack. TCP performance problems are notoriously difficult to diagnose. Lightweight TCP counters are too coarse-grained, while the gold standard for diagnosis—tracing TCP executions on every packet—is unscalable in datacenters. To scalably provide the precise and fine-grained details, we extend the replay technique in distributed systems to the network. In the new context, the unique challenge of replay is a butterfly effect—a small timing variation causes a chain reaction between TCP and the network that drives the system to a completely different state in the replay. We find the root cause to be the closed-loop nature of TCP, and identifies the minimum runtime information to record that breaks the closed loop into open loops. With the proper division of labor, hosts only record runtime data that is 0.03% in size of the traffic, while the controller can replay precise, per-packet, per-line-of-code information.

1.5 Impact

As the performance requirement continues to grow in the post-Moore era, developing more hardware is inevitable. However, hardware is much slower to develop and debug than software, and takes several years to finally produce the chips. So it is extremely hard to directly turn existing software into hardware. Thus, hardware-software codesign is inevitable. However, unlike software engineering which we have more than 50 years of experience[1], hardware-software codesign is just beginning.

In this dissertation, we design and build several concrete systems, in the context of networking, as an initial exploration of hardware-software codesign. We show the feasibility of using just slightly improved hardware (flexibility) to robustly achieve hardware-level performance, precision, and granularity.

The success of our designs also changes the traditional mindset of designing systems. The cloud provider is usually viewed as a single entity, but different components are still managed by different teams (e.g., server team, switch team, controller team), which usually still carry on the mindset of designing self-contained systems within the teams. The success of our designs shows that a paradigm shift is extremely beneficial. Our designs have influenced many companies. HPCC is not only deployed in Alibaba cloud[6], but also supported by many switch and NIC vendors (Intel, Mellanox, Broadcom, Cisco, Innovium, Marvell, etc.); in addition, Alibaba, Intel, and Mellanox are actively writing an IETF draft of HPCC, with the latest version in September 2020[5]. We have built a prototype of Sundial at Google and deployed it in a test cluster with >500 servers[127] in mid 2020, and we show performance improvements in Spanner[77] and in Swift[118] brought by Sundial. FlowRadar and LossRadar also result in a joint patent with Barefoot/Intel, and Alibaba is very interested in using the technique for loss detection. Finally, DETER's Linux kernel-based implementation is open-sourced, and we reveal several TCP problems when running large Spark and RPC systems in the dissertation.

2 HPCC: Robustly Achieving High Performance in Congestion Control

2.1 Introduction

The link speed in datacenter networks has grown from 1Gbps to 100Gbps in the past decade, and this growth is continuing. Ultra-low latency and high bandwidth, which are demanded by more and more applications, are two critical requirements in today's and future high-speed networks.

Specifically, in major cloud providers in the world, there are two critical trends in datacenters that drive the demand on high-speed networks. The first trend is new datacenter architectures like resource disaggregation and heterogeneous computing. In resource disaggregation, CPUs need high-speed networking with remote resources like GPU, memory, and disk. According to a recent study[87], resource disaggregation requires 3-5μs network latency and 40-100Gbps network bandwidth to maintain good application-level performance. In heterogeneous computing environments, different computing chips, e.g., CPU, FPGA, and GPU, also need high-speed interconnections, and the lower the latency, the better. The second trend is new applications like storage on high I/O speed media, e.g., NVMe (non-volatile memory express) and large-scale machine learning training on high computation speed devices, e.g., GPU and ASIC. These applications periodically transfer large volume of data, and their performance bottleneck is usually in the network since their storage and computation speeds are very fast.

Given that traditional software-based network stacks in hosts can no longer sustain the critical latency and bandwidth requirements[188], offloading network stacks into hardware is an inevitable direction in high-speed networks. In recent years, major cloud provider such as Alibaba and Microsoft deployed large-scale networks with RDMA (remote direct memory access) over Converged Ethernet Version 2 (RoCEv2) in datacenters as the hardware-offloading solution.

Unfortunately, according to our study at Alibaba's large-scale RoCEv2 networks, we find that RDMA networks face fundamental challenges to reconcile low latency, high bandwidth utilization, and high stability. This is because high speed implies that flows start at line rate and aggressively grab available network capacity, which can easily cause severe congestion in large-scale networks. In addition, high throughput usually results in deep packet queueing, which undermines the performance of latency-sensitive flows and the ability of the network to handle unexpected congestion. We highlight two representative cases among the many encountered in practice to demonstrate the difficulty:

Case-1: PFC (priority flow control) storms. A cloud storage (test) cluster with RDMA once encountered a network-wide, large-amplitude traffic drop due to a long-lasting PFC storm. This was triggered by a large incast event together with a vendor bug which caused the switch to keep sending PFC pause frames indefinitely. Because incast events and congestion are the norms in this type of cluster, and the operators are not sure whether there will be other vendor bugs that create PFC storms, the operators decided to try our best to prevent any PFC pauses. Therefore, they tuned the CC algorithm to reduce rates quickly and increase rates conservatively to avoid triggering PFC pauses. They did get fewer PFC pauses (lower risk), but the average link utilization in the network was very low (higher cost).

Case-2: Surprisingly long latency. A machine learning (ML) application complained about $>100\mu s$ average latency for short messages; its expectation was a tail latency of $<50\mu s$ with RDMA. The reason for the long latency, which finally dug out, was the in-network queues occupied majorly by a cloud storage system that is bandwidth-intensive in the same cluster. As a result, the operators have to separate the two applications by deploying the ML application to a new cluster. The new cluster had low utilization (higher cost) given that the ML application is not very bandwidth hungry.

To address the difficulty to reconcile latency, bandwidth/utilization, and stability, we believe a good design of CC is the key. This is because CC is the primary mechanism to avoid packet buffering or loss under high traffic loads. If CC fails frequently, backup methods like PFC or packet retransmissions can either introduce stability concerns or suffer a large performance penalty. Unfortunately, we found state-of-art CC mechanisms in RDMA networks, such as DCQCN[188] and TIMELY[142], have some essential limitations:

Slow convergence. CC continuously observes network states (based on feedback signals such as ECN or RTT) and make rate adjustments accordingly. With existing coarse-grained signals, current

CC schemes do not know exactly how much to increase or decrease sending rates to match the bottleneck capacity, and do not know what the network states will be after the rate adjustment. Therefore, they use heuristics to guess the rate adjustment and try to iteratively converge to a stable rate distribution, in a trial-and-error manner. Such iterative methods are slow for handling large-scale congestion events[106], as we can see in Case-1.

Persistent packet queueing. The slow convergence forces CC to keep persistent packet queueing to avoid bandwidth waste. If the queue is zero or small, when some bandwidth is freed (e.g., when some flows finish), CC takes a long time to ramp up the running flows to fully utilize the bandwidth. Such bandwidth waste is especially severe in production where traffic is highly dynamic—flows start and finish frequently. To avoid wasting bandwidth, CC has to persistently maintain queues, which occupy the bandwidth released by finished flows before the remaining flows ramp up. The higher the queues are, the higher bandwidth can be utilized, but with the consequence of higher latency. This is exactly the issue met by the ML application at the beginning in Case-2.

Complicated parameter tuning. The above problems entail that CC cannot robustly deliver high throughput and low latency under various network environments, especially under dynamic traffic patterns. So CC algorithms employ many heuristics for handling different environments and traffic patterns, which have many parameters. For instance, DCQCN has 15 parameters to set up. As a result, operators usually face a complex and time-consuming parameter tuning stage in daily RDMA network operations, which significantly increases the risk of incorrect settings that cause instability or poor performance.

The fundamental cause of the preceding three limitations is that *CC cannot robustly ramp up or ramp down rates to match the capacity immediately*. This is unavoidable because of the gap between the network states CC observes (ECN or RTT) and the control knob of CC (sending rate): there is no way to tell the exact rate adjustment given the observed states, and no way to predict exactly how states will change after the adjustment. However, this situation has recently changed. With

In-network telemetry (INT) features that have become available in new switching ASICs[46,48,47], obtaining fine-grained network states and using it to improve CC has become possible in production networks.

In this paper, we propose a new CC scheme, HPCC (High Precision Congestion Control), for large-scale, high-speed networks. The key idea of HPCC is to close the gap between observation and control by using the number of inflight bytes for both of them, so HPCC can converge to high bandwidth utilization with zero queueing in one RTT in most cases. To realize the design, we leverage the switches to provide fine-grained information via INT, and compute the number of inflight bytes on the hosts. This addresses the three limitations in current CC schemes. First, HPCC senders can quickly ramp up or ramp down flow rates to match the link capacity, in just one RTT under most circumstances. Second, HPCC can maintain near-zero queues—it does not need persistent queues to avoid bandwidth waste, because it can grab the free bandwidth in just one RTT (a few microseconds). Finally, since the sending rates can quickly adapt to the traffic dynamics, HPCC requires merely 3 independent parameters that are used to tune fairness and efficiency.

During the design of HPCC, we face two main challenges. First, the feedback can be delayed by link congestion, which can defer the flow rate reduction for resolving the congestion. This leads to our decision of controlling inflight bytes, as opposed to controlling the rate in DCQCN and TIMELY, because it prevents senders from sending extra traffic even if the feedback gets delayed. To observe the number of inflight bytes going through the bottleneck, which cannot be directly reported by switches, we let hosts calculate it based on INT information. Second, despite that INT information is in all the ACK packets, there can be destructive overreactions if a sender blindly reacts to all the information for fast reaction (§2.3.2). Our CC algorithm selectively uses INT information by combining per-ACK and per-RTT reactions, achieving fast reaction without overreaction.

HPCC meets our goals of achieving ultra-low latency, high bandwidth, and high stability simultaneously in large-scale high-speed networks. In addition, HPCC also has the following essential properties for being practical: ***(i)** Deployment ready*: It merely requires standard INT features (with a trivial and optional extension for efficiency) in switches and is friendly to implementation in NIC hardware. ***(ii)** Fairness*: It separates efficiency and fairness control. It uses multiplicative increase and decrease to converge quickly to the proper rate on each link, ensuring efficiency and stability, while it uses additive increase to move towards fairness for long flows.

HPCC's stability and fairness are guaranteed in theory (§2.4). We implement HPCC on commodity NIC with FPGA and commodity switching ASIC with P4 programmability. With testbed experiments and large-scale simulations, we show that compared with DCQCN, TIMELY, and other alternatives, HPCC reacts faster to available bandwidth and congestion and maintains close-to-zero queues. In our 32-server testbed, even under 50% traffic load, HPCC keeps the queue size zero at the median and 22.9KB (only 7.3μs queueing delay) at the 99-th percentile , which results in a 95% reduction in the 99-th percentile latency compared to DCQCN without sacrificing throughput. In our 320-server simulation, even under incast events where PFC storms happen frequently with DCQCN and TIMELY, PFC pauses are not triggered with HPCC.

Note that despite HPCC having been designed from the experiences with RDMA networks, we believe its insights and designs are also suitable for other high-speed networking solutions in general.

2.2 Experience and Motivation

In this section, we present our study at Alibaba's production datacenter networks that demonstrate the difficulty to operate large-scale, high-speed RDMA networks due to current CC schemes' limitations. We also propose some key directions and requirements for the next-generation CC of high-speed networks.

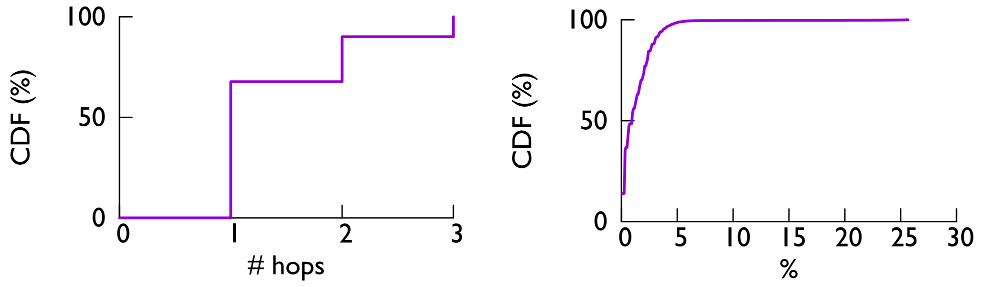
2.2.1 Large RDMA Deployments in Alibaba

Alibaba adopts RDMA in datacenters for ultra-low latency and large bandwidth demanded by multiple critical applications, such as distributed storage, database, and deep learning training frameworks.

The datacenter network is a Clos topology with three layers—ToR, Agg, and Core switches. A PoD (point-of-delivery), which consists of tens of ToR switches that are interconnected by a number of Agg switches, is a basic deployment unit. Different PoDs are interconnected by Core switches. Each server has two uplinks connected with two ToR switches for high availability of servers, as required by the customers. In the current RDMA deployment, each PoD is an independent RDMA domain, which means that only servers within the same PoD can communicate with RDMA.

Alibaba uses the latest production-ready version of RoCEv2: DCQCN is used as the congestion control (CC) solution which is integrated into hardware by RDMA NIC vendors. PFC[22] is enabled in NICs and switches for lossless network requirements. The strategy to recover from packet loss is "go-back-N", which means a NACK will be sent from receiver to sender if the former finds a lost packet, and the sender will resend all packets starting from the lost packet.

There have been tens of thousands of servers supporting RDMA, carrying Alibaba's databases, cloud storage, data analysis systems, HPC, and machine learning applications in production. Applications have reported impressive improvements by adopting RDMA. For instance, distributed machine learning training has been accelerated by 100+ times compared with the TCP/IP version, and the I/O speed of SSD-based cloud storage has been boosted by about 50 times compared to the TCP/IP version. These improvements majorly stem from the hardware offloading characteristic of RDMA.



(a) Propagation depth

(b) Suppressed bandwidth

Figure 2.1: The impacts of PFC pauses in production.

2.2.2 Goals for RDMA Networks

Besides ultra-low latency and high bandwidth, network stability and operational complexity are also critical in RDMA networks, because RDMA networks face more risks and tighter performance requirements than TCP/IP networks.

First of all, RDMA hosts are aggressive for resources. They start sending at line rate, which makes common problems like incast much more severe than TCP/IP. The high risk of congestion also means a high risk to trigger PFC pauses.

Second, PFC has the potential for large and destructive impacts on networks. PFC pauses all upstream interfaces once it detects a risk of packet loss, and the pauses can propagate via a tree-like graph to multiple hops away. Such spreading of congestion can possibly trigger PFC deadlocks[95,101,161] and PFC storms (Case-1 in §2.1) that can silence a lot of senders even if the network has free capacity. Despite the probability of PFC deadlocks and storms being fairly small, they are still big threats to operators and applications, since currently there are no methods to guarantee they won't occur[101].

Third, even in normal cases, PFC can still suppress a large number of innocent senders. For instance, by monitoring the propagation graph of each PFC pause in a PoD, we can see that about 10% of PFC events propagate three hops (Figure 2.1a), which means the whole PoD is impacted due to a single or a small number of senders. Figure 2.1b shows that more than 10% PFC pauses suppress more than 3% of the total network capacity of a datacenter, and in the worst case the capacity loss can be 25%! Again, we can see that the 25% capacity loss is rare, but it is still a threat that operators have to plan for.

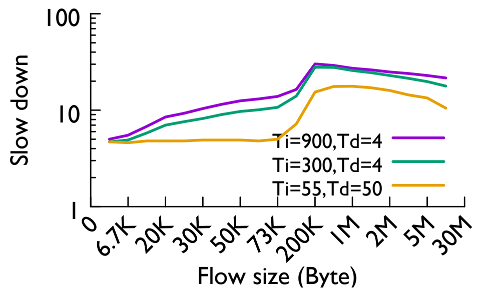
Finally, operational complexity is an important factor that is previously neglected. Because of the high-performance requirement and stability risks, it often takes months to tune the parameters for RDMA before actual deployment, in order to find a good balance. Moreover, because different applications have different traffic patterns, and different environments have different topologies, link speeds, and switch buffer sizes, operators have to tune parameters for the deployment of each new application and new environment.

Therefore, we have four essential goals for the RDMA networks: (i) latency should be as low as possible; (ii) bandwidth/utilization should be as high as possible; (iii) congestion and PFC pauses should be as few as possible; (iv) the operational complexity should be as low as possible. Achieving the four goals will provide huge value to the customers and the cloud provider, and we believe the key to achieving them is a proper CC scheme.

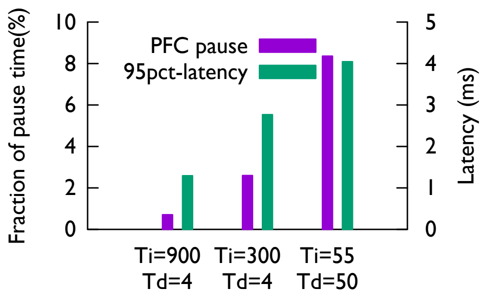
2.2.3 Tradeoffs in Current RDMA CC

DCQCN is the default CC in the RDMA networks. It leverages ECN to discover congestion risk and reacts quickly. It also allows hosts to begin transmitting aggressively at line rate and to increase their rates quickly after transient congestion (e.g., FastRecovery[188]). Nonetheless, its effectiveness depends on whether its parameters are suitable for specific traffic patterns and network environments.

In practice, operators always struggle to balance two tradeoffs in DCQCN configurations: throughput v.s. stability, e.g., Case-1 in §2.1, and bandwidth v.s. latency, e.g., Case-2 in §2.1. To make it



(a) 95-th percentile FCT under normal case with 30% network load



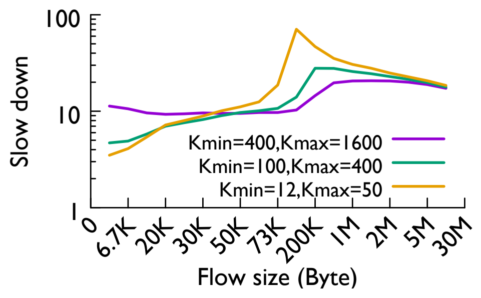
(b) PFC duration and latency with 30% network load + incast

Figure 2.2: FCT slowdown and PFC pauses with different rate increasing timers in DCQCN, using *WebSearch*.

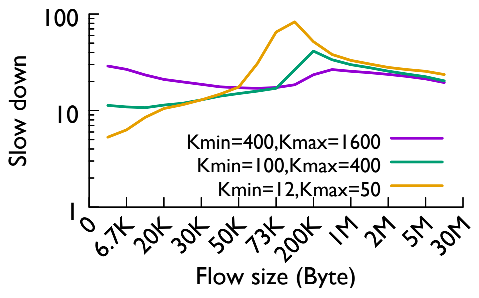
concrete, since we cannot directly change configurations in production, we highlight the two trade-offs with experiments on a testbed that has similar hardware/software environments but a smaller topology compared to production networks. The testbed is a PoD with 230 servers (each has two 25Gbps uplinks), 16 ToR switches, and 8 Agg switches connected by 100Gbps links. We use public traffic workloads, e.g., *WebSearch*[57] and *FB_Hadoop*[156] for reproducibility.

Throughput vs. Stability It is hard to achieve high throughput without harming the network's stability in one DCQCN configuration. To quickly utilize free capacity, senders must have high sensitivity to available bandwidth and increase flow rates fast, while such aggressive behavior can easily trigger buffer overflows and traffic oscillations in the network, resulting in large-scale PFC pauses. For example, Figure 2.2 approximately shows the issue in Case-1 of §2.1. Figure 2.2a shows the FCT (flow completion time) slowdown* at 95-th percentile under different DCQCN rate-increasing timers (T_i) and rate-decreasing timers (T_d) with 30% average network load from *WebSearch*. Setting $T_i = 55\mu s$, $T_d = 50\mu s$ is from DCQCN's original paper; $T_i = 300\mu s$, $T_d = 4\mu s$ is a vendor's default; and $T_i = 900\mu s$, $T_d = 4\mu s$ is a more conservative version from Alibaba. Figure 2.2a shows that smaller (T_i) and larger (T_d) reduce the FCT slowdown because they make senders more ag-

*"FCT slowdown" means a flow's actual FCT normalized by its ideal FCT when the network only has this flow.



(a) 30% network load.



(b) 50% network load.

Figure 2.3: 95-th percentile FCT slowdown distribution with different ECN thresholds, using *WebSearch*.

gressive to detect and utilize available bandwidth. However, smaller T_i and larger T_d is more likely to have more and longer PFC pauses compared to more conservative timer settings during incast events. Figure 2.2b shows the PFC pause duration and 95-th percentile latency of short flows when there are incast events whose total load is 2% of the network's total capacity. Each incast event is from 60 senders to 1 receiver. We can see that smaller (T_i) and larger (T_d) suffer from longer PFC pause durations and larger tail latencies of flows. We also have tried out different DCQCN parameters, different average link loads, and different traffic traces, all of which draw the same conclusion.

Bandwidth vs. Latency Though "high bandwidth and low latency" has become a "catchphrase" of RDMA, we find it is practically hard to achieve them simultaneously. This is because for consistently low latency the network needs to maintain steadily small queues (which means low ECN marking thresholds), while the bandwidth cannot be highly utilized if queues are too small because of the bandwidth waste under dynamic traffic, as discussed in §2.1. For example, Figure 2.3 approximately shows the issue in Case-2 of §2.1. It shows the FCT slowdown with different ECN marking thresholds (K_{min}, K_{max}) in switches and *WebSearch* as input traffic loads. Figure 2.3a shows that when we use low ECN thresholds, small flows which are latency-sensitive have lower FCT, while big flows which are bandwidth-sensitive suffer from larger FCT. The trend is more obvious when the network load is higher (Figure 2.3b when the average link load is 50%). For instance, the 95-th percentile RTT is about 150μs—30 (slowdown) $\times$ 5μs (baseline RTT)—when $K_{min} = 400$KB, $K_{max} = 1600$KB, which is a lot worse than the ML application's requirement (<50μs) in Case-2. We have tried out different DCQCN parameters, different average link loads, and different traffic traces, and the tradeoff between bandwidth and latency remains.

As mentioned in §2.1, network administrators are usually forced to sacrifice utilization (or money) to achieve latency and stability. The unsatisfactory outcome made us rethink about the fundamental reasons for the tight tensions among latency, bandwidth, and stability. Essentially, as the first generation of CC for RDMA designed more than 5 years ago, DCQCN has several design issues due to the limitations of hardware when it was proposed, which results in the challenges to network operations. For instance, with ECN-based observation, it cannot robustly adjust the sending rate, so it is either too aggressive causing instability or too conservative causing low throughput. Also, it has to maintain persistent queues to avoid bandwidth waste, because it cannot robustly ramp up rate to grab available bandwidth with just ECN information. Other than the preceding two tradeoffs, the timer-based scheduling can also trigger traffic oscillations during link failures; the queue-based feedback also creates a new tradeoff between ECN threshold and PFC threshold. We omit the details due to space limit.

Further, though we have less production experience with TIMELY, Microsoft reports that TIMELY's performance is comparable to or worse than DCQCN[189], which is also validated in §2.6.3.

2.2.4 Next Generation of High-speed CC

We advocate that the next generation of CC for RDMA or other types of high-speed networks should have the following properties *simultaneously* to make a significant improvement on both application performance and network stability:

(i) Fast and robust convergence. The network can quickly converge to high utilization while avoiding congestion. The timing and amount of traffic adjustments should be adaptive to specific traffic patterns and network environments rather than manually configured.

(ii) Close-to-empty queue. The queue sizes of in-network buffers are maintained steadily low, close-to-zero.

(iii) Few parameters. The new CC should not rely on lots of parameters that require the operators to tune. Instead, it should adapt to the environment and traffic pattern itself, so that it can reduce the operational complexity.

(iv) Fairness. The new CC ensures fairness among flows.

(v) Easy to deploy on hardware. The new CC algorithm is simple enough to be implemented on commodity NIC hardware and commodity switch hardware.

Nowadays, we have seen two critical trends that have the potential to realize a CC which satisfies all of the preceding requirements. The first trend is that switches are more open and flexible in the data plane. Especially, in-network telemetry (INT) is being popularized quickly. Almost all the switch vendors we know have INT feature enabled already in their new products (e.g., Barefoot Tofino[46], Broadcom Tomahawk3[47], Broadcom Trident3[48], etc.). With INT, a sender can know exactly loads of the links along a flow's path from an ACK packet, which facilitates the sender to make accurate flow rate adjustments. The second trend is that NIC hardware is becoming more capable and programmable. They have faster speed and more resources to expose packet-level events and processing. With these new hardware features, we design and implement HPCC, which achieves the desired CC properties simultaneously.

2.3 Design

The key idea of HPCC is to use the number of inflight bytes for both observation and control. This direct mapping between observation and control closes the gap between them, in contrast to existing CC schemes [188,142,57,109,83], so HPCC can converge to high bandwidth utilization with zero queueing in one RTT for each bottleneck link. To realize the design, we leverage the switches to provide fine-grained load information, such as queue size and accumulated tx traffic, and compute the number of inflight bytes on the hosts.

During our design, we face two major challenges. First, in existing pure rate-based CC schemes (DCQCN and TIMELY), during congestion, feedback signals can be delayed, causing a high rate to persist for a long time. This results in much more inflight data from each sender than needed to sustain high utilization; as our experiment in §2.2.3 shows, each sender can have significantly more inflight data than the BDP (Bandwidth-delay product)[†]. To avoid this problem, HPCC directly controls the number of inflight bytes as opposed to rate, so even if feedback signals are delayed, the senders do not send excessive packets because the total inflight bytes are limited. Consequently, we also derive a new way to measure the number of inflight bytes passing the bottleneck based on INT. Second, while a HPCC sender can react to network load information in each ACK, it must carefully navigate the tension between reacting quickly and overreacting to congestion feedback. We combine RTT-based and ACK-based reactions to overcome this tension.

2.3.1 HPCC Framework

HPCC is a sender-driven CC framework. As shown in Figure 2.4, each packet a sender sends will be acknowledged by the receiver. During the propagation of the packet from the sender to the receiver,

[†]In Figure 2.2b, the PFC being propagated to hosts means at least 3 switches (intra-PoD) has reached the PFC threshold. So the inflight bytes is at least $11\times$BDP per flow on average, calculated based on the datacenter specification and the incast ratio.

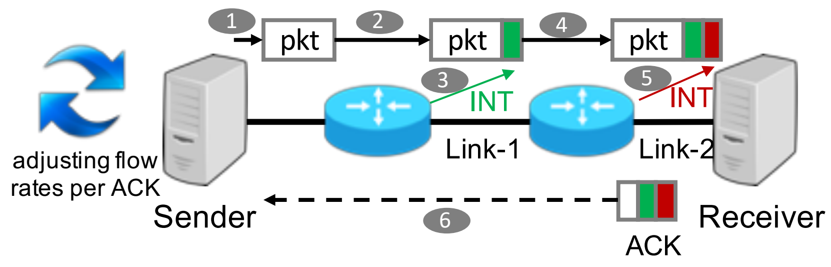


Figure 2.4: The overview of HPCC framework.

each switch along the path leverages the INT feature of its switching ASIC to insert some meta-data that reports the current load of the packet's egress port, including timestamp (*ts*), queue length (*qLen*), transmitted bytes (*txBytes*), and the link bandwidth capacity (*B*).

When the receiver gets the packet, it copies all the meta-data recorded by the switches to the ACK message it sends back to the sender. The sender decides how to adjust its flow rate each time it receives an ACK with network load information.

2.3.2 CC Based on Inflight Bytes

HPCC controls the number of inflight bytes. The inflight bytes mean the amount of data that have been sent, but not acknowledged at the sender yet.

Controlling inflight bytes has an important advantage compared to controlling rates. In the absence of congestion, the inflight bytes and rate are interchangeable with equation $inflight = rate \times T$ where T is the base propagation RTT. However, controlling inflight bytes greatly improves the tolerance to delayed feedback during congestion. Compared to a pure rate-based CC scheme which continuously sends packets before feedback comes, the control on the inflight bytes ensures the number of inflight bytes is within a limit, making senders immediately stop sending when the limit is reached, no matter how long the feedback gets delayed. As a result, the whole network is greatly stabilized.

Senders control inflight bytes with sending windows. Each sender maintains a sending window, which limits the inflight bytes it can send. Using a window is a standard idea in TCP, but the benefits for tolerance to feedback delays are substantial in datacenters, because the queueing delay (hence the feedback delay) can be orders of magnitude higher than the ultra-low base RTT[107]. The initial sending window size should be set so that flows can start at the line rate, so we use $W_{init} = B_{NIC} \times T$, where B_{NIC} is the NIC bandwidth.

In addition to the window, we also pace the packet sending rate to avoid bursty traffic. Packet pacer is generally available in NICs[188]. The pacing rate is $R = \frac{W}{T}$, which is the rate that a window size W can achieve in a network with base RTT T.

Observing congestion based on inflight bytes. Since the control knob is the number of inflight bytes, we observe congestion based on inflight bytes—such a direct mapping between observation and control enables one-RTT convergence for each bottleneck link.

For a link, we need to know the total inflight bytes of all flows traversing it. Assume a link's bandwidth is B, and the i-th flow traversing it has a window size W_i. The number of inflight bytes for this link is $I = \sum W_i$.

If $I < B \times T$, we have $\sum \frac{W_i}{T} < B$. $\frac{W_i}{T}$ is the throughput that flow i achieves if there is no congestion. So in this case, the total throughput of all these flows is lower than the link bandwidth.

If $I \geq B \times T$, we have $\sum \frac{W_i}{T} \geq B$. In this case, there must be congestion (otherwise, the total throughput would exceed the link capacity which is impossible), and queues form. The congestion can be on this link, or somewhere else if there are multiple bottlenecks.

So our goal is to control I to be slightly smaller than $B \times T$ for every link, such that there is no queue and bandwidth is highly utilized.

Estimating the number of inflight bytes for each link. The first question is how a sender uses INT information to estimate I_j for each link j on its path. Specifically, the inflight bytes consist of data packets in the queues and in the pipeline. So for each link j, we estimate I_j using its queue length (*qlen*) and its output rate (*txRate*), as in Eqn. (2.1):

$$I_j = qlen + txRate \times T \tag{2.1}$$

where *qlen* is directly from INT. *txRate* is calcualted with *txBytes* and *ts*: $txRate = \frac{ack_1.txBytes - ack_0.txBytes}{ack_1.ts - ack_0.ts}$ where ack_1 and ack_0 are two ACKs. $txRate \times T$ estimates the number of bytes in the pipeline. Eqn. (2.1) assumes all flows have the same known base RTT. This is possible in datacenters, where the RTT between most server pairs are very close due to the regularity of the topology.[‡]

In the most common congestion scenario where there is one bottleneck j, I_j is the estimation of the total inflight bytes of all flows traversing link j. In the case where some flows traverse multiple bottlenecks, I_j is the lower bound of total inflight bytes.

Reacting to the observed number of inflight bytes. Each sender should adjust its window so that I_j for each link j on its flow's path is slightly lower than $B_j \times T$—specifically, to be $\eta \times B_j \times T$ (η is a constant close to 1, e.g., 95%). Thus, for link j, each sender can multiplicatively adjust its window by a factor of $k_j = \frac{I_j}{\eta \times B_j \times T} = U_j/\eta$, where U_j is the normalized number of inflight bytes of link j:

$$U_j = \frac{I_j}{B_j \times T} = \frac{qlen_j + txRate_j \times T}{B_j \times T} = \frac{qlen_j}{B_j \times T} + \frac{txRate_j}{B_j} \tag{2.2}$$

Sender i should react to the most congested link:

$$W_i = \frac{W_i}{max_j(k_j)} + W_{AI} = \frac{W_i}{max_j(U_j)/\eta} + W_{AI} \tag{2.3}$$

where W_{AI} is an additive increase (AI) part to ensure fairness, which is very small. Note that the first term in Eqn. (2.3) is an MIMD term. This decouples the utilization control from the fairness

[‡]Shorter RTT exists, such as within a rack, but we show it is not a problem in evaluation.

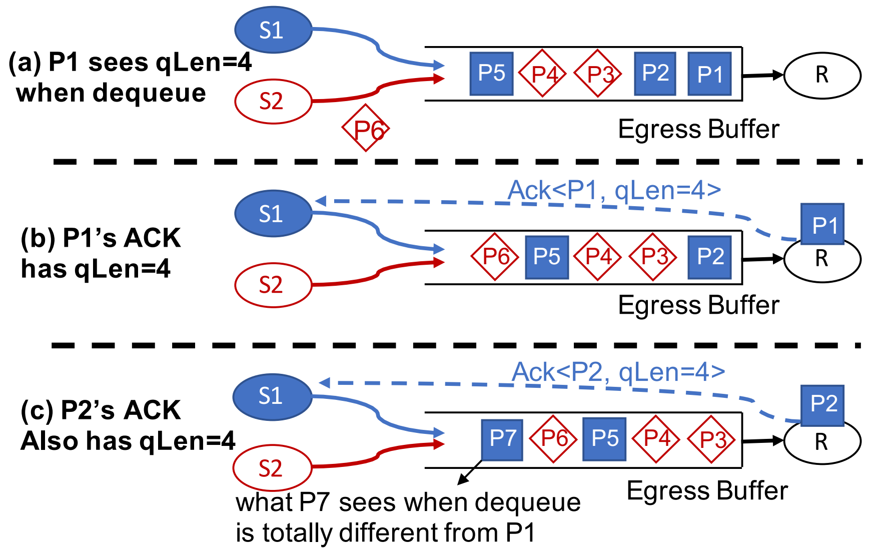


Figure 2.5: Overreaction to two sequential ACKs.

control to ensure senders quickly grab free bandwidth or avoid congestion, which is inspired by XCP[109].

Note that if there are multiple bottlenecks, Eqn. (2.1) under-estimates the number of inflight bytes, so HPCC needs multiple rounds of adjustments to resolve congestion, one round for each bottleneck. However, during incast cases, which are the most common congestion cases in datacenter[162], there is only one bottleneck, so HPCC can resolve the congestion with only one round of adjustment. Theoretical details are in §2.4.1.

Fast reaction without overreaction. With Eqn. (2.2) and Eqn. (2.3), an HPCC sender can react with every ACK. Reacting to each ACK enables fast congestion avoidance, but it reacts multiple times to ACKs describing the same packets and queues. Figure 2.5 shows an example of how reacting to each ACK causes overreaction. At the beginning, in Figure 2.5(a), there are three packets (P_1, P_2 and P_5) from sender S_1 and two packets (P_3 and P_4) from sender S_2 queued in an egress buffer of a link. When P_1 is dequeued, the buffer has 4 packets. So its ACK to S_1 indicates $qLen = 4$ (Figure 2.5(b)). Suppose S_1 uses Eqn. (2.2) and Eqn. (2.3) to decide $W(1) = W(0)/2$, where $W(0)$ is the window size of S_1 in Figure 2.5(a). However, P_2 also sees 4 packets when it is dequeued in Figure 2.5(b) and its ACK to S_1 has $qLen = 4$ in Figure 2.5(c). If S_1 blindly updates its window size based on P_2's ACK, it ends up with $W(2) = W(1)/2 = W(0)/4$. This conservative window size is an overreaction because P_1 and P_2's ACKs report the link conditions for almost the same set of packets.

One way to prevent overreaction is to make sure that the window is only adjusted when an ACK that describes a brand new set of packets is received. For instance, in Figure 2.5(c), P_7 is a packet which is sent out from S_1 after S_1 gets the ACK of P_1. Therefore, the packets P_7 sees in any queue are totally different from what P_1 sees. In other words, the network status reported in P_1's and P_7's ACKs have no overlap. Therefore, for avoiding overreaction, we always remember the first packet (Q) sent right after the window is adjusted and only adjust the window again when the sender gets Q's ACK. However, the drawback of this strategy is that merely updating the window each RTT might be too slow for handling urgent cases like failures and incasts (§2.6.4).

HPCC combines the per-ACK and per-RTT strategies to achieve fast reaction without overreaction. The key idea is to introduce a reference window size W_i^c, a runtime state updated on a per-RTT basis. Hence, only when receiving the ACK of the first packet sent with the current W_i^c, we update it with $W_i^c = W_i$, i.e., assigning the current window size W_i to the reference window size W_i^c. With W_i^c, the sender can safely update its window size using Eqn. (2.4):

$$W_i = \frac{W_i^c}{\max_j(U_j)/\eta} + W_{AI} \tag{2.4}$$

Since W_i is computed from W_i^c which is fixed in a RTT, the sender does not overreact to the same network loads. For example, in the case of Figure 2.5, we have $W(2) = W(1) = W(0)/2 = W^c/2$ even if S_1 recomputes window sizes on ACKs of both P_1 and P_2. Meanwhile, if the inflight bytes dramatically change within an RTT, the window size is still adjusted by Eqn. (2.4) because U_j is updated.

The overall workflow of the sender side CC algorithm. Algorithm 1 illustrates the overall process of CC at the sender side for a single flow. Each newly received ACK message triggers the procedure NEWACK at Line 21. At Line 22, the variable *lastUpdateSeq* is used to remember the first packet sent with a new W^c, and the sequence number in the incoming ACK should be larger than *lastUpdateSeq* to trigger a new sync between W^c and W (Line 14-15 and 18-19). The sender also remembers the pacing rate and current INT information at Line 27. The sender computes a new window size W at Line 23 or Line 26, depending on whether to update W^c, with function MEASUREINFLIGHT and COMPUTEWIND.

Function MEASUREINFLIGHT estimates normalized inflight bytes with Eqn. (2.2) at Line 5. First, it computes *txRate* of each link from the current and last accumulated transferred bytes *txBytes* and timestamp *ts* (Line 4). It also uses the minimum of the current and last *qlen* to filter out noises in *qlen* (Line 5). The loop from Line 3 to 7 selects $\max_i(U_i)$ in Eqn. (2.3). Instead of directly using $\max_i(U_i)$, we use an EWMA (Exponentially Weighted Moving Average) to filter the noises from timer inaccuracy and transient queues. (Line 9).

Function COMPUTEWIND combines multiplicative increase/decrease (MI/MD) and additive increase (AI) to balance the reaction speed and fairness. If a sender finds it should increase the window size, it first tries AI for *maxStage* times with the step W_{AI} (Line 17). If it still finds room to increase after *maxStage* times of AI or the normalized inflight bytes is above η, it calls Eqn. (2.4) once to quickly ramp up or ramp down the window size (Line 12-13).

Algorithm 1 Sender algorithm. $ack.L$ is an array of link feedbacks in the ACK; each link $ack.L[i]$ has four fields: $qlen$, $txBytes$, ts, and B. L is the sender's record of link feedbacks at the last update.

```
1: function MeasureInflight(ack)
2:     $u = 0$;
3:     for each link $i$ on the path do
4:         $txRate = \frac{ack.L[i].txBytes - L[i].txBytes}{ack.L[i].ts - L[i].ts}$;
5:         $u' = \frac{\min(ack.L[i].qlen, L[i].qlen)}{ack.L[i].B \cdot T} + \frac{txRate}{ack.L[i].B}$;
6:         if $u' > u$ then
7:             $u = u'$; $\tau = ack.L[i].ts - L[i].ts$;
8:     $\tau = \min(\tau, T)$;
9:     $U = (1 - \frac{\tau}{T}) \cdot U + \frac{\tau}{T} \cdot u$;
10:    return $U$;
11: function ComputeWind($U$, $updateWc$)
12:    if $U >= \eta$ or $incStage >= maxStage$ then
13:        $W = \frac{W^c}{U/\eta} + W_{AI}$;
14:        if $updateWc$ then
15:            $incStage = 0$; $W^c = W$;
16:    else
17:        $W = W^c + W_{AI}$;
18:        if $updateWc$ then
19:            $incStage++$; $W^c = W$;
20:    return $W$;
21: procedure NewAck(ack)
22:    if $ack.seq > lastUpdateSeq$ then
23:        $W = $ ComputeWind(MeasureInflight(ack), $True$);
24:        $lastUpdateSeq = snd\_nxt$;
25:    else
26:        $W = $ ComputeWind(MeasureInflight(ack), $False$);
27:    $R = \frac{W}{T}$; $L = ack.L$;
```

2.3.3 Parameters of HPCC

HPCC has three easy-to-set parameters: η, *maxStage*, and W_{AI}. η controls a simple tradeoff between utilization and transient queue length (due to the temporary collision of packets caused by their random arrivals. See §2.4.3), so we set it to 95% by default, which only loses 5% bandwidth but achieves almost zero queue. *maxStage* is not needed in theory (i.e., $maxStage = 0$ works perfectly fine, as shown in our simulation), although in our hardware implementation we set $maxStage = 5$ to deal with the pacing imperfection due to a short of time for implementation. W_{AI} controls the tradeoff between the maximum number of concurrent flows on a link that can sustain near-zero queues and the speed of convergence to fairness (§2.4.2). Normally we set a very small W_{AI} to support a large number of concurrent flows on a link, because slower fairness is not critical. A rule of thumb is to set $W_{AI} = \frac{W_{init} \times (1-\eta)}{N}$, where N is the expected maximum number of concurrent flows on a link. The intuition is that the total additive increase every round ($N \times W_{AI}$) should not exceed the bandwidth headroom, and thus no queue forms. Even if the actual number of concurrent flows on a link exceeds N, the CC is still stable and achieves full utilization, but just cannot maintain zero queues. Note that none of the three parameters are reliability-critical.

2.3.4 Properties of HPCC

HPCC has fewer parameters and the tuning is simpler than previous CC schemes. Most previous CC schemes, such as DCQCN[188], TIMELY[142], and DCTCP[57] which are productionized, do not have precise feedback, so they have to use heuristics to infer the current network state. These heuristics work differently in different environments, so they have parameters for operators to tune for the environments. On the other hand, HPCC uses precise feedback to know the exact network state, so HPCC does not need heuristics, and thus no need for the associated parameters.

For example, they heuristically maintain the equilibrium during steady states (i.e., there are a fixed number of flows). Specifically, the AI step and MD factor should be in a dynamic equilibrium. The smaller AI step, the stabler equilibrium and slower fairness convergence, but once some flow finishes, it takes a longer time to reclaim the free bandwidth. So operators have to carefully tune the AI step to achieve a good tradeoff; DCQCN and TIMELY even have two parameters for AI steps. In contrast, HPCC reclaims bandwidth through MI, enabled by the precise link load information, so the AI step can be small and is easy to set.

For the MD factor, they use EWMA to gradually find out the right value. Average over longer terms give more precise MD factor and thus stabler equilibrium, but once more flows join, it takes a longer time to resolve the congestion. So operators have to also carefully tune α (the parameter that controls the weight of new feedback). In contrast, HPCC directly knows the MD factor (Eqn. (2.3)), so it does not have this parameter. Note that the EWMA in HPCC is parameterless: the weights of new ACKs are automatically scaled with inter-packet time gaps.

There are also CC schemes that use fine-grained feedback, such as XCP[109] and RCP[83]. However, they use heuristic combinations of different types of feedback, so they have scaling parameters to tune their relative significance. The next bullet elaborates on this.

Advantages over CC schemes that also use fine-grained feedback from switches. XCP and RCP[109,83] combines *rxRate* and *qlen* from switches to observe congestion, which may seem similar to HPCC's use of *txRate* and *qlen*. The key difference is that using *txRate* in Eqn. (2.3) allows HPCC to accurately estimate the amount of inflight bytes, which closes the gap between observation and control and enables one-RTT convergence for each bottleneck. In XCP and RCP, their combination of *rxRate* and *qlen* has no concrete physical meaning, so they still have the gap between observation and control: one cannot predict exactly how *rxRate* and *qlen* will change after adjusting the window or rate, so they still take multiple rounds to converge. In fact, *rxRate* and *qlen* overlap in terms of the congestion that they measure: a high *rxRate* increases the queue occupancy, which implies a large *qlen*. Therefore XCP and RCP require scaling parameters (α and β) to tune

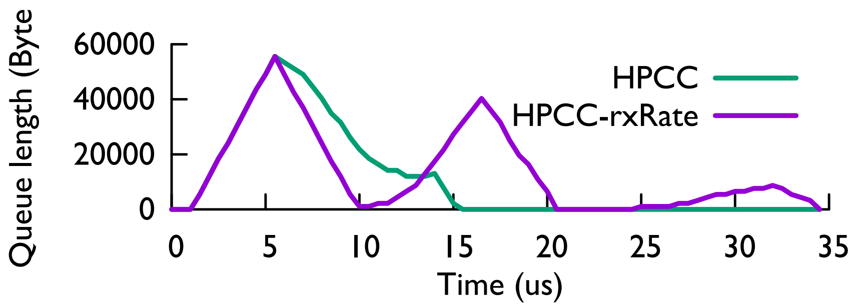


Figure 2.6: Comparison of txRate and rxRate.

the weight of *rxRate* and *qlen*, complicating the tuning process.

Viewed from another perspective, as first observed in ABC[93], using *txRate* improves the accuracy of the feedback signal in a window-based CC scheme like HPCC. The reason is that the *txRate* at a switch queue reflects the *rxRate* at that queue *one RTT in the future*, since packet transmissions from senders are clocked by acknowledgments in a window-based scheme. Therefore, by adjusting the window size based on *txRate*, the senders can anticipate what the extent of congestion will be one RTT after the measurement is taken at the switch, and react more accurately compared to when using *rxRate*.

We perform a simple experiment to compare the use of *txRate* and *rxRate*. We use HPCC and HPCC-rxRate (replacing *txRate* with *rxRate* in all calculations) in a simple 2-to-1 congestion scenario. Figure 2.6 shows the queue length over time. We can see that using *rxRate* has oscillation before it finally converges, which is a result of the aforementioned problems. On the other hand, using *txRate* gracefully converges without oscillation.

2.4 Theoretical Analysis of HPCC

For simple models of a system with arbitrary network topology and with multiple bottleneck links, we prove that HPCC has rapid convergence to a Pareto optimal rate allocation (§2.4.1), followed by a slower convergence to fairness (§2.4.2). We also derive theoretical understanding of the queueing delay (§2.4.3) and the importance of limiting inflight bytes (§2.4.4).

2.4.1 Fast Convergence of Utilization

Next we consider how quickly loads on the resources can be brought back under control following a perturbation (perhaps caused by a new source or sources starting).

We begin with a very simple discrete time model, where sources all share the same RTT and rates at sources are updated synchronously once per RTT. First note that if there is a single bottleneck resource then we could achieve the target utilization (η) in one RTT with the update $R(t + RTT) = R(t)(U_{target}/U)$ where U is the observed utilization (it is simpler to work with rates rather than windows in the analysis since rates and capacity constraints have the same units and fairness is traditionally defined with respect to rates).

Now suppose there are resources $i = 1, 2, \cdots, I$ and paths $j = 1, 2, \cdots, J$. Let A be the incidence matrix defined by $A_{ij} = 1$ if resource i is used by path j and $A_{ij} = 0$ otherwise; assume each path uses at least one resource, so that no column of A is identically zero. Let $C_i > 0$ be the (target) capacity of resource i, for $i = 1, 2, \cdots, I$, and define the vector $C = (C_i, i = 1, 2, \cdots, I)$. A rate allocation is a vector $R = (R_j, j = 1, 2, \cdots, J)$. Let Y_i be the load on resource i and let $Y = (Y_i, i = 1, 2, \cdots, I)$. From the definition of the matrix A we have that $Y = AR$. Say that R is *feasible* if the vector inequality $Y \leq C$ is satisfied, so that the load on each resource is not greater than the (target) capacity of the resource.

Suppose the initial state $R(0)$ has $R_j(0) > 0$ for $j = 1, 2, \cdots, J$ and suppose rates are updated in discrete time by the recursions

$$Y(n) = AR(n) \tag{2.5}$$

$$R_j(n+1) = \frac{R_j(n)}{\max_i\{Y_i(n)A_{ij}/C_i\}}. \tag{2.6}$$

Lemma

(i) $Y(n) \leq C$ for $n = 1, 2, \cdots$; hence after one step rates are all feasible.

(ii) $R(n+1) \geq R(n)$ for $n = 1, 2, \cdots$; hence after the first step rates are either constant or increase.

(iii) $R(n) = R$ for $n = I, I+1, \cdots$ where R is Pareto optimal; hence after at most I steps $R(n)$ is constant and is then Pareto optimal.

Proof

$$Y_i(n+1) = \sum_j A_{ij} \frac{R_j(n)}{\max_k\{Y_k(n)A_{kj}/C_k\}} \leq \sum_j A_{ij} \frac{R_j(n)}{\{Y_i(n)/C_i\}} = C_i,$$

and so rates are feasible for $n \geq 1$. Also $\max\{Y_i(n)/C_i\} \leq 1$ for $n = 1, 2, \cdots$, and hence after the first step rates are non-decreasing. Furthermore if $k = \text{argmax}\{Y_i(0)/C_i\}$ then after one time step $Y_k(1) = C_k$—the resource k is *bottlenecked*. Thereafter the rates on paths through the bottlenecked resource k remain unchanged. We can remove resource k from the network, together with all paths through it. At each subsequent step at least one more resource becomes bottlenecked and can be removed. After at most I steps either all resources are bottlenecked or all paths have been removed. At the resulting rate allocation R all paths pass through at least one bottleneck, and so no path can have its rate increased without decreasing the rate of another path: hence the rate allocation R is Pareto optimal.

The recursions Eqn. (2.5)-(2.6) thus give convergence to feasibility after just one RTT and fast convergence to a Pareto optimal allocation $R(n) = R$. However the allocation will not in general be fair, and indeed R will in general depend on the initial state $R(0)$. Next we consider how an additive increase term encourages convergence to a form of fairness.

2.4.2 Additive Increase and Fairness

Consider a network with multiple resources where RTTs vary and updates are asynchronous. For a given source let U_i be the utilization at resource i observed by the source and let $U = \max_i\{U_i\}$ where the maximum is over the resources on the path associated with the source. Suppose the rate at the source is updated once per RTT by

$$R(t + RTT) = R(t)\frac{U_{target}}{U(t + RTT)} + a$$

where $a > 0$ is a small additive increase. (In this sub-section we use R for the rate from a typical source, rather than the vector giving the rates over all sources.) Then at an equilibrium point (where $U(t) = U$ and $R(t) = R$ do not vary with time for any of the sources) we have that

$$U = \max_i\{U_i\}, \quad R = a\left(1 - \frac{U_{target}}{U}\right)^{-1}.$$

Let $U_{(1)}$ be the equilibrium utilization at the most congested bottleneck (i.e. the resource for which U_i is highest), and let $R_{(1)}$ be the equilibrium rate on paths through this bottleneck. Let $U_{(2)}$ be the equilibrium utilization at the next most congested bottleneck, and let $R_{(2)}$ be the equilibrium rate on paths which pass through this as their most congested bottleneck. Similarly define $U_{(i)}, R_{(i)}$ for $i = 3, 4, \cdots$. Thus $R_{(1)} \leq R_{(2)} \leq \cdots$. Then, by recursion of the above analysis,

$$R_{(i)} = a\left(1 - \frac{U_{target}}{U_{(i)}}\right)^{-1}.$$

Thus

$$U_{(i)} = U_{target}\left(1 - \frac{a}{R_{(i)}}\right)^{-1}$$

confirming that $U_{(1)} \geq U_{(2)} \geq \cdots$. Observe that the equilibrium utilizations are above U_{target} by an amount that increases with a. The highest utilization $U_{(1)}$ will be less than 100% if $a < R_{(1)}(1 - U_{target})$. For example, if $U_{target} = 95\%$ then a should be less than 5% of the flow rate $R_{(1)}$.

There is a tradeoff in the choice of the additive increase a: smaller values of a will produce smaller fluctuations about the equilibrium, but at the cost of slower convergence to the equilbrium. If a is small enough the rates will be approximately max-min fair.

Next consider the impact of stochastic fluctuations in the observed utilizations and hence in the rates. The distribution of $\max_i\{U_i\}$ will depend mainly on the most congested link but will be biased upwards by other congested links on the path. The rates achieved along different paths will be biased away from max-min fairness towards proportional fairness. This is not of itself a major problem: relative to max-min fairness, rate allocations under proportional fairness give an improvement of utilization across multiple resources (since the absolute priority max-min fairness gives to smaller flows can cause starvation at some resources[66]).

With more registers at sources we can exercise more control over the form of fairness achieved, as we now briefly describe. Suppose a source maintains a distinct register R_i for each resource on its path, updated by

$$R_i(t + RTT) = R_i(t)\frac{U_{target}}{U_i(t + RTT)} + a$$

where $U_i(t + RTT)$ is the utilization at resource i observed by the source over the preceding RTT and again $a > 0$ is a small additive increase. Update the rate R of the source by

$$R = \left(\sum_i R_i^{-\alpha}\right)^{-1/\alpha} \tag{2.7}$$

where $\alpha \in (0, \infty)$ is a fixed parameter.

At an equilibrium point

$$R_i = a\left(1 - \frac{U_{target}}{U_i}\right)^{-1}$$

and R is given by Eqn. (2.7). Here we interpret R_i as the rate which would be allocated to a source whose path went through just one resource, resource i, and then rate R is the α-fair rate allocation. Note that as $\alpha \to \infty$ the expression (2.7) approaches $\min_i\{R_i\}$ corresponding to max-min fairness. The case $\alpha = 1$ corresponds to proportional fairness. The case $\alpha \to 0$ approaches the rate allocation which maximizes the sum of the rates over all sources[113].

2.4.3 Bounding Queueing Delays

To achieve ultra-low latency at the queues within the network we control the windows of sources to keep the utilization of resources around a target utilization of less than 100%, and each source paces packet transmissions so that the time between two packets entering the network is the reciprocal of the source's pacing rate.

With a fixed number of long flows the natural model for a queue at a resource is then a $\sum D_i/D/1$ queue, a deterministic server with an arrival process that is a heterogeneous mix of periodic sources. This model has been analyzed extensively[149,155]. A superposition of homogeneous streams yields the greatest buffer requirement for a given load, and a Brownian bridge approximation is accurate in heavy traffic. As an example of the numerical results, if the load is 95% of capacity and there are 50 sources then the mean number of packets in the queues is about 3 and the probability there are more than 20 packets in the queue is about 10^{-9}. Even if the load is 100% the mean number of packets in the queue is only about $(\pi N/8)^{0.5}$ where N is the number of streams (and thus less than 5 with 50 sources; note that since sources are periodic, the queue remains stable even at 100% load).

In practice the arrival process at a resource is likely to be more variable than a heterogeneous mix of periodic sources, even with just a fixed number of long flows. Nonetheless we expect little queueing unless the load on the queue exceeds 100%. Conversely, if more than a small number of packets are queued then it is almost certainly because the load on the resource exceeds its capacity.

The stability analysis of XCP and RCP[64] aims for 100% utilization, and requires feedback based on queue size, as well as rate mismatch, to shrink an otherwise persistent queue. This complicates the stability analysis, which is available for only a single congested resource. When the target utilization is less than 100%, feedback based on queue size is not particularly helpful for steady state stability; and the stability analysis for systems using just feedback based on rate mismatch can be simplified and extended to a network with multiple resources and RTTs[112].

In HPCC, measurement of queue size is essential for dealing rapidly with transient overloads. New sources are allowed to start transmitting at line rate (to allow short flows to finish quickly), and this may cause queue lengths to grow rapidly especially for incasts. The interaction of window limits with the queueing term *qlen* in the algorithm is designed to drain queues rapidly (§2.4.4).

2.4.4 Importance of Limiting Inflight Bytes

Since the earliest days of packet-switching the importance of controlling the number of inflight packets traversing the network has been understood[78] and congestion control in TCP makes explicit use of window flow control[103]. In our algorithm we also limit the number of inflight packets a source has. We illustrate the benefit in this subsection.

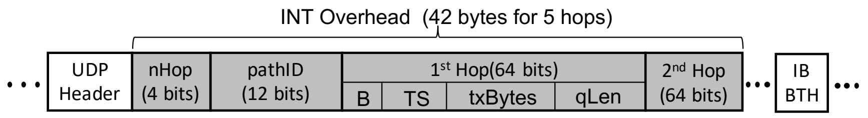
We suppose that a source limits its transmissions so that at any time the source has no more packets unacknowledged than its window limit. If an update indicates congestion, then the window limit decreases, so the source may be restricted from transmitting if acknowledgements are slow in returning to the source.

As an example, suppose a new source starts transmitting at line rate. Then it can continue to do so for the first RTT: if after this acknowledgements start returning at line rate then the window limit will not restrict the source. So if a new source transmitting at line rate does not observe congestion, then it can continue at line rate.

As a second example, suppose a set of 64 new sources begin transmitting together, each at line rate, and that the paths used by the sources converge on an intree to a single root queue which was already busy at its target utilization. Further suppose the 64 new sources continue to transmit at line rate for one base RTT - their initial burst. This is a very stressful case: the queueing time at the root of the intree will build up to nearly 64 times the base RTT of the new sources by the time the last packets of the initial bursts arrive at the root queue. But after the first acknowledgements start arriving at the new sources their windows will decrease rapidly, since these acknowledgements carry early news of the queue building up at the root queue. Consequently very few packets will be sent by the source following its initial burst until the receipt of the last packet from its initial burst, by which time it will decrease its window to about 1/65 of its initial window (note that the RTT time of the last packet from the initial burst is about 65 times the base RTT, and that this packet has observed approximately the peak queue at the root of the intree). Thus the window limits on sources allows the queue at the root of the intree to empty as fast as is possible, and the queueing term *qlen* in the algorithm forces new sources to moderate their windows following receipt of the delayed acknowledgements from the initial burst. The rates of the new sources and the rates of the existing flows through the root queue will not yet be fair - that will take longer, as a consequence of the additive increase term in the algorithm. But utilization has been brought under control as quickly as possible in a very stressful case without triggering PFC.

2.5 Implementation

We implement a prototype of HPCC in commodity NICs with FPGA programmability to realize the CC algorithm (§2.3.2) and commodity switching ASICs with P4 programmability to realize a



nHop (4 bits): it is initialized as 0 by the sender host; each switch add 1 to this counter
pathID (12 bits): XOR of all switch IDs along the path for detecting path changes
B (4 bits): the type of speed of the egress port (e.g. 40Gbps, 100Gbps, etc.)
TS (24 bits): the timestamp when the packet is emitted from its egress port
txBytes (20 bits): the accumulative total bytes sent from the egress port
qLen (16 bits): the current queue length of the egress port

Figure 2.7: The packet format of HPCC.

standard INT feature[§] (§2.3.1). We also implemented DCQCN on the same hardware platform for fair comparisons.

2.5.1 INT Padding at Switches

HPCC only relies on packets to share information across senders, receivers, and switches. Figure 2.7 shows the packet format of the INT padding after UDP header and before IB BTH (Base Transport Header) as in RoCEv2 standard. The field *nHop* is the hop count of the packet's path. The field *pathID* is the XOR of all the switch IDs (which are 12 bits) along the path. The sender sets *nHop* and *pathID* to 0. Each switch along the path adds *nHop* by 1, and XORs its own switch ID to the *pathID*. The sender uses *pathID* to judge whether the path of the flow has been changed. If so, it throws away the existing status records of the flow and builds up new records.

Each switch has an 8-byte field to record the status of the egress port of the packet when the packet is emitted. *B* is a *enum* type which indicates the speed type of the port. The timestamp (*TS*), total bytes sent so far (*txBytes*, in units of 128 Bytes) and the queue length (*qLen*, in units of 80 Bytes) are all standard INT information.

The overhead of the INT padding for HPCC is low. Inside a datacenter, the path length is often

[§]INT only requires a tiny subset of P4 programmability. Other non-P4 ASIC vendors also provide the same INT features in their new releases.

no more than 5 hops, so the total padding is at most 42 bytes, which is only 4.2% in a 1KB packet.

The switch also has modules such as destination-based ECMP routing, QoS, WRED, PFC, etc..

2.5.2 Congestion Control at NICs

Figure 2.8 shows HPCC implementation on a commodity programmable NIC. The NIC provides an FPGA chip which is connected to the main memory with a vendor-provided PCIe module and the Ethernet adapter with a vendor-provided MAC module. Sitting between the PCIe and MAC modules, HPCC's modules realize both sender and receiver roles.

The Congestion Control (CC) module implements the sender side CC algorithm. It receives `ACK` events which are generated from the RX pipeline, adjusts the sending window and rate, and stores the new sending window and rate for the flow of the current ACK in the flow scheduler via an `Update` event.

The flow scheduler paces flow rates with a credit-based mechanism. Specifically, it scans through all the flows in a round-robin manner and assigns credit to each flow proportional to its current pacing rate. It also maintains the current sending window size and unacknowledged packets for active flows. If a flow has accumulated sufficient credits to send one packet and the flow's sending window permits, the flow scheduler invokes a `PktSend` event to TX pipe.

The TX pipe implements IB/UDP/IP stacks for running in RoCEv2. It maintains the flow context for each of concurrent flows, including 5-tuples, the packet sequence number (PSN), destination QP (queue pair), etc. Once it receives the `PktSend` event with QP ID from the flow scheduler, it generates the corresponding packet and delivers to the MAC module.

The RX pipe parses the incoming packets from the MAC module and generates multiple events to other HPCC modules. (1) On receiving a data packet, the RX pipe extracts its flow context and invokes a `PktRecv` event to the TX pipe to formulate a corresponding ACK packet. If the packet

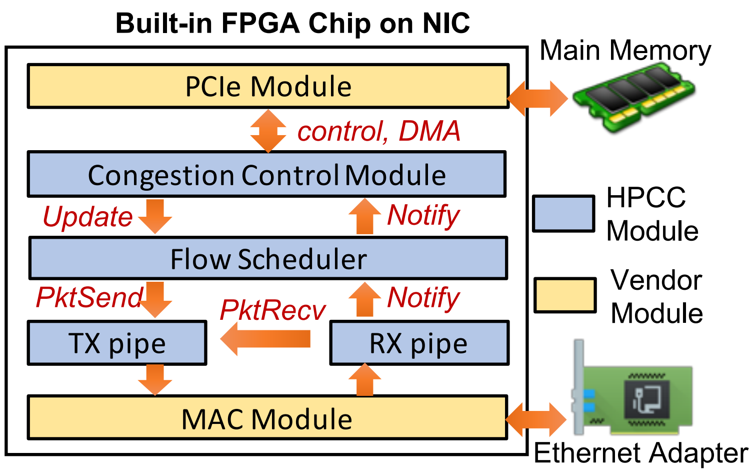


Figure 2.8: Overview of HPCC's NIC implementation.

is out-of-sequence (OOS), the TX pipe sends a NAK instead. (2) On receiving an ACK packet, the RX pipe extracts the network status from the packet and passes it to the CC module via the flow scheduler. (3) On receiving a NAK, the RX pipe notifies the TX pipe to start go-back-to-N retransmission. (4) On receiving a control packet with an RDMA operation, the RX pipe notifies the flow scheduler to create a flow with a new QP ID, or remove an existing flow. Currently, HPCC supports two operations: RDMA WRITE and RDMA READ. We leave the full support of IB verbs as future work.

2.5.3 Performance Optimization

We did many performance optimizations in our hardware implementation. Here are two examples:

Accelerating Divisions in Hardware. After receiving an ACK, the CC module needs to recompute the window, which requires divisions in Eqn (2.4). However, divisions are expensive operations especially in FPGA. We design a lookup table to replace division operations by applying the multiplication operation on the value of $\frac{1}{n}$, where n is an integer. To reduce the table size while constraining the estimation error, we choose store n values whose difference with the previous stored one is larger than ε (i.e., $\frac{1}{n_{k+1}} - \frac{1}{n_k} \geq \varepsilon \times \frac{1}{n_k}$). As a result, we can speed up the division option in Eqn (2.4) by about 8 times. The hardware memory overhead is negligible. In our current implementation, we merely use about 10KB to represent $\{\frac{1}{n}|1 \leq n \leq 2^{22}, n \in \mathbb{N}\}$.

Supporting many concurrent flows. The bottleneck to supporting a large number of concurrent flows is the speed of the clock engine in the hardware. Because the flow scheduler uses round robin over a fixed size array to schedule different flows, it can only support up to 50 concurrent flows at line rate with a single engine. To support more concurrent flows, we use multiple independent engines to schedule multiple independent arrays of flows. The FPGA we use in our prototype has six engines, which means we can support 300 concurrent flows per 25GE interface. We expect to be able to support 9K flows in ASIC implementations given that the ASIC's clock is much faster (e.g., 0.2ns per tick) than our FPGA's clock (e.g., 5ns), which is sufficient in datacenters.

2.5.4 Complexity and Overhead

Our NIC implementation takes about 12000 lines of Verilog code for the flow scheduler, RX/TX pipes, register profiles, and top flow controls. The CC modules for HPCC and DCQCN have about 2000 and 800 lines of Verilog code respectively. The hardware resources (e.g., CLB LUTs, CLB register, Block RAM, DSP, etc.) used by both HPCC and DCQCN are less than 2% of the total in the FPGA.

We believe HPCC can be easily implemented in the next-generation RoCE NICs. This is because HPCC conforms to the paradigm of existing RoCE NICs, so it just needs simple logic changes, rather than architectural changes. Specifically, it has a simple receiver and three components at the sender: measurement, calculation, and traffic enforcement, which are already in RoCE NICs, unlike some other paradigms that are architecturally different, such as receiver-driven or credit-based CC[88,74,144,99]. The major changes are incremental. We just need an INT parser at the measurement part which is just another type of header parsing, and changes to the calculation part are also simple as demonstrated in our prototype.

Our switch side implementation consists of about 300 lines of P4 code and 700 lines of configurations via Program-Dependent (PD) APIs in the control plane. The regular modules (e.g., QoS, WRED, etc.) are all standard modules used in today's commodity switches. The additional resources used to support INT function are small over baseline *switch.p4*[52] with 25% more stateful ALU usage and a 5% increase in memory and Packet Header Vector resources.

Difficulty to implement TCP-like CC in hardware. From our experience in implementation of HPCC and DCQCN and our conversations with NIC vendors, we found it is hard to implement TCP-like CC algorithms which use sliding windows. The primary reason is sliding windows should support retransmissions of arbitrary packet losses, so they needs random access to memory which is complex to implement in the hardware even for a single flow. It is even harder when the number of flows goes up to hundreds or thousands. However, implementing a sending window as we did in HPCC, which is just a sequential array per flow without random memory access, is straightforward and effective.

2.6 Performance Evaluation

In this section, we use testbed experiments with our prototype and large scale ns-3 simulations[50] to evaluate the performance of HPCC and compare to existing alternatives.

2.6.1 Evaluation Setup

Network topologies. The testbed topology mimics a small scale RDMA PoD in Alibaba's production. The testbed includes one Agg switch and four ToRs (ToR_1-ToR_4) connected via four 100Gbps links. There are 32 servers in total and each server has two 25Gbps NICs. 16 servers are connected to ToR_1 and ToR_2 via two uplinks, and the other 16 servers are connected to ToR_3 and ToR_4. The base RTT is 5.4μs within a rack and 8.5μs cross racks.

The topology in the ns-3 simulations is a FatTree[55]. There are 16 Core switches, 20 Agg switches, 20 ToRs and 320 servers (16 in each rack), and each server has a single 100Gbps NIC connected to a single ToR. The capacity of each link between Core and Agg switches, Agg switches and ToRs are all 400Gbps. All links have a 1μs propagation delay, which gives a 12μs maximum base RTT. The switch buffer size is 32MB which is derived from real device configurations. The whole network is a single RDMA domain.

Traffic loads. We use widely accepted and public available datacenter traffic traces, *WebSearch*[188] and *FB_Hadoop*[156] in both testbed experiments and simulations. We adjust the flow generation rates to set the average link loads to 30% and 50% respectively. We also create some simple artificial traffic loads to evaluate the microbenchmarks of HPCC.

Alternative approaches. We compare HPCC with DCQCN and TIMELY[142], which are CC schemes designed for RDMA. Since neither of them limits inflight bytes, we also try to improve them by adding a sending window (same as we use for HPCC), and we call the improved version "DCQCN+win" and "TIMELY+win". We also compare with DCTCP[57] which is a host-based TCP-like CC for high throughput and low latency in datacenter networks. We remove the slow start phase in DCTCP for fair comparisons.

Parameters. For HPCC, we set $W_{AI} = 80\text{bytes}$[¶], $maxStage = 5$ in testbed and $maxStage = 0$ in simulation, and $\eta = 95\%$[‖] in Algorithm 1. We set T to 9μs for testbed and 13μs for simulations, which are slightly greater than the maximum RTT of the networks. For DCQCN, we use the parameters suggested by a major NIC vendor; For TIMELY, we use the parameters suggested in[142].

[¶]Calculated based on 100 concurrent flows under 100Gbps according to §2.3.3.

[‖]We tried η from 95% to 98%, all of which give similar results. Here we show the most conservative setting.

For DCQCN and DCTCP, we scale the ECN marking threshold proportional to the link bandwidth (Bw). For DCQCN, we set $K_{min} = 100KB \times \frac{Bw}{25Gbps}$ and $K_{max} = 400KB \times \frac{Bw}{25Gbps}$ according to Alibaba's experiences (no vendor suggestion available). For DCTCP, we set $K_{min} = K_{max} = 30KB \times \frac{Bw}{10Gbps}$ according to[57]. We set the dynamic PFC threshold so that the PFC is triggered when an ingress queue consumes more than 11% of the free buffer.

Performance metrics. We have five performance metrics. (i) FCT slow down; (ii) Real-time bandwidth of individual flows; (iii) network latency; (iv) PFC pause duration; (v) Size of in-network queues.

INT overhead. For considering the impact of INT overhead on the performance, we assume each packet in HPCC has an additional 42 bytes in the header. This is a worst-case assumption because a data packet merely has 42 bytes INT meta-data at the last hop.

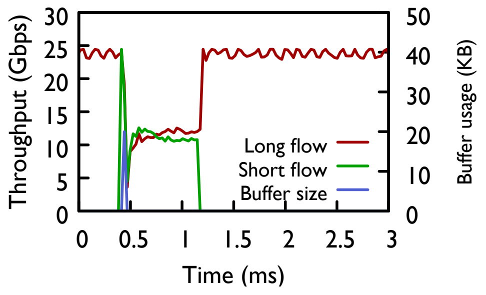
2.6.2 Testbed Experiments

We run our prototype (§2.5) on the testbed. We compare HPCC with DCQCN with both microbenchmarks and realistic traffic loads.

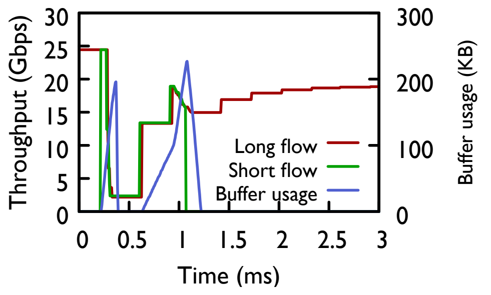
Microbenchmarks

HPCC has faster and better rate recovery Figure 2.9a and 2.9b illustrate the behaviors of HPCC and DCQCN with *Long-Short* traffic. A long flow sends at full line rate, and later a short flow with 1MB size joins sharing the same links as the long flow and leaves after a period of time. HPCC recovers the rate of the long flow right after the short flow ends, while DCQCN cannot recover to line rate even after 2ms (>350 RTTs). HPCC ramps up quickly because its feedback does not rely on the queue.

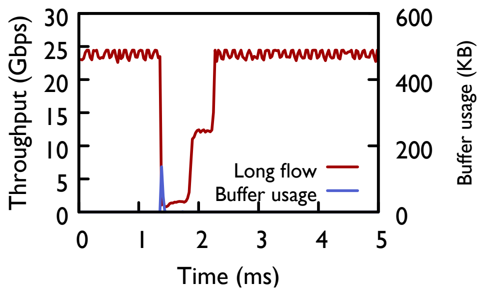
HPCC has faster and better congestion avoidance Figure 2.9c and 2.9d show how HPCC and DCQCN react to congestion caused by *Incast*. Seven senders start to send flows at the same time to-



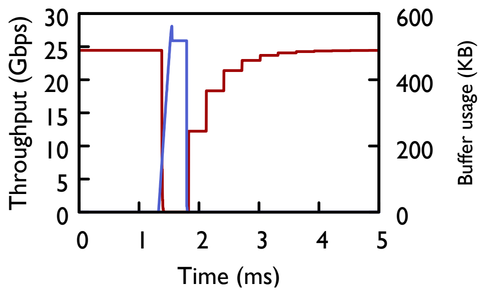
(a) *Long-Short* HPCC



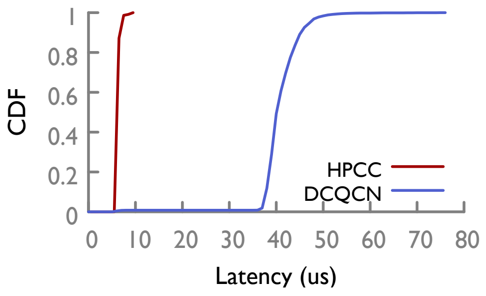
(b) *Long-Short* DCQCN



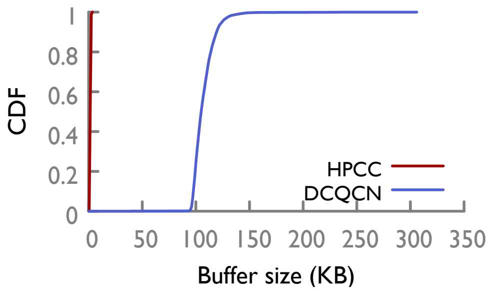
(c) *Incast* HPCC



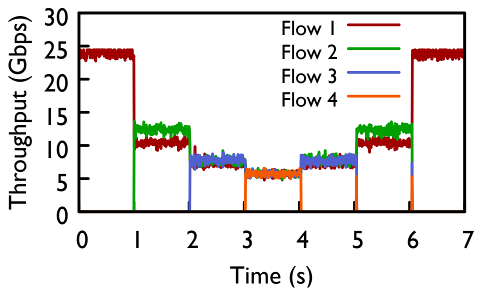
(d) *Incast* DCQCN



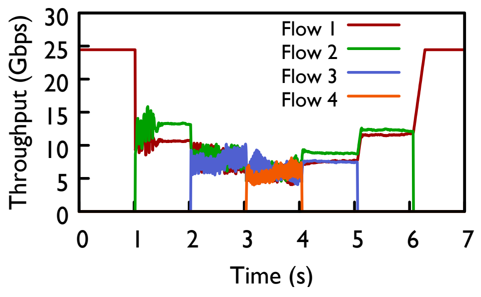
(e) *Elephant-Mice* latency



(f) *Elephant-Mice* queue size



(g) *Fair-Share* HPCC



(h) *Fair-Share* DCQCN

Figure 2.9: Comparing HPCC and DCQCN on testbed with four microbenchmark traffic loads.

wards the receiver of a long-running flow. HPCC quickly reacts after just one round trip, so the queue drains quickly. With DCQCN, the queue builds up to 550KB due to two reasons: (1) it waits for the queue build up for ECN, and (2) it does not limit the inflight bytes.

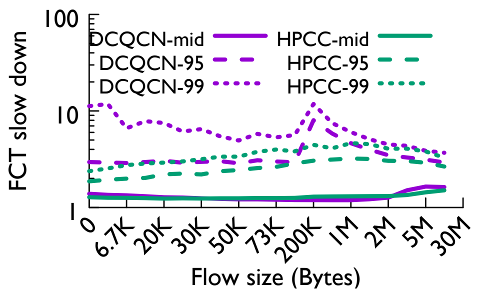
HPCC has lower network latency We keep sending mice flows (1KB each) through a link that is saturated by two elephant flows, and measure the mice flow latency and the buffer size. Figure 2.9e and 2.9f show HPCC keeps a near-zero queue and therefore the latency of mice flows is close to 5.4μs, the base RTT. DCQCN keeps a standing queue around the ECN marking threshold, so the latency is consistently higher than 35μs.

HPCC has fairness Figure 2.9g and 2.9h show the fairness of HPCC and DCQCN. 4 flows join a link one by one every second and leave afterwards. HPCC provides good fairness even in a short time scale.

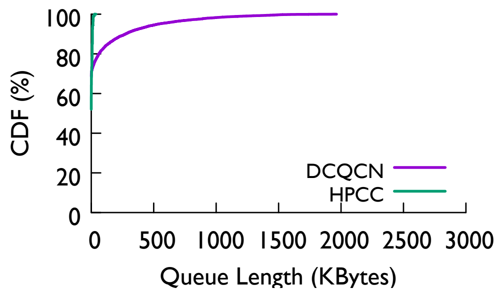
End-to-end performance We evaluate HPCC and DCQCN under *WebSearch*, at 30% and 50% loads. We also run similar experiments under *FB_Hadoop*, which show similar trends.

HPCC significantly reduces FCT for short flows HPCC and DCQCN achieve similar FCT slowdowns in the median, but at 95-th and 99-th percentile, HPCC achieves a much better FCT slowdown especially for short flows (Figure 2.10a and 2.10c). For example, at 30% load, HPCC reduces the 99-th percentile FCT slowdown from 11.2 down to 2.38 for flows shorter than 3KB, which is only 16.9μs. The gap is larger with higher loads. For example, at 50% load, HPCC achieves a 95% reduction on the 99-th percentile FCT slowdown, from 53.9 down to 2.70, for the flows shorter than 3KB, which is only 19.2μs.

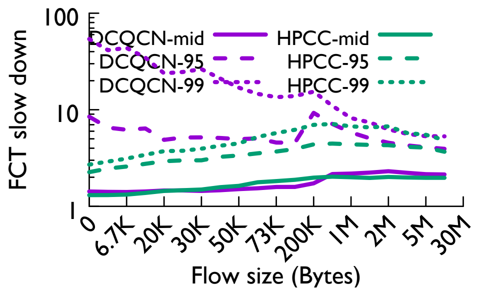
HPCC has steadily close-to-zero queues Figure 2.10b and 2.10d show the CDF of queue lengths at switches, which provides more insight into the achieved performance. In both cases, the median queue size is 0, which explains the closeness of median FCT slow down. However, HPCC keeps the ultra-low queue size even at the very tail, thus achieving much lower FCTs for short flows. For example, at 50% load, HPCC's 95-th and 99-th percentile queue sizes are 19.7KB and 22.9KB, whereas



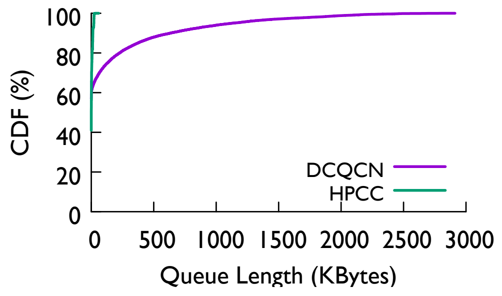
(a) 30% Avg. Load



(b) 30% Avg. Load



(c) 50% Avg. Load



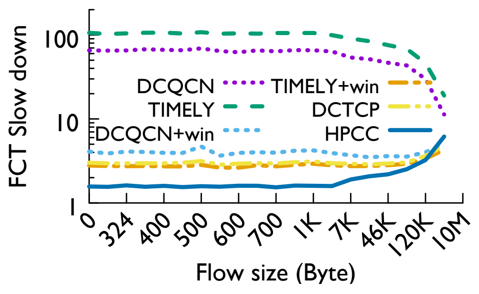
(d) 50% Avg. Load

Figure 2.10: FCT slow down and queue size of HPCC and DCQCN in testbed with *WebSearch* (30% and 50% avg. load).

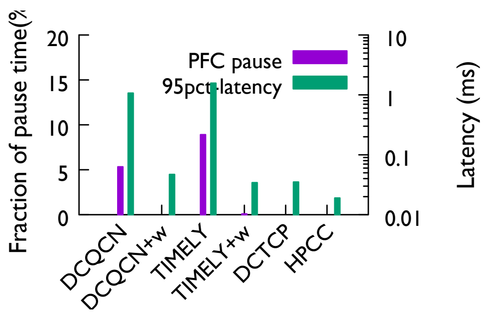
DCQCN's sizes are 1.1MB and 2.1MB. These experiments confirm that HPCC is very effective at keeping the queue near zero under realistic traffic patterns. As a result, there is no packet loss and PFC is not necessary because the queue size never reaches the PFC threshold.

2.6.3 Large-scale Event-driven Simulations

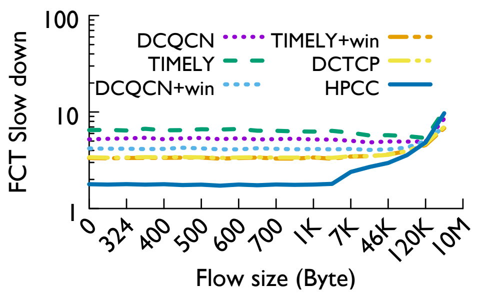
We verified the fidelity of simulation by performing the same experiment as the testbed, which matches testbed results well. We then use simulations to evaluate HPCC on a larger network topology and higher line rates. Figure 2.11 shows the comparison of HPCC and other CC schemes for *FB_Hadoop* traffic. To stress test with diverse traffic patterns, we either add incast traffic to 30% load traffic or run 50% load traffic. We generate the incast traffic by randomly selecting 60 senders and one receiver, each sending 500KB. The incast traffic load is 2% of the network capacity. We also tried various levels of traffic load, incast sizes and incast ratios, as well as with *WebSearch*, all of



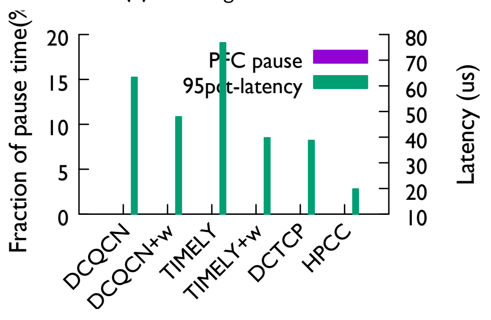
(a) 30% Avg. Load + incast



(b) 30% Avg. Load + incast



(c) 50% Avg. Load



(d) 50% Avg. Load

Figure 2.11: FCT slow down at 95-th percentile, PFC and latency with *FB_Hadoop* (30% avg. load + 60-to-1 incast and 50% avg. load).

which show similar trends. Here are the key observations:

HPCC is beneficial to short flows. Since HPCC keeps near-zero queues and resolves congestion quickly, it is beneficial to short flows. All other CC schemes maintain standing queues, so they cannot keep the latency low. Figure 2.11a and 2.11c show that for the flows shorter than 120KB, HPCC achieves much lower FCTs than all the other schemes at 95-th percentile. This is beneficial to applications with many short flows. This is the case for *FB_Hadoop*, where 90% of the flows are shorter than 120KB.

Figure 2.11b and 2.11d show that on the tail, HPCC still achieves very low round trip latency, under 20μs. For example, the 95-th percentile latency at 50% load is 19.8μs, which is less than 8μs extra latency compared to the 12μs base RTT.

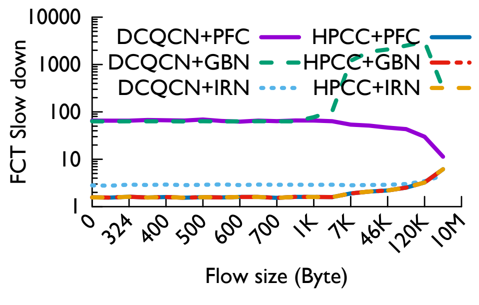
DCTCP outperforms DCQCN and TIMELY because it controls the queue better as its window limits the inflight bytes. This may seem to contradict previous statements about their relative performance[188,142]. However, this discrepancy is mainly because they compare hardware DCQCN/kernel-bypassing TIMELY with DCTCP based on kernel[188,142], which is known to have huge performance cost and requires higher ECN threshold, while we only simulate the effect of CC, excluding the cost introduced by the software nature for a fair comparison. That said, HPCC reduces DCTCP's latency by more than 2 times, which is significant in datacenters.

Throughput for long flows. Since HPCC explicitly controls the bottleneck links to have a 5% bandwidth headroom, and the INT header consumes extra bandwidth, the long flow has a higher slowdown as shown in Figure 2.11a and 2.11c. The slowdown increases with a higher load as the theory in[58] shows. The reason is that the long flow slowdown is inversely proportional to the residual capacity of the network. Other CC schemes aim to fully utilize the bandwidth, so their residual capacity is $(100\% - \mathit{load})$ which is 44.6% at 50% load (including header and ACK); for HPCC, the residual capacity is $(95\% - \mathit{load} \times (1 + \mathit{INT\_overhead}))$ which is 36.1%. So at 50% load the long flows are 1.24 times slower with HPCC than with other schemes, which matches the FCT quite well. This is a fundamental tradeoff we have to make in favor of short flows.

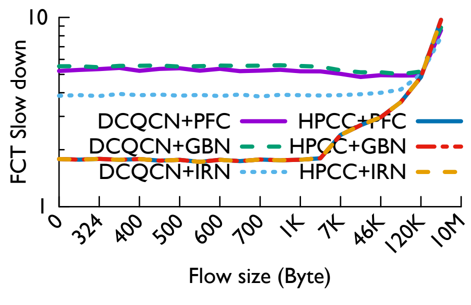
CC is the key to achieve stability and high performance. As Figure 2.11b shows, large scale PFC pauses only appear when using DCQCN and TIMELY, which confirms our insight that CC is the key to the stability problem. Specifically, controlling the inflight bytes is the key: just adding a sending window to DCQCN and TIMELY reduces PFCs to almost zero.

We further show that a good CC scheme lessens the importance of the flow control choices. We use PFC, go-back-N retransmission, and IRN[143]**, in combination with DCQCN and HPCC, and perform the same experiment. Figure 2.12 shows that with HPCC, different flow control

**When using go-back-N or IRN, where packet losses are not prevented by PFC, we set the dynamic threshold for the egress queues with $\alpha = 1$, which allows a single congested egress port to consume up to half of the buffer.



(a) 30% Avg. Load + incast



(b) 50% Avg. Load

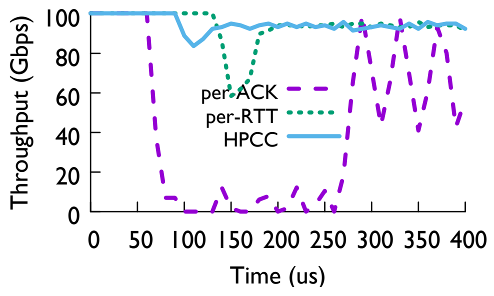
Figure 2.12: FCT slow down at 95-th percentile, with different flow control choices. GBN stands for go-back-N.

schemes do not affect the performance. On the other hand, since DCQCN controls the queue poorly, a better flow control does improve its performance (it is worth noting that, IRN adds a fixed sized window which also limits the inflight bytes, and thus has improvement over the other two schemes). But even with IRN, DCQCN still cannot match HPCC's performance, which confirms that CC is the key problem.

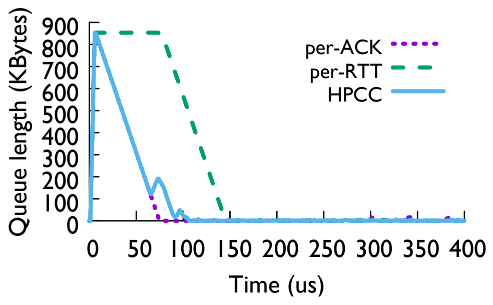
2.6.4 Design Choices

We use a simple 16 to 1 incast scenario to show the design choices of HPCC. The 16 senders and 1 receiver are connected via 100Gbps links through a single switch, with $1\mu s$ link propagation delay.

HPCC achieves fast reaction without overreaction. We illustrate the benefit of HPCC's strategy of combining per-ACK and per-RTT reactions. Figure 2.13 shows the time series of aggregate throughput and queue lengths. The queue builds up at the beginning of all flows. Since the first few ACKs of each flow already see the long queue, per-ACK reaction reacts to the queue quickly. However, it incurs a significant overreaction, so the aggregate throughput soon drops to almost 0 and then oscillates. Per-RTT reaction reacts to the queue slowly (only after all the ACKs in the first round are received) and wastes the information brought by the first few ACKs. As a result, the long queue persists for a long time. HPCC introduces a reference rate that is updated every RTT and re-

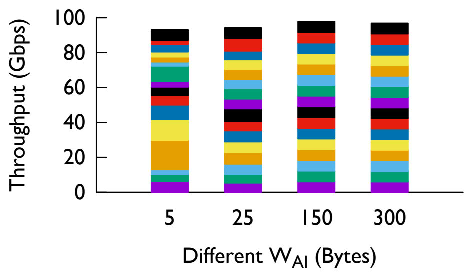


(a) Total throughput.

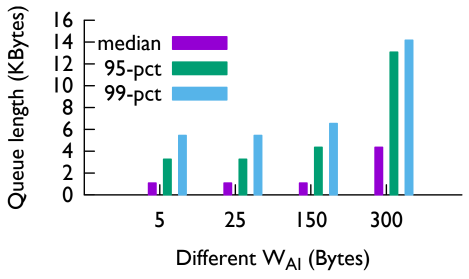


(b) Queue length.

Figure 2.13: Different ways of reacting to ACK.



(a) Total throughput.



(b) Queue length.

Figure 2.14: Fairness and queue size with W_{AI}.

acts to every ACK based on the reference rate, so HPCC achieves fast reaction without overreaction.

Tuning W_{AI} for HPCC. Figure 2.14a shows the first 10ms of the throughput of different flows. Figure 2.14b shows the queue length distribution, sampled every 1μs.

With 16 concurrent flows, W_{AI} should not exceed 100 Gbps$\times$4μs $\times(1-0.95)/16 \approx 150$bytes (4$\mu$s is the base RTT), which is confirmed by the result. Specifically, all W_{AI} within 150bytes achieve a queue length within 4KB at 95-th percentile, while W_{AI}=300bytes has a queue length of 13KB at 95-th percentile (Figure 2.14b). Within the 150 bytes range, a higher W_{AI} has better fairness. That said, since we need to prepare for the worst case, we set W_{AI} to sustain 100 concurrent flows (25 bytes in this case) that has good fairness.

It is worth noting that, HPCC's performance degrades gracefully with a high W_{AI}. In the 300 bytes case, the queue length is still very low in general: a 13KB queue just means 1μs queueing delay.

2.7 Related Work

CC is an enduring research topic. Here we try to cover several related works which are closely related.

RDMA CC. TIMELY[142] is a pioneer in RDMA CC, which uses RTT as a congestion signal. §2.6 shows that TIMELY suffers from incast congestion because it gradually adjusts its rate. Furthermore, it can converge to much longer queues than DCQCN[189]. iWarp[154] is an alternative to RoCEv2. It puts the full TCP stack into hardware NIC. As a result, iWarp suffers from well-known TCP problems in datacenters[172], such as high latency and vulnerability to incast. Furthermore, due to the complexity to implement TCP stack in hardware, iWarp NICs in general have a higher cost[90].

General CC for datacenter networks. DCTCP[57] is implemented in host software which suffers from high CPU overhead and high latency, while implementing them in hardware raise problems similar to iWarp. In addition, because they both use ECN similar to DCQCN, they can hardly achieve small queues.

There are several proposals aiming to reduce latency with changes in both host software and switch hardware. pFabric[59] needs to run sophisticated priority scheduling logics in switches and to correctly prioritize traffic in hosts, which are hard to deploy[99]. HULL[58] advocates leaving a bandwidth headroom for ultra-low latency in datacenters, which is similar to HPCC. However, it is based on DCTCP, which cannot slowly ramp up the sending rate to grab free bandwidth. DeTail[184] needs a new switch architecture for lossless fabric and performs per-packet adaptive load balancing of packet routes in switches. HOMA[144] and NDP[99] are receiver-driven, credit-based solutions, which is a big shift from the state-of-the-art in practice, since they have complex receivers. They, together with PIAS[63] also require priority queues, while the number of priority queues is limited in switches, and production networks often have to reserve them for application QoS. Different from these solutions, one HPCC's design goal is ease of implementation in hardware for offloading and deployability with the start-of-the-art commodity NICs and switches. In addition, HPCC works with a single priority queue.

There are also explicit CCs such as XCP[109] and RCP[83]. Besides the key difference discussed in § 2.3.4, both of them require switches to perform computation which is not widely available in most commodity switches, but INT is widely available (§ 2.2.4). Moreover, because HPCC allows line rate start, new flows can finish faster than with slow start (XCP) and processor sharing (RCP). The line rate start is allowed by our control on inflight bytes, which drains queues rapidly. HPCC's decoupling of utilization and fairness is inspired by XCP.

The stability of CC algorithms has been investigated by several authors using a variety of simplified models[64,109,112,176]. We draw two main insights from previous work. Firstly the speed of adaptation to new observations of congestion should be scaled to round-trip times in order to avoid destabilizing oscillations. Secondly, while feedback based on queue size is important for dealing with sudden overloads, it is not particularly helpful for steady state stability when queueing delays are short compared with round-trip times (§2.4.3). So we aim to keep link utilizations less than 100% to keep steady state queueing delays very short.

Flow controls for RDMA. IRN[143] and MELO[135] are recent proposals to reduce hardware-based selective packet re-transmissions to prevent PFC pauses or even replace PFC. These efforts are orthogonal and complementary with HPCC. Different from the fixed window used in IRN, the sending window in HPCC is proportional to flow's sending rate with better network stability.

2.8 Summary

We share our study at Alibaba production networks on the difficulties to operate RDMA networks with the state-of-the-art high-speed CC. We propose HPCC as a next-generation CC for high-speed networks to achieve ultra-low latency, high bandwidth, and stability simultaneously.

HPCC achieves fast convergence, small queues, and fairness by leveraging precise load information from INT. It has been implemented with commodity programmable NICs and switches and shows remarkable gains. We believe HPCC is a start towards CC for future hyper-speed networks.

3

Sundial: Robustly Bounding Precision at Submicrosecond Level in Clock Sync

3.1 Introduction

Clock synchronization is increasingly important for datacenter applications such as distributed transactional databases[77,159], consistent snapshots[69,117], network telemetry, congestion control, and distributed logging.

One key metric for clock synchronization is the *time-uncertainty bound* for each node, denoted as ε in this paper, which bounds the difference between local clock and other clocks. This concept is used by TrueTime in Spanner[77]. Spanner leverages TrueTime to guarantee the correctness properties around concurrency control and provide consistency in distributed databases. Another example is consistent snapshots, which are commonly used for debugging or handling failures in distributed systems. To ensure consistency among snapshots, each node needs to wait for its time-uncertainty bound (ε) before recording the states.

Traditional clock synchronization techniques provide ε at the millisecond level (e.g., <10ms in TrueTime[77]), which is no longer effective for modern datacenter applications with increasing performance requirements and ultra low latency datacenter networks (e.g., with latency around 5μs[131]). Today's applications can benefit significantly from submicrosecond-level ε. For example, FaRMv2[159], an RDMA-based transactional system, observes the median transaction delay can drop by 25% if we improve ε from $\sim$20μs to 100ns. CockroachDB[18] can significantly reduce the retry rate when ε drops from 1ms to 100ns based on an experiment in[91].

Providing submicrosecond-level ε can also enable new network management applications. For example, with submicrosecond-level clock differences across devices, we can measure one-way delay, locate packet losses, and identify per-hop latency bursts[129,130]. It also enables synchronized network snapshots[181] which are useful for identifying RTT-scale network imbalance and collect global forwarding state. Accurate one-way delay provides a better congestion signal to delay-based congestion control[142,118] to differentiate between forward and reverse path congestion.

There are several systems that achieve submicrosecond-level clock precision. The state-of-the-art commercial solution on precise clock synchronization is Precision Time Protocol (PTP)[20]. PTP is widely available in switches and NICs[35,54,53]. Each switch or NIC has a hardware clock driven by an oscillator, generates timestamped synchronization messages *in software*, and sends them over a spanning tree to synchronize with other nodes. Normally, oscillator drifts stay within $\pm 100\mu s$ per second and the devices synchronize every 15ms to 2 seconds[53,20]. A recent proposal DTP[124] sends messages in the physical layer every few microseconds and can also achieve $\sim$100ns precision. Huygens[91] is a clock-synchronization system built in software that achieves <100ns precision by using Support Vector Machines to accurately estimate one-way propagation delays.

While these works provide high clock precision under normal cases, the time-uncertainty bound ε grows to 10-100s of μs as datacenters are subject to a variety of failures. In large-scale datacenters, there are common temperature-related failures which affect oscillator drifts. There are also frequent link, device, and domain failures (i.e., a domain of links and devices that fail together) that affect the synchronization across nodes (see §3.3).

In this paper, we present Sundial, which provides $\sim$100ns time-uncertainty bound (ε) under failures including temperature-related, link, device and domain failures and reports ε to applications – two orders of magnitude better than current designs. Even in cases of simultaneous failures across domains, Sundial provides correct ε to applications. Sundial achieves this with a hardware-software codesign that enables fast failure detection and recovery:

Fast failure detection based on frequent synchronous messaging on commodity hardware: Sundial exchanges messages every $\sim 100\mu s$ in hardware without changing the physical layer. The frequent message exchange enables fast failure detection and recovery, and frequent reduction of ε. To ensure fast failure detection for remote nodes in the spanning tree, Sundial introduces *synchronous* messaging which ensures that each node sends a new message only when it receives a message from the upstream.

Fast failure recovery with precomputed backup plan that is generic to all types of failures: To enable fast failure recovery, Sundial controller precomputes a backup plan consisting of one backup parent for each node and a backup root, so that each device can recover locally. The backup plan is generic to different types of failures (i.e., link, device failures, root failures, and domain failures) and ensures that after failure recovery, the devices remain connected without loops. We introduce a new search algorithm for the backup plan that extends a variant of edge-disjoint spanning tree algorithm[167] but with additional constraints such as no-ancestor condition (the edge in the current tree cannot be a forward edge in the backup tree) and disjoint-failure-domain condition (no domain failure can take down both the parent and the backup parent for any device). Our algorithm only takes 148ms on average to run on an example Jupiter[162] topology with 88K nodes.

We evaluate Sundial with experiments in a $>$500 machine prototype implementation and via large-scale simulations. Sundial achieves $\sim$100ns time-uncertainty bound both under normal time and under different types of failures, which is more than two orders of magnitude lower than the state-of-the-art solutions such as PTP[20], Huygens[91], and DTP[124]. Sundial reduces the commit-wait latency of Spanner[77] running inside a datacenter by 3-4x, and improves the throughput of Swift congestion control[118] by 1.6x under reverse-path congestion.

3.2 Need for *Tight* Time-uncertainty Bound

A clock synchronization system for datacenters need not only a current value of time but also time-uncertainty bound that is used by applications for correctness as well as performance. We describe several datacenter applications and how tight time-uncertainty bound benefits them below.

Distributed Transactional Databases: Spanner[77], FaRMv2[159] and CockroachDB[18] are some examples of distributed databases deployed at scale in production that directly use time-uncertainty bound to guarantee consistency—transactions wait out time-uncertainty bound before committing a transaction. Spanner is the first to use ε in production transactional systems. While it is globally distributed, its idea of using ε is adopted in many intra-datacenter systems such as FaRMv2[159]. However, inside datacenters, with recent software and hardware improvements such as RDMA, NVMe, and in-memory storage, transaction latencies are going towards microsecond level. For example, FaRMv2 is built atop RDMA for datacenters and has ε of $\sim$20μs which already accounts for 25% of median transaction latency! Tight ε improves the performance of these systems both in terms of latency and throughput.

Consistent snapshots: Consistent snapshots[69,117] is another important application for datacenters for debugging, failure handling, and recovery for cloud VMs. The consistency across servers can be guaranteed by waiting out ε to ensure the scheduled snapshot time is passed. With recent software and hardware improvements, ε becomes a performance bottleneck at a similar level as in distributed databases, limiting the frequency of taking snapshots.

Network telemetry: As network latency reduces to the order of a few microseconds, millisecond-level ε is too coarse-grained. Tight ε enables a wide range of fine-grained network telemetry. For example, per-link latency or packet losses can be measured by comparing the timestamps or counters at both ends of a link read at the same time[130,129,190]. Synchronized network snapshots at RTT scale can be enabled with tight time-uncertainty bound, and can be used for various telemetry needs such as measuring traffic imbalance across different links/paths in the dataceter[181].To achieve these, switch clocks also need to be synchronized.

One-way delay (OWD): Synchronized clocks enable the measurement of one-way delays. Small ε provides a tighter bound on the error in the measurement especially under failures. Measurement of OWD is useful for many applications including telemetry and congestion control. For example, OWD differentiates between forward and reverse-path congestion improving performance of delay-based congestion control algorithms such as Swift[118] (§3.6.3).

Distributed logging: A key challenge for debugging large-scale distributed systems is to analyze logs collected from different devices with clock differences. Tighter ε enables more useful analysis and opens up more distributed debugging opportunities. Our $\sim$100ns ε is about the same as L3 cache miss time, so it can help order all log messages in a datacenter. We note that this class of applications has an additional requirement in that the synchronized clocks follow a master clock that reflects the physical time of day (§3.4.5).

3.3 Failures in Clock Synchronization System

In this section, we discuss the different failure scenarios affecting a clock synchronization system and their respective impacts. We start with a brief background on clock synchronization to aid the discussion.

3.3.1 Background on Clock Synchronization

The clock is driven by a crystal oscillator. Every device has a clock, whose value is incremented on every tick of a hardware oscillator. Different oscillators, even of the same type, have slightly different frequencies. The frequency of an oscillator may change over time, due to factors such as temperature changes, voltage changes, or aging resulting in clocks to drift away over time. As an example, oscillators in production networks can have a frequency variation of ± 100 ppm (parts per million)[42], meaning that the oscillator can drift within the range of $\pm 100\mu$s per second compared to running at the nominal frequency. More stable oscillators (e.g., atomic clocks based on Cesium, Hydrogen or Rubidium particles or oven-controlled oscillators) are too expensive to deploy on every device in production.

Clocks exchange messages with each other for synchronization. To ensure that clocks remain close to each other, we need to periodically adjust the clocks to account for potential drift. Figure 3.1 shows an example where clock B synchronizes to A. A sends a synchronization message (abbreviated as sync-message in this paper) with a timestamp T_1^A based on A's clock, and B records the receiving time (timestamped by B) of the sync-message T_1^B. Now, if B knows the message delay d_{AB} from A to B, B can compute the *offset* between A and B as $T_1^A + d_{AB} - T_1^B$. To know d_{AB}, B sends another message to A to measure RTT, and use half of RTT to estimate: $d_{AB} = (T_2^A - T_1^A - (T_2^B - T_1^B))/2$.

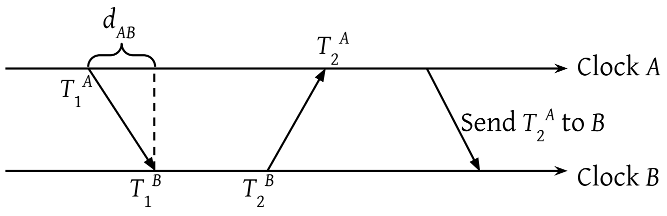


Figure 3.1: Message exchanges to synchronize B to A.

B uses *offset* to adjust its clock. A periodically sends out these sync-messages at an interval denoted by sync-interval. The accuracy of d_{AB} depends on multiple factors and we discuss them below.

A network of clocks synchronize using a synchronization structure. A common way to do this is to construct a spanning tree over which sync-messages are sent, e.g., PTP which is the most widely available system for datacenter clock synchronization uses a spanning tree with one device serving as the root (called master or grandmaster). The model for best case synchronization is that each device's parent is one of its direct neighbors in the physical network and sync-messages flow periodically from the root across the spanning tree.* This has two advantages. First, it allows switch clocks to also be synchronized enabling additional telemetry applications (§3.2). Second, it significantly improves the measurement of d_{AB} as shown in Figure 3.2. Noises in estimation of d_{AB} by halving the RTT can arise due to (1) asymmetric propagation delays of the forward path and the reverse path, and (2) queuing delays. For direct neighbors in the physical network, propagation delay asymmetry is near zero, and there is no queuing delay†. There are proposals that do not use a spanning tree as the synchronization structure but either they don't reflect the physical time[124] (§3.4.5) or they cannot provide submicrosecond-level precision[138,141,77] (§3.7).

Time-uncertainty bound. As clocks can drift apart over time, time-uncertainty bound (ε) can be calculated as:

*Note that PTP doesn't require this to be the case.

†While the devices may have local queues, the timestamp is marked at dequeue/egress time and is not subject to local queuing delay.

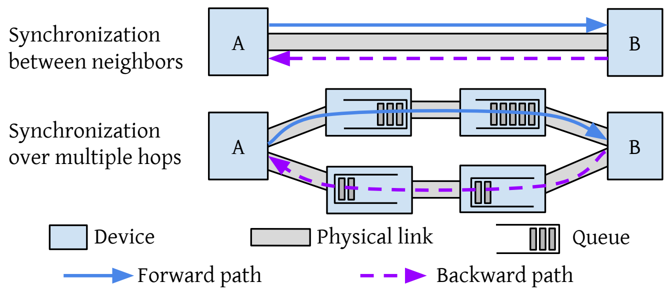

Synchronization between neighbors
A
B
Synchronization over multiple hops
A
B
Device
Physical link
Queue
Forward path
Backward path

Figure 3.2: Benefit of synchronization between neighbors: symmetric forward and backward paths, and no noises from queuing delay.

$$\varepsilon = \varepsilon_{base} + (now - T_{last\_sync}) \times max\_drift\_rate \tag{3.1}$$

ε of a clock exhibits a sawtooth function. T_{last_sync} is the last time when the clock is synchronized to the *root* (not just its direct parent), $now - T_{last\_sync}$ increases with time and goes back to zero after synchronization to the root, and max_drift_rate is a constant representing the maximum possible drift rate between the local clock and the root's clock. The ε_{base} is a small constant (a few nanoseconds) that accounts for other noises (e.g., timestamping errors, bidirectional delay asymmetry of physical links, etc.).

We will show that in the face of failures in production environments, max_drift_rate should be conservatively derived (§3.3.2.1), and $now - T_{last\_sync}$ can be large (§3.3.2.2).

3.3.2 Impact of Failures on ε

We classify failures affecting clock synchronization into three categories and study their impact on ε—failures that induce large frequency variations and need a conservative setting of max_drift_rate, connectivity failures that affect T_{last_sync}, and incorrect behaviors due to broken clocks and message corruption that need to be detected and addressed.

3.3.2.1 Failures that induce large frequency variations

An oscillator's frequency can incur a large variation in the event of sudden temperature or voltage fluctuation. Cooling failures are common and can affect thousands of machines. In an cooling incident that occurred in production recently, it resulted in errors related to clock synchronization in a large fraction of machines (and not just the ones affected by the failure). The temperature variation resulted in oscillator frequency variation to exceed *max_drift_rate* and the operator had to shut down many machines.[‡] Thus, the *max_drift_rate* needs to be set very conservatively (e.g., 200ppm in TrueTime[77]) to tolerate frequency variations under a wide range of temperature (e.g., up to 85 °C) even though in normal cases, temperature variations occur slowly[91]. This entails that in order to keep ε small, we need to reduce $now - T_{last\_sync}$ through frequent messaging—ε of 100ns with *max_drift_rate* of 200ppm needs sync-interval to be <500μs. Software cost of reducing sync-interval to such low values is high—PTP takes one core to process thousands of sync-messages and associated computations per second[2], and Huygens consumes 0.44% CPU for a sync-interval of 2s (which grows proportionally as the interval is reduced). We need hardware support for efficiency (§3.4.1).

3.3.2.2 Connectivity failures

Failures that break the connectivity of the spanning tree also affect ε. For example, if a device or a link in the spanning tree fails, the whole subtree under this device or link loses connectivity to the root[§], until a new spanning tree is reconfigured by the SDN controller. ε grows proportionally to the time it takes for recovery—if it takes 100 ms, ε grows to more than 20μs. Even a distributed

[‡]Normally, after a cooling system failure, operators let machines continue running for 10s of minutes before the recovery of cooling system or a gradual shutdown of machines, because this is usually safe and a sudden total shutdown should be avoided as much as possible.

[§]PTP is configured on a per-port basis (not per-device), so sync-message cannot bypass the failed link or the link associated with the failed device.

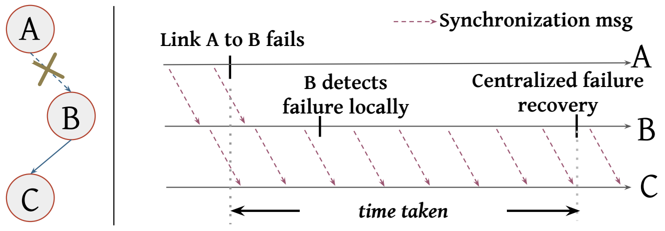


Figure 3.3: Challenge of determining T_{last_sync}. Node C cannot determine if it is synchronized to the root or not, so C has to always set T_{last_sync} conservatively early to account for possible down time.

spanning tree protocol supported by PTP (best master clock algorithm) is slow to converge.

What is worse, is that the inflation of ε is not only for a device affected by the failure at a given time; instead, **almost all devices have to report high ε, all the time** and not only during the failure duration. This is because a device cannot distinguish whether it is affected by a failure or not. Consider a 3-node setup as depicted in Figure 3.3 with A as the root of the spanning tree and B and C as A's child and grandchild respectively. When A fails, B detects the failure but C continues synchronizing to B without noticing the failure. This means at any time, there is no way for C to tell if it is in-sync or not, no matter if there is an actual failure or not and thus, it has to **always** report large ε (i.e., > $20\mu s$) even during normal periods.¶ Another way to look at this is in the context of Equation 3.1, C cannot set T_{last_sync} to the time it receives the last sync-message from its parent T_{last_msg}; instead, for correctness, C has to **always** set $T_{last\_sync} = T_{last\_msg} - T_{recovery}$, where $T_{recovery}$ is the maximum time to recover from any failure that may break its connectivity to the root. All non-direct descendants of the root are affected by this.

There are many possible causes of connectivity failures: besides the common link or switch down, there are incidents that can take down massive (10s to 100s) devices or links, such as failures related to patch panels, link bundles, power domains, or human operations[186,185]. Figure 3.4 shows the time series of link down events in a 1000-machine cluster during a failure incident. The

¶Without changing the PTP standard, B cannot explicitly communicate to C about the failure.

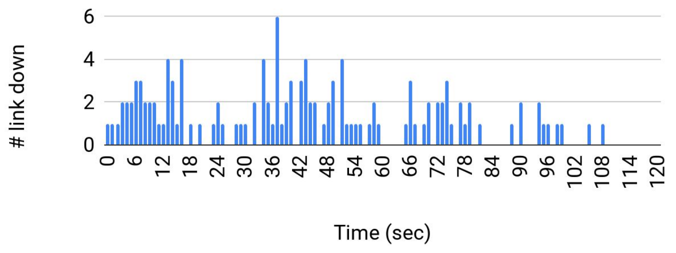


Figure 3.4: Number of link down events per second in a 1000-machine cluster during a near two-minute window of a failure incident.

suspected cause was a software bug related to a patch panel but its impact on device/link failures lasted across nearly two minutes—a total of 133 links go down. Thus, in order to provide small ε, the system must recover from connectivity failures quickly.

3.3.2.3 Broken clocks and message corruption

Clocks may break and stop functioning well resulting in actual drift rate to exceed *max_drift_rate*. While this is rare relative to more severe hardware problems—statistics from production show that broken CPUs are 6 times more likely than broken clocks[77]—they need to be taken care of to provide correct ε to applications. Similarly, sync-message corruption may garble the associated timestamp and affect correctness of reported ε. A fault-tolerant clock synchronization system must detect and address such anomalies.

3.4 Sundial Design and Implementation

Motivated by the discussion above, we identify two key requirements to build a fault-tolerant clock synchronization system for datacenters that achieves performant time-uncertainty bounds. First is a small sync-interval (§3.3.2.1)—this is well served with a hardware implementation to avoid high CPU overhead of receiving and transmitting synchronization messages in software. Second is fast failure recovery so that ε continues to be small even when failures happen (§3.3.2.2). The challenge

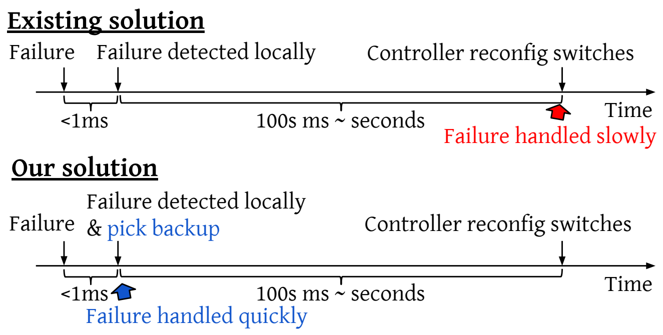


Figure 3.5: Fast failure recovery using precomputed backup plan.

here is that recovering solely via a centralized controller is slow for our target ε requirements. Instead, as we show later, we can recover from most failures locally by adding redundancy to the synchronization graph, where in addition to the primary spanning tree, each device maintains a backup parent, such that it can transition to the backup parent locally upon detecting a failure. As shown in Figure 3.5, this takes the round trip time to the controller and the computation time out of the critical path of failure handling.

Thus, Sundial uses a hardware-software codesign. Figure 3.6 depicts Sundial's framework, which has three main components. Sundial implement the most essential functions of exchanging synchronization messages and detecting failures in hardware such that it can synchronize frequently and quickly detect failures. Sundial relies on software components to take action once a failure is detected, by invoking a failure handler in software which reconfigures the hardware to transition to the backup parent pre-programmed by a centralized controller (also in software). We use the topology in Figure 3.7(a) as a toy example to aid with the discussion in this section with Figure 3.7(b) as an example spanning tree.

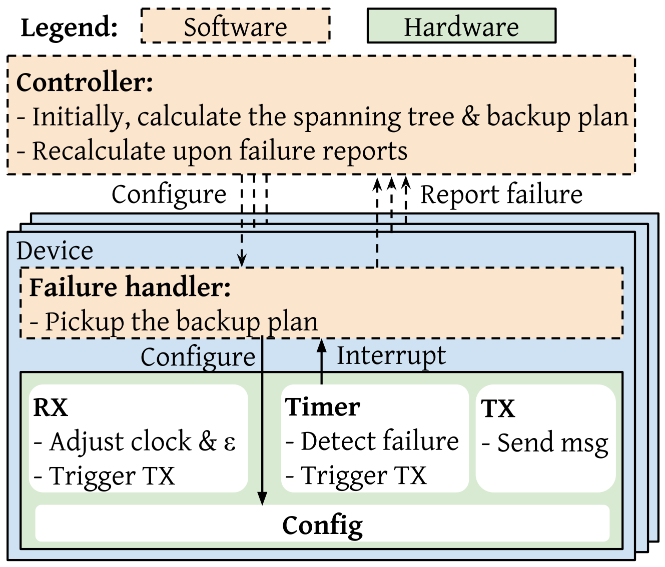


Figure 3.6: Sundial Framework. Solid arrows are the fast local recovery. Dashed arrows are slower but non-critical paths of recovery.

3.4.1 Sundial Hardware Design

Sundial's hardware has three main components. It implements *frequent* transmission of sync-messages in a *synchronous* fashion, i.e., sync-messages are sent downstream only upon their receipt. The hardware is also responsible for detecting failures and triggering software handlers for quick recovery. Finally, the hardware maintains the current value of ε that is exported to applications. We detail out these components below.

3.4.1.1 Frequent synchronous messaging

Sundial's hardware supports frequent message sending to prevent clocks from drifting apart significantly. On the root, this is done via a hardware timer maintaining a counter that increments on every oscillator cycle, and triggers message transmission when the time since last transmission exceeds the configured sync-interval. We configure sync-interval on the root device to be around 100μs. The sync-messages are sent at the highest priority, but the network overhead remains extremely small—a

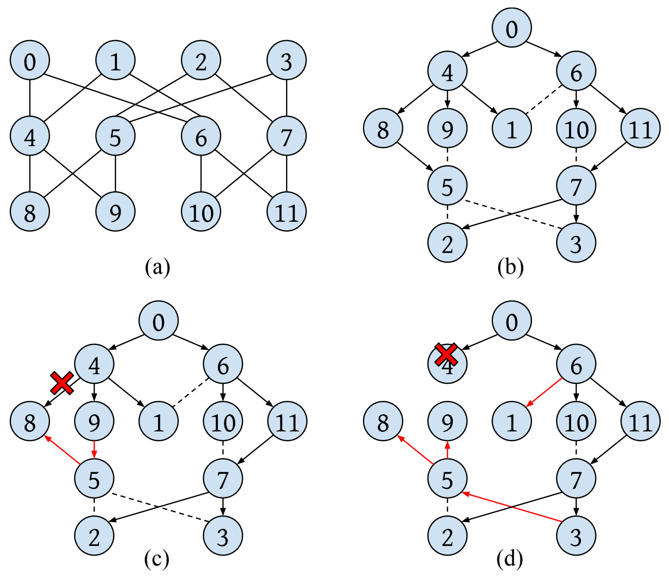


Figure 3.7: Failure cases in a k=4 FatTree. (a) is the raw FatTree. To show the spanning tree clearer, we draw an equivalent topology in (b) and a spanning tree in it. An arrow is from a parent to its child, and a dashed line indicates an edge not used in the spanning tree. (c) shows one way of adjusting the spanning tree when the link between 4 and 8 fails; not only the directly impacted nodes (node 8), but also other nodes (node 5) have to change parent. (d) shows one way of adjustment when node 4 fails; the way node 5 changes its parent (to node 3) is different from the case in (c) (change to node 9).

100-byte packet every 100μs only consumes less than 0.01% bandwidth and adds at most 10ns queuing delay for other traffic.

For non-root devices, a challenge is that an upstream failure can affect all devices in that subtree. Consider the case in Figure 3.7(c), if link 4-to-8 goes down, 8 needs to switch to 5 as its parent, which needs 5 to change its parent as well. A potential solution is explicit notification of failures to other devices, but this has two issues—not only can this be unreliable (since the notification messages may get dropped), it also adds complexity to the hardware. Instead, we solve this via *synchronous messaging* where message transmission is triggered only upon receipt of a message from upstream. In this way, an upstream failure implies that messages stop propagating downstream, and devices can take corrective actions.

3.4.1.2 Fast failure detection

Sundial's hardware uses a timeout to detect if it stops receiving messages indicating an upstream failure. The timeout is set to span multiple sync-intervals, such that occasional message drop or corruption doesn't trigger it. It's implemented using a counter that is incremented on every oscillator cycle, and cleared on receiving a sync-message—once it's exceeded, the hardware issues an interrupt to the software.

To detect broken clocks and message corruption, each device verifies the incoming timestamp value (adjusted for link delay). If the adjusted value lies outside the local ε, the message is marked invalid and doesn't trigger an update and message transmissions. A broken clock can cause continuous invalid messages and thus, we don't reset the timeout counter on their receipt. Once a broken clock is detected, handling broken clocks is done by the failure handler in device software (§3.4.2.2).

3.4.1.3 Time-uncertainty bound calculation

The hardware maintains ε according to Equation 3.1. In our implementation, we configure *max_drift_rate* = 200ppm and ε_{base} = 5ns$\times$*depth* where *depth* is the distance of the device from the root in the tree.

T_{last_sync} is updated when receiving a sync-message. In PTP, T_{last_sync} should be set to earlier than T_{last_msg}. Thanks to synchronous messaging, Sundial sets it to T_{last_msg} since a device stops receiving messages on an upstream failure. This lowers $now - T_{last\_sync}$ which in turn lowers ε.

3.4.2 Sundial Software Design

There are two main components to round out the fault-tolerant design of Sundial–a centralized SDN controller that pre-calculates backup plans and programs them on the devices and a failure handler in device software that quickly moves to the backup when a failure is detected by the hardware.

3.4.2.1 Centralized controller

The centralized controller in Sundial is responsible for computing the primary spanning tree along with the backup plan based on the current topology and configures the devices accordingly. Comparing Figure 3.7(c) and 3.7(d), we see that not all neighbors of a node (e.g., node 5 in the figure) can be the backup parent under different failures. Sundial uses a search algorithm (detailed below) to compute a fault-tolerant backup plan that is *generic* to link, non-root node, root node, and domain failures (which can take down multiple links or devices). We break down this requirement into 5 properties.

Properties of a fault-tolerant backup plan. We briefly introduce the terminology used. The **primary spanning tree** is one that is currently being used to propagate sync-messages. In addition, the **backup plan** consists of a backup-parent for each node/device and a backup root. Terms like parent, edges, paths, and ancestors apply separately to the primary and the backup graph (graph formed by the edges in the backup plan).

(1) No-loop condition: For any primary subtree, the backup edges of nodes in the subtree do not form a loop. This is a necessary and sufficient condition to be generic to any single link failure. The necessity is obvious: if there is a loop, the nodes in the loop do not synchronize to the root after a failure. We prove the sufficiency by induction as follows. Suppose a k-node subtree is affected by a link failure, and the k backup edges do not form a loop (Figure 3.8); the nodes other than the k nodes are unaffected and still form a tree (called the *main tree*). At least one of the k nodes' (say, C) parent is in the main tree; otherwise, all k nodes' parents are in the k nodes, which must form a loop, contradicting the no-loop condition. We can now expand the main tree to include C since C is connected to the main tree via its backup edge. We can then iteratively add the remaining $k-1$ nodes to the main tree.

(2) No-ancestor condition: The backup parent of a node is not its ancestor. This and property (1) to-

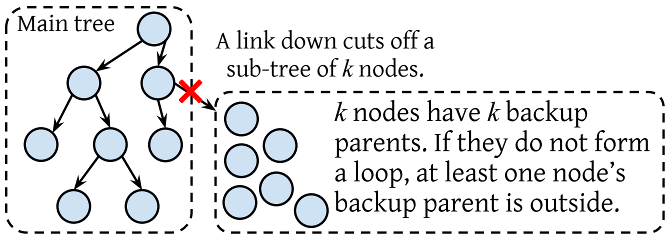

Main tree
A link down cuts off a sub-tree of k nodes.
k nodes have k backup parents. If they do not form a loop, at least one node's backup parent is outside.

Figure 3.8: No-loop condition. It is sufficient to guarantee connectivity after any link failure.

gether ensure that the backup plan is generic to any non-root node failure. Otherwise, if that ancestor fails, that node has no backup parent.

(3) Reachability condition: The backup root must be able to reach all nodes through backup paths. This is necessary and sufficient to be generic to the root failure. When the root fails, all nodes change to their backup parents, and the backup root will become the new root. To synchronize all nodes, they must be reachable from the backup root.

(4) Disjoint-failure-domain condition: Domain failures present a unique challenge, because they may take down multiple devices or links. If a domain failure breaks the connectivity of a device s to the root, s will turn to its backup parent; but if the domain failure also takes down its backup parent, then s cannot recover its connectivity.

The following property solves this problem: *for any node s, there shouldn't be a single domain failure that both breaks s's connectivity to the root and takes down the backup parent or backup edge, unless that failure also takes down the node s.*

Formally, if the set of failure domains that can break s's connectivity to the root[‖] is D_p, the set of failure domains that can take down s's backup parent or backup edge is D_b, and the set of failure domains that s belongs to is D_s, we should have $D_p \cap D_b \subseteq D_s$.

The necessity is obvious. We present the intuition behind the proof of the sufficiency. If a domain failure happens, s has two possibilities: either s's connectivity is unaffected, or s connects to its

[‖]Any device or link failure along the primary path from the root to s can break s's connectivity.

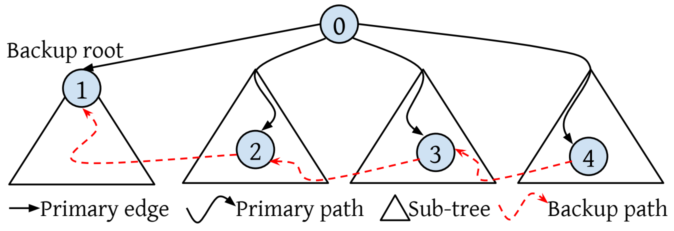


Figure 3.9: Root failure detection. Under any non-root failure, the backup root continues receiving messages, which can be used to distinguish other failures.

backup parent *b*. If it is the latter, then the questions is whether *b* is connected to the root, which also has two possibilities. Doing this recursively, *s* keeps connecting to more nodes along a backup path. The backup path will not go indefinitely due to the no-loop condition, so it finally reaches either an unaffected node or the root.

(5) Root failure detection: Upon root failure, the backup root needs to collect sufficient information to elect itself. Figure 3.9 describes the approach—the backup root is chosen amongst root's children so it has one source of information by itself.

To get information from additional sources, we set up the backup graph to have a backup path from the subtree of another child of the primary root (i.e., the backup path from node 2 to 1 in Figure 3.9). In this way, if the link between the primary root and the backup root fails (i.e., link from 0 to 1), the backup root knows the primary root is still alive because it continues receiving sync-messages that come through the backup path. We can continue this backup path to cross more subtrees of children of the primary root to get additional sources of information (e.g., crossing node 3 and 4 in Figure 3.9).

In this way, as long as the root is alive, the backup root continues receiving sync-messages. Only when the root fails, the backup root stops receiving messages. So the backup root can detect the primary root failure using a second timeout of not receiving messages after it first turns to its backup parent, and it elects itself as the new root after the second timeout.

Putting all 5 properties together. Only non-root nodes have backup parents, so there are *N*-1

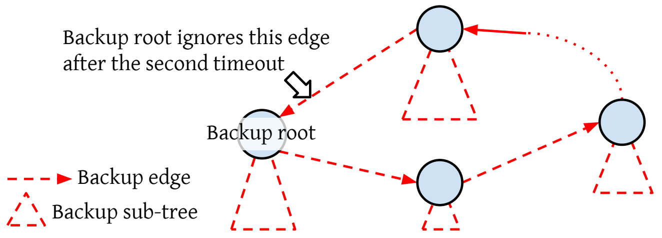


Figure 3.10: Backup Graph. There is exactly one loop with the backup root in it. Each node in the loop is the root of a subtree.

nodes and *N*-1 edges in the backup graph (*N* is the total number of nodes), so there must be exactly one loop[**] in the backup graph, and each node in the loop has a backup subtree (can be a single node) under it (Figure 3.10). With property (3), the backup root must be in the loop, so that the backup root can reach all nodes. The loop should cross multiple primary subtrees of root's children, so it meets both property (1) and property (5) (it delivers multiple sources of primary root's information to the backup root). Lastly, the backup graph should meet properties (2) and (4).

Figure 3.11(a) shows an example of primary tree and backup graph for the topology in Figure 3.7. Note that the computed primary tree is different to support a backup graph. The backup graph has a loop (between node 4 and 8) with the backup root 4 on it; the loop crosses the two primary subtrees of root's children (node 8 is under node 6's primary subtree). To show how property (4) handles domain failures, we add a failure domain that includes both node 11 and 3 (primary and backup parents of node 7 in Figure 3.11(a)). Now in the new backup graph (Figure 3.11(b)), to meet property (4), node 7's backup parent becomes node 2, so that even if both node 3 and 11 go down, node 7 (and other nodes) is still connected.

We want to highlight how the system recovers when the root fails. All backup edges get enabled forming a loop, but no sync-messages flow at this time. At the second timeout, the backup root elects itself and ignores incoming messages, effectively disabling the edge towards it (Figure 3.10). In

[**]A graph with equal numbers of nodes and edges has at least one loop. In addition, if there is more than one loop, then the graph is not fully connected.

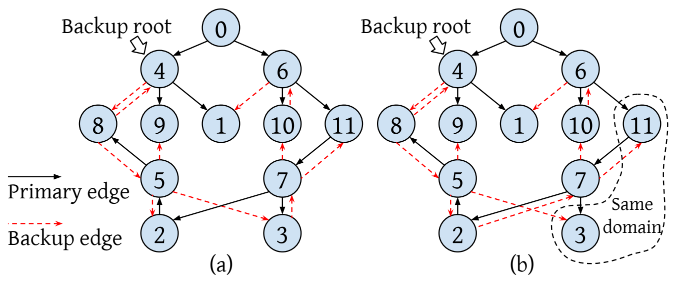


Figure 3.11: (a) A primary tree and a backup graph that meet all properties in Figure 3.7. But if node 3 and 11 are in the same domain, node 7 cannot have them as its primary and backup parents, so its backup parent becomes node 2 in (b).

this way, sync-messages do not loop.

Algorithm for computing backup plan. Sundial uses a search algorithm to calculate the backup plan which includes a primary tree and the backup graph. Note that not every primary tree has a valid backup graph. Thus, the goal is to search for a primary tree and its backup graph together. The search heuristic is based on the *score* of a primary tree—the maximum number of edges in the backup graphs it supports. The corresponding backup graphs are called the largest backup graphs (of the primary tree).

Algorithm 2 describes the algorithm. *pending* is the set of primary trees that are pending to be checked, initialized with a simple BFS (Line 1). After initialization (Line 2), we start the SEARCH function (Line 3) that will return a pair of primary tree and backup graph. In SEARCH, each time, we pick the primary tree p with the highest score (Line 6)—the most promising one—and find the largest backup graphs for it (Line 7). If some backup graph is complete, i.e., every device (including the backup root) has a backup parent, the search successfully returns (Line 8 - 9). Otherwise, we update the best score so far (Line 10), and mutate p (Line 11) to get a new set of primary trees in *pending* and iterate.

In MUTATE, for each backup graph b (Line 14), we try to expand b to include edge $<x, y>$ (Line 15). Since $<x, y>$ is not usable in backup graphs of p[††] (i.e., USABLEINBACKUP($<x, y>, p$) is false),

[††] $<x, y>$ is not usable for sure; otherwise b is not the largest because it can readily include $<x, y>$.

Algorithm 2 Searching for a primary tree and a backup graph.

```
1:  $pending = \{BFS(prim\_root)\}$;
2:  $tested = \emptyset$; $best\_score = 0$;
3:  return SEARCH();
4:  function SEARCH
5:      while $pending$ is not empty do
6:          $p = pending.get\_best()$; $tested \cup = \{p\}$;
7:          $backup\_set = $ FINDLARGESTBACKUP($p$);
8:          if $\exists b \in backup\_set | b$ is complete then
9:              return $p$, $b$;
10:         $best\_score = \max\{best\_score, calc\_score(p)\}$;
11:         MUTATE($p$, $backup\_set$);
12:     return NotFound;
13: procedure MUTATE($p$, $backup\_set$)
14:     for $b$ in $backup\_set$ do
15:         for each $<x, y> \mid x \in b, y \notin b$ do
16:             $new\_prim\_set$=IMPROVE($p$, $<x, y>$, $b$);
17:             for $p'$ in $new\_prim\_set$ do
18:                 if $p' \notin tested$ then
19:                     $pending \cup = \{p'\}$;
20:                     if $calc\_score(p') > best\_score$ then
21:                         return ;
22:     $pending$-=$p$;
```

we IMPROVE p to make $<x, y>$ usable (Line 16). We then add each improved version p' to *pending* if not already tested (Line 19). After all the mutations, p is removed from *pending* (Line 22). We will discuss the optimizations in Line 20 - 21 later.

FINDLARGESTBACKUP and IMPROVE are key functions:

- FINDLARGESTBACKUP conforms to the 5 properties. Properties (2) and (4) decide what edges can be used in backup graphs given a primary tree p, as expressed in function USABLEINBACKUP (Algorithm 3). Properties (1), (3), and (5) decide how the backup graph should look like. We can simply use BFS starting from the backup root (property (3)) to find

Algorithm 3 Check if <x, y> is usable in backup graphs of p.

```
1: function UsableInBackup(<x, y>, p)
2:     return (x is not y's ancestor in p) && (y's ancestor in p and x meet disjoint-failure-domain condition);
```

the tree (property (1)) that is largest, and then enumerate the backup parent for the backup root and see if it forms a loop that meets property (5).

- Improve's goal is to change p to p' so that <x, y> becomes usable (i.e., UsableInBackup(<x, y>, p') is true). It finds the set of p' that meets this goal.

As long as FindLargestBackup and Improve are exhaustive, the search is exhaustive—it will find a solution if one exists. The search process is similar to an algorithm that finds two edge-disjoint spanning trees[167], because our backup graph is composed of a more restricted spanning tree that is edge-disjoint with the primary tree, and an extra edge towards the backup root. The problem seems to be NP-hard although we don't have a proof yet.

In practice, our implementation of Search is extremely fast—it only takes 148ms on average in a simulated Jupiter topology with 88,064 nodes[162] leveraging the following three strategies.

- In Line 20 - 21 of Algorithm 2, we prune enumerations as per Line 14 - 15 as long as we find a p' that is heuristically better than any primary trees so far (including p). This significantly speeds up the search, as we can immediately make progress—after return (Line 21), the search immediately starts a new iteration at Line 6 based on p', which is heuristically better than continuing mutating p. Note this strategy does not miss any primary trees, as the original p remains in *pending*.
- FindLargestBackup only returns one of the largest backup graphs, rather than all of them. This is sufficient as all largest backup graphs for a primary tree connect the same set of nodes and we use this strategy in Algorithm 4.
- Improve just returns the set of p' that *keeps the largest number of b's edges usable*. This tries

Algorithm 4 Finding the largest backup graph of p.

```
1: function FINDLARGESTBACKUP($p$)
2:     $b$=BFS_ForBackup($backup\_root, p$);    ▷ BFS uses USABLEINBACKUP to avoid unusable edges.
3:     Find <$y$, $backup\_root$> where $y \in b$ && USABLEINBACKUP(<$y$, $backup\_root$>, $p$) && the loop crosses multiple subtrees of $prim\_root$ in $p$; Add <$y$, $backup\_root$> to $b$;
4:     return {$b$};
```

to keep as many useful fruits of past iterations as possible, so it speeds up the search. Algorithm 5 is based on this strategy.

These three strategies significantly reduce the computation time per iteration (Line 6 - 11). While the latter two strategies make the search non-exhaustive, all practical datacenter topologies have high redundancy such that in our experiments, we quickly found a backup plan even after injecting 50 successive failures. Also owing to the high redundancy in practical topologies, the number of iterations is small since the initial p already has a very high score, only a few hundreds below the total number of nodes.

Finally, another consequence of high redundancy is that in practice, the search iterates with almost monotonically increasing scores [‡‡], sometimes with jumps of tens or hundreds, reaching the final backup plan in tens of iterations on average.

Mutation for meeting property (5) follows a similar process as MUTATE.

Calculating $\varepsilon_{base,backup}$. When a node turns to its backup parent, its *depth* may change, so we also precompute $\varepsilon_{base,backup}$ to which a device set ε_{base} upon timeout. The exact *depth* is failure-dependent as shown in Figure 3.12.

So we calculate the maximum possible *depth* for each node after *any* failures. A naive approach is to enumerate all possible combinations of failures, which can be slow. Instead, Sundial uses a simple dynamic-programming (DP) based scheme. If a node s turns to its backup parent b, we calculate s's

[‡‡]IMPROVE can easily find paths while keeping all b's edges usable, because of the high redundancy.

Algorithm 5 Changing p to make $<x, y>$ usable and keep as many b's edges usable as possible.

```
1:  function IMPROVE($p$, $<x, y>$, $b$)
2:      if $x$ is $y$'s ancestor in $p$ then
3:          for each edge $<u, v>$ on the path $x \rightsquigarrow y$ in $p$ do
4:              $new\_prim\_set \cup = $ RECONNECT($v, x, p, b$);
5:      if $<x, y>$ fails disjoint-failure-domain condition then
6:          $new\_prim\_set \cup = $ RECONNECT($y, x, p, b$);
7:      return $new\_prim\_set$;
8:  function RECONNECT($v, x, p, b$)
9:      BFS from $v$ along reverse edges, and stops at nodes outside $x$-subtree in $p$, while
    keeping as many $b$'s edges usable as possible. It gives a set of paths $S$=$\{w \rightsquigarrow v | w$ is
    outside $x$-subtree in $p\}$
10:     for $path$ in $S$ do
11:         For each $<i, j> \in path$ set $j$'s parent to $i$ in $p$ to get $p'$;
12:         $new\_prim\_set \cup = \{p'\}$;
13:     return $new\_prim\_set$;
```

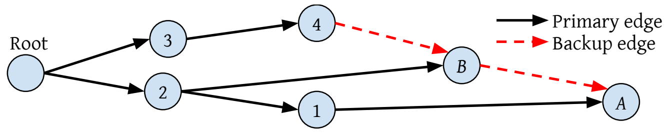


Figure 3.12: Node A's *depth* is dependent on the failure. If node 1 fails, A's *depth* is 3 (A, B, 2, root). But if node 2 fails, A's *depth* is 4 (A, B, 4, 3, root).

maximum possible depth $s.depth_{backup}$:

$$s.depth_{backup} = 1 + \max(b.depth_{primary}, b.depth_{backup})$$

where the max function considers two possible cases: b is unaffected by failures ($b.depth_{primary}$ denotes b's depth in the primary tree, a deterministic value), or affected by failures. $depth_{backup}$ can be calculated top-down.

DP works for all nodes except the nodes on the loop in the backup graph, whose DP calculations inter-depend. But we can easily calculate their maximum possible depths. On an L-node loop, for each node we enumerate all $L + 1$ possible ways it connects to the root (Figure 3.13). So the overall

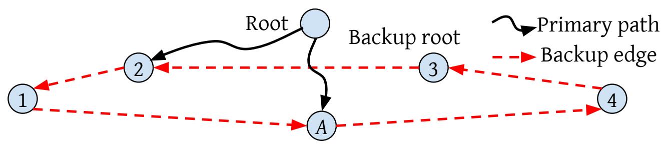


Figure 3.13: A has 6 possible paths to the root, of 3 types. **(1) Backup path**: if the root is down, the backup path (A, 1, 2, 3) is in effect. **(2) Primary path**: when A is unaffected by failures. **(3) Mixed path**: when failures affect A and some other nodes on the loop, A connects to the root first along the loop for one or more hops, and then along the primary path (e.g., A, 1, 2, ..., root). There are 4 possible mixed paths, starting the primary paths from respective node 1, 2, 3, and 4.

time complexity is $O((N-L)+L(L+1))$ for a total of N nodes.

3.4.2.2 Failure handler in the device software

A daemon running in firmware serves as the failure handler and responds to interrupts generated by the hardware once it detects a failure—the hardware is reconfigured to move to the backup parent based on the backup plan and set ε_{base} to $\varepsilon_{base,backup}$. For the backup root, if an interrupt is triggered, the failure handler also continues to monitor incoming sync-messages for the second timeout. At the second timeout, the device sets itself as the primary root.

Handling broken clocks. If a clock is broken[77], it can drift away faster than *max_drift_rate*. In Sundial, we detect such clocks in two steps: (1) detect the existence of a broken clock when receiving an invalid message, and (2) confirm which one is broken. Figure 3.14 illustrates the process. As such, a broken clock is isolated without affecting other clocks.

The failure handler is triggered by a hardware interrupt upon receiving an invalid message to handle broken clocks. For the node with a broken clock, it evicts itself (no longer participates in synchronization). For the node whose parent has a broken clock, it turns to its backup parent.

3.4.3 Implementation

Controller. We implement a module in the network controller. The module registers a function to be called by the controller framework for failure notifications. When notified, this module reads the

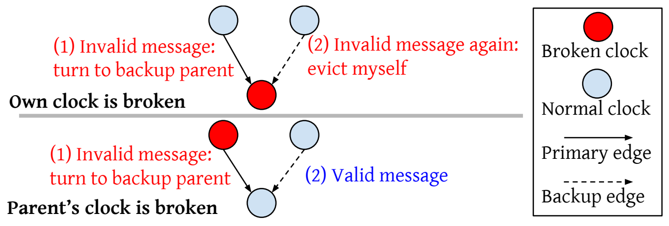


Figure 3.14: Handling a broken clock in two steps. If a node's own clock is broken, the messages from both its primary and backup parents are marked invalid by itself (the timestamp is outside local ε), so it evicts itself. If a node's parent's clock is broken, after receiving an invalid message it turns to its backup parent, and continues synchronization thereafter.

current device/link/port states, and computes a new backup plan. For each device, it compares the existing configuration and the new configuration, and only reconfigures the devices whose configuration changes, through RPC. It also configures the TX side of both primary and backup edges to send sync-messages.

RPC Interface between the Controller and Device Firmware. The controller and the device firmware communicates through RPCs. These RPCs have the following parameters: backup parent, first timeout, and second timeout which are used to configure the device hardware.

Firmware. The RPC handler configures the backup parent, the first timeout, and the second timeout accordingly. The backup parent and the second timeout are maintained in the firmware, and the first timeout is maintained in the hardware registers to enable failure detection in hardware. Only the backup-root has a non-zero value for the second timeout.

The firmware also registers a handler function for the interrupt triggered by the first timeout. This function first reconfigures the hardware to accept sync-messages from the backup parent; then, if the second timeout is non-zero, it waits for the timeout to see if it receives any new sync-messages; if not, it configures the hardware to become the root.

We cannot reveal hardware details due to confidentiality.

3.4.4 Practical Considerations

Concurrent connectivity failures may happen in practice, and may not be recovered by the backup plan, which needs to involve the controller. Sundial maintains the correctness of ε in this case. The only impact is that ε grows larger before being recovered by the controller: if it takes 100ms to recover, ε grows up to 20μs during this time (still $\sim$100ns during normal time). The impact is negligible, because compared to single failures, concurrent failures are already rare, and only a very small subset of them cannot be recovered by the backup plan, as discussed below.

The most commonly seen concurrent failures are caused by a domain failure, which is not an issue because of the disjoint-failure-domain condition of the backup plan (§3.4.2.1).

If cross-domain failures happen, whether they impact Sundial depends on their locations and time proximity. For the nodes whose connectivity is affected, the backup plan is ineffective only if these failures also take down their backup parents/edges (special locations) within a short period of time before the controller recomputes a new backup plan (time proximity). The chance is very small, because cross-domain failures are random in locations and time proximity.

Small window of error before evicting a broken clock. The broken clock detection only happens when messages arrive. There is a small time window between when the failure actually happens and when the next message arrives, during which errors could arise. This can be solved via hardware redundancy—each node physically keeps two clocks, and each clock query reads the two clocks and checks if they match (their time-uncertainty ranges overlap). Once a clock is broken, the next read immediately detects it. Additionally, Sundial prevents this failure to affect other clocks, because its children ignore the invalid messages.

False positives. If a device timeouts without a failure, it will turn to the backup parent. Such false positives are harmless, except extra controller processing. We do not observe false positives in our experiments.

3.4.5 Sundial's Innovation in the Design Space

3.4.5.1 Design space of clock synchronization

At the submicrosecond level, Sundial is the first to support time-uncertainty bound. We identify three key aspects of the design that a clock synchronization system must answer.

1. Type of message: There are multiple options, synchronization messages can either be sent directly with specialized physical layer (PHY) with zero-overhead messages, or at higher layers (L2, L3, L4) with increasing bandwidth overhead and increasing ease of deployment.

2. Noise due to message delay between a pair of clocks. The message delay in the forward and reverse direction may not be equal due to queuing or asymmetric paths. There are three options to deal with such noise: (1) Only synchronize between neighboring devices, such that there is no noise (§3.3.1). (2) Use multiple messages to filter out noise; (3) Tolerate the noise.

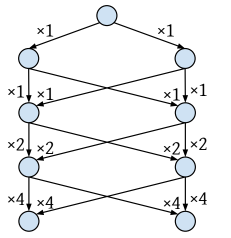
Option (1) is the best if all devices (switches and hosts) can participate. Otherwise, option (2) and (3) face a tradeoff between noise and overhead.

3. Network-wide synchronization structure: three options.

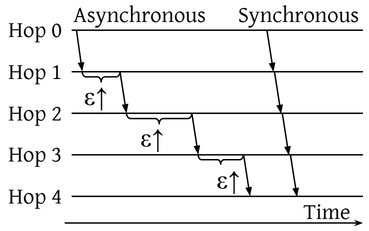
(1) Master clock distributed through a tree. A master clock distributes its time to other clocks through a tree. The master clock can synchronize to the physical time (e.g., via GPS), so that all clocks reflect the physical time.

(2) Master clock distributed through a mesh. Similar to (1), but instead of a tree, each clock receives sync-messages from multiple other clocks, forming a mesh.

(3) No master clock (no physical time). Clocks synchronize independently with each other without regards to a master clock. For example, in DTP[124] each clock follows the fastest of its neighbors. In this option, all clocks converge to a function (e.g., max() in DTP) of all clocks, which has nothing to do with the physical time. This option is worse than (1) and (2) because access to physical time is important for many datacenter applications.



(a) Synchronous messaging: exponential inflation of sync-messages.



(b) Asynchronous messaging has much larger ε.

Figure 3.15: Mesh structure: higher ε due to asynchronous messaging.

Tradeoff between (1) and (2). While (2) is clearly more fault-tolerant, it cannot get ε as low as (1). The reason is that mesh-based solutions cannot use synchronous messaging. As shown in Figure 3.15a, if a clock receives sync-messages from k other clocks, synchronous messaging inflates the number of messages by k per hop, causing exponential inflation. So mesh-based solutions have to use asynchronous messaging, which has much larger ε—as shown in Figure 3.15b, ε increases per hop from the master to other clocks. On the other hand, tree-based solutions can use synchronous messaging, achieving much lower ε. §3.6.2 evaluates this effect.

3.4.5.2 Innovation and stand in the design space

Sundial's innovation is in the third design choice, which exhibits fundamental tradeoff between small ε and fault tolerance. Sundial aims to achieves the best of both worlds, by combining tree and mesh structures: Sundial sends messages through a mesh, such that it still has available edges upon failures; but the effective synchronization only happens over a primary tree, enabling it to use synchronous messaging.

The first two design choices have clear best options, and they are mainly determined by hardware availability. In our implementation, synchronizes neighboring devices at the L2 level as the specialized PHY layer is not available. That said, Sundial can benefit from such a layer if it's available.

Comparison with other schemes is in §3.7.

3.5 Application Access to Synchronized Clocks

In Sundial, the primary mechanism to access synchronized clocks is via hardware Rx/Tx timestamps. Additionally, for applications that want to access host clock directly, Sundial provides local host to NIC/switch clock synchronization via a PI controller.

Access via hardware timestamps. NIC and switch hardware timestamps marked on the packets[142] are the primary access mechanism in Sundial, for which it provides ∼100ns time-uncertainty bound. Applications such as distributed databases that have strict ε requirements rely directly on NIC-Rx-timestamps marked on the last packet in a message to order them to provide consistency properties. Networking stacks such as Snap[137] can provide op-stream interface to applications (preventing out-of-order delivery) and export the NIC timestamps. Telemetry and congestion control applications also rely directly on NIC timestamps to measure one-way delays.

Host clock synchronization. In addition, we synchronize the local host clock to the NIC/switch clock for applications that want to directly read the host clock (and don't require strict guarantees on ε). We use a Proportional-Integral controller based on clock-skew between the host and NIC clocks as depicted in Figure 3.16. We measure the offset, $o(t)$ and skew, $s(t)$, every T time-units (we use T=10ms) and apply the rate adjustment to the host clock to tick faster or slower. The constants P and I need to be tuned in production. One challenge is that the two clock-measurements are subject to local delays such as PCIe jitter and we use linear regression to filter the noise out.

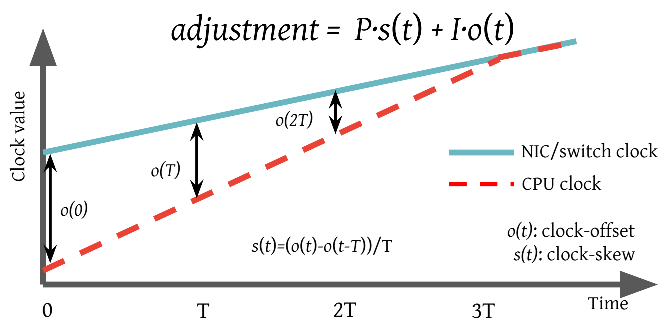


Figure 3.16: PI controller based on clock-skew; offset and skew are measured periodically and an adjustment is computed using suitable P and I constants.

3.6 Evaluation

Through experiments in a >500-machine testbed-prototype (§3.6.1) and through large-scale simulations (§3.6.2), we show that Sundial's time-uncertainty bound is $\sim$100ns under different types of failures, and discuss application improvements enabled by Sundial in §3.6.3.

3.6.1 Time-uncertainty Bound (ε) in Testbed

3.6.1.1 Methodology

Testbed. The testbed consists of 23 pods, 276 switches and 552 servers. A pod including 12 switches and 24 servers acts as a failure domain. The oscillators used in the hardware have a frequency specification of $\pm$100ppm. The depth of the base spanning tree in the topology is 5.

Schemes for comparison. We compare Sundial with recent submicrosecond-level clock synchronization schemes: PTP[20], Huygens[91], and DTP[124]. While they do not consider time-uncertainty bound (ε) and how it is reported to applications, we augment the designs to provide ε, according to Equation 3.1 in §3.3.1 and describe them below.

***Sundial*:** We set the sync-interval to 90μs.[§§] The timeout is 185μs (>2$\times$sync-interval). The second

[§§] 90μs is just enough for $\sim$100ns ε, although lower ε is achievable.

timeout for the backup root to elect itself is set to 180μs (185+180>4×sync-interval). The backup plan has a maximum $depth_{backup}$ of 6.

PTP+ε: PTP is the most common submicrosecond-level synchronization protocol with a default sync-interval of two seconds. To add ε, we set ε_{base} to 5ns×$depth$, and max_drift_rate to 200ppm. T_{last_sync} is updated as follows—for root's children, we set $T_{last\_sync} = T_{last\_msg}$; but for other descendants, we set $T_{last\_sync} = T_{last\_msg} - T_{recovery}$ to account for possible out-of-sync duration caused by remote connectivity failures that are oblivious to them (§3.3.2.2). We set $T_{recovery}$ to 2s, since it takes 2s to recover from failure.¶¶

PTP+DTP+ε: What if we could set lower sync-interval in PTP+ε? We evaluate another scheme that leverages DTP—DTP allows very small sync-interval (a few microseconds) with low bandwidth overhead by modifying the physical layer protocol. Since DTP requires hardware support, we emulate it in our testbed by setting 5μs sync-interval (much smaller than 90μs).‖‖ All devices that are not direct children of the root set $T_{recovery}$=100ms, where 100ms is the typical connectivity failure recovery time measured from datacenters.* * *

Huygens+ε: Huygens gathers network-wide sync-messages during each 2-second sync-interval, and uses machine learning to decide the best adjustment for each device at the beginning of the next sync-interval. While we do not have its implementation, we report the best possible ε it can achieve. Specifically, we assume it is not affected by connectivity failures because of its use of network-wide information, so T_{last_sync} is set to the beginning time of each sync-interval (without minus $T_{recovery}$).

¶¶In favor of low ε, $T_{recovery} = 2$ seconds is already a very optimistic setting for PTP+ε, because recovery may take longer if the next sync-message is also dropped by another failure that just happens at that time. Setting $T_{recovery}$ larger results in even higher ε. But we show that even with this optimistic setting, PTP+ε still has much higher ε than Sundial.

‖‖This is sufficient to show the improvement of ε, even though we don't have the physical layer protocol to keep the bandwidth overhead low.

* * *This is already friendly to PTP+DTP+ε because to *guarantee* correct ε, $T_{recovery}$ should be the maximum recovery time, which is several seconds.

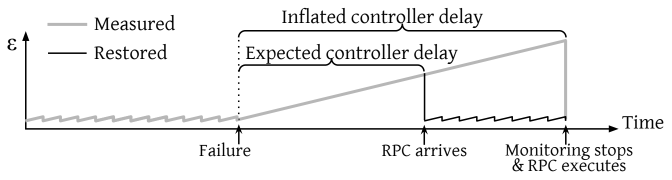

Measured
Restored
ε
Inflated controller delay
Expected controller delay
Time
Failure
RPC arrives
Monitoring stops & RPC executes

Figure 3.17: Restoration of ε under inflated controller delay.

We also assume it can filter out delay noises entirely and optimistically set ε_{base} to 0.

Failure injection. We evaluate the impact of failures on ε in Sundial and above schemes by injecting link failures, non-root device failures, root failures, and domain failures (where multiple devices can go down).

Metrics and measurement approach. We measure ε on every device by running a daemon in the firmware to read ε every 10μs. After a failure, the controller sends an RPC to configure the devices for recovery. The frequent monitoring interferes with processing RPCs that are sent by the controller in the event of failures. As a workaround, we set a stop time which allows the controller RPC to execute after the monitoring stops. In this way, the monitoring tells us which devices are affected by failures and their ε. But it also inflates the controller delay, which is unfair to other schemes as they heavily rely on the controller for failure recovery. With knowledge of the expected controller delay, we can easily restore the expected ε based on the measured ε (Figure 3.17), because ε's behavior is deterministic during failures recovery: ε keeps increasing, and goes back to normal when the failure is recovered. To get the expected controller delay, we use its lower bound, the delay on the controller (without network delay), which is more friendly to schemes other than Sundial.

3.6.1.2 ε distribution without failures

Figure 3.18 shows the distribution of ε over all devices under different schemes. In Sundial, $\varepsilon \leq 43$ns, which matches the calculated value—the deepest device in the tree has ε_{base} of 25ns, and 90μs sync-interval leads to an additional 18ns.

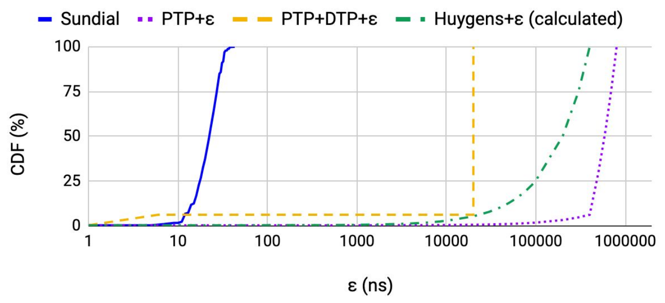


Figure 3.18: CDF of ε measured across devices without failures.

In contrast, all other schemes have a much higher ε. In PTP+ε and PTP+DTP+ε, devices that do not directly synchronize to the root have to set T_{last_sync} earlier than T_{last_msg} by 2s and 100ms respectively, to account for possible failure-induced out-of-sync periods, so their ε can go up to 800μs and 20μs respectively during a sync-interval. Devices directly synchronizing to the root can set T_{last_sync} to T_{last_msg} and achieve lower ε. So their ε increases from 5ns (1-hop ε_{base}) to $\sim$400μs and 6ns respectively (2s and 5μs sync-intervals lead to 400μs and 1ns additional ε respectively at the end of each sync-interval). For these devices ($\sim$6.3% of all), PTP+DTP+ε's low ε shows the benefit of extremely small sync-interval when failure is not a concern. Note that if available, Sundial can also benefit from DTP's physical layer design to futher reduce sync-interval. In Huygens+ε, during each 2s interval, ε increases from 0 to 400μs. Reducing sync-interval comes with CPU cost (Huygens already consumes 0.44% CPU of the whole cluster). However, even if the sync-interval was halved, ε is still 3 orders of magnitude higher than Sundial's.

3.6.1.3 ε distribution during failures

To understand the behavior under failures, we inject 50 random failures over a course of 6 minutes including 24 single link failures, 23 non-root single device failures, 2 domain failures and 1 root failure.

Figure 3.19 shows the time series of ε of a device affected by a link failure. In Sundial, ε is saw-

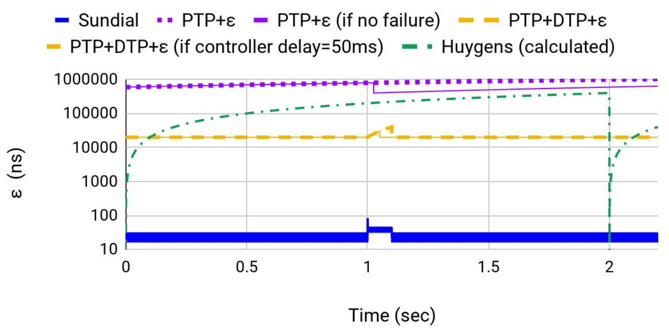


Figure 3.19: Time series of ε of a device affected by a link failure. The failure happens at 1s and the controller reacts to it near 1.1s.

tooth between 15ns and 33ns during normal time, because this device has a depth of 3 in the tree.[† † †] When the link failure happens, ε increases to a maximum of 84ns and goes down in just 270μs (after the 185μs timeout, the next message is at 270μs). After that, ε is sawtooth between 30ns and 48ns, because its ε_{base} is set to $\varepsilon_{base,backup}$ by the local recovery, which is 30ns ($depth_{backup}$=6). Once the controller reconfigures the spanning tree, ε goes back to between 15ns and 33ns because its depth is 3. In PTP+ε, since the sync-message is dropped due to this failure, ε continues to increase for the next 2 seconds. Even if the sync-message was not dropped, ε for PTP+ε (w/o failure) remains high. PTP+DTP+ε's ε increases to 40μs and recovers to 20μs when the controller recovers the connectivity. However, even if the controller delay was lower (50ms), it only reduces the peak ε to 30μs, but the normal ε is still around 20μs. Huygens+ε is not affected by failures, but its ε is normally very large (200μs at median and up to 400μs).

The behavior is similar under other failures—ε depends on the recovery time. For PTP+ε and PTP+DTP+ε, the recovery time depends on how long it takes for the controller to recover from it. For Sundial, the recovery time is much smaller as it's local. Any non-root failure recovery time is around 270μs, as is the case in Figure 3.19. The root failure takes slightly longer to recover from (365μs after the two timeouts) and ε increases to up to 103ns. The devices at different levels in the

[† † †] Figure 3.21 shows the behavior at smaller timescales.

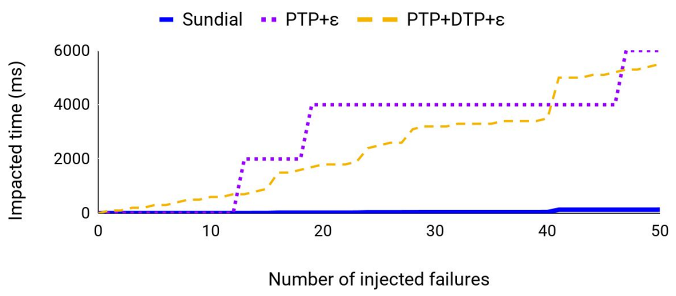


Figure 3.20: Blast radius of failures under different schemes. Impacted device time is the summation of per-device impacted time—duration when a device stops receiving sync-messages—over all devices .

tree have slightly different ε (discussed in §3.6.1.4).

We now study the spatial and temporal impact range (blast radius) of failures. Figure 3.20 shows that Sundial's blast radius is very small. Even after 50 failures, the total impacted time summarized over all devices is only 131ms. The most significant jump happens when the root fails (40-th failure). PTP+ε and PTP+DTP+ε's blast radius is much higher owing to their longer recovery time. Note that more devices are affected by failures under Sundial (401 in total) than under PTP+ε (3 in total) and PTP+DTP+ε (55 in total) as Sundial's backup-plan-based recovery can affect remote devices as well (those under the subtree of the failure). Even then the total impacted time for Sundial remains significantly smaller.

PTP+ε exhibits a step function because only failures occuring close to sync-interval boundaries affect it as the sync-interval of 2s is longer than the time to recover in most cases. The impact, however, is larger than in other schemes because it takes 2s for the next sync-message. PTP+DTP+ε's sync-interval is only 5μs and thus, every failure affects it. While Huygens+ε is not affected by connectivity failures, its ε remains high as shown before.

3.6.1.4 Microbenchmarks

How Sundial's different techniques improve ε. We zoom into details of how each technique improves ε. Specifically, starting with PTP+ε, we add (1) frequent sync-messages, (2) synchronous messaging, and (3) backup plan to it one by one, resulting in four schemes: PTP+ε, PTP+ε+freq_msg, PTP+ε+freq_msg+sync_msging, and Sundial itself.

Figure 3.21 shows the time series of ε during a link failure. Frequent sync-messages improve ε by an order of magnitude. Synchronous messaging further reduces ε during normal time as it helps each device detect connectivity failures: as long as a device receives a sync-message, it is connected to the root, so T_{last_sync} can be safely set to T_{last_msg}. Finally, adding the backup plan significantly speeds up the failure recovery—ε only increases for 270μs to a maximum of 84ns before the backup plan is activated, two orders of magnitude lower.

To show how Sundial's backup plan handles domain failures, we also run Sundial without considering domain failures (called Sundial w/o domain). We find that if a domain failure simultaneously takes down both the primary and backup parents of a device, the device's ε is like PTP+ε+freq_msg+sync_msging in Figure 3.21. This is expected because a down backup parent is equivalent to no backup parent. But if the failure domain is considered in the backup plan, ε is similar to Sundial in Figure 3.21, because the backup plan guarantees that no device loses both its primary and backup parents due to this domain failure. We also try another domain failure, which gradually takes down the primary and backup parents of a device, mimicing the domain failure that gradually takes down multiple devices or links (e.g., Figure 3.4). The result is similar.

Distribution of ε at different levels of the tree. We plot the maximum ε across devices at different depths, under different scenarios, shown in Figure 3.22. Root's ε is always 0. ε increases linearly with depth, which is expected as each level increments ε_{base} by 5ns.

3.6.2 Large-scale Simulations

We compare Sundial vs Marzullo's algorithm[138], an agreement algorithm for fault-tolerant clock-synchronization which is used by NTP[141] and TrueTime[77]. Marzullo's algorithm also introduces

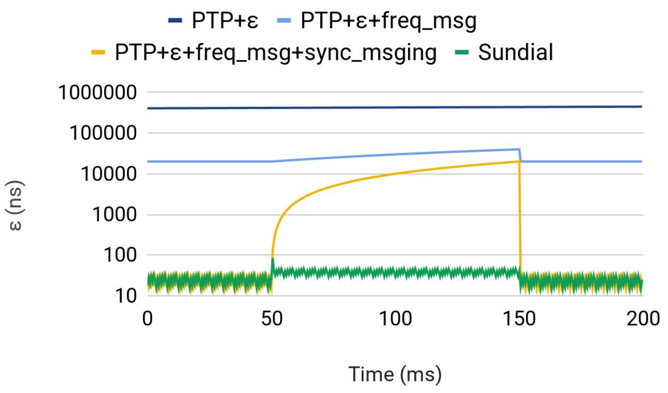


Figure 3.21: A link failure happens at 50 ms. The controller reacts to the failure at around 150 ms.

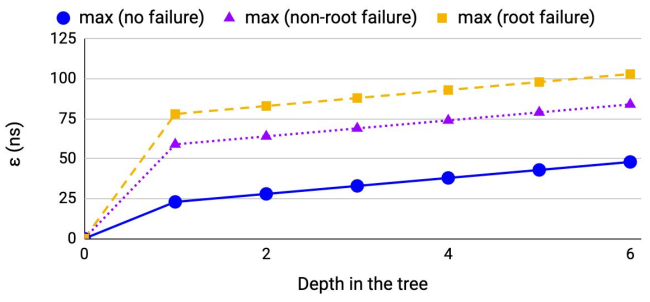


Figure 3.22: Distribution of ε at different levels in the tree.

time-uncertainty bound (ε) (called as error-bound in the original version). Since it is not supported in hardware due to its complexity, we use large scale simulations to demonstrate the performance characteristics.

Marzullo's algorithm synchronizes clocks through a mesh, so it can tolerate connectivity failures but has higher ε (§3.4.5). To reconcile the different time values and ε from multiple clocks, each node does intersection of time-uncertainty ranges of different clocks as the correct time should be within all ranges. A set of master clocks (1 or more masters synchronized via GPS) serve as the source of synchronization, whose ε is always close to zero. Broken clocks can also be detected when the intersection result is empty. We simulate in a Jupiter topology[162] with 88,064 devices, where each node sends sync-messages to all its neighbors to maximize the tolerance to failures. We set 2

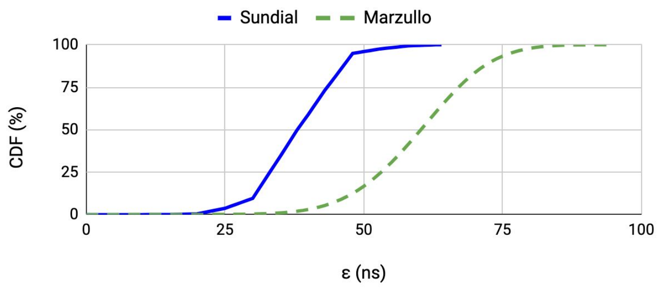


Figure 3.23: CDF of ε during normal time in Jupiter in simulation.

masters to tolerate master failures. The sync-interval is 90μs, same as Sundial.

Figure 3.23 shows that during the normal time, Sundial has smaller ε than Marzullo's algorithm. Under failures, Marzullo's algorithm's ε is affected insignificantly. For Sundial, ε increases during failure recovery; the largest ε is 178ns, which is under the root failure.

3.6.3 Application Performance Improvement

Distributed transactional system. We evaluate the impact of smaller time-uncertainty bound using a load-test provided to us by Spanner team[77]. We run the load-test inside a datacenter. The load-test does 4KB transactions and we measure commit-wait gap—time to wait out time-uncertainty before committing the transaction. Results are in Table 3.1 where we show that our system improves performance by 3-4× not only in the median but also at the 99-th percentile.

Table 3.1: Sundial improves commit-wait latency by 3-4× for Spanner running inside a datacenter.

	Baseline	With Sundial
Median	211μs	49μs
99-%ile	784μs	238μs

Congestion Control. Delay-based congestion control such as Swift[118] is widely used in datacenters relying on end-to-end RTT measurements to control sending rate. A key challenge with such schemes is how to differentiate between forward and reverse-path congestion. As an example, congestion in the reverse path can also inflate RTT causing a sender to slow down even though there is no congestion in the forward path.[‡‡‡] Synchronized clocks solve this problem as they enable the measurement of one-way delay (OWD) which can pinpoint the direction in which congestion is occurring.

We perform a microbenchmark with 3 servers—A, B and C. First, we only send traffic from A to B which achieves line-rate throughput. Next, we introduce reverse-path congestion by adding traffic from B and C to A. In Table 3.2, we observe A's throughput goes down to 50Gbps even though there was no congestion in the forward path. Replacing RTT with OWD as measured using Sundial resolves this completely and A continues to send at line rate.

Table 3.2: Using one-way delay (OWD) improves throughput in the presence of reverse-path congestion.

	RTT	OWD
No reverse congestion	80.1Gbps	80.5Gbps
Reverse congestion	50.5Gbps	80.9Gbps

3.7 Related Work

Other clock synchronization schemes. Table 3.3 compares state-of-the-art solutions, in the design space outlined in §3.4.5.

DTP[124] is the one introducing the special PHY to achieve zero bandwidth overhead of sync-messages. If this modified PHY can be standardized and productionized in the future, Sundial can readily benefit from it to have lower sync-interval and ε. Separately, DTP does not reflect physical time since it has no master clock.

[‡‡‡]While prioritizing the ACK may solve the problem, it is impractical in production because of two reasons. (1) Network priorities are typically tied to business priorities; and we simply cannot send ACKs for lower business priority traffic on a higher network priority. (2) Sending ACKs on a higher network priority precludes ACK piggybacking on data packets, thereby increasing the packets-per-second to process. This is especially detrimental for CPU-efficient networking stacks such as PonyExpress in Snap[137].

Table 3.3: Design choices of state-of-the-art clock synchronization schemes. Italic options are the best.

	DTP[124]	**Huygens**[91]	**Marzullo**[138]	**PTP boundary clock**[20]	**Sundial**
Message type	*Special PHY*	L3	Unspecified	L2	L2
Dealing with delay noises	*Neighbor*	Multi. msg	Unspecified	*Neighbor*	*Neighbor*
Synchronization structure	No master	Master, mesh	Master, mesh	Master, tree	*Master, mesh+tree*
Support time-uncertainty bound	No	No	*Yes*	No	*Yes*

Huygens[91] does not synchronize switches, so it uses multiple messages between each pair to filter out noises. As a result, Huygens' sync-interval is limited, so it cannot achieve tight ε. We believe Huygens is more suitable for environments where switches are out of control. Huygens also does not consider ε; if added, ε will be large because of the large sync-interval. While it assumes clocks drift slowly during normal time, it cannot set a small *max_drift_rate* because of failures in §3.3.2.1; otherwise it risks datacenter-wide application-level errors (e.g., inconsistent transactions), which is unacceptable.

Marzullo's algorithm[138] sends messages through a mesh, so its ε is not tight. It is the first to introduce ε. PTP boundary clock[20] is based on tree, so it is not fault-tolerant.

Other solutions are too expensive (e.g., GPS[125]), too complex[123,121,158] or don't provide physical time[178,163,157].

Fault tolerance in other systems. In distributed systems and networking, fault tolerance is provided through redundancy[122,28,55,162,94,185]. However, Sundial's backup plan cannot be chosen arbitrarily and needs to satisfy a set of properties (§3.4.2.1) to be generic to different types of failures.

Ethernet uses spanning tree protocols[17,100] that can recompute a spanning tree in a distributed fashion after a failure happens, but they usually take 100s of milliseconds to a few seconds to converge[100]. In contrast, Sundial proactively computes a backup plan aiding fast local recovery.

3.8 Summary

Sundial is the first submicrosecond-level clock synchronization system that is resilient to failures. It uses hardware-software codesign to quickly detect failures and recover from them, enabling tight bounds on time-uncertainty. Our evaluation shows that Sundial provides $\sim$100ns time-uncertainty bound under different types of failures. We also show performance improvements in Spanner and in Swift brought by Sundial.

4

FlowRadar: Precise Per-flow Telemetry

4.1 Introduction

NetFlow[8] is a widely used monitoring tool for over 20 years, which records the flows (e.g., source IP, destination IP, source port, destination port, and protocol) and their properties (e.g., packet counters, and the flow starting and finish times). When a flow finishes after the inactive timeout, NetFlow exports the corresponding flow records to a remote collector. NetFlow has been used for a variety of monitoring applications such as accounting network usage, capacity planning, troubleshooting, and attack detection.

Despite its wide applications, the key problem to implement NetFlow in hardware is how to maintain an active working set of flows using a data structure with low time and space complexity. We need to handle collisions during flow insertion and remove old flows to make room for new ones. These tasks are challenging given the limited per-packet processing time at merchant silicon.

To handle this challenge, today's NetFlow is implemented in two ways: (1) Using complex custom silicon that is only available at high-end routers, which is too expensive for datacenters; (2) Using software to count sampled packets from hardware, which takes too much CPU resources at switches. Because of the lack of usable NetFlow in datacenters, operators have to mirror packets based on sampling or matching rules and analyze these packets in a remote collector[98,177,190,153]. It is impossible to mirror all the packets because it takes too much bandwidth to mirror the traffic, and too many storage and computing resources at the remote collector to analyze every packet. (§ 4.2)

However, in datacenters, there is an increasing need to have visibility of the counters for all the flows all the time. We need to cover all the flows to capture those transient loops, blackholes, and switch faults that only happen to a few flows in the Network and to perform fine-grained traffic analysis (e.g., anomaly detection). We need to cover these flows all the time to identify transient losses, bursts, and attacks in a timely fashion. (§ 4.3)

In this paper, we propose FlowRadar, which keeps counters for all the flows with low memory overhead and exports the flow counters in short time scales (e.g., 10ms). The key design of FlowRadar is to identify the best division of labor between cheap switches with limited per-packet processing time and the remote collector with plenty of computing resources. We introduce *encoded flowsets* that only require simple constant-time instructions for each packet and thus are easy to implement with merchant silicon at cheap switches. We then decode these flowsets and perform network-wide analysis across time and switches all at the remote collector. We make the following key contributions in building FlowRadar:

Capture encoded flow counters with constant time for each packet at switches: We introduce encoded flowsets, which is an array of cells that encode the flows (5 tuples) and their counters. Encoded flowsets ensure constant per-packet processing time by *embracing* rather than handling hash collisions. It maps one flow to many cells, allows flows to collide in one cell, but ensure each cell has constant memory usage. Since encoded flowsets are small, we can afford to periodically export the entire flowsets to the remote collector in short time scales. Our encoded flowset data structure is an extension of Invertible Bloom filter Lookup Table (IBLT), but provides better support for counter updates.

Network-wide decoding and analysis at a remote collector: While each switch independently encodes the flows and counters, we observe that most flows traverse multiple switches. By leveraging the redundancies across switches, we make the encoded flowsets more compact. We then propose a network-wide decoding scheme to decode the flows and counters across switches. With the network-wide decoding, our encoded flowsets can reduce the amount of memory needed to track 100K flows by 5.6% compared to an ideal (and hence impractical) implementation of NetFlow with *perfect hashing* (i.e., no collisions) while providing 99% decoding success rate*. (§ 4.4 and 4.5)

FlowRadar can support a wide range of monitoring applications including both existing monitoring applications on NetFlow, and new ones that require monitoring all the flows all the time. As demonstrations, we design and build two systems on top of FlowRadar: one that detects transient loops and blackholes using a network-wide flow analysis and another that provides a per-flow loss map using temporal analysis (§ 4.6).

We discuss the implementation issues in § 4.7, compare with related work in § 4.8, and conclude in § 4.9.

*The decode success rate is defined as the probability of successfully decoding all the flows.

Table 4.1: Comparing FlowRadar with hardware-based NetFlow in custom silicon, sampling-based software NetFlow in merchant silicon, and sFlow/EverFlow

	Hardware-based NetFlow in custom silicon	**Software-based NetFlow (Sampled)**	**sFlow[177], EverFlow[190]**	**FlowRadar**
Division of labor				
State in switch hardware	active working set of flows	none	none	encoded flows records
State in switch software	none (or some active flows)	active working set of flows	none	none
Data exported to collector	flow records after finish	flow records after finish	selected pkts&timestamps	periodic encoded records
Coverage of traffic				
Temporal coverage	No	No	No (if select control pkts)	**Yes (milliseconds)**
Flow coverage	**All** or sampled packets	sampled packets	sampled/selected packets	**All**

4.2 Motivation

In this section, we discuss the key challenges of implementing NetFlow. We then describe three alternative monitoring solutions (Table 4.1): NetFlow in high-end routers with custom silicon, NetFlow in cheap switches with merchant silicon, and selective mirroring. To address the limitations of these approaches, we present FlowRadar architecture, which identifies a good division of labor between the switches and the remote collector.

4.2.1 Key Challenges of Supporting NetFlow

Since NetFlow has been developed for over 20 years, there have been many implementations and extensions of NetFlow in routers and switches. We cannot capture all the NetFlow solutions here, and in fact many solutions are proprietary information. Instead, we focus on the basic function of NetFlow: storing the flow fields (e.g., 5 tuples) and the records (e.g., packet counter, flow starting time, the time that the flow is last seen, etc.) in a hash table. The key challenge is how to maintain the active working set of flows in the hash table given the limited packet processing time.

Maintain the active working set of flows: There are two key tasks in maintaining the active working set of flows:

(1) How to handle hash collisions during flow insertion? When we insert a new flow, it may experi-

ence collisions with existing flows. One solution is to store multiple flows in each cell in the hash table to reduce the chances of overflow (e.g., d-left hashing[62,174]), which requires atomic many-byte memory accesses. Another solution to move existing flows around to make room for new flows (e.g., Cuckoo hashing[150]), which requires multiple, non-constant memory accesses per packet in the worst case. Both are very challenging to implement on merchant silicon with high line rate. The detailed challenges are discussed in § 4.8.

(2) How to remove an old flow? We need to periodically remove old flows to make room for new flows in the hash table. If a TCP flow receives a FIN, we can remove it from the table. However, in datacenters there are many persistent connections reused by multiple requests/responses or messages. To identify idle flows, NetFlow keeps the time a flow is last seen and periodically scan the entire hash table to check the inactive time of each flow. If a flow is inactive for more than the inactive timeout, NetFlow removes the flow and exports its counters. The inactive timeout can only be set between 10 and 600 seconds with a default value of 15 seconds[7]. When the hash table is large, it takes a significant time and switch CPU resources to scan the table and clean up the table entries.

Limited per-packet processing time at merchant silicon: It is hard to maintain the active working set of flows at the merchant silicon—the commodity switch design in datacenters. The key constraint of the merchant silicon is the limited time we can spend on each packet. Suppose a switch has 40Gbps per port, which means 12ns per packet processing time for 64-byte packets[†]. Let's assume the entire 12ns can be dedicated to NetFlow by performing perfect packet pipelining and allocating all other packet processing functions (packet header parsing, Layer 2/3 forwarding, ACLs, etc.) to other stages. Yet inside NetFlow, one needs to calculate the hash functions, look up SRAM, run a few ALU operations, and write back to the SRAM. Even with on-chip SRAM which has roughly 1ns access time, to finish all these actions in 12ns is still a challenge. (Similar arguments are made

[†]This becomes worse when datacenters move to 100Gbps.

in[86] about the difficulties of implementing data streaming at routers.)

4.2.2 Alternative Monitoring Solutions

Due to the limited per-packet time in merchant silicon, one cannot process complex and non-constant time insertion and deletion actions as required in NetFlow. Therefore, there are three alternatives (Table 4.1):

Hardware-based NetFlow in custom silicon: One solution is to design custom silicon to maintain the active working set of flows in switch hardware. We can cache popular flow entries in on-chip SRAM, but the rest in off-chip SRAM or DRAM. We can also combine SRAM with expensive and power-hungry TCAM to support parallel lookup. Even with the expensive custom silicon, the test of Cisco high-end routers (Catalyst series)[76,24] shows that there is still around 16% switch CPU overhead for storing 65K flow entries in hardware. Cisco highly recommends NetFlow users to choose sampling to reduce the NetFlow overhead on these routers[76].

Sampled software-based NetFlow in merchant silicon: Another solution is to sample packets and mirror them to the switch software, and maintain the active working set of flows in software. This solution works with cheap merchant silicon, but takes even more CPU overhead than hardware-based NetFlow in high-end routers. To reduce the switch CPU overhead of NetFlow and avoid interrupting other processes (e.g., OSPF, rule updates) in CPU, operators have to set sampling rate low enough (e.g., down to 1 in 4K). With such low sampling rate, operators cannot use NetFlow for fine-grained traffic analysis (e.g., anomaly detection) or capturing those events that only happen to some flows (e.g., transient loops or blackholes).

Selective mirroring (sFlow[177], EverFlow[190]): The final solution datacenter operators take today is to only sample packets or select packets based on match-action rules, and then mirror these packets to a remote collector. The remote collector extracts per flow information and performs detailed

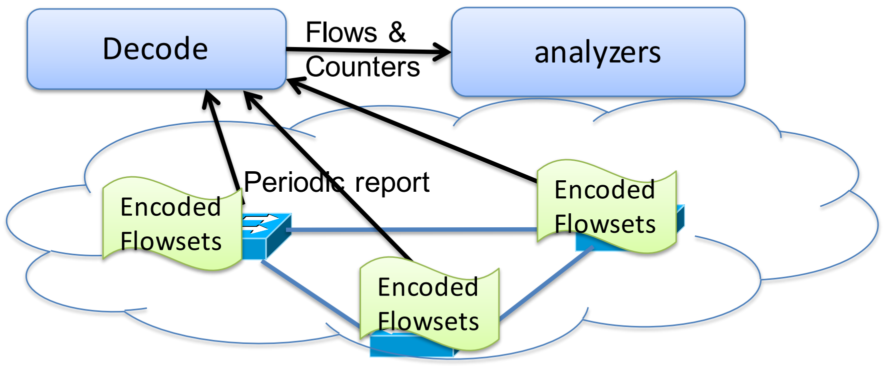


Figure 4.1: FlowRadar architecture

analysis. This solution works with existing merchant silicon, and best leverages the computing resources in the cloud. However, it takes too much bandwidth overhead to transfer all the packets to the collector and too much storage and computing overhead at the collector[190]. Therefore, operators can only get a partial view from the selected packets.

4.2.3 FlowRadar Architecture

Instead of falling back to sampling in existing monitoring solutions, we aim at providing full visibility to all the flows all the time (see example use cases in § 4.3). To achieve this, we propose to best leverage the capabilities at both the merchant silicon at switches and the computing power at the remote collector (Figure 4.1).

Capturing encoded flow counters at switches: FlowRadar chooses to encode flows and their counters into small fixed memory size that can be implemented in merchant silicon with constant flow insertion time. In this way, we can afford to capture all the flows without sampling, and periodically export these encoded flow counters to the remote collector in short time scales.

Decoding and analyzing flow counters at a remote collector: Given the encoded flows and counters exported from many switches, we can leverage the computing power at the remote collector to perform network-wide decoding of the flows, and temporal and flow space analysis for different monitoring applications.

4.3 Use Cases

Since FlowRadar provides per flow counters, it can easily inherit many monitoring applications built on NetFlow such as accounting, capacity planning, application monitoring and profiling, and security analysis. In this section, we show that FlowRadar provides better monitoring support than sampled NetFlow and sFlow/EverFlow in two aspects: (1) Flow coverage: count all the flows without sampling; and (2) Temporal coverage: export these counters for each short time slot (e.g., 10ms).

4.3.1 Flow Coverage

Transient loop/blackhole detection: Transient loops and blackholes are important to detect, as they could cause packet loss. Just a few packet losses can cause significant tail-latency increase and throughput drops (especially because TCP treats losses as congestion signals)[139,15], leading to violations of service level agreements (SLAs) and even a decrease of revenue[81,175]. However, transient loops and blackholes are difficult to detect, as they may only affect a few packets during a very short time period. EverFlow or sampled NetFlow only select a few packets to monitor, and thus may miss most of the transient loops and blackholes. In addition, the transient loops and blackholes may only affect a certain kind of flows, so probing methods like Pingmesh[96] may not even notice the existence of them. Instead, if we can capture all the packets in each flow and maintain a corresponding counter in real time at every switch, we can quickly identify flows that are experiencing loops or blackholes (see § 4.6).

Errors in match-action tables: Switches usually maintain a pipeline of match-action tables for packet processing. Datacenters have reported table corruptions when switch memory experiences soft errors (i.e., bit flips) and these corruptions can lead to packet losses or incorrect forwarding for a small portion of the traffic[96,190][‡]. Such corruptions are hard to detect using network verification tools because they cannot see the actual corrupted tables. They are also hard to detect by sampled NetFlow or EverFlow because we cannot pre-decide the right set of packets to monitor. Instead, since FlowRadar can monitor all the packets, we can see problems when they happen (§ 4.6).

Fine-grained traffic analysis: Previous research has shown that packet sampling is inadequate for many fine-grained monitoring tasks such as understanding flow size distribution and anomaly detection[85,82,136]. Since FlowRadar monitors all the packets, we can provide more accurate traffic analysis and anomaly detection.

4.3.2 Temporal Coverage

Per-flow loss map: Packet losses can be caused by a variety of reasons (e.g., congestion, switch interface bug, packet corruptions) and may have significant impact on applications. Although each TCP connection can detect its own losses (with sequence numbers or with switch support[72]), it is hard for the operators to understand where the losses happen inside the network, how many flows/applications are affected by such loss, and how the number of losses changes over time. NetFlow with low sampling rates cannot capture losses that happened to flows that are not sampled; and even for those sampled flows, we cannot infer losses from estimated flow counters. EverFlow can only capture control packets (e.g., NACK (Negative Acknowledgment)) to infer loss and congestion scenarios. Instead, if we can deploy FlowRadar at switches, we can directly get an overall map of the per-flow loss rate for all the flows soon after a burst of packets passes by (see § 4.6).

Debugging ECMP load imbalance: ECMP load imbalance can lead to inefficient bandwidth usage in network and can significantly hurt application performance[16]. Short-term load imbalance

[‡]For example, the L2 forwarding table gets corrupted. The packet that matches the entry can be flooded or mis-forwarded, leading to transient blackholes or loops before the entry is relearnt and corrected.

can be caused by either (1) the network (e.g., ECMP not hashing on the right flow fields) or (2) the application (e.g., the application sends a sudden burst). If operators can quickly distinguish the two cases, they can make quick reactions to either reconfigure the ECMP functions for the network problem or to rate limit a specific application for the application problem.

EverFlow can diagnose some load imbalance problems by mirroring all the SYN and FIN packets and count the number of flows on each ECMP paths. However, it cannot diagnose either of the two cases above because it does not have detailed packet counters for each flow and does not know the traffic changes for these flows over time. Traditional NetFlow has similar limitations (i.e., no track of flows over time).

Timely attack detection: Some attacks exhibit specific temporal traffic patterns, which are hard to detect if we just count the number of packets per flow as NetFlow, or just capture the SYN/FIN packets as EverFlow. For example, TCP low-rate attacks[119] send a series of small traffic bursts that always trigger TCP's retransmission timeout, which can throttle TCP flows to a small fraction of the ideal rate. With per-flow counters at small time scale, we can not only detect these attacks by temporal analysis, but also report these attacks quickly (without waiting for the inactive timeout in NetFlow).

4.4 FlowRadar Design

The key design in FlowRadar is an encoding scheme to store flows and their counters in a small fixed-size memory, that requires constant insertion time at switches and can be decoded fast at the remote collector. When there is a sudden burst of flows, we can leverage network-wide decoding to decode more flows from multiple encoded flowsets. We also analyze the tradeoff between memory usage and decoding success rates.

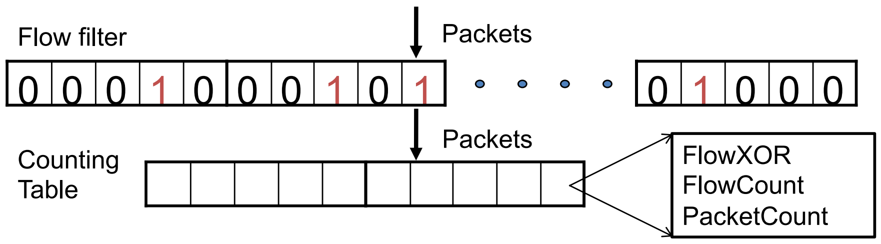


Figure 4.2: IBLT based flow counters

4.4.1 Encoded Flowsets

The key challenge for NetFlow is how to handle flow collisions. Rather than designing solutions to *react* to flow collisions, our design focuses on how to *embrace* collisions: We allow flows to collide with each other without extra memory usage, and yet ensure we can decode individual flows and their counters at the collector.

There are two key designs that allow us to embrace collisions: (1) First, we hash the same flow to multiple locations (like Bloom filters). In this way, the chance that one flow collide with other flows in one of the bins decreases. (2) When multiple flows fall in the same cell, it is expensive to store them in a linked list. Instead, we use a XOR function to the packets of these flows without using extra bits. In this way, FlowRadar can work with a fixed-size memory space shared among many flows and has constant update and insertion time for all the flows.

Based on the two designs, the *encoded flowset* data structure is shown in Figure 4.2, which includes two parts: The first part is the *flow filter*. The flow filter is just a normal Bloom filter with an array of 0's and 1's, which is used for testing if a packet belongs to a new flow or not. The second part is the *counting table* which is used to store flow counters. The counting table includes the following fields:

- *FlowXOR:* which keeps the XOR of all the flows (defined based on 5 tuples) mapped in the bin

- *FlowCount:* which keeps the number of flows mapped in the bin
- *PacketCount:* which keeps the number of packets of all the flows mapped in the bin

Algorithm 6 FlowRadar packet processing

```
1: if ∃ i ∈ [1,$k_f$], s.t. FlowFilter[$H_i^F$(p.flow)]==0 then
2:     FlowFilter.add(p.flow);
3:     for j= 1..$k_c$ do
4:         l = $H_j^C$(p.flow);
5:         CountTable[l].FlowXOR = CountTable[l].FlowXOR ⊕ p.flow;
6:         CountTable[l].FlowCount ++;
7: for j= 1..$k_c$ do
8:     CountTable[$H_j^C$(p.flow)].PacketCount ++;
```

As indicated in Algorithm 6, when a packet arrives, we first extract the flow fields of the packet, and check the flow filter to see if the flow has been stored in the flowset or not. If the packet comes from a new flow, we update the counting table by adding the packet's flow fields to FlowXOR and incrementing FlowCount and PacketCount at all the k_c locations. If the packet comes from an existing flow, we simply increment the packet counters at all the k_c locations.

Each switch sends the flowset to the collector every a few milliseconds, which we defined as *time slots*. In the rest of the paper, we set the value of the time slot to 10ms, unless explicitly setting it to other values in the context.

When FlowRadar collector receives the encoded flowset, it can decode the per flow counters by first looking for cells that include just one flow in it (called *pure cell*). For each flow in a *pure cell*, we perform the same hash functions to locate the other cells of this flow and remove it from all the cells (by XORing with the FlowXOR fields, subtracting the packet counter, and decrementing the flow counter). We then look for other *pure cells* and perform the same for the flows in each *pure cell*. The process ends when there are no *pure cells*. The detailed procedure is illustrated in Algorithm 7.

Algorithm 7 Decoding at a single node

```
1:  function SingleDecode($A$)
2:      $flowset = \emptyset$;
3:      for each $c$ where CountTable[$c$].FlowCount==1 do
4:          $flow = A$.CountTable[$c$].FlowXOR;
5:          $flowset$.add($flow$);
6:          $count = A$.CountTable[$c$].PacketCount;
7:          for $j$=1..$k_c$ do
8:              $l=H_j^C(flow)$;
9:              $A$.CountTable[$l$].FlowXOR = CountTable[$l$].FlowXOR $\oplus$ $flow$;
10:             $A$.CountTable[$l$].FlowCount -= 1;
11:             $A$.CountTable[$l$].PacketCount -= $count$;
12:     return $flowset$;
```

4.4.2 Network-wide Decoding

Operators can configure the encoded flowset size based on the expected number of flows. However, there can be a sudden burst in terms of the number of flows. In that case, we may fail to decode some flows, when we do not have any cell with just one flow in the middle of the SingleDecode process. To handle a burst of flows, we propose a network-wide decoding scheme that can correlate multiple encoded flowsets at different switches to decode more flows. Our network-wide decoding process has two steps: decoding flows across switches and decoding counters inside a single switch.

FlowDecode across switches: The key observation is that if we use different hash functions at different switches, and if we cannot decode one flow in one encoded flowset, it is likely that we may be able to decode the flow at another encoded flowset at a different switch the flow traverses. For example, suppose we collect flowsets at two neighboring switches A_1 and A_2. We know that they have a common subset of flows from A_1 to A_2. Some of these flows may be single-decoded at A_1 but not A_2. If they match A_2's flow filter, we can remove these flows from A_2, which may lead to more one-flow cells. We can run SingleDecode on A_2 again.

The general process of FlowDecode is described in Algorithm 8. Suppose we have the N encoded

Algorithm 8 FlowDecode

```
1:  for i=1..N do
2:      $S_i$ = SingleDecode($A_i$);
3:  $finish$ = false;
4:  while not $finish$ do
5:      $finish$ = true;
6:      for each $A_i$,$A_j$ are neighbor do
7:          for $flow$ in $S_i - S_j$ do
8:              if $A_j$.FlowFilter.contains($flow$) then
9:                  $S_j$.add($flow$);
10:                 for p=1..$k_c$ do
11:                     $l = H_p^{j,C}(flow)$;
12:                     $A_j$.CountTable[$l$].FlowXOR = $A_j$.CountTable[$l$].FlowXOR $\oplus$
    flow;
13:                     $A_j$.CountTable[$l$].FlowCount -= 1;
14:         for $flow$ in $S_j - S_i$ do
15:             Update $S_i$ and $A_i$ same as $S_j$ and $A_j$;
16:     for i=1..N do
17:         $result$ = SingleDecode($A_i$);
18:         if $result \neq \emptyset$ then
19:             $finish$ = false;
20:         $S_i$.add($result$);
    =0
```

flowsets: $A_1..A_N$, and the corresponding sets of flows we get from SingleDecode $S_1..S_N$. For any two neighboring A_i and A_j, we check the all the flows we can decode from A_i but not A_j (i.e., $S_i - S_j$) to see if they also appear at A_j's flow filter. We remove those flows that match A_j's flow filter from A_j. We then run SingleDecode for all the flowsets again, get the new groups of $S_1..S_N$ and continue checking the neighboring pairs. We repeat the whole process until we cannot decode any more flows in the network.

Note that if we have the routing information of each packet, FlowDecode can speed up, because for one decoded flow at A_i, we only check the previous hop and next hop of A_i instead of all neighbors.

CounterDecode at a single switch: Although we can easily decode the flows using FlowDecode, we cannot decode the counters of them. This is because the counters at A and B for the same flow may not be the same due to the packet losses and on-the-fly packets (e.g. packets in A's output queue). Fortunately, from the FlowDecode process, we may already know all the flows in one encoded flowset. That is, at each cell, we know all the flows that are in the cell and the summary of these flows' counters. Formally, we know $CountTable[i].PacketCount = \sum_{\forall f, \exists j, H_j^C(f)=i} f.PacketCount$ for each cell i. Suppose the flowset has m_c cells and n flows, we have a total of m_c equations and n variables. This means we need to solve $MX = b$, where X is the vector of n variables and M and b are constructed from the above equations. We show how to construct M and b in Algorithm 9.

Algorithm 9 Linear equations for CounterDecode

```
1: function ConstructLinearEquations($A$, $S$)
2:     $M$=ZeroMatrix; $b$=ColumnVector;
3:     for $flow_t$ in $S$ do
4:         for $j$=1..$k_c$ do
5:             $l = H_j^C(flow_t)$; $M[l,t]$ = 1;
6:     for CountTable[$j$] in $A$ do
7:         $b[j]$ = CountTable[$j$].PacketCount;
```

Solving a large set of sparse linear equations is not easy. With the fastest solver lsqr (which is based on iteration) in Matlab, it takes more than 1 minute to get the counters for 100K flows. We speed up the computation from two aspects. First, we provide a close approximation of the counters, so that the solver can start from the approximation and reach the result fast. As the counters are very close across hops for the same flow, we can get the approximated counters during the FlowDecode. That is, when decoding A_i with the help of A_j's flows (Algorithm 8 line 7 to 21), we treat the counter from A_j as the counter in A_i for the same flow. We feed the approximated counters to the solver as initial values to start iteration, so that it can converge faster. Second, we use a loose stopping criterion for the iteration. As the counter is always an integer, we stop the iteration as long as the result is floating within a range of ± 0.5 around an integer. This significantly reduces the rounds of iteration. By these two optimizations, we reduce the computation time by around 70 times.

4.4.3 Analysis of Decoding Errors

SingleDecode: We now perform a formal analysis of the error rate in an encoded flowset. Suppose the flow filter uses k_f hash functions and m_f cells; and the counting table has k_c hash functions and m_c cells with s_c bits per cell. The total memory usage is $m_c \cdot s_c + m_f$. Assume there are n flows in the encoded flowset. For the flow filter, the false positive for a single new flow (i.e., the new flow being treated as an existing flow) is $(1 - e^{-k_f n/m_f})^{k_f}$. Thus the chance that none of the n flows experience false positives is $\prod_{i=1}^{n-1}(1 - (1 - e^{-k_f i/m_f})^{k_f})$. When the flow filter has a false positive, we can detect it by checking if there are non-zero PacketCounts after decoding. In this case the counters are not trustful, but we still get all the flows.

For the counting table, the decoding success rate of SingleDecode (i.e., the chance we can decode *all* the flows) is proved to be larger than $O(1 - n^{-k_c+2})$, if $m_c > c_{k_c} n$, where c_{k_c} is a constant associates with k_c[92]. When we fail to decode some flows in the counting table, the already decoded flows and their counters are correct.

We choose to use separated flow filter and counting table rather than a combined one (i.e. the counting table also serves as a bloom filter to test new flow), because a combined one consumes much more memory. For a combined one, for each packet, we check the k_c cells it is hashed to, and view this flow as a new flow if and only if at least one of these k_c cells' FlowCount is 0. However, this solution requires far more memory than the separated solution. This is because for the counting table, a good parameter setting is about $k_c = 3$ and $m_c = 1.24n$ when n is larger than 10K based on the guidelines in[92] and our experiences in § 4.5. In such a parameter setting, when we treat the counting table as a Bloom filter, the false positive rate for a new flow is $(1 - e^{-k_c n/m_c})^{k_c}$ is larger than 99.9%. To keep the false positive rate low enough for all the n flows, we would have to significantly increase k_c and m_c.

NetDecode: We discuss FlowDecode and CounterDecode separately. For FlowDecode, we first consider a simple *pair-decode* case, where we run NetDecode between two nodes with the same set of flows. This can be viewed as decoding n flows in a large counting table with $2k_c$ hashes and $2m_c$ cells. This means we will need only half of the number of cells of the counting table with $2k_c$ hashes with SingleDecode. In our experiment, we only need $m_c = 8K$ for decoding 10K flow appear at both sides, which is even fewer than the number of flows.

For the more general network-wide FlowDecode, if all nodes in the network have more flows than expected and require FlowDecode, the decode success rate is similar to the *pair-decode* case. This is because for each node A, decoding its flows is similar to decoding the pair of A's flowset and the sum of flowsets from all the neighbors containing A's flows. However, it is more likely that only a portion of the nodes have more flows than expected, and the rest can SingleDecode. In this case, the decode success rate is higher than the *pair-decode* case.

For CounterDecode, we need at least the same number of linear equations as the number of variables (per flow counters). Because we have one equation per cell, we need the number of cells m_c

to be at least the number of variables n. In practice, m_c should be slightly larger than the n, to obtain a high enough chance of having n linearly independent equations.

The complete NetDecode process is bottlenecked by CounterDecode not FlowDecode. This is because CounterDecode requires more memory and takes more time to decode. Since CounterDecode only runs on a single node, the memory usage and decoding speed of NetDecode at a node mostly depends on the number of flows in its own decoded flowset, rather than the number of other flowsets that contain similar flows.

4.5 Evaluation

In this section, we demonstrate that FlowRadar can scale to many flows and large networks with limited memory, bandwidth, and computing overhead, through simulations on FatTree topologies.

4.5.1 Scale to Many Flows

Parameter settings We set up a simulation network of FatTree with $k = 8$ (80 switches). We set the number of flows on each switch in 10ms from 1K to 1000K. We generate an equal number of flows between each inter-Pod ToR pair. We then equally split these flows among ECMP paths. In this way, each switch has the same number of flows. We set the flow filter to ensure that the probability that one of the n flows experiences a false positive is 1/10 of the SingleDecode failure rate of the counting table. We set the optimal k_f and m_f according to the formulas in § 4.4.3. We set $k_c = 4$ because it is the best for NetDecode. We select m_c based on the guidelines in [92]. We set the size of FlowCounter according to the expected number of flows. We conservatively set both NetFlow and FlowRadar packet counters as 4 bytes, although in FlowRadar we collect statistics in a short time scale and thus would see much fewer packets and needs fewer bytes for the packet counter. Since our results are only related to the number of flows but not the packets, we generate a random set of

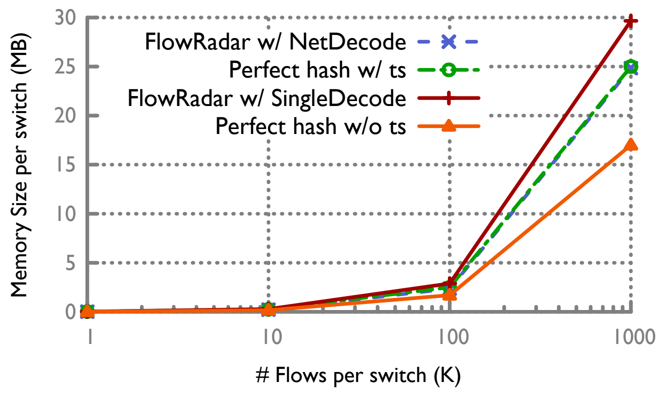


Figure 4.3: Memory usage per switch

flows as input.

We run decoding on 3.60GHz CPU cores, and parallelize decoding different flowsets on multiple cores.

The memory usage of FlowRadar is close to NetFlow with a perfect hash table: We first compare the memory usage between NetFlow and FlowRadar. As discussed in § 4.2, it is almost impossible in merchant silicon to implement a hash-based design that handles flow insertions and collisions within the per packet time budget. If we implement a simple hash table, it would take 8.5TB to store 100K flows to ensure a 99% chance that there are no collisions. The actual data structure used in custom silicon would be proprietary information. Therefore, we compare with the best possible case for NetFlow—a perfect hash table without any collisions.

Even with a perfect hash table, NetFlow still needs to store in each cell the starting time of a flow and the time the flow is last seen for calculating inactive timeout (4 bytes each). However, in FlowRadar, we do not need to keep timestamps in hardware because we use frequent reporting in a short scale. To fully decouple the benefit of FlowRadar data structure and removing timestamps, we also compare with perfect hashing without timestamps, which can be viewed as the optimal case we can reach.

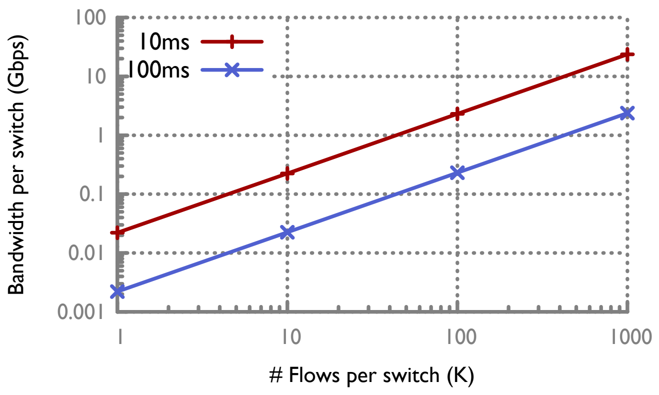


Figure 4.4: Bandwidth usage per switch

Figure 4.3 shows that NetFlow with perfect hashing needs 2.5MB per switch. FlowRadar needs only 2.88MB per switch with SingleDecode and 2.36MB per switch with NetDecode to store 100K flows with 99% decoding success[§], which is +15.2% and -5.6% compared to 2.5MB used by NetFlow. The best possible memory usage with perfect hashing without timestamps is 1.7MB per switch. With 1M flows, we need 29.7MB per switch for SingleDecode and 24.8MB per switch for NetDecode, which is +18.8% and -0.8% compared to NetFlow with perfect hashing and timestamps.

FlowRadar requires only a small portion of bandwidth to send encoded flowsets every 10ms. Figure 4.4 shows that we only need 2.3Gbps per switch to send encoded flowsets of 100K flows with 10ms time slot, and 0.23Gbps with 100ms time slot. In Facebook datacenter and traffic setting[156], a rack switch connects to 44 hosts with 10Gbps links, where each host send at most 100s to 1000s of concurrent flows in 5ms. Suppose there are a total of 2K*44 flows in 10ms in the rack switch, FlowRadar only incurs less than 0.52% of bandwidth overhead (2.3Gbps/(44*10Gbps)) with 10ms time slot.

[§]Note that even in the 1% of cases we cannot successfully decode all flows, we can still decode 61.7% of the flows on average.

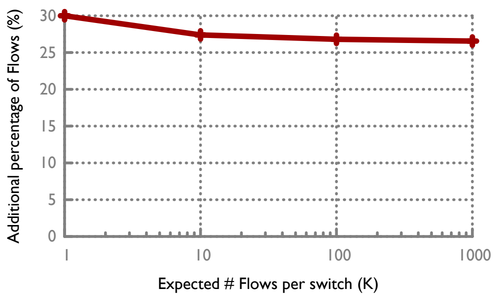


Figure 4.5: Extra #flows using NetDecode

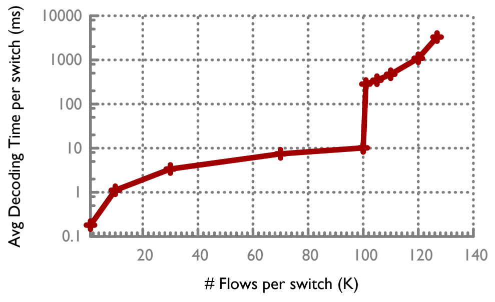


Figure 4.6: Decoding time

FlowRadar with NetDecode can support 26.6-30% more flows than SingleDecode, with more decoding time. Operators can configure FlowRadar based on the expected number of flows. When the number of flows goes beyond the expected number, we can use NetDecode to decode more flows given the same memory. Figure 4.5 shows with 1K to 1M expected number of flows, NetDecode can decode 26.6-30% more flows than SingleDecode given the same memory. So our solution can tolerate bursts in the number of flows.

Figure 4.6 shows the average decoding time of each flowset for the case with 100K expected flows. When the traffic is below 100K flows, the collector can run SingleDecode to quickly detect all the flows within 10ms. When the traffic goes beyong 100K flows, we need NetDecode, which takes 283ms and 3275ms to decode flowsets with respective 101K flows and 126.8K flows.

We break down the NetDecode time into CounterDecode and FlowDecode. The result is shown in Figure 4.7. As the number of flows increases, the CounterDecode time increases fast, but the FlowDecode time remains low. If we just need to decode the flows, we need only 135ms, which is very small portion compared to CounterDecode's 3140ms. Note that the burst of flows does not always happen, so it is fine to wait for extra time to get the decoded flows and counters.

We do not rely on the routing information to reduce the NetDecode time, because it only helps reduce the FlowDecode time, which is only a small portion of the NetDecode time. The routing information can help reduce the FlowDecode time by 2 times.

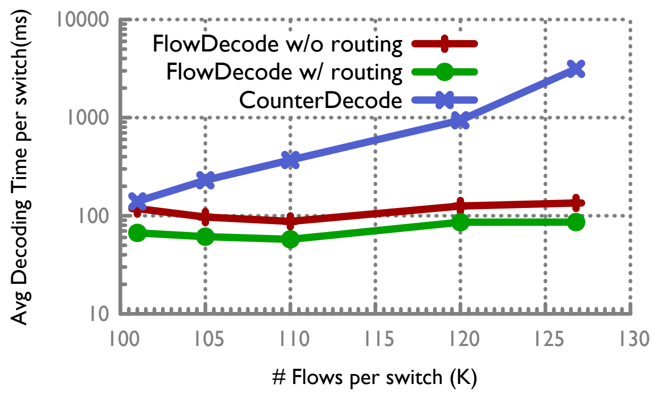


Figure 4.7: Breakdown of NetDecode Time

4.5.2 Scale to Many Switches

We now investigate how FlowRadar scales with larger networks. For direct comparison, we assume the same number of flows per switch with different network sizes.

The memory and bandwidth usages per switch do not change with more switches: This is because the decoding success rate only relates to the number of flows and number of cells. Obviously this is true for SingleDecode. For NetDecode this is also true, because as long as all flows appear in at least 2 flowsets, NetDecode's decoding rate is similar no matter how many flowsets the flows appear in. The reason is that the bottleneck of the number of flows can be decoded is from CounterDecode, which is independent from other flowsets. For flowsets with 102.5K cells, two such flowsets can already decode more than 110K flows, but the CounterDecode can only support 100K flows (limited by the number of linearly independent equations).

Decoding requires proportionally more cores with more switches: The SingleDecode time per switch only relates to the number of flows in a flowset. For example, to decode 100K flows within 10ms, we need the same number of cores at the remote collector as the number of switches. This means for a network with 27K servers (K=48 FatTree) and 16 cores per server, we need about 0.65% of the servers for the decoding.

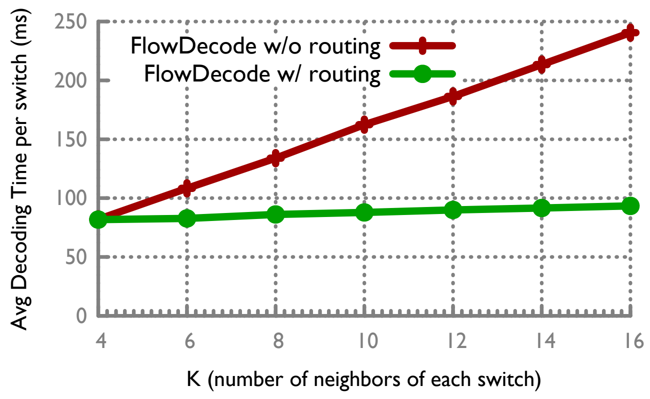


Figure 4.8: FlowDecode Time with different network size

NetDecode only happens during bursts of flows. The decoding time per switch increases slowly with more switches, because most time is spent on CounterDecode, which only relates to the number of flows in a flowset.

The FlowDecode time increases with larger networks, because it takes more time to check a decoded flow with the neighboring switches, when there are more neighbors in a larger network. In a FatTree network, suppose each switch has k neighbors. The total number of switches in the network is $n = \frac{5}{4}k^2$, so each flowset only checks with $O(\sqrt{n})$ other flowsets. We tested the FlowDecode time with different FatTree network sizes by increasing k from 4 to 16. The memory on each switch is set expecting 100K flows for SingleDecode. We generate traffic such that the number of flows on each switch reaches the maximum number (126.8K) that could be NetDecoded. Figure 4.8 shows the result. The FlowDecode time increases linearly with k. However, it is still a small portion compared to CounterDecode time. For 126.8K flows per switch and $k = 16$ FatTree, FlowDecode only takes 0.24 seconds, which is 7.1% of the total decoding time. Routing information can speed up FlowDecode to 0.093 seconds, which is 2.9% of the total decoding time.

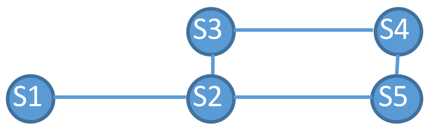


Figure 4.9: A flow path that has cycle

4.6 FlowRadar Analyzer

We show two use cases of FlowRadar: transient loop and blackhole detection with network-wide flow analysis and providing per-flow loss map with temporal analysis.

4.6.1 Transient Loop/Blackhole Detection

With FlowRadar, we can infer the path for each flow by concatenating the switches that have records for that flow. As a result, we can easily provide a network-wide map of all the loops and blackholes, the time they happen, and the flows they affected.

Loops: We first identify all the switches that see the same flow during each time slot. If the switches form a cycle, then we suspect there is a loop. We cannot conclude that there is a loop because this may be caused by a routing change. For example, in Figure 4.9, we may observe counters at all the switches in one time slot with FlowRadar, which forms a cycle (S2,S3,S4,S5). However, this may be caused by a routing change from S1 $\rightarrow$ S2 $\rightarrow$ S5 to S1 $\rightarrow$ S2 $\rightarrow$ S3 $\rightarrow$ S4 $\rightarrow$ s5 within the time slot. To confirm, we need to compare the counter on the hop that is not in the cycle (counter1), and the counter on one hop in the cycle (counter2). If counter1 $<$ counter2 then we can conclude that there is a loop. For example, if counter on S1 $<$ counter on S3, we know this is a loop.

Blackholes: If a transient blackhole is longer than a slot's time, we can detect it by seeing the path of some flows stopped at some hop. If a transient blackhole is shorter than a slot's time, we still see a large difference between the counters before and after the blackhole at one slot. Note that we do not need the counters, but only the flow information to detect blackhole. Thus, during flow bursts, we can run FlowDecode without CounterDecode to detect blackholes faster.

Evaluation: We create a FatTree k=4 topology with 16 hosts and 20 switches in DeterLab[3]. We modify Open vSwitch[10] to support our traffic collection. We direct all the packets to the user space and maintain the encoded flowsets. We install forwarding rules for individual flows with different source and destination IP pair.

We send persistent flows from each host to all the other hosts, which send one packet every 5ms. This is to make sure that each flow has at least one packet in each time slot even if some packets is close to the slot's boundary.

We simulated a case that a VM migration causes a transient loop when the routing table on the edge switch S1 of the old VM location is updated so it sends packets up to the aggregation switch S2. But S2 has not been updated so it sends packets back to S1. We manually updated a rule at the edge switch S1 at around 10ms, which forms a loop S1 $\rightarrow$ S2 $\rightarrow$ S1, where S2 is an aggregation switch. We can detect the loop within 10ms.

To generate a blackhole, we manually remove a routing rule at an edge switch. We can detect the blackhole within 20ms. This is because there are still traffic in the first 10ms when the blackhole happens. So we can only confirm in the next 10ms.

4.6.2 Per-flow Loss Map

FlowRadar can generate a network-wide loss map by comparing the per-flow counters between the upstream and downstream switches (or hosts) in a sequence of time slots. A simple approach is that for each flow, the difference between the upstream and downstream counters is the number of losses in each time slot. However, this approach does not work in practice because it is impossible for the two switches capture exactly the same set of packets, even though today's datacenters often have well synchronized clocks across switches at milliseconds level. This is because there are always packets on the fly between upstream and downstream switches (e.g., in the switch output queue).

To address this problem, we can wait until the flow finishes to compare its total number of packets at different hops. But this takes too long. Instead, we can detect losses faster by comparing counters for flowlets instead of flows. Suppose a time slot in FlowRadar is 10ms. We define *flowlets* as bursts of packets from a flow that are separated by gaps larger than a time slot[108]. With FlowRadar, we can identify flowlets between two time slots with counters equal to zero. Given a flowlet *f*, the upstream and downstream switches collect sequences of counters: $U_1 ... U_t$ and $D_1 ... D_t$ (D_0 and D_{t+1} are zero). We compute the total number of losses for the flowlet *f* as $\sum_{i=1}^{t}(U_i) - \sum_{i=1}^{t}(D_i)$. This is because if a packet does not arrive at the downstream switch for at least 10ms, it is very likely this packet is lost.

With this approach, we can get the accurate loss numbers and rates for all the flowlets that have finished. The key factor for our detection delay is the duration of flowlets. Fortunately, in datacenters, many flows have short flowlets. For example, in a production web search workload[57], 87.5% of the partition/aggregate query flows are separated by a gap larger than 15ms. 95% of query flows can finish within 10ms. Moreover, 95% of background large flows have 10-200ms flow completion times with potential flowlets in them.

Evaluation: We evaluate our solution in a k=8 FatTree topology in a ns-3 simulator[50]. The FatTree has 128 hosts connected with 80 switches using 10G links. We take the same workload distribution from a production web search datacenter[57], but add the 1000 partition-aggregate queries per second with 20 incast degree (i.e., the number of responding nodes) and packet sizes of 1.5KB. The queue size of each port in our experiment is 150KB which means 100 packets of size 1.5KB. The flowlet durations are mostly shorter than 30ms with the maximum as 160ms. 50% of background traffic has 0ms interarrival time indicates application sends a spike of flows. The rest at least 40% of background traffic has interarrival time larger than 10ms for periodical update and short messages.

We run FlowRadar to collect encoded flowsets every 10ms at all the switches. We define detec-

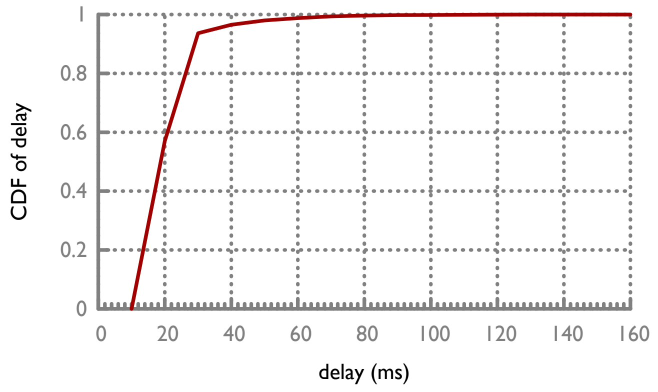


Figure 4.10: CDF of loss detection delay

tion delay as the time difference between when the loss happens and when we report it. Figure 4.10 shows the CDF of loss detection delay. We can detect more than 57% of the losses within 20ms, and more than 99% of the losses within 50ms.

4.7 Implementation

We now discuss the implementation issues in FlowRadar.

Encode and export flow counters at switches: FlowRadar only requires simple operations (e.g., hashing, XOR, and counting) that can be built on existing merchant silicon components. For example, hashing is already used in Layer 2 forwarding and ECMP functions. With the trend of programmable switches (e.g., P4[13]), FlowRadar can be easier to implement.

We have implemented our prototype in P4 simulator[14], which will be released at[4]. We use an array of counters to store our counting table and flow filter. On each packet's arrival, we use the modify_field_with_hash_based_offset API to generate the k_c hash values for counting table and k_f hash values for flow filter, and use bit_xor API to xor the header into the flowXOR field. In the control plane, we use the stateful_read_counter API to read the content in our data.

Since the encoded flowset is small, we can export the entire encoded flowset to the collector rather than exporting them on a per flow basis. To avoid the interruptions on the data plane during the exporting phase, we can use two encoded flowset tables: the incoming packets update one table while we export data in another table. Note that there is a tradeoff between the memory usage and exporting overhead. If we export more often (with a smaller export interval), there are fewer flows in the interval and thus require fewer memory usage. Operators can configure the right export interval based on the number of flows in different time scales and the switch performance. For this paper, we set the time interval as 10ms.

Deployment scenarios: Similar to NetFlow, we can deploy FlowRadar's encoded flowset either per port or per switch. The per-switch case would use less memory than per-port case because of multiplexing of flows. That is, it is unlikely that all the ports experience a burst in terms of the number of flows at the same time.

In the per-switch case, we still need to distinguish the incoming and outgoing flows (e.g., the two unidirectional flows in the same connection). One way to do this is to store the input port and output port as extra fields in the encoded flowset such as InputPortXOR and OutputPortXOR as what we did for the 5-tuple flow fields.[¶] Another way is to maintain two encoded flowsets, one for incoming flows and another for outgoing flows.

FlowRadar can be deployed in any set of switches. FlowRadar can already report the per-flow counters in short time scales independently at each deployed switch. If FlowRadar is deployed at more switches, we can leverage network-wide decoding to handle more number of flows in a burst. Note that our network-wide decoding does not require full deployment. As long as there are flows that traverse two or more encoded flowsets, we start to gain benefits from network-wide decoding. Operators can choose where to deploy, and they know the flows where they deployed FlowRadar. In

[¶]Similarly, one can easily add other flow properties (e.g., VLAN) as XOR sum fields.

the ideal case, if all switches are deployed, then we know the per-flow counters at all locations, and the paths of the flows. Operators could also choose a subset of switches. For example, if we deploy only on ToR switches, the counters still cover all the events (e.g. loss) in the network, but we no longer know the exact locations where the flows appear in the network. As we mentioned in §4.5.2, the decoding success rate does not change as long as we have at least 2 flowsets, so partial deployment does not affect decoding success rate.

4.8 Related Work

4.8.1 Monitoring Tools for Datacenters

Due to the problems of NetFlow, datacenter operators start to invent and use other monitoring tools. In addition to sFlow[177] and EverFlow[190], there are other in-network monitoring tools. OpenFlow[140] provide packet counters for each installed rules, which is only useful when the operators know which flows to track. Planck[153] leverages sampled mirroring at switches, which may not be sufficient for some monitoring tasks we discussed in § 4.2. There are also many end-host based monitoring solutions such as SNAP which captures TCP-level statistics[182] and pingmesh[96] which leverages active probes. FlowRadar is complementary to the end-host based solutions by providing in-network view for individual flows.

4.8.2 Measurement Data Structures

There have been many hash-based data structures for measurement. Compared to them, FlowRadar has three unique features: (1) Store flow-counter pairs for many flows; (2) Easy to implement in merchant silicon; (3) Support network-wide decoding across switches.

Data structures for performance measurement and volume counting: Varghese et. al. proposed a group of data structures for loss, latency, and burst measurement[116,171]. However, none of these solutions can maintain per flow metrics and scale to a large number of flows. There are many hash-based data structures that can keep per-flow state with small memory[67,183,164,187]. However, most of them do not suit for NetFlow because they can only keep the values (i.e., per flow state). Instead, FlowRadar provides the key-value pairs (i.e., the flow tuples and the packet counters) and can scale to a large number of flows.

Hash-based data structures for storing key-value pairs: Cuckoo hashing[150] and d-left hashing[62,174] are two hash table designs that can store key-value pairs with low memory usage. However, both are hard to implement in merchant silicon for NetFlow. This is because NetFlow requires inserting a flow immediately for an incoming packet so that follow up packets can update the same entry (i.e., *atomic read-update* operations). Otherwise, if one packet reads a cell that is being updated by a preceding packet, the counters become incorrect. Today, merchant silicon already has transactional memory that supports *read-update* operations in an atomic way for counters. However, typical merchant silicon can handle read-update operations against only a few (up to four) 4B- or 8B-long counters for each packet[‖]. This is because to support high link rate of merchant silicon (typically a few Tbps today), merchant silicon must resort to a highly-parallelized packet-processing design, and the atomic-execution logic is at odds with such parallelism. In fact, to support such atomic read-update semantics for a small number of counters, merchant silicon has to employ various complicated hardware logic similar to operand forward[11].

A d-way Cuckoo hash table[150] hashes each key to d positions and stores the key in one of the empty positions. When all the d positions are full, we need to rebuild the table by moving items around to make room for the new key. However, this rebuilding process can only be implemented with switch software (i.e., the control plane), because it requires multiple, and often-unbounded

[‖]Note the total number of counters can still be larger; only the number of concurrently read-and-updatable counters is small.

number of memory accesses[150]. Running the rebuilding process in switch software is not suitable for NetFlow, because NetFlow requires atomic read-update semantics.

d-left hashing splits a hash table with n buckets into d equal subtables each with n/d buckets, where each bucket contains L cells to hold up to L keys. d-left hashes a new key to d buckets, one in each subtable, and put the key in the bucket with the least load, breaking ties to the left. d-left requires first reading all Ld cells and testing if there is any match for an incoming flow. If there is a match, we increment the counter; otherwise, we put a new entry in an empty cell in the least-loaded bucket. There are two key challenges in supporting d-left: First, rather than read-update operations, d-left requires *atomic read-test-update* operations. The testing logic requires not only more ALUs and MUXes but also significantly increase the complexity of the atomic operation logic, making the critical section much longer in time. Second, d-left can only make insertion decisions after the testing on all Ld cells (each cell with 13 bytes 5-tuple fields and 4 bytes counter) are finished, which also increases the size of the atomic operation logic. Longer atomic operation duration can be a disaster for highly parallelized packet processing in merchant silicon.

In contrast, FlowRadar is easier to implement in merchant silicon, because of three reasons: First, FlowRadar only requires *atomic read-update* operations (i.e., increment/xor) rather than *atomic read-test-update*, which is much simpler in silicon design and has shorter atomic operation duration. Second, FlowRadar only requires atomic operations on a single cell and packets can update different cells in parallel. Thus FlowRadar requires significantly shorter atomic operations and is better fit for merchant silicon with high line rate.

It is impossible to support d-left with today's merchant silicon because the smallest d-left configuration (i.e., $d = 4$ and $L = 1$) needs to atomically read-test-update 4*17=68B, but today's silicon only supports 4*8B=32B. Thus, we compare FlowRadar with the basic d-left setting (i.e., $d = 4$ and $L = 1$) that may be supported in future silicon, and the setting recommended by[68] (i.e., $d = 3$ and $L = 5$) which is even harder to implement. To hold 100K flows on a memory of 2.74MB, the basic d-left has an overflow rate of 1.04%; both FlowRadar and the recommended d-left have no overflow. During flow bursts, FlowRadar can still report flows even when the counters cannot be decoded. Such flow information can be used for a variety of tasks like transient blackhole detection, route verification, and flow duration measurement. For example, to hold 152K flows in 2.74MB memory, the basic d-left has an overflow rate of 10%; the recommeded d-left has an overflow rate of 1.2%; FlowRadar can still decode all 152K flows (but not their counters).

Invertible Bloom filter Lookup Table (IBLT): FlowRadar is inspired by Invertible Bloom filter (IBF)[84] and Invertible Bloom filter Lookup Table (IBLT)[92]. IBF is used to keep a set of items. By comparing two IBFs, one can easily extract the differences between two sets. Rather then keeping a set of elements, FlowRadar needs to collect a key-value store of flows and their packet counters.

IBLT is an extension of IBF that can store key-value stores. Our counting table is built upon IBLT, but has two key extensions: *(1) How to handle value updates.* Since IBLT does not have a flow filter before it to identify if a key is new or old, it treats an existing key with a new value as a new key-value pair which has duplicated keys with existing key-value pairs. It then uses an arithmetic sum instead of a XOR sum in FlowXOR field, and a sum of hash values of the flows instead of a simple flow counter. This design takes more bits in both FlowXOR and FlowCount fields, which takes as much memory as FlowRadar uses for the flow filter. It also requires computations over large numbers (beyond 64bit integer), and more complex hash functions. Our experiments show that IBLT saves only 2.6% of memory for 100K keys but at the expense of 4.6 times more decoding time. *(2) How to decode the keys.* Our single node encoding scheme is similar to IBLT's, but takes much less time because of the simple FlowXOR and FlowCount fields. Moreover, with an extra flow filter, we support network-wide flow and counter decoding across multiple encoded flowsets.

4.9 Summary

We present FlowRadar, a new way to provide per-flow counters for all the flows in short time scales, which provides better visibility in datacenter networks. FlowRadar encodes flows and their counters with a small memory and constant insertion time at switches. It then introduces network-wide decoding of flowsets across switches to handle bursts of flows with limited memory. Our design can be improved in many aspects to further reduce the cost of computation, memory, and bandwidth, such as reducing the NetDecode time and better ways to leveraging redundancies across switch hops.

5

LossRadar: Precise Per-loss Telemetry

5.1 Introduction

Packet losses are common in datacenter networks and can happen for a variety of reasons. For example, Jeff Dean's keynote[79] indicates on average in a production datacenter for one year, up to 40-80 machines can experience packet losses, 4 network maintenance jobs can cause 30-minute random connectivity losses, and 3 router failures can cause immediate traffic blackholing for an hour. Such losses can significantly affect application tail latency and throughput especially when applications often view losses as a signal of congestion. It also takes operators tens of hours to diagnose the root causes of losses and recover from them.

It is important to detect losses *fast* and *independent of the loss types* (e.g., congestion, blackholes). With fast loss detection, operators have enough time to diagnose their root causes and mitigate the impact of losses. Being independent of loss types, we can capture all kinds of losses, especially unexpected ones such as those caused by hardware and software errors. Moreover, to diagnose the root causes, operators often need the detailed location on where losses happen, and the packet header information on what kind of packets get lost, the timing of losses, and other loss patterns.

Unfortunately, existing monitoring tools that are generic in capturing all types of network events often fall short in capturing losses fast with enough details and low overhead. End-host based solutions[137,145,96] cannot provide the exact location information on where losses happen. Packet mirroring[190,98] either incurs large bandwidth overhead of mirroring all the traffic, or selectively mirrors some packets and thus miss capturing all types of losses. Some counters at switches (e.g., SNMP counters) can report a few loss types (e.g., ACL drops) but often miss the other losses that are harder to capture (e.g., caused by hardware and software errors). Flow-based monitoring tools (e.g., FlowRadar[129], NetFlow[8]) keep counters for individual flows and compare them across hops to identify losses, which do not have the timing and sequence patterns of losses and have an overhead associated with the number of flows (which can be large in large datacenters[156]). One common theme of these approaches is to detect loss by keeping records of ongoing flows and identify the missing ones.

Due to the importance of loss in datacenters, we propose a specific monitoring system designed for loss detection. Instead of keeping records whose overhead is proportional to the ongoing traffic or flows, we propose to keep records whose overhead is proportional to the number of losses.

We propose LossRadar, a lightweight packet loss detection service that quickly reports the locations and 5-tuple flow information of individual lost packets in 10s of milliseconds. The very basic component we introduce for LossRadar is a meter, a light-weight traffic digest generation logic. We place several meters at various vantage points in a network. Then, each meter periodically generates small digests summarizing all the traffic passing through itself and exports the digests to a remote collector. The collector then decodes the digests exported by the meters. The way collector performs such analysis is inspired by the flow conservation rule in graph theory (and the current conservation rule in electrical-circuit theory as well). Given any segments (a link, switch, or group of switches) in a network and all uni-directional flows passing through the segment, the collector compares between the traffic digests generated by all the meters surrounding the segment. Any mismatch across the digests indicates packet losses, and the collector decodes the digest mismatch further to restore the identifiers of each lost packet. We also introduce an analyzer that can identify several (but not all) types of root causes of packet losses.

We design a Bloom-filter based data structure to collect traffic digests at each meter in real time with three key benefits: (1) Its memory requirement only grows linearly with the number of lost packets instead of all the transferred packets. (2) It keeps all the details of the lost packets such as 5 tuples, timing, and sequences of packets. (3) These traffic digests are generic to capture all types of losses and can be easily implemented in today's commodity switches.

Our extensive testbed evaluations with open vSwitch based prototype and large-scale packet-level simulations show that LossRadar uses only 1.4% of memory usage compared to the state-of-the-art approach when the loss rate is lower than 0.1%, and 0.5% of the bandwidth overhead of full mirroring, while providing detailed information for all the lost packets.

Table 5.1: Example types of packet losses

Loss types	Example error places	Mitigation	Details
Congestion	Buffer	Re-schedule flows	Flows, timing
Persistent blackholes	Switch match-action tables, Controller	Reboot, fix config	Flows, sequences
Transient blackholes	Controller, switch software	Debug controller or switch software	Flows, sequences, timing
Random drops	switch hardware	direct traffic away	Flows, timing

5.2 Packet Loss in Datacenters

Packet losses not only significantly affect application performance but also make network management tasks such as traffic engineering and diagnosis hard[96]. In this section, we first discuss the key requirements of loss detection: fast detection to minimize the impact of losses, generic detection of all types of losses, and capturing the locations and the headers of lost packets to help diagnosis. Next, we discuss existing tools in datacenters which fall short in supporting these requirements of loss detection with low overhead.

5.2.1 Requirements on Loss Detection

We summarize three key requirements on loss detection: (1) We need to detect packet losses *fast* to minimize their impact. (2) We need a generic approach to capture all types of packet losses ranging from congestion losses to random losses. (3) We need to know the locations and all the details of lost packets (e.g., header fields) to infer their root causes.

Fast detection. Packet losses are common in datacenter networks. Studies have shown that, on average in a production datacenter for one year, up to 40-80 machines can experience up to 50% packet loss rates, four network maintenance jobs can cause 30-minute random connectivity losses, and three router failures can cause immediate traffic blackholing for an hour[79]. Microsoft also reports one to ten new blackholes every day in a datacenter[96].

Packet losses often have significant impacts on application performance and network operations.

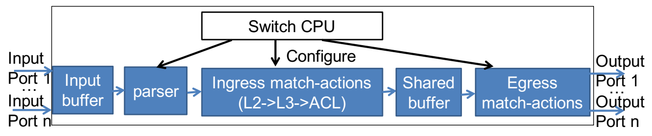


Figure 5.1: Switch packet processing pipeline

Just a few non-congestion-related losses can cause significant tail latency increase and throughput drop (especially because TCP treats them as congestion signals)[139,15], leading to violations of service level agreements (SLAs) and revenue drops of online services[81,175]. Moreover, it often takes operators tens of hours to identify whether a failure or performance problem is caused by packet losses, where losses happen, which flows are affected, and finally identify the root causes of losses and figure out fixes[9,23].

Instead, we need a loss detection solution that can identify losses and narrow down their root causes fast (e.g., in tens of milliseconds). In this way, we can give operators enough time to identify their root causes or take actions to eliminate these losses, in order to minimize their impact on applications and network operations.

Generic detection of all loss types. Losses can be related to different components in the network (Figure 5.1) and be caused by different reasons. It is impossible to enumerate all types of losses, so we highlight a few common ones normally reported in datacenters to illustrate the diversity of losses.

Congestion: Congestion loss happens at the output buffer and is caused by multiple flows competing for the same output port, and the total rate exceeds the capacity of that port.

Persistent blackholes: The persistent blackhole is a kind of problem that drops all packets matching a certain "pattern" at a switch[96,190]. Persistent blackholes could be caused by match-actions table corruptions[96,190] or the controller misconfiguring the rules.

Transient blackholes: The transient blackhole is similar to the persistent blackhole, but only during a short period. Transient blackholes could be caused by non-atomic rule updates[120] or network-wide inconsistent updates[105].

Random drops: The random drop is a type that the switch persistently drops packets randomly without reporting[96,190]. Random drops could be caused by component problems such as not-well-seated linecards or faulty links[96].

Of course, there are many other losses we have not mentioned here. Such a diversity of losses demands a tool that can detect all losses regardless of the types.

Capturing location information. The first step to diagnose a packet loss is to identify the location of it—which switch, NIC, or host that incur the packet loss. Knowing the location, we can take quick actions (e.g., reboot the switch, reroute the traffic) to quickly mitigate the impact[179] or analyze the flows at the location to help diagnose the root causes[96].

Capturing packet header information. We need to distinguish different types of losses to identify the best solution to mitigate these losses (Table 5.1). For example, if there are congestions, operators can re-schedule the flows to avoid congested links or move the applications to other servers. For persistent blackholes, we need to reboot the switches to fix table corruptions, or to correct the misconfigurations at the controller. If there is a transient blackhole, it helps operators to identify and fix the bugs in the controllers or switch softwares. If there are silent random drops, operators have to shut down and RMA (return merchandise authorization) the faulty components[96].

To distinguish different types of losses and understand the impacts of losses, we need different types of details. (1) flow information such as 5 tuples (source, destination IP addresses, ports, and protocol). Different flows may have different loss patterns, so it is important to know the 5 tuples. Also, such 5-tuple information can indicate the applications affected by the losses and allow operators to take fine-grained actions to help these applications. (2) Timing information. Many types of losses are very short, such as congestions or transient blackholes, so it is good to know the timing of each individual loss, which can help identify the temporal pattern of the losses. (3) Loss pattern. It is also important to identify the loss pattern, because different types of losses have different loss

Table 5.2: State-of-the-art datacenter monitoring solutions

Loss detection tools	**Detection time**	**Generic**	**Location**	**Header info**	**Overhead**
Host monitoring [137,145]	10s of ms	Yes	No	Yes	Host CPU
Pingmesh [96]	10s of seconds	Only persistent losses	Infer	No	Host CPU
Mirroring [190]	10s of ms	Miss congestion	Yes	Yes	B/w prop. to # packets
Flow-level counters [129]	After a flow(let) ends	Yes	Yes	Only flow info	Memory prop. to # flows
LossRadar	10s of ms	Yes	Yes	Yes	Memory prop. to # losses

patterns (e.g., blackhole losses are continuous, while others may not be).

5.2.2 Existing Monitoring Tools

In datacenters, there have already been a lot of monitoring tools. Some of them can be used for detecting packet losses. However, we now show that these solutions fall short in achieving the above goals. Due to the importance of loss detection in datacenters, we need a new monitoring tool that is specifically designed for packet losses.

End host based solutions. End host based solutions [137,145] can capture losses at the TCP level or by inspecting packets at the hypervisors. Although these solutions can capture all the details of lost packets independent of their types, it is hard to pinpoint the exact location of packet losses.

Pingmesh [96] can infer the location of packet losses through all-to-all probing, but cannot identify the exact switch that causes the problem within the leaf or spine groups. However, it does not track the packet losses of the actual applications. Moreover, its probing frequency is at least 10 seconds to reduce its overhead. Thus, it may miss many transient losses that happen between probes.

Packet mirroring at switches. Everflow [190] selectively mirrors packets to the collector, so the operators can know the traces of packets matching pre-selection filters and can detect losses by identifying those packets that do not have complete traces to reach the destination. However, it is not good at capturing most types of losses, and it is unclear if the packets get lost or if the report to the collectors gets lost.

Counters at switches. Switch vendors provide various counters (e.g., SNMP) to indicate the health conditions and operations at switches, some of which are related to losses. However, due to the diversity of loss types, it is difficult to identify and implement the right set of counters.

FlowRadar[129] captures per-flow counters at each switch at a fine time scale (e.g., 10s of milliseconds). One can use FlowRadar to detect packet losses by comparing the counters at two nearby hops. However, since it is hard to synchronize the counters to capture the same batch of packets, we often have to wait for a flow ends (or a flowlet ends with enough idle time after it[56]). The memory usage for FlowRadar is proportional to the number of flows we need to capture.

5.2.3 LossRadar Design

Instead of relying on generic monitoring tools which often fall short in loss detection, we propose to build a monitoring tool specific to loss detection given the prevalence and importance of packet losses in datacenters. We propose to design a new loss detection solution that can capture all types of losses that happen at anywhere anytime and identify individual lost packets, while keeping the overhead low.

Generic to all types of losses. To detect losses regardless of their types and root causes, our key idea follows the *Flow Conservation Rule*: The set of packets that comes into any network domain (a link, a switch module, a switch, or an area of network devices) should be equal to the set of packets that leaves the domain, as long as the domain does not include any packet sources and sinks (e.g., middleboxes, hosts)*.

Fast detection of losses and their locations. Based on the Flow Conservation Rule, we design LossRadar as shown in Figure 5.2. LossRadar installs *meters* to capture uni-directional traffic

*We recognize that a network switch can occasionally behave as a packet source or sink for certain types of packets (e.g., routing protocol messages, switch management protocol messages). Those packets, however, can be easily and unambiguously distinguished and hence can be filtered out from the LossRadar mechanism.

before it enters and after it leaves a domain in the network (called *upstream* and *downstream meters*). Each meter encodes the unique identifiers for individual packets into a traffic digest, and periodically reports traffic digests to a central LossRadar collector. LossRadar collector compares traffic digests from upstream and downstream meters, detects violations of the Flow Conservation Rule, and reports detailed information of lost packets.

Capture packet header information with low overhead. We want to store detailed information (e.g., 5 tuples, timing, sequences) to help the diagnosis of lost packets. However, switches have limited memory (only tens of MB for all the counters and match-action tables). Thus, it is important to capture important information about packet losses in limited memory. Rather than keeping information about all the traffic (e.g., FlowRadar[129]), we propose to build a traffic digest whose size is proportional to the number of packet losses independent of the total traffic or flows in the system.

With LossRadar, operators only need to provision the switch memory based on the number of losses they expect and the amount of details they need. When the number of losses goes beyond the expected number, LossRadar can still report the total number of losses at each location but may miss the detailed information such as 5 tuples for some losses. This is reasonable because when there is a large number of losses at a place, it is more important to reboot the entire switch or direct traffic away from it.

To collect more useful details of unknown lost packets, operators may also choose to pre-filter expected losses (e.g., those packets dropped by access control list) by specifying the right set of packets that LossRadar needs to keep in its traffic digests.

Challenges: There are three key challenges in building LossRadar:

Small size digests: To capture individual lost packets and their detailed information with low overhead, we proposed data structures that can capture all the packets in the digest while keeping the digest small enough to reduce the storage, bandwidth, and processing overhead. The key insight is we leverage coding techniques[84] to only keep information about lost packets. (§5.3)

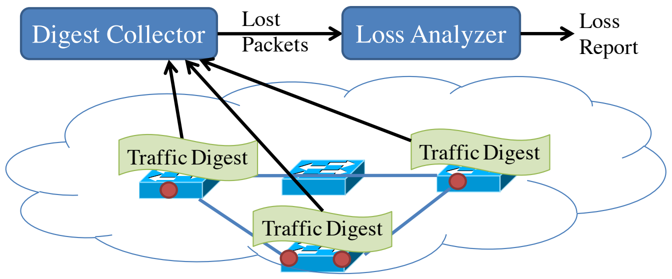


Figure 5.2: LossRadar architecture (Red circles are meters)

Meter placement: To locate where losses happen in the network, we identify the right places to install meters that can cover the entire network. In some networks where we cannot deploy meters at all places, we can still install the meters at some locations that can narrow down the potential location of losses. (§5.4)

Loss analyzer: Given the details of losses, we design temporal and spatial loss analysis algorithms that can quickly identify some root causes such as blackholes, congestions and random drops. Moreover, we develop algorithms that can identify the set of rules that may cause the blackhole. (§5.5)

5.3 LossRadar Traffic Digests

In this section, we describe the design of traffic digests with low memory overhead and the benefits of our design, and then discuss the challenges in collecting the digests.

5.3.1 Capturing Losses with Small Digests

We observe that although the number of packets in the network is too large for the meters to keep track of, the number of lost packets are much smaller. Thus if we can only keep track of the lost packets rather than all packets, we can reduce the overhead a lot. The question is how to track lost packets? Our idea is to keep traffic digests at an upstream meter and a downstream meter. When we

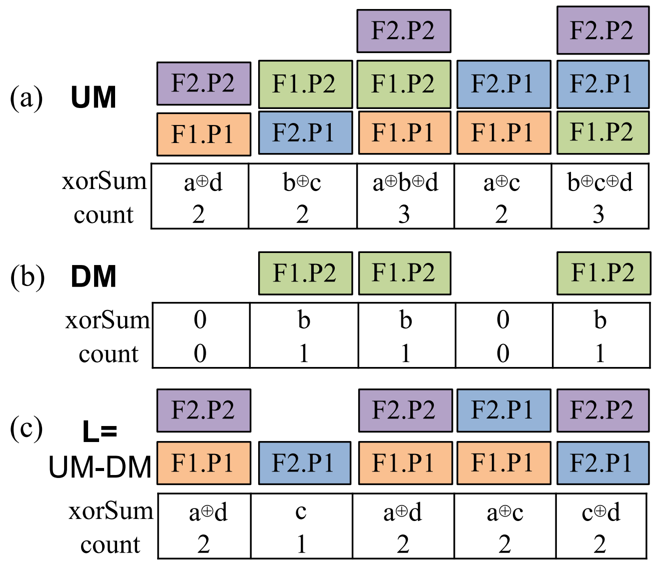


Figure 5.3: Traffic digests to get individual lost packets. (a) two flows, each with two packets at the upstream meter, indicated as a (F1.P1), b (F1.P2), c (F2.P1), d(F2.P2); (b) only F1.P2 is received at the downstream meter; (c) the result of the subtraction contains only lost packets

compare the two meters, the same packets at both digests will cancel out, so only the packets that do not appear at the downstream (i.e., lost packets) remain in the result. In this way, the digest size only needs to be large enough for the number of losses, not the total number of packets.

We build traffic digests using Invertible Bloom Filter (IBF)[84], as shown in Figure 5.3. The digest has an array of cells. Each cell contains two values: *xorSum* and *count*. When a packet x arrives, we insert it into k cells. Each cell is updated as $xorSum = xorSum \oplus x.sig$, $count = count + 1$. The $x.sig$ denotes x's *signature*, which includes the immutable header fields that can uniquely identify the packet (e.g., 5-tuple and IP_ID) and other information about the packet, which we will discuss in §5.3.2. The k cells are indexed by k hash functions (h_1 ... h_k) calculated on $x.sig$.

Figure 5.3 (a) and (b) shows two traffic digests, each of which is taken at the upstream meter UM and the downstream meter DM respectively. The upstream digest contains four packets from two flows, and the downstream one contains only one packet—the other three are lost.

To identify lost packets, the digest collector subtracts each cell i in DM's digest from the corresponding one in UM's digest. That is: $L_i.xorSum = UM_i.xorSum \oplus DM_i.xorSum$; $L_i.count = UM_i.count - DM_i.count$. We can decode lost packets from the digest difference L as follows: We first find the cells of L that contain exactly one packet (i.e., $count = 1$) which are called *pure cells*. The *xorSum* of a pure cell is exactly the signature of the packet in it. We then remove the packet from all the k cells in which the packet was hashed into. Once this is done, some other pure cells may appear, and we repeat the same process for them. By doing this iteratively, we can retrieve all packets in L with a very high probability (we will explain this in §5.3.2). For example, in Figure 5.3 (c), we first identify a pure cell whose count is 1 (F2.P1). After removing F2.P1, we get the packet F1.P1 (the 4^{th} cell's updated count is 1); and finally after removing F1.P1, we decode F2.P2.

5.3.2 Benefits of Digests

There are two benefits of the digests.

The size of the digest in the meter is proportional to the number of lost packets, but not the total number of packets that go through the meter. Suppose both *UM* and *DM* have n_{cell} cells. Then L, as a result of the subtraction, also has n_{cell} cells. Because L only needs to be large enough to decode the n_{loss} lost packets, n_{cell} is determined by the n_{loss}. Existing works have shown that n_{cell} only needs to be proportional to n_{loss} (i.e., $c = \frac{n_{cell}}{n_{loss}}$ is a constant) to achieve a high success rate for decoding all losses[†]. Eppstein et al.[84] shows empirically that $c = 1.3$ to achieve a 99% success rate when $n_{loss} > 1000$ for $k = 3$. Thus, we only need to set the size of the digest in data plane with a small c times the expected number of losses to achieve a high success rate.

Because packet loss rate is very low (less than 0.01%) in datacenters[96], we only need small sized digests. Thus we set our digest size small enough to capture lost packets based on an upper bound of

[†]A success for decoding all losses means we can always find a pure cell until all lost packets are decoded.

expected loss rate R (e.g., 0.1%). When the loss rate is below R, we can decode individual lost packets with high probability (e.g., >99%). When the loss rate is above R, however, we can still detect the total number of lost packets per port, but may fail to identify individual lost packets. This is because the digest may have so many lost packets in each cell, and thus the occurrence of pure cells become progressively less probable. Note, however, this may be still be quite acceptable for most network admins because, when the loss rate is this high, it is much more crucial to take urgent action to fix the problem (e.g., shut down the link or the switch) than to diagnose the problem or to account for individual losses to particular applications that are affected.

We can include a lot of details we are interested in of each lost packet. The basic setting is to include two pieces of information in the signature of a packet. (1) Flow-level information. We store the 5-tuple information to provide enough details of individual packets. (2) Packet identifier. We use the IP_ID field because the IP_ID field is consecutive within each 5-tuple, so it can be used to uniquely identify a packet. In case the IP_ID field is not sufficient (e.g., if more than 64K packets from a single flow are monitored in a single monitoring window), we can include other fields such as fragment offset, TCP sequence number or a small fraction of the payload in the signature. In the basic setting, each signature is 120 bits[‡].

Additionally, operators can configure to include any other fields of a packet that they are interested in (e.g., TCP flags). Note that if the field changes across hops, we need to consider that. For example, if we want to include TTL, and the TTL will decrement h in between, operators need to configure the DM to insert TTL + h instead of TTL into the digest. Operators can also include other metadata that are related to but not part of the packet, such as the timestamp when the packet arrives at the UM. In this case, the UM needs to tag the metadata in the packet header, so that the DM can use the same value. Including other fields require more bits in the signature. For example,

[‡]104 bits for 5-tuple, and 16 bits for IP_ID.

if we include the TCP-SYN bit, TTL (only need 1 bit to distinguish zero and non-zero) and timestamp, we need to extend the signature by 16 bits[§], which are used in our loss root cause analysis (§5.5).

5.3.3 Digest Configuration

With LossRadar, operators only need to provision the switch memory based on the number of losses they expect and the amount of details they need. When the number of losses goes beyond the expected number, LossRadar can still report the total number of losses at each location but may miss the detailed information such as 5 tuples for some losses. This is reasonable because when there is a large number of losses at a place, it is more important to reboot the entire switch or direct traffic away from it.

To collect more useful details of unknown lost packets, operators may also choose to pre-filter expected losses (e.g., those packets dropped by access control list) by specifying the right set of packets that LossRadar needs to keep in its traffic digests. Recent programmable switches can export information of which packets they decide to drop because of access control lists or buffer overflow (e.g., using negative mirroring, which is presented at P4 workshop). With these techniques, we can also exclude the lost packets known by switches, and thus focus LossRadar digests on those losses the switches cannot diagnose.

5.3.4 Measurement Batch

To capture packet losses in near real time, we capture packets in batches at both upstream and downstream. So the operator needs to ensure the same packet goes to the same batch in both the upstream meter and the downstream meter.

[§] 1 bit for TCP-SYN, 1 bits for TTL, and 14 bits for microsecond-level timestamp in 10ms (2^{14}us>10ms).

To report packet losses in near real time, meters should keep collecting traffic digests for a small batch of packets every T (e.g., 10ms) and immediately report the digests to the central collector as soon as the batch is over. To compare traffic digests correctly across meters, however, we need to ensure that a pair of upstream and downstream meters must agree on the boundary of batches so that they can put the same packet in the same batch. Otherwise, digests will always end up with mismatches. Unfortunately, relying on time-synchronization protocols or devices (e.g., NTP[141], PTP[20], GPS) to let meters agree on batch boundaries has fundamental limitations: no matter how accurate the time-sync protocol can ensure, there could always be packets on the fly, i.e. packets that leave the upstream meter, but have not arrived at the downstream meter. Hence, if we rely on the time-sync protocols, batch disagreement is inevitable.

Another possible solution is to use signal packets to synchronize the starting and ending times of a pair of upstream and downstream batches. However, if the signal packet is dropped or gets reordered, some packets may be classified into the wrong batch.

To work with packet losses and reordering, we propose to let each packet carry a batch identifier to tell the downstream meter which upstream batch the packet belongs to. Thus which batch at the downstream a packet belongs to is not based on the packet's arrival time, but the batch_ID it carries.

If there is no packet reordering, a downstream batch can be terminated when a packet with a new batch_ID arrives. However, if there is reordering (e.g., due to different priority queues), packets from different batches may interleave at the downstream meter, so multiple downstream batches have to coexist. As a result, we set the lifetime of a downstream batch to $T + \textit{timeout}$, starting at the first packet's arrival of this batch. The downstream batch is *timeout* longer than the upstream batch, which is for the delay in between. The operator can set the value of *timeout* based on their expectation of the maximum delay.

There is one question raised that how many downstream batches can coexist at the same time, which is related to the memory consumption. In datacenters, the RTT is a few hundreds of microseconds[57], and per-hop delay should be shorter, so for $T > 1ms$, there will be at most two batches coexist at a downstream meter.

5.4 Network-wide Deployment

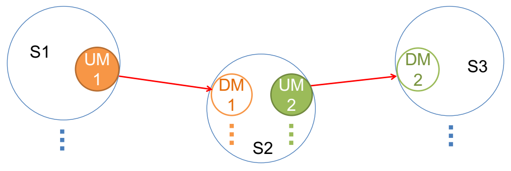
In this section, we discuss where to install the meters to detect packet losses that happen at any place in the network, and pinpoint the exact loss locations. We consider two cases: When we have access to all the devices in the network, or we only have access to some switches.

5.4.1 Cover the Entire Network

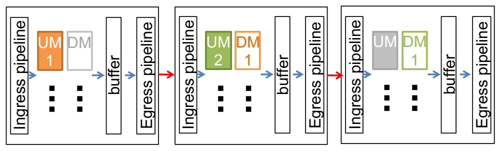
We need to install meters to meet two goals: (1) Each pair of meters covers a segment of paths where packets have a unique path. In this way, we can easily compare the traffic digests taken at a pair of meters (upstream and downstream) to identify lost packets; (2) All pairs of meters together cover the entire network. In this way, we would not miss any packet loss event.

The high-level idea is to install one upstream meter at each output port of every switch, and one downstream meter at each input port of every switch as shown in Figure 5.4a. In this way, both directions of every link are covered. Note that although we do not draw in Figure 5.4a, there is also traffic from S2 to S1, so actually we also need an upstream meter at S2's output port connected to S1, and a downstream meter at S1's input port connected to S2. In fact, we need two pairs of upstream meter and downstream meter for each bidirectional link. We only draw one pair in Figure 5.4a for conciseness.

Then, given the packet processing pipeline of a switch, we need to figure out at which stages to install the meters. Considering that losses could happen at any stage of the pipeline, we need to make sure the meters cover every part of the data plane. We choose to put the meters at the end of the ingress pipeline of tables, and before the shared buffer, as shown in Figure 5.4b. And, we



(a) Full deployment at all the switches



(b) Meter deployment details inside switches

Figure 5.4: Placing meters to cover the entire network (solid and empty circles are respective upstream and downstream meters)

put the upstream meters before the downstream meters within a switch, rather than the other way around, so that the segments between the upstream meters and downstream meters are also covered. In addition, we install meters for each network interface at hosts.

We choose this place to install meters for 2 reasons. First, at the place where upstream meters are installed, the decision of which output port(s) a packet goes to need to have been made, so that the switch knows which upstream meter (output port) to insert the packet into. Thus, the meters must be after all the ingress pipeline of tables, because the decision could be made at any of the ingress tables. Note, by doing so, we can handle multicast by inserting the packet into the upstream meters of all the output ports it goes to. Second, the timestamp collected at this stage is the closest to the time a packet is enqueued, so the timestamp can be used to describe the temporal pattern of congestion losses. By doing so, we make the most use of the timestamp of the packet.

5.4.2 Incremental Deployment

We do not expect LossRadar to be deployed at all the switches in datacenters at once. Now we consider the incremental deployment case where only a subset of the switches support LossRadar. We define blackboxes as those areas where we cannot install the meters. The strawman approach is to install one meter at each port of the blackbox and compare across meters. When meters do not have synchronized clocks, we propose to let each meter maintain multiple small digests instead of one. Moreover, our solution can reduce the memory usage of the strawman approach by almost 2 times.

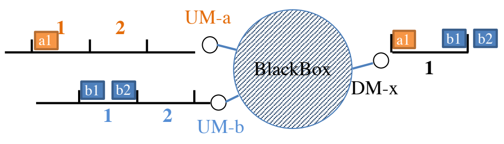
Strawman approach. Intuitively, we can insert an upstream meter at each entrance to the blackbox, and a downstream meter at each exit of the blackbox. Each upstream meter and downstream meter maintains one digest. To retrieve the losses, we can add up the digests at all upstream meters and the digests at all downstream meters respectively, and compare the two sums. Here adding up two digests a and b means for each corresponding cell i, $a_i.xorSum \oplus b_i.xorSum$, $a_i.count + b_i.count$. So the sum is also a digest of the same size with a and b.

The memory usage of each digest in this case is larger than in a non-blackbox case. The reason is as follows. Let us call the size of a digest in a non-blackbox case as *normal size*. Suppose there are n upstream meters and n downstream meters. During the comparison of the two sums, the total number of losses could be n times of the number of lost packets from a single upstream meter. Thus, the size of the sum digest has to be n times of the normal size. As a result, the size of the digest at each meter has to be the same with the sum, which is n times of the normal size.

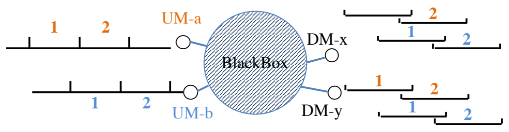
Handle unsynchronized clock across meters. The problem with the strawman solution is that if the clocks of the upstream meters are not well synchronized, packets from the same batch of the same upstream meter may arrive at different batches at one downstream meter. For example (Figure 5.5a), UM-a's batch 1 is earlier than UM-b's batch 1. The packet a1 from UM-a's batch 1 goes to DM-x, and starts the downstream batch 1. The packets from UM-b's batch 1 also goes to DM-x. The packet b1 from the *early* part of UM-b's batch 1 arrives at the *late* part of DM-x's batch 1, since DM-x's batch 1 has already started for a while. The packet b2, however, arrives after DM-x's batch 1 ended. So b1 and b2 are in the same upstream batch, but end up in different downstream batches.

As long as the clocks of the upstream meters are not accurately synchronized, there is always a chance of batch disagreement.

Solution. Our solution is that each downstream meter maintains separate digests for each upstream meter, as shown in Figure 5.5b. To detect the packet losses, we can add up all the digests for the



(a) Sync. Issue for blackbox



(b) Solution for blackbox case

Figure 5.5: Challenge and solution to blackbox case

same upstream meter across all the downstream meters (e.g., the digests for UM-a's batch 1 at DM-x and DM-y respectively), and compare the sum with the digest at the upstream meter (e.g., UM-a's batch 1). Thus, we can identify all the lost packets between one upstream meter and all its downstream meters.

Note that the downstream meters do not always know which upstream meter a packet comes from. Our solution is to let each upstream meter mark its meter ID in the packet header.

This solution has another benefit that it reduces the memory usage. In the strawman approach, the size of each digest (no matter at an upstream or a downstream) is n times of normal size. In our solution, however, we compare one upstream digest with the sum of its corresponding downstream digest, so the total number of losses in each comparison is just the number of lost packet from one upstream meter, not all upstream meters. So the size of each digest only needs to be normal size. Thus, although each downstream meter maintains n separate digests, the total size is still n times of normal size. But for each upstream meter, the total size is only normal size, which is reduced by n times.

In practice, each port of a blackbox can be both entrance and exit, so each switch connects to the blackbox need to install both upstream meter and downstream meter. In this case, the memory usage reduction is $2n/(n+1)$ per switch, which is almost 2 times.

5.5 Loss Analyzer

Using the detailed information of lost packets, we build an analyzer to classify the causes of packet losses by analyzing traffic digests across the network over time. The loss analyzer consists of (1) a root cause analyzer, which uses only the loss information from LossRadar to analyse the root causes of the losses, and (2) an ACL rule corruption analyzer as an application of the root cause analyzer, which can infer the corrupted rules of an ACL table based on the correct version of the ACL (from controller) and the blackhole losses from the root cause analyzer.

5.5.1 Root Cause Analyzer

The root cause analyzer classifies the reasons for the losses on each individual switch. In a short time (e.g., 10ms) at one switch, we assume losses are dominated by one reason, so we can identify the root causes by recognizing the unique patterns of different types of losses, given the details collected by LossRadar. We then relax the assumption to handle multiple problems all causing losses.

5.5.1.1 Classification based on loss patterns

We can classify the losses based on the unique patterns of different types of losses, given the details such as 5 tuples, IP_ID, TTL, timestamps and TCP flags. We list the common types of losses here. For other uncommon types, we classify them as unknown. However, after diagnosis, we can extract patterns of them based on the details we have, and use the patterns for future classification.

Congestion. We observed that *the congestion losses are bursty, and the gap between back-to-back losses is only a few microseconds*. The TCP sends a batch of packets every RTT, and the flows that experience losses will shrink their congestion windows, so in the next RTT congestion is much less likely to happen. Thus the congestion losses are bursty.

The gap between back-to-back losses is only a few microseconds because the transmission time of a packet is microsecond level. For 10G link, the transmission time of a 1500B packet is 1.2μs. Thus, even if the total receiving rate is only 10% higher than the sending rate of an output port, there will be one loss every 11 packets, which is one loss every 13.2μs.

Blackholes. We observed three different loss patterns.

(1) Bursty and consecutive losses within a flow . If a blackhole happens and affects ongoing large flows (e.g., file transferring, video delivering), the affected ongoing large flows have consecutive lost packets, and the losses are bursty.

(2) Non-bursty and consecutive losses within a flow . If a blackhole happens and affects some ongoing small flows, which only carries small pieces of data (e.g., sending messages), the affected ongoing small flows have consecutive, but non-bursty losses.

(3) Only a SYN packet loss from each flow . In the above two cases, each flow has multiple losses, but there are cases some flows only have one loss for more than a few seconds. After further investigation, we found that these lost packets are mostly SYN or SYN-ACK. This means that for the new flows, their handshake packets are dropped, so there are no follow-up packets until the retransmission of the SYN packet, which waits for a few seconds (normally 3 seconds for the first retransmission, then the waiting time doubles).

Random drops. *The random drops are evenly distributed over time.* This is because random drops are caused by problematic hardware components of a switch, which randomly drops packets passing it with certain drop rates.

Loop. *Packets dropped by loops have TTL = 0.* So we can use the TTL field to detect the losses caused by loops.

5.5.1.2 Root cause inference algorithms

There is still a small chance that during a given time period, there may be multiple types of losses together, which blurs the patterns. Thus, we design the root cause inference algorithms based on two principles. **(1) Identify bursty losses first**. We identify bursty losses first because other types of losses have long term patterns (e.g., random drops are evenly distributed over time, and the SYN packets dropped by blackholes should represent a large fraction of losses), which can be blurred by the burst of losses. **(2) It is fine to correctly classify only one type of losses when multiple types are mixed**. This is because we can fix the problems for one type of losses first. Moreover, if the missed types are caused by persistent problems, we can detect them in the future when they are not mixed. **(3) Keep the false alarm rate low for blackholes and random drops**, because both of them require human involvement: operator needs to check the switch configuration for blackholes, and hardware components for random drops. Thus, when blackholes/random drops are mixed with congestion, we believe classifying the losses as congestion losses is better[¶].

Before performing the inference algorithm, we first exclude the expected losses such as packets dropped by ACL tables. All the expected losses can be pre-filtered by programmable switches. If the switch does not support pre-filtering, the collector can easily identify this kind of losses by matching the lost packets against the table rules. We also exclude losses with TTL = 0, which could be dropped by loops, or are expected by the applications (e.g., traceroute). We can map the 5-tuple of the lost packets to the application to decide if the loss is expected or not.

The algorithm has two steps on different time scales: (1) For each new batch, we classify the bursty losses; (2) Every *long_term_interval* (e.g., 100ms), we classify the remaining losses (from multiple batches) into non-bursty types. We now describe the algorithm.

Analysis of bursty losses. We first perform an analysis of bursty losses for each new batch at each port (Algorithm 10). A burst is identified by at least n_{burst} losses with gaps less than t_{gap} between back-to-back losses. A burst of losses may be caused by either congestion or blackholes. If a flow's losses are non-consecutive in this burst, it cannot be caused by a blackhole, so we know there is con-

[¶]We understand that different networks have different requirements, so operators may use other principles if necessary.

gestion. In this case, according to the design principle (3), we classify the whole burst as congestion losses (Line 13 to 16). If all losses are consecutive, we only classify the flows that lost almost one congestion window ($\#losses > n_{bb}$[||]) as blackhole losses, and the rest as congestion losses (Line 17 to 19).

Our idea to measure the consecutiveness is to examine the IP_ID of the lost packets. In current Linux, the IP_ID of each 5-tuple is consecutive[70], so we can use the IP_ID to determine if the losses are consecutive or not.

We should set t_{gap} large enough and n_{burst} small enough to detect slight and short congestion, but still keep the probability of randomly dropping n_{burst} packets with less than t_{gap} back-to-back gap low. n_{bb} should be set slightly lower than the bandwidth-delay product for flows with still-growing windows, while large enough to keep the probability of congestion dropping n_{bb} consecutive packets of a flow low.

Long term analysis. Every *long_term_interval* (e.g., 100ms), we classify the rest of the losses (Algorithm 11).

We first classify the consecutive losses within a flow as blackhole losses (Line 2 to 4), because its per-flow pattern is not affected by other types of losses, and after removing these losses, the pattern of other losses are clearer. We set a small threshold n_{nb}[**] (line 3) for the length of the consecutive losses. n_{nb} needs to be large enough to keep the probability of randomly dropping n_{nb} consecutive packets of a flow low.

After that, for the rest of the packets, we first classify SYN losses as blackhole losses (line 6 to 8) if the fraction of SYN packets in an interval is large ($> p_{syn}$). p_{syn} should be set large enough to keep false alarm rate low. Finally, we measure the distribution of the rest of the losses over time (line 9 to 10). We divide the time interval into n (number of losses) equal bins, and count the fraction of

[||] bb stands for Bursty Blackhole

[**] nb stands for Non-bursty Blackhole

non-empty bins (bins with at least one loss) in all bins. We empirically estimate this fraction using a simple experiment: we randomly throw m balls into m bins following even distribution, and count the fraction of non-empty bins. The fraction is around 0.63 for different m. This fraction gives an upper bound of p_{rand}, because in practice packets' arrivals are non-uniform.

In practice, we should set $n_{burst} \leq n_{nb}$, because otherwise if a congestion loss burst of n packets where $n < n_{burst}$ contains $\geq n_{nb}$ consecutive losses of a flow, the losses of that flow would be classified as blackhole losses.

Algorithm 10 LossRadar bursty loss analysis

```
1: function BURSTY_LOSS_ANALYSIS($B$: a new batch of losses)
2:     Mark the types of all losses in $B$ as unknown;
3:     $burst$ = find_next_burst($B$);
4:     while $burst$ != Null do
5:         classify_burst($burst$);
6:         $burst$ = find_next_burst($B$);
7: function FIND_NEXT_BURST($B$)
8:     Find the next $B_{i...j}$ that $B_{k+1}.time - B_k.time < t_{gap}$, and the gap before $B_i$ and after
       $B_j$ is larger than $t_{gap}$;
9:     while $j - i + 1 < n_{burst}$ do
10:        Find the next $B_{i...j}$;
11:    return $B_{i...j}$;
12: function CLASSIFY_BURST($burst$)
13:    for flow $f$ in burst do
14:        if $f$'s lost packets have non-consecutive IP_ID then
15:            Mark all losses in $burst$ as congestion;
16:            return ;
17:    for flow $f$ that has more than $n_{bb}$ losses in $burst$ do
18:        Mark all losses of $f$ as blackhole;
19:    Mark losses in $burst$ whose types are unknown as congestion;
```

Algorithm 11 LossRadar root cause inference

```
1: function LONG_TERM_ANALYSIS($L$: all losses at a port in $long\_term\_interval$)
2:     for flow $f$ in $L$ do
3:         if $f$'s lost packets have consecutive IP_ID and #(unclassified losses) >= $n_{nb}$ then
4:             Mark losses in $f$ whose types are still unknown as blackhole;
5:     $rest$ = all the losses in $L$ whose types are still unknown;
6:     if fraction of SYN packets in $rest > p_{syn}$ then
7:         Mark all the SYN packets' types in $rest$ as blackhole;
8:         Remove the SYN packets from $rest$;
9:     if random_distribution($rest$) then
10:        Mark all packets' types in $rest$ as random;
11: function RANDOM_DISTRIBUTION($L$)
12:     $n$ = number of losses in $L$;
13:     Divide the time period of $L$ into $n$ equal time interval $T_{1...n}$;
14:     Count the number of losses in each $T_i$;
15:     if fraction of $T_i$ that has losses $> p_{rand}$ then
16:         return True;
17:     return False;
```

5.5.2 Rule Corruption Detection

For blackhole losses, after confirming with the configuration, we can determine if there are misconfigurations or not. If not, there must be rule corruptions. We develop algorithms that can locate the set of rules whose corruption may cause the blackhole. We take the access control list (ACL) table as an example. Given the list of blackhole losses and ACLs, we need to identify the rule that may get corrupted. For simplicity, we only consider one bit flip in the table, which is the most common case. There are three cases for unexpected losses: (a) a deny rule's match field corrupted, so it denies flows originally not covered by it; (b) an accept rule's match field corrupted, and the flows originally covered by it now match other deny rules; (c) an accept rule's action changes to deny. We enumerate all three cases. For case (a), we enumerate all deny rules in the correct ACL table, and find the candidates whose match field corrupted by one bit would match all unexpected loss. For case (b) and (c), we check if all unexpected loss all match the same rule in the correct ACL table. If so, then this rule is a candidate. Our evaluation in §5.7.4 shows that this algorithm often narrows down the candidate rule set of 2-2.5 rules and always successfully cover the corrupted rules.

5.6 Implementation

We implemented LossRadar in both Open vSwitch[10] and P4 behavioral model[12].

5.6.1 Open vSwitch Implementation

We modify Open vSwitch to support traffic digest collection. We direct all the packets to the user space and collect traffic digests there. We create a sending thread which sends digests at the end of each batch to a digest collector via a persistent TCP connection. Each time the collector receives both the traffic digests from a pair of upstream meter (UM) and downstream meters (DM), it runs the decoding process to report lost packets.

5.6.2 P4 Implementation

At a high level, we put LossRadar's functions in two tables, at the end of the ingress pipeline. The first table is for updating UMs, and the second is for DMs. We now inspect the different components needed by LossRadar.

Storing traffic digests. LossRadar's traffic digest is implemented on the *register* in P4. We defined two arrays of registers for *xorSum* and *count* respectively. The width of each register in the *xorSum* array is set to the total number of bits in a packet's *signature* (5-tuple, IP_ID, TTL, TCP flag, timestamp, etc.). The lengths of the arrays are the number of cells n_{cell}, which is defined by users according to their expected number of losses.

Each port of a switch has a UM and a DM. Each meter keeps two digests for different batches of packets. We put all $2n_{port}$ upstream digests in an array[††], and all $2n_{port}$ downstream digests in another array. The i-th cell in the b-th batch of the p-th port is indexed by $i + (b \cdot n_{port} + p) \cdot n_{cell}$.

Updating digests. This includes calculating hash values based on the packets' *signatures*, and updating the cells indexed by these hash values. We define 3 hash functions, calculated on a *field_list* that is defined to include the *signature* of each packet. The *signature* is a bit different for UM and DM. The mutable part of the *signature* (e.g., timestamp and TTL) for the DM should be the same as the part for the UM at the *previous hop*. So each UM uses the timestamp at this hop, and each DM uses the timestamp carried by the packet. The DM uses TTL+1 instead of TTL to cancel out the decrement between hops.

Getting timestamps. LossRadar uses timestamps when packets are inserted into the UM. In current P4 behavioral model[12], among the timestamps exposed, the closest to our need is the times-

[††]For brevity, we conceptually treat the two arrays for respective *xorSum* and *count* as one array.

tamp when the switch starts processing the packet (*ingress_global_timestamp*). This choice sacrifices the time spent in the ingress pipeline, which, however, is stabler and smaller (sub-microsecond) than the queuing delay, so it is also useful for congestion detection. We believe in future programmable switch design, exposing timestamp at arbitrary stages in the pipeline is possible.

Pre-filtering. In P4, every packet go through the whole pipeline, including the ones dropped by tables. So we can pre-filter the table-dropped packets by inserting them into only the DM (not the UM). In this way, these packets will cancel out with the UM in the previous hop, without affecting decoding other lost packets. We add a metadata *drop* with a default value 0 for each packet, and update it to 1 in the dropping actions of the tables. At the UM, we use a matching field to filter out packet with $drop = 1$.

Header modification. LossRadar needs the header to carry batchID and optionally the timestamp and meterID (if in a blackbox case). We either store these data in unused header fields (e.g., VLAN, DSCP) or add a header, and modify parsers or add a parser for these data accordingly.

Each hop needs to modify the batchID and timestamp in the header to the ones the UM uses. However, this modification cannot be performed before or at the UM, because after the UM, the DM needs the previous hop's batchID and timestamp, which is carried by the header. Thus, we store the new batchID and timestamp in a metadata at the UM, and copy the metadata to the header after the DM finishes the update.

5.7 Evaluation

Our evaluation focuses on answering five key questions:

1. What is the memory and bandwidth usage of LossRadar, and how they compare to other solutions such as FlowRadar[129] and full packet mirroring[190,98]? We test them with a single pair of switches with traffics of different loads and different loss rates. We use simulation to test up to

100G traffic load. The result shows that LossRadar always uses less memory than FlowRadar in large datacenters where the number of flows is large and less bandwidth than full mirroring.

2. Under realistic network traffic, how effective are the detection and the root cause inference of LossRadar? We run a network with 80 switches and 128 hosts in ns-3 simulations and evaluate LossRadar with different traffic, different loss patterns and loss rates. The result shows that we only need a few KB per digest to capture more than 99.99% lost packets. Our root cause inference algorithm can capture all the root causes within a few 100s ms.

3. What is the memory usage per switch for different deployments? We evaluate a full deployment and different types of blackboxes cases, based on the same topology, traffic and loss problems in the previous question. The result shows that the memory usages per switch in blackbox cases are higher than in a full deployment, but are still acceptable.

4. How fast is the loss detection? Our result in a testbed with Open vSwitch[10] on a FatTree topology shows that the detection delay is only 12ms.[‡‡]

5. Is the information LossRadar provides useful? We show two applications that use the loss details to improve performance and identify corrupted rules.

5.7.1 Bandwidth and Memory Overhead

We set up a simulation with a simple topology where there are two switches. By default, we use 10G traffic, and the average packet size is 175B according to Facebook's datacenter traffic[156]. We change the traffic volume and the number of concurrent flows during the experiment for comparison with different state-of-the-art solutions. For LossRadar, each meter in LossRadar reports the digests every 10ms. Because it is possible that we fail to decode a digest, the size of the digest we tested is large enough so that it can decode all losses with 99% probability.

[‡‡]We evaluate the delay with Open vSwitch rather than P4 behavior model due to performance concerns.

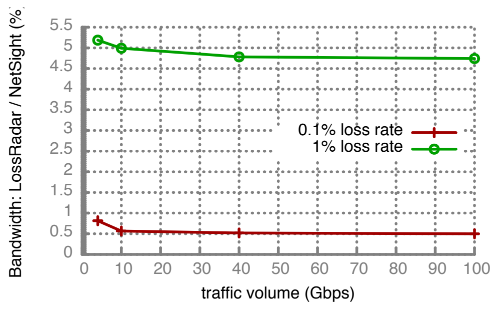


Figure 5.6: Bandwidth usage: LossRadar vs. NetSight

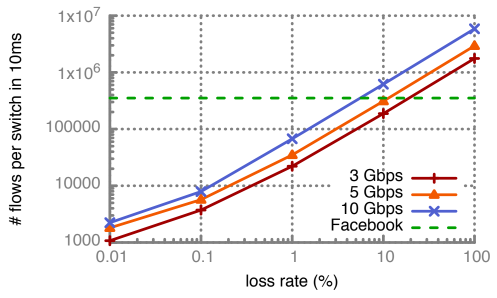


Figure 5.7: Memory usage: LossRadar vs. FlowRadar.

Bandwidth usage of LossRadar is much less than full packet mirroring. Both Everflow[190] and NetSight[98] capture packets at every hop in the entire network and sends them to a centralized analyzer. While Everflow mirrors raw packets, NetSight introduces compression to reduce the overhead. Thus we compare with NetSight's best compression (Van Jacobson style compression plus gzip compression), which is reported 10Bytes per packet. In order to detect every loss, we configure it to capture every packet.

We evaluated the bandwidth usage with different traffic volume from 4Gbps to 100Gbps, and two loss rates 0.1% and 1%, and show the percentage of LossRadar bandwidth usage to full packet mirroring bandwidth usage in Figure 5.6. The result shows that LossRadar saves 95% and 99.5% of the bandwidth usage, for 1% and 0.1% loss rates respectively. For example, for 10Gbps traffic, full packet mirroring uses 513Mbps, while LossRadar only uses 2.9Mbps for 0.1% loss rate, and 25.6Mbps for 1% loss rate.

LossRadar saves memory when the number of concurrent flow is large or loss rate is low, compared with FlowRadar. FlowRadar[129] maintains per-flow counters at switches. It can get the loss counter by comparing the packet counters across hops. Its memory usage is proportional to the number of concurrent flows.

We perform experiments to find the number of concurrent flows, loss rate and throughput at which LossRadar and FlowRadar need the same amount of memory, and plot the curves in Figure 5.7. The red, orange and blue curves are for 3Gbps, 5Gbps and 10Gbps (which is also a 40G link at 25% utilization) throughput respectively. In the area above the curves LossRadar uses less memory; below the curves FlowRadar uses less memory.

We also calculate and plot the average number of flows in 10ms at a cluster switch in Facebook. In Facebook, each server talks to 250 different racks in 5ms on average, and a cluster has 4 cluster switches and at least 64 racks. Thus, assuming each rack has 44 hosts (assuming the top of rack switch has 48 ports, 4 of which is connected to cluster switches), the total number of concurrent flows on average could be 352K in 10ms at a cluster switch. The green dotted line represents this number. This means that as long as the loss rate <5% at 10 Gbps, LossRadar uses less memory than FlowRadar. At a lower throughput, LossRadar can even support a higher loss rate (up to 10% and 20% for 5Gbps and 3Gbps) while using less memory than FlowRadar.

5.7.2 Detection and Inference Effectiveness

We evaluate the fraction of losses being detected given different amounts of memory and the accuracy of the root cause inference, under realistic traffic with injected random drops and blackholes.

We run larger networks and higher link speed in the ns-3 simulator. We simulate a k=8 FatTree topology, which includes 16 core switches, 8 pods (8 switches per pod), and 128 hosts, all connected with 10G links. The simulated switch per-packet processing time is 5μs (not including queueing delay). We deploy 1536 meters in total at switches and hosts.

To get realistic packet inter-arrival times, traffic distribution across switches, and realistic loss patterns, we implement DCTCP[57] and ECMP in the simulator. We take the same workload distribution from production datacenter[57], but add 1000 partition-aggregate queries per second to generate incast-related congestion losses. We set the default incast degree (i.e., the number of senders to one

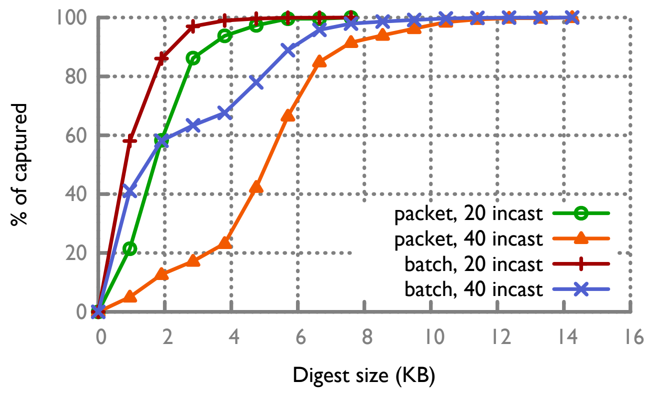


Figure 5.8: Memory usage under realistic traffic

receiver in a query) as 20. The query traffic characteristics also follow the study in[57]. By default, we set the total traffic volume as 40% of the input bandwidth (i.e., 40% × 1280Gbps in our topology), and a network-wide loss rate of 0.1%. (Note that the per switch port loss rate can go much higher.)

We also inject blackhole and random drops. We select 3 bidirectional links to have 1% random drops, one for each layer (host to ToR, ToR to Aggr, Aggr To Core). We also select 3 switches (a ToR, an Aggr and a Core) to have blackholes. The blackhole is blocking a randomly selected specific destination IP address. Both the random drops and the blackholes are inserted in the middle of the simulation (not at the beginning). Moreover, we choose the time to inject the blackholes so that they will drop some ongoing large flows, in order to have all types of loss patterns (discussed in §5.5.1.1).

5.7.2.1 Loss detection accuracy

LossRadar needs a small digest size to capture most losses. We evaluate different digest sizes and show the percentage of lost packets being captured in Figure 5.8 for different incast degrees. A captured lost packet means the packet is retrieved from the digest. With a 20 incast degree, we just need 2KB digest size to capture around 90% of lost packets. Note that LossRadar can still get the right number of losses at each switch, but only provide details for 90% of these packets. We need 7.6KB to capture the details for more than 99.99% of the lost packets. With a 40 incast degree, the network-wide loss rate grows from 0.1% to 0.3%. Thus we need 14.25KB to capture more than 99.99% of the lost packets.

LossRadar can capture most batches successfully with a small digest. We also evaluate the percentage of batches where *all* lost packets in the digest of that batch are successfully retrieved (Figure 5.8) as compared with all the batches with at least one loss here. With a 40 incast degree, we need 6KB per digest to capture around 90% of the batches. The percentage of captured batches is more than the percentage of captured lost packets because LossRadar sometimes fails to decode some packets when there are too many lost packets in a batch. For these batches, it is ok to just get the total numbers of losses because they have large numbers of losses and operators should inspect the entire switch port.

5.7.2.2 Root cause inference accuracy

We run our root cause inference on all the losses we get from the trace, and classify each loss into congestion loss, random drop, blackhole loss, or unknown reason. We set the digest size to 7.6KB according to §5.7.2.1. We set the parameters according to the guideline described in §5.5.1.2, as shown in Table 5.3. We compare the inference result with the ground truth, and show the precision and recall of the three types of losses in Figure 5.9. We also vary each parameter within a range while keeping other parameters with default values; the error bars in the figure shows the range of the precisions and recalls over all these different settings.

The precision is close to 100%. We design the algorithm to keep the false alarm rates of blackholes and random drops low. The high precision shows our algorithm achieves this goal. The precision of blackholes is 100%, which means there is no false alarm.

With different parameter settings, the precisions of random drops are all above 97.6%. The preci-

Table 5.3: Root cause inference parameter settings

	t_{gap} (us)	n_{burst}	n_{bb}	n_{nb}	p_{syn}	p_{rand}
Default	20	5	30	5	0.9	0.5
Range	[10,30]	[3,5]	[20,50]	[5,10]	[0.85,0.95]	[0.45,0.55]

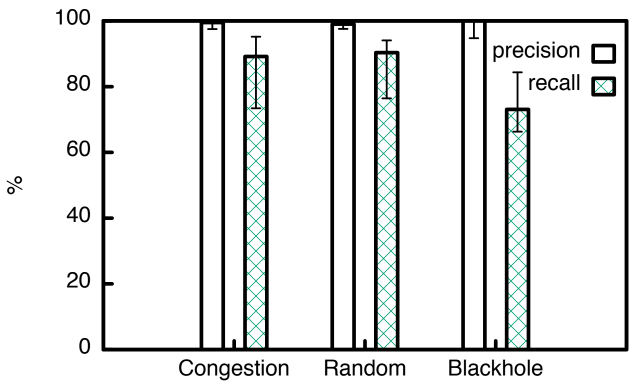


Figure 5.9: Root cause inference accuracy

sion of blackhole losses are almost all 100%, except when $n_{bb} = 20$ it becomes 94.7%. We inspect this case, and find that all false positives are 27 consecutive losses of a flow in a congestion. The reason for such long consecutive losses is similar to the outcast problem[152]. However, this phenomenon is exaggerated in simulation because all delays are deterministic.

Every injected problem is captured. We correctly classify 90.3% of the random drops and 73.1% of the blackhole losses. The missed blackhole losses are all short bursts (e.g., 5 losses) of consecutive losses, so they are classified as congestion losses. The missed random drops have two types: (1) a small number of random drops happen closely (e.g., 3 random drops with gaps less than 10ms), so they are classified as congestion losses; (2) during some intervals, there are other losses being classified as unknown together with random drops, so the distribution is uneven, and we classify all the losses in these intervals as unknown. However, because the blackholes and the random drops are all persistent, we always correctly classify the losses caused by them in the next few intervals. Thus, the classified losses are sufficient for us to detect all problems.

With different parameter settings, most losses of each type are correctly classified, and we detect all problems as well.

5.7.3 Memory Usage in Partial Deployments

We evaluate the memory usage per switch in different deployment scenarios based on the same topology, traffic pattern and injected problem in §5.7.2.

With a full deployment, we need 243.2KB for all the digests in each switch[§§]. We also evaluate the memory usage when we do not have access to some parts of the network, which is treated as blackboxes. In datacenters, upgrading a ToR switch requires notifying applications or tenants running under the ToR on the planned downtime, or migrating the applications or tenants to other racks. On the other hand, upgrading an aggregation or a core switch is easier, because the traffic can be automatically rerouted to other paths when the switch is shut down without affecting applications.

Thus we evaluate the case that we have access to aggregation and core switches but treat ToR switches as blackboxes. We deploy more meters at the devices around the ToR switches. On an aggregation switch, each of the 4 ports that connect to ToR switches receives packets from the 4 hosts under the ToR switch, which requires 4 downstream meters; each port that connects to a core switch only needs 1 downstream meter. So each aggregation switch needs 20 downstream meters, plus 8 upstream meters (one per port). We need 425.6KB for all the digests in each aggregation switch.

We also evaluate other scenarios. If each aggregation switch is a blackbox, we need 608KB in each core switch and each ToR switch. If each core switch is a blackbox, we need 608KB in each

[§§]7.6KB for each digest, two digests (batches) per meter, two meters (UM and DM) per port and eight ports.

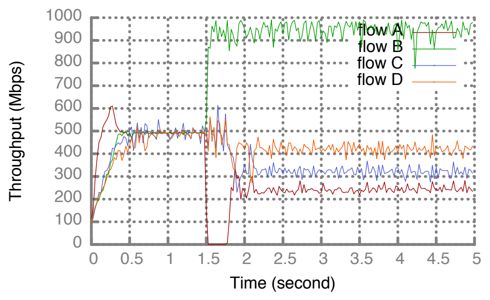


Figure 5.10: Recovery from blackhole

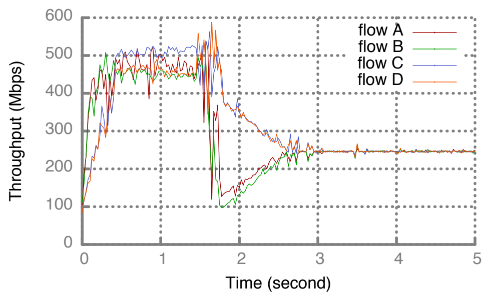


Figure 5.11: Recovery from random drops

aggregation switch.

5.7.4 LossRadar Applications

Improving flow throughput. We run four iperf flows (A, B, C, D) through two ECMP paths in our FatTree testbed, with two flows on each path. The two paths share the same edge and aggregation switches but differ at the core. We inject two types of losses. One is blackholing flow A (Figure 5.10). LossRadar quickly identifies that flow A is experiencing blackhole, and installs an entry matching flow A at the aggregation switch to moves A's path to the other. The throughput recovers after 250ms. The other case is randomly dropping 1% packets on A and B's link (Figure 5.11). LossRadar detects random drops on this link, so it moves both flows to the other path.

Correlating lost packets with ACLs. We use ClassBench ACL rules [168] and staggered data center traffic matching these rules as used in [146]. ClassBench has been shown to generates rules representative of real-world access control with accept and deny actions. The traffic is created by first generating staggered traffic for VMs and then assigning VMs with IP address ranges that match the ACL rules. The details are described in [146]. We take four rule sets of 50K, 100K, 150K, and 200K rules. We run the experiment 100 times by flipping a different random bit each time. We install the rules at a single switch, and send the traffic through the switch.

We compare the upstream and downstream traffic digests collected to get the set of lost packets.

Given the set of lost packets, our loss analyzer first excludes the intended drops based on the correct set of rules, and then runs flow space correlation algorithm to narrow down the set of candidate rules that may cause the loss. The whole process takes 0.7-2.5ms. Our result shows that we have zero false negatives (i.e., the rule with flipped bit is always in our reported set). From 50K to 200K rules, we narrow down to 2-2.5 candidate rules that may get bit flip. Sometimes a high priority rule which covers all the losses is treated as a candidate rule but sometimes it is not the corrupted rule.

5.8 Discussion

Avoid modifying packet header fields. It is possible to avoid tagging headers with the batch ID (§5.3.4) and the meter ID (§5.4.2), by tolerating on-the-fly packets.

Without the batch ID and the meter ID to synchronize batches, an on-the-fly packet would appear in two different batches at the upstream meter and the downstream meter, so it would appear as a difference in both the two batches' subtractions. Thus, we need to compare the results across batches and identify the true losses (only appear once not twice). The number of batches to compare depends on the time a packet takes from the upstream to the downstream, which is less than an RTT—100s of microseconds in datacenters. Thus, as long as the batch length $T >$ 1ms, we only need to wait for one more batch to conclude a packet is lost or on-the-fly.

Now the downstream digest is no longer a subset of the upstream digest. We leverage the original IBF design[84], which can handle two-way set difference. We also need larger digests, because the difference also include the on-the-fly packets besides the losses. With Sundial[127], we expect the extra size is extremely small. We leave the evaluation to future works.

Improving root cause inferences. Our root cause inference algorithms take the first step in identifying root causes for packet losses. There are a few ways to improve the inference: One way is to use machine learning instead of threshold-based conditions to better classify different types of losses with dynamics. Another way is to incorporate the header patterns of blackholes (e.g., specific 5-tuple or specific dest IP) to help better distinguishing blackhole losses from other types. In addition, when a previously unknown type of losses happen, how the analyzer correctly reports the existence of an unknown type is an open question.

Reducing memory usage for incremental deployment cases. In a blackbox case, each downstream meter needs to maintain one digest for each upstream meter, and the size of each downstream digest has to be equal to the size of an upstream digest, which results in a high memory usage per switch. If we can partition the the upstream digest based on which downstream each packet goes to, the downstream digests can be smaller. This requires path-awareness at the upstream; we leave it to future works.

5.9 Related Work

We discussed host passive mirroring[137,145], Pingmesh[96], Mirroring[190], and FlowRadar[129] in §5.2.2. FlowRadar is the closest work to LossRadar, but differs in four aspects: (1) FlowRadar collects per-flow counters at each switch, which can also be used for counting the aggregated number of losses per flow. In contrast, LossRadar provides details of individual losses. (2) The memory usage of FlowRadar is proportional to the number of flows, while LossRadar's is proportional to the number of losses. Thus, the memory usage of LossRadar is lower than FlowRadar when the loss rate is low, as shown in §5.7.1. (3) To detect loss, FlowRadar needs to compare flow counters between two switches, and thus has to wait till an idle reporting interval (at least 10ms) to synchronize the two counters. In contrast, LossRadar immediately detects the packet loss soon after a reporting interval. (4) FlowRadar uses Invertible Bloom Lookup Table[92] which also uses multiple hash functions per item and has a similar decoding process to IBF[84]. The key difference is that LossRadar utilizes the subtraction operation to cancel out the packets between upstream and downstream meters. FlowRadar cannot use the subtraction, because each flow's 5-tuple would cancel out in the subtraction, resulting in useless data.

Some other research solutions are available for loss detection. LDA[116] leverages packet counters to identify losses, but cannot identify individual lost packets and their flow information. NetSight[98] can detect packet loss by sending the packet headers of all the packets at each hop to a centralized controller, but incurs significant bandwidth overhead. In contrast, LossRadar provides detailed loss information with low memory and bandwidth overhead.

Many research works on network verification[111,110,114] performs static analysis of forwarding tables, and thus can detect blackholes caused by misconfiguration. Complementary to network verification, LossRadar capture all lost packets in the network in real time.

The paper[72] allows the switch to send to end hosts the packet header information of those packets that are dropped due to congestion. Rather than focusing on one type of losses (misconfiguration or congestion), LossRadar is generic to all types of packet losses.

Our traffic digest is inspired by Invertible Bloom Filters (IBFs)[84]. IBFs were introduced to identify *two-way* set differences for application-layer systems such as peer-to-peer networks and link-state databases. Our loss detection is a *one way* set difference problem, where one set is always a subset of the other. Therefore, we simplify the IBF design with XOR-based codes and counters, but store more flow information in the XOR-based codes. Moreover, because our data structure is used at switches with high-speed links, we face new challenges of synchronizing the data collection across nodes, and identifying the lost packets among multiple upstream and downstream meters around a blackbox. Counter Braids[134] are novel data structures to store *estimated* flow-based counters with low memory usage, but does not work for loss detection because it does not provide accurate counts and does not store flow information with counters.

5.10 Summary

With diverse types of packet losses in datacenter networks, and their significant impact on application performance and network operations, it is important to detect these losses in time and with detailed information of individual lost packets. We presented LossRadar, a generic, light-weight loss detection system that can identify individual lost packets, their locations, and the flows they belong to in near real time. LossRadar is easy to implement with programmable switch hardware, and can capture all the individual lost packets with small memory and bandwidth overhead. We also propose root cause inference algorithms based on the loss information from LossRadar.

6

DETER: Precise Per-packet Per-line-of-code Telemetry in TCP

6.1 Introduction

Modern datacenter applications increasingly rely on high throughput and low latency TCP performance. Yet, these applications often experience TCP performance problems that are hard to diagnose. This is because the TCP stack is a complex system that involves many heuristics to deal with network conditions and application behaviors, and it has many variations that optimize for different traffic scenarios and application objectives.

As a result, there is simply no single best setting for all scenarios. Researchers invent more than two TCP variations every year and there are already tens of congestion control algorithms to choose in Linux. TCP in Linux 4.4 has 63 parameters to configure, some of which are less known to normal application developers, such as early retransmission flag and TCP low latency flag which provides options for optimizing specific traffic settings. Other parameters are hard to configure even for TCP experts, as they have to run TCP multiple times to fully understand the influences of different parameter settings and the interactions of various TCP features. For example, thin-dupACK dynamically changes the threshold of the number of dupACK for fast retransmission based on the size of the current transfer. TSO window divisor affects the Nagle test for TSO, which decides how many packets to wait in order to form a larger packet.

Moreover, TCP is under continuous, error-prone development. There are 16 bugs identified in Linux TCP[49] in just July and August of 2018. As an example, one bug is related to DCTCP, where the DCTCP CC's ACK generation conflicts with the basic TCP framework's ACK generation, resulting in some packets never being acknowledged[43].

Many misconfigurations and bugs are hard to diagnose because they are sporadic and intermittent. However, they are still sufficient to degrade application performance, especially in datacenters where large scale distributed systems often involve thousands of requests to fulfill a task[148,104], because a single long latency may delay the entire task[80,126].

Although diagnosing TCP performance problems is notoriously hard, the gold standard tools are still the same as what have been used for tens of years: capturing packet traces[39] and tracing TCP executions[34,19]. While these tools are useful for diagnosing individual connections, using them in large-scale datacenter environments is hard, because there are millions of flows from hundreds of thousands of hosts interfering with each other constantly. Collecting packets and tracing TCP executions at all hosts and switches takes large quantities of storage, computing, and bandwidth resources. TCP counters [137,61] are useful lightweight tools in production, but they are not detailed enough to diagnose the complex settings and interactions mentioned above (see more examples of complex TCP performance problems in §6.5).

A common way to debug complex large-scale systems is deterministic replay [165,115,89,151,60,97]. Deterministic replay is proven to be an effective tool for developers to recreate performance problems, identify their root causes, and uncover many long-standing bugs in popular software. It would be ideal if we can deterministically replay TCP (i.e., deterministically re-execute the TCP code).

However, deterministically replaying a large network of TCP connections is difficult because TCP is a tightly coupled system with multiple interacting parties: applications, the network, other TCP connections traversing through a common switch, and the kernel at hosts.

In particular, the closed-loop nature of TCP creates a *butterfly effect*, where even small timing variations (e.g., clock drifts) between the runtime and the replay can drive the system to an entirely different state. Better time synchronization cannot solve this problem: even a nanosecond of timing variation leads to completely different TCP behaviors (§6.2.2). This is because small timing variations at hosts can cause different packet arriving orders at switches and therefore different packet drops. The differences in packet drops cause different TCP behaviors (e.g., congestion control) in turn, leading to different traffic rates from TCP senders and causing more differences in switch behaviors such as packet drops. Such butterfly effect propagates to many flows in the entire network after many rounds.

To eliminate the butterfly effect, we propose DETER, a DEterministic TCP Replay system, which breaks the closed loop interactions by replaying each TCP connection separately. We identify the minimal set of signals that capture all the interactions between a TCP connection with the application and the network, and record these signals at hosts in a lightweight manner. Specifically, DETER captures application socket calls and any impact on packets (e.g., if they are dropped or marked ECN) in the runtime. In the replay, we no longer need switches because all their actions to packets have been recorded and can be simply replayed. Since all the switch actions are deterministically replayed, we break the butterfly effect. We also isolate the TCP connection with other connections in the network because they only interact through switch actions.

The next question is how to deterministically replay an individual TCP connection. Although we already capture the interactions with the application and the network, there are still non-determinisms in the kernel at hosts. We design a customized solution for TCP which captures TCP-kernel interactions such as the kernel calling TCP handler functions, TCP reading kernel variables, and locks in thread scheduling. Note that we do not need to capture every packet, as the sender and receiver can generate packets and ACK for each other. The size of our total recording is just 2.1~3.1% of the size of fully compressed packet traces.

Since the recording is lightweight, DETER can run at all times for every connection on each host. Upon observing a performance problem, we can use DETER to zoom into any TCP connection, deterministically replay its exact same execution, capture packet traces, and examine TCP state during the execution—all after the fact. We can also iteratively debug the same performance problem instance multiple times to collect different levels of detail each time.

Once we have the packet traces for each connection using DETER, we can also replay network queues in a physical network, emulator, or simulator as long as the setup has the same topology, routing, buffer size, and switch queuing algorithms as the runtime. During the replay, we push all the outgoing packets for all the senders into the network based on their timestamps. We also introduce a heuristic that significantly improves the accuracy of replaying packet drops.

We demonstrate the benefits of DETER by showing how we diagnose TCP performance problems in a Spark application with 6.2K connections, tail latency problems in an empirical web search workload with incasts, and example performance problems in a local testbed. With DETER, we can also diagnose a wide range of performance problems that require tracing the TCP execution, such as long latency related to receive buffer shrinking, zero windows, late fast retransmission, frequent retransmission timeout, and problems related to the switch shared buffer. The main limitation of Deter is that it requires recording at both the sender and the receiver of a connection and therefore cannot work when we do not have access to both ends.

6.2 Diagnosis Example and Challenges

We use a diagnosis example to demonstrate the benefits of deterministic replay. We then use the example to show the key challenge to enable the deterministic replay—the butterfly effect. Even a nanosecond of sending timing variation leads to completely different TCP behaviors.

6.2.1 A Diagnosis Example

We use an example to show how Deter can help diagnose TCP performance problems. We run a network with two senders (A and B) and one receiver, which are connected to a single switch and 10Gbps links between them. Each sender sends two long flows of 20MB each. 30ms after the long flow starts, sender A sends a short flow of 30KB to the same receiver. In one run, the short flow takes 49ms to complete, which is two orders of magnitude higher than its expected completion time. In comparison, the RTO is just 16ms.

Usually, people diagnose a problem by reproducing it. However, this problem is very hard to reproduce (shown in §6.2.2). If we cannot reproduce a problem, we have to rely on the information captured online, such as the TCP counters that datacenters usually continuously monitor[137,61]. Unfortunately, TCP counters are not enough for diagnosing this problem. The counter for retransmission timeout is two, but twice the RTO (2*16ms) is still much less than 49ms.

With Deter, we can deterministically replay the connections using the lightweight data recorded

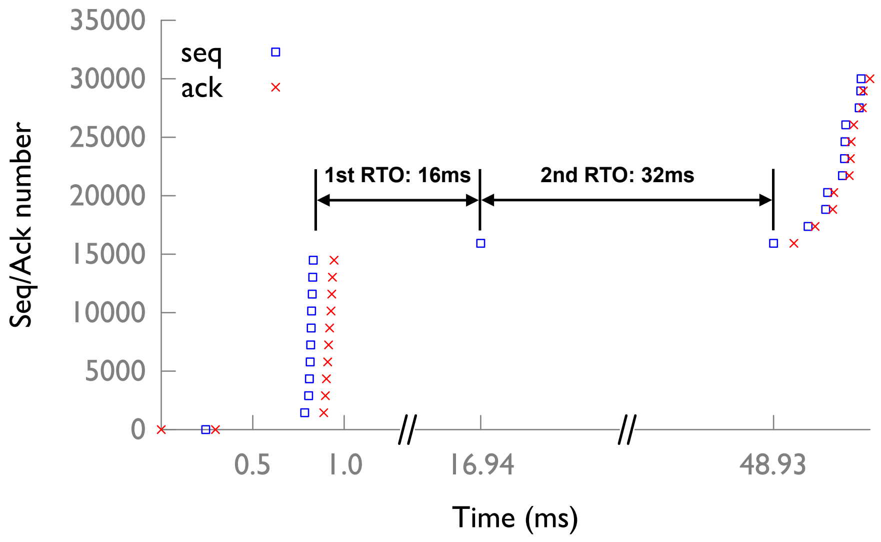


Figure 6.1: Receiver side Seq and Ack number of the short flow that experiences 49ms FCT.

in the runtime (Table 6.1), and capture packet traces during the replay. Here we show the packet trace at the receiver side for the short flow (Figure 6.1). The trace shows that the second timeout is 32ms. This is because the two timeouts are consecutive and thus trigger exponential backoff. The trace also shows the reason why the sender experiences the second timeout: the receiver receives the first retransmitted packet at 16.94ms, but it does not send an ACK. Without the ACK, the sender has to retransmit again at 48.93ms.

Why does the receiver not send an ACK for the first retransmitted packet? DETER allows us to replay multiple times, in order to collect more data and iteratively diagnose the problem. We use Ftrace[19] to get the function call graph on the processing of the first retransmitted packet. It shows that TCP enters the delayed ACK function, which means TCP decides to delay the ACK for the first retransmitted packet. The delayed ACK timeout is 40ms (which is a hardcoded value in the kernel and not configurable), which is longer than 2*RTO, so the second retransmission triggers first.

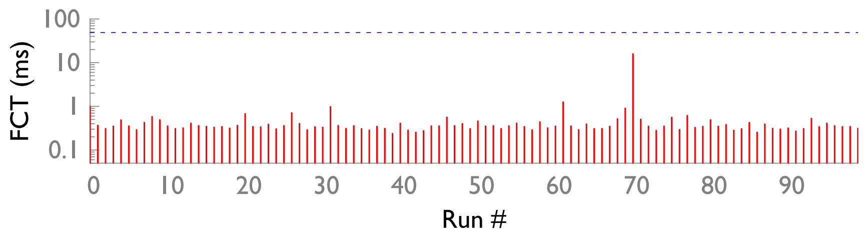


Figure 6.2: FCT of the short flow across 100 attempts of replay with socket calls. The blue dashed-line is 49ms.

The root cause of this problem is that delayed ACK is very risky in the presence of RTO, because after RTO the sender can only send one packet. Ideally, the receiver needs a way to identify retransmissions (e.g., the sender marks the retransmitted packets), so it does not delay the ACK for them. As a workaround today, reducing the delayed ACK timeout can mitigate the problem.

6.2.2 Butterfly Effect

While deterministic TCP replay is a powerful tool for diagnosing TCP performance problems, it is not easy to ensure determinism. For the above example, if we simply replay with the same socket calls at the same times as the runtime, we cannot reproduce the problem.* Figure 6.2 shows that when we replay 100 times, the short flow always has way less than 49ms flow completion times (FCT). In the production where there are more flows and more dynamic traffic than our testbed, it is more difficult to reproduce the same problem.

The key challenge for the deterministic replay is the butterfly effect. Packet sending times at hosts often have microsecond-level variation between the replay and the runtime. This is caused by the inherent host non-determinism, such as the clock drift, context switching, kernel scheduling, and cache state[126].

*We synchronize the clocks among the senders and receivers to 100s of nanoseconds precision by PTP (Precision Time Protocol[20]).

The small variation gets amplified by the butterfly effect—the closed loop interactions between switches and TCP. A small packet sending time variation may change the order of packets from different hosts at a switch, which causes *switch action variations*—the switch may drop or mark ECN on a different set of packets. This starts the butterfly effect in the closed loop between switches and TCP: *Switch action variations* cause *TCP behavior variations* (e.g., TCP changing congestion window size differently). TCP behavior variations change its flow sending rates, which affect the queue lengths at all the switches the flow traverses ever since and lead to more switch action variations. Such a chain reaction between switches and TCP affects more and more flows all over the network in multiple rounds.

One may expect that reducing the sending time variation (e.g., better clock synchronization, more deterministic packet processing time) can improve the replay accuracy. However, our experiment shows that even a nanosecond of variation can lead to completely different packet-level behaviors.

We run an ns-3 simulation[50] to control the sending time variation. We use the same topology and traffic as in §6.2.1. For the runtime, we set the host packet processing delay to 10μs, the same as what we measure in the testbed. The short flow incurs a long flow completion time because of the correlated RTO and delayed ACK. We then replay the experiment with the same socket calls and timings. To simulate different levels of sending time variation, we simulate a normal distribution of host packet processing delay with the same mean delay of 10μs but with a standard deviation ranging from 0 to 1000ns. For each level, we replay 100 times.

Figure 6.3 shows the percentage of replays that reproduce the correlated RTO and ACK delay on the short flow. Once the sending time variation exceeds zero, even just 1ns, the probability of reproducing the same problem suddenly drops.

This is because with a non-zero sending time variation, there is always a chance that a switch takes different actions on a packet between the runtime and the replay. Smaller timing variation

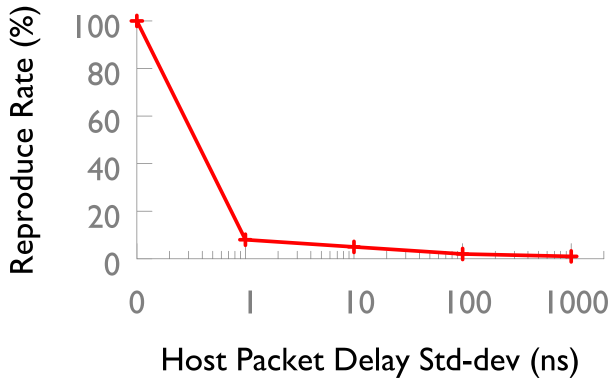


Figure 6.3: The rate of reproducing the correlated RTO and ACK delay.

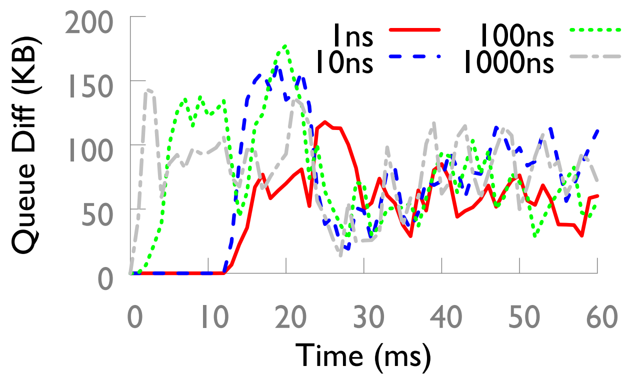


Figure 6.4: The time series of queue length difference.

can only delay the appearance of different actions, but cannot prevent it. Once the switch takes a different action, the butterfly effect starts, causing a chain reaction of changing sending rates and queue lengths. The chain reaction persists regardless of the level of the sending time variation.

Figure 6.4 illustrates this. We show the time series of queue length difference between runtime and replay experienced by each packet. For each level of sending time variation, we show a typical one of the 100 replays. For 1ns variations, although the queue length difference starts later than with higher variations, once the difference starts at 12ms, it never goes down to 0.

This result indicates that we cannot simply rely on reducing the sending time variation. This motivates our Deter design, which decouples the TCP and the network so that switch action variations cannot affect TCP.

6.3 Deter Design

In this section, we discuss Deter design with four key ideas: first, we break the butterfly effect by replaying individual TCP connections separately and record TCP's interactions with the application and the network. Second, to deterministically replay each TCP connection, we record all the non-determinisms that happen in the interactions between TCP and the kernel. Third, we introduce a rate-based sampling solution to reduce the overhead of recoding packet sending times. Finally, with the packet traces of all the connections, we show how to replay switch queuing behaviors.

6.3.1 Breaking the Butterfly Effect

We break the closed loop between TCP and the switches by replaying individual connections separately. We identify the minimal set of signals that capture all the interactions of a TCP connection with the application and the network.

TCP interacts with applications through socket calls. Deter captures all socket calls and its input arguments such as the number of read/recv bytes and socket flags.

TCP interacts with the network through packets. TCP sends packets into the network and receives packets from the network. We do not need to record most incoming packets because we replay the sender and the receiver of a connection together and they can automatically generate packets for each other. We only need to record how the switches inside the network change the packet stream such as dropping packets or marking them with ECN bits. At the receiver, we detect packet drops by checking if the IP_ID fields are continuous and ECN by checking the ECN bits (see §6.4

for details) and record them there.

Note that for a TCP connection, it does not matter which switch drops or marks the packets. Only the final changes to the packets matter. So in the replay, we no longer need switches because their actions to packets have been recorded and we can just replay them. Since the switch actions are deterministically replayed, we break the butterfly effect.

A TCP connection interacts with other connections when they share switch resources in the network and cause different switch actions[†]. Since we recorded switch actions, we also isolate the interactions among TCP connections.

In summary, in the runtime, we record socket calls and switch changes to packets at all the hosts. Users can specify which connections to replay. To replay a connection, we set up a simple two-host testbed that runs as a sender and a receiver for every single connection without involving any switches. We run a socket call generator to generate socket calls at the right time and run a packet corrector to inject actions on packets before they arrive at the TCP sender and receiver. We can easily parallelize the replay of multiple connections because we replay each connection independently.

6.3.2 Handling Non-determinisms in the Kernel

The next question is how to deterministically replay a single TCP connection. It is complex to replay a general system[151,60], which requires record and replay lots of non-determinisms. We use the knowledge of TCP to design a customized replay for TCP, which is lightweight. Specifically, besides the interaction with the application and the network, TCP also has three non-determinisms from interacting with the kernel: the kernel may call TCP handler functions, the TCP may read kernel variables, and there is thread scheduling.

(1) TCP handler function calls from kernel: The kernel may call some TCP handler functions.

[†]We discussion TCP connections on the same host in the next subsection.

For example, the OS timer may call TCP timeout handler. The kernel may also call resume transmission handler, which sends more packets in the send buffer. We need to record them.

(2) Reading kernel variables: TCP reads a few variables that are updated by other kernel programs (or hardware), such as memory pressure indicator, the jiffies (a low-resolution clock), the mstamp (a microsecond-resolution clock), and the send queue byte count. We should record the return value of each read.

(3) Thread scheduling: TCP works in a multi-threaded environment. Different threads, such as applications, NIC interrupts, and OS timers, access the shared socket variables by calling TCP handler functions. For example, an application thread calls a socket call handler to copy data into the socket send buffer; a NIC interrupt may call the TCP receive packet handler to frees up some space of the send buffer; OS timer may call the timeout handler to send a pending packet in the send buffer. It is important to ensure the order of different threads accessing the same variable. Fortunately, TCP uses a single socket lock to ensure that only one thread can access all the shared variables at a time. Thus, we just record the order of lock acquisition of different threads by giving a sequence number for each lock acquisition.

In the replay, we run the same TCP stack with the same TCP configuration as the runtime. In addition to the socket call generator and the packet corrector, we also generate handler calls from the kernel based on the recorded logs. We feed in the recorded kernel variables when TCP reads them. We also enforce the order of lock acquisition of different threads (see §6.4 for more details).

6.3.3 Sampling Packet Sending Times

So far we have ensured the ordering of TCP behaviors (e.g., the sequence of packets, state updates, loss detections, timeouts). One remaining question is how to replay packet sending times accurately. Recording the sending times for all the packets takes high storage overhead. To reduce the overhead,

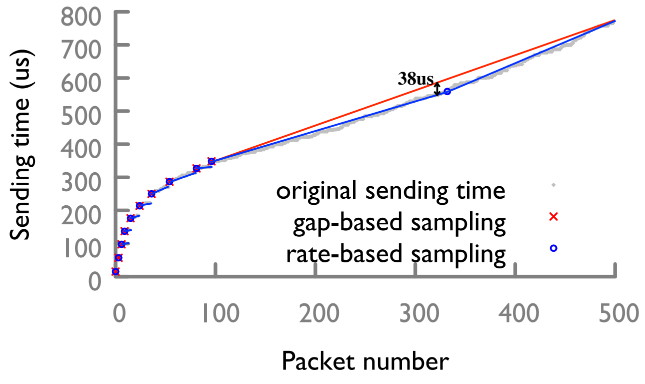


Figure 6.5: Inference error of sampling approaches. (The lines indicate the inferred sending time of each packet. The end points of the lines are sampled sending times.)

we choose to sample packets, record the sending times for sampled packets, and infer the times for the other packets. The question is how to select the samples in real-time while bounding the inference error within a given threshold *th*.

Strawman solution: gap-based sampling. TCP usually sends packets in bursts. So intuitively for each burst, we can keep the sending time of the first packet and the burst length. Assuming all the packets in the same burst follow the same sending rate, we can then infer the sending times of all the unsampled packets. We can identify packets in the same bursts if their interarrival time is below a threshold.

We perform a simple experiment to show that this approach has an unbounded error. We send two flows from two senders through a shared 10Gbps link. The second flow starts 500μs after the first flow. Figure 6.5 shows the packet sending time series of the first flow. All the packets from the 96-th to the 499-th are in the same burst (i.e., no gap of packet sending time), but the rate changes. As a result, the inferred sending time of the 323-th packet is 38μs later than the actual time.

Our solution: Rate-based sampling. Gap-based sampling fails to sample packets when the packet rate changes. Therefore, instead of recording the burst length, we propose to record the packet rate. When the inferred sending time based on the recorded packet rate is wrong (i.e., the difference with the actual time is above the threshold th), we sample a new packet. We set th to 5μs by default.

Specifically, in the runtime, we follow Algorithm 12. s is the previous sampled packet and p is the new packet. Given the sending time of s ($s.time$) and a packet rate r, we can infer the sending time of p ($p.time$). In reverse, to ensure that our inferred sending time of p falls in the range of $[p.time - th, p.time + th]$, we must ensure our recorded packet rate r falls in the range of $p\_range = [\frac{p.index - s.index}{p.time + th - s.time}, \frac{p.index - s.index}{p.time - th - s.time}]$ (Line 2). Thus, we compare the recorded rate range rec_range and p_range. If they overlap, it means we can find a rate, in the intersection of rec_range and p_range, that can be used to infer a bounded sending time for both p and all the previous packets between s and p. Thus, we do not need to sample p (Line 4). Otherwise, if the two ranges do not overlap, we sample p, record a rate in rec_range, and reset rec_range (Line 6-7).

DETER can generate the full packet trace for each connection, by combining the recorded (inferred) sending times with the packets generated by the replay of TCP execution.

Algorithm 12 DETER Sampling sending time. $p.index$ is its index within its 5-tuple flow, and $p.time$ is its sending time.

```
1: procedure SAMPLE(p: a new packet)
2:     $p\_range = [\frac{p.index - s.index}{p.time + th - s.time}, \frac{p.index - s.index}{p.time - th - s.time}]$
3:     if $p\_range \cap rec\_range \neq \emptyset$ then
4:         $rec\_range = p\_range \cap rec\_range$
5:     else
6:         record($s.index, s.time, rec\_range.mid$)
7:         $s = p$; $rec\_range = [-\infty, \infty]$
```

6.3.4 Replaying Switch Queues

Because we can get all the packets, their sizes, and sending times for each connection in the network (§6.3.2 and §6.3.3), we can use them to replay switch queues in simulators (e.g., ns-3[50]) by pushing all the packets at the right time into the network. Replaying switch queues can help us understand the interaction between different connections at the switches (e.g., which flows contend for the queues).

The simulator needs the same topology and switch data plane (e.g., forwarding tables, buffer sharing policies, switching delay, and link propagation delay) as the runtime. Today, many vendors build high-fidelity simulators for their own devices[25,41,31]. One can also choose to replay switch queues in a physical network if available. Replaying switch queues also requires that host clocks are tightly synchronized during the runtime, so that the relative packet sending time error across hosts are small. Precision Time Protocol[20] is widely available and can synchronize clocks to submicrosecond level. With Sundial[127], we can tightly bound the clock precision at $\sim$100ns.

Replaying the exact queueing behavior is both impractical and unnecessary. It requires recording the exact order of enqueue and dequeue, which is too heavy for the runtime. On the other hand, it is often good enough to show the contending flows and their occupancies with high accuracy.

Thus, we opt for a simple design that can achieve high accuracy. We simply push all the packets into the network at the right time. It can achieve high accuracy because the switch queue occupancy is a continuous function with respect to packet sending times. Since the difference in packet sending times between the runtime and the replay is bounded, the difference of switch queue occupancy is also bounded. Specifically, suppose a packet's arrival time at a port differs by k packets transmission time, and the fan-in of that port is f, the queue difference is at most $(f-1)k$. k is small because our sampling bounds the sending time error to 5μs, and there are limited hops to amplify it. f is also small because the destinations of flows traversing a switch are random[‡]. Even if f is large, such as during incast, the queue occupancy is also large, so the difference is a small fraction of the queue.

However, one exception is packet drops. Because dropping packets or not is a binary decision (not a continuous function), even if a microsecond level difference can cause different drops. Specif-

[‡]In theory, the fan-in is within 4 for 99.7% of the time for a 64-port switch with random traffic.

Table 6.1: Runtime recorded data.

Type	Data recorded
Interaction w/ network	losses, ECN, reordering
Interaction w/ applications	socket calls
Handler call from kernel	Timeout handler, resume transmission handler, packet receive handler
Kernel variables	Infrequently updated variables, e.g., jiffies, memory pressure indicator
	Influence of frequently updated variables e.g., RACK loss detection
Order of lock acquisitions	sequence number of lock acquisitions for diff. threads
Timestamp samples	Sampled packet sending time (time, index, rate)

ically, a runtime dropped packet may get through, which we call a *false-accept*. It also occupies some free space in the queue, leaving less space for later packets that should be in the queue, so one of the later packets may get mistakenly dropped, which we call a *false-drop*.

We propose to reduce the probability of false-accepts and false-drops by letting the hosts tag *should-be-dropped* packets. In this way, we ensure that the switches only drop packets with tags (for eliminating false-accept) and always deliver packets without tags (for eliminating false-drop).

The key challenge is how to know which switch to drop the tagged packets. Since the switch queue occupancy is a continuous function, it has bounded differences with respect to the sending time difference. We propose to decide whether to drop packets at a switch based on the switch's queue occupancy upon packet arrivals. That is, when a *should-be-dropped* packet arrives at a switch, and the queue occupancy is above a threshold (e.g., > queue max length - 5 MTU), the switch drops the packet.

When a packet only experiences one congested switch on its path, which is the most common case, our solution works well. In the rare case when there are multiple congestion spots on the path, Deter may drop the packet at a wrong location. Our evaluation shows that this heuristic reduces the error of dropping packets[§] from 58.3% to 2.87%.

[§]Percentage of false-accept, false-drop, and drop at wrong location in all drops.

6.4 Implementation

In this section, we discuss the implementation details of DETER. We just need 139 extra lines of code in the Linux kernel. Then we accomplish the record and replay with two kernel modules and two userspace programs (3000 lines of C and C++ code in total).

Runtime recording. For each connection, we first record its configurations, and then record the data listed in Table 6.1 during its runtime. We note that the configurations of connections on the same server are mostly the same, so we only record the parameters that differ from the default values. Our current prototype starts the recording after the connection is successfully built[¶]. We now discuss the runtime recording.

Interaction with the network: This includes packet drops, ECN, and packet reordering. In our design, we use the IP_ID field to detect packet drops: Linux sends packets of each connection with consecutive IP_ID values, so the receiver can check if there are gaps in the series of incoming packets to detect drops (Similarly, the sender can detect drops in the incoming ACKs)[‖]. On other platforms that do not have the consecutive IP_ID feature, we can use LossRadar[130] to detect drops, which only takes O(#loss) space. The host also checks the ECN of the IP header of each incoming packet, and record 1 bit (CE) for it. Sometimes there may be packet reordering, which we can detect also using the IP_ID field.

Recording the interactions with the network is lightweight. In datacenters, the packet drop rate is just 10^{-5} to 10^{-4}[162,96]. For ECN we just need 1 bit per packet. Reordering is rare, so it does not cost much. We instrument the TCP receive packet handler to record them.

Socket calls from the application: We hook the TCP socket call handler functions to record the

[¶]Record and replay for connection setup is not very different. The only difference is detecting the drop of the first packet (SYN and SYN-ACK). This can be solved by recording the IP_ID of all SYN packets at both sender and receiver, which just adds 8 bytes for each connection.

[‖]This is different from TCP's drop detection: TCP sender does not distinguish drop of a data packet or its ACK. We must distinguish them because both the sender and the receiver must replay accurately.

#bytes and flag, so that we do not need to change the application.

We can reduce the storage overhead of socket calls a lot. We find that there are often identical socket calls. For example, distributed files systems break large files into fixed-size chunks, so most of the send and receive sizes are the same. Thus, we store all the common patterns of socket calls (the common #bytes and flag pairs) for different applications, and only record the pattern numbers in the runtime. DETER associates connections to applications via their TCP port numbers.

Other TCP handler calls from the kernel: We hook the timeout handlers and the resume transmission handler, and record them when they get called.

Kernel variables read by TCP: We record the memory pressure indicator and jiffies with low overhead because their values change infrequently. The memory pressure indicator is very rarely set, and the jiffies increments by 1 every 4ms. So we just maintain the values of the last read and only record the reads that return a new value.

The mstamp and the send queue byte count are updated frequently. We reduce the overhead by recording their *influences* instead of their values. Specifically, the variables influence the TCP executions by serving as the metrics of if-conditions in TCP. For example, TCP uses the mstamp to detect losses (RACK[38]). We just need to record the loss detection result, rather than the actual value of the clock. We identify and record all the if-conditions they affect (1 bit for each), which relates to loss detection, cwnd reset, TCP segmentation offload, and TCP small queue. Moreover, most of the if-conditions have a dominant result (e.g., loss detection mostly return false), so we reduce the overhead further by only recording when they have the uncommon result.

We use a special reader function to record these values. For example, in the TCP code, we replace a=jiffies with a=reader(jiffies) to record the value of jiffies and replace if (mstamp>b) with if (reader(mstamp>b)) to record the influence of mstamp. The reader function simply records the value passed to it and returns this value.

Lock acquisition: We instrument TCP's lock acquisition function to record which thread calls this function, so we know the order of lock acquisition by different threads. We also optimize the overhead. Specifically, one thread may acquire the lock many times consecutively. For example, NIC interrupt acquires one lock for each incoming packet, so there are often tens of lock acquisition by NIC interrupt in a row. Therefore, we record the number of consecutive lock acquisitions, instead of recording them individually.

Sampled sending times: To get the most accurate timestamps, we sample and record the sending times in the NIC driver, just before TCP pushes packets into the NIC ring buffer.

Replay. We now discuss the replay.

Replay TCP stacks. Figure 6.6 shows the replay setup. We implement the *packet corrector* with NetFilter[27]. It injects drops and CE bits to the incoming packets[**]. We also enforce the reordering here.

To replay the socket calls, we implement a *socket call generator* in the user space to inject socket calls from the applications according to the log.

We also implement a *TCP handler caller*, which is a kernel thread that calls TCP handler functions according to the log. The handler functions include the packet receive handler, the timeout handler, and the resume transmission handler. When calling the packet receive handler, it gets a packet from the packet corrector as an argument to the handler.

To enforce the order of different threads acquiring the lock, we implement a *sequencer*. It knows the order of different threads acquiring the lock based on the log. We instrument the lock acquisition function to check with the sequencer before it actually acquires the lock. If the current thread is not the next to acquire the lock, it waits for other threads until itself is the next to acquire the lock.

We reuse the reader function that we introduced before to feed the recorded kernel variables or their influences. During the replay, the reader function reads the log and return the corresponding

[**]We require no packet drops before packets entering the packet corrector, so we must make sure no packets get unexpectedly dropped in the queues on the hosts (e.g., NIC ring buffer, softirq queue, qdisc) during the replay. We can set the sizes of these queues large enough to avoid unexpected drops.

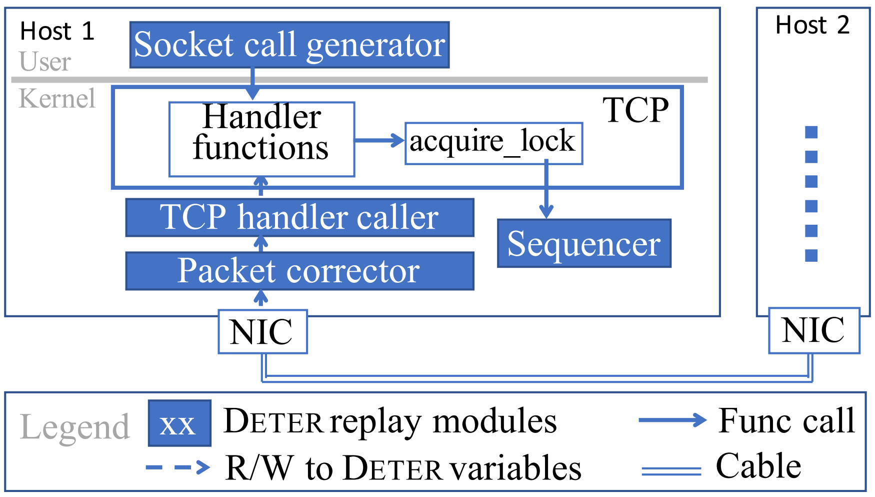


Figure 6.6: Replay implementation in Deter

value.

Replay sending and receiving timestamps. We only record packet sending times for replay. We can infer receiving times from sending times: for the received packet which triggers a new packet to send, we can estimate its receiving time as the sending time of the new packet minus the average packet processing time, which is measured separately. For the received packet that does not trigger a new packet, its gap with the previous received packet is close to their sending time gap, because they experience similar network conditions. Note that only the sending times affect the switch queue replay, but not the receiving times.

Switch queue replay. We run Precision Time Protocol[20] in our testbed. We implement the switch queue replay in both testbed and simulation. For the testbed, we implement a DPDK packet generator that reads the packet trace, tags packets, and sends packets to the NIC at the right time. We use a NetFPGA-based switch to implement the drop accuracy improvement (§6.3.4). It is also implementable in P4[13]. We also implement the replay in a packet-level simulation in ns-3[50], with the same topology, link delay and bandwidth, switch queueing algorithm, and routing state as the testbed.

6.5 Evaluation

In this section, we demonstrate the benefits of deterministic replay in DETER by showing how we diagnose TCP performance problems in a Spark application with 6.2K connections, the tail latency problem in an empirical web search workload with incasts, and example problems in a local testbed.

We also measure the CPU and storage overhead of DETER recording and the accuracy of DETER replay. Our evaluation shows that DETER only uses 2.1~3.1% compared to fully compressed packet traces and requires 0.094%~1.49% of CPU overhead. DETER also fully replays the sequences of packets at hosts and replays switch queues with lower than 1 MTU differences on average.

6.5.1 Diagnosis in Spark

Evaluation setting. We run a TeraSort job in Spark[45] that sorts 200GB data on 20 servers connected with 10Gbps network in Amazon EC2[40]. We use 4 executors (i.e., 4 cores) and 20GB memory on each server. The NIC MTU is 1500B. We enable TCP segmentation offload, and disable generic receive offload[††]. We run DETER on all servers to record data for all connections during the runtime and also run Tcpdump[39] to collect the packet traces as the groundtruth.

Diagnosis. We can use DETER to identify and diagnose tail latency problem in Spark. We define each flow as all the packets belonging to the same Spark message. Spark usually sends one large message with multiple socket calls. So if a socket call starts after all the previous packets are acknowledged, we treat the socket call as a new message. Otherwise, we treat it as part of the previous message.

[††]We have not implemented replay for it, but it is not hard (§6.7).

Table 6.2: Reasons for 99.9-th percentile latency for flows of different sizes in Spark.

Flow size (MB)	<0.1	[0.1, 1]	[1, 10]	>10
RTO	8	3	4	0
FR	74	0	0	0
Delayed ACK	0	0	18	0
Rwnd=0	0	0	1	1
Slow start	0	0	1	0

We find that the tail latency of flows from HDFS are mostly caused by receiver limit, because their receive windows frequently reach zero.

The 99.9-th percentile latencies for flows between Spark workers experience a variety of problems as summarized in Table 6.2. For flows shorter than 1MB, their tail latency are mostly caused by packet drops (RTO or fast retransmission (FR)). For flows longer than 10MB, their tail latency are mostly caused by receive window frequently reaching zero (Rwnd=0).

The flows in the range [100KB,1MB] are of particular interests, because most of their tail latencies (18 out of 24) are caused by multiple delayed ACKs. We show the sender-side packet trace for one of them in Figure 6.7; others have similar patterns. The sender frequently gets blocked after sending a burst of packets, until around 40ms later when the ACK comes back. Such burst-40ms-ACK pattern repeats multiple times and causes excessive delay. This is out of our expectation, because the receiver should acknowledge every two data packets.

With Deter, we can easily replay the problem, and use TCP Probe to print the variables that decide whether to delay the ACK. We check many of these connections, and find that TCP explicitly delays the ACK because the free space in the receive buffer is shrinking. This suggests that the root cause is the application not reading the data in the receive buffer in time. Our guess is that the application is busy with processing data, so the CPU is the bottleneck in this case.

Deter can help us effectively diagnose the problems caused by the network (e.g., RTO, fast retransmission). In addition, it can also help us identify problems caused by applications. This is help-

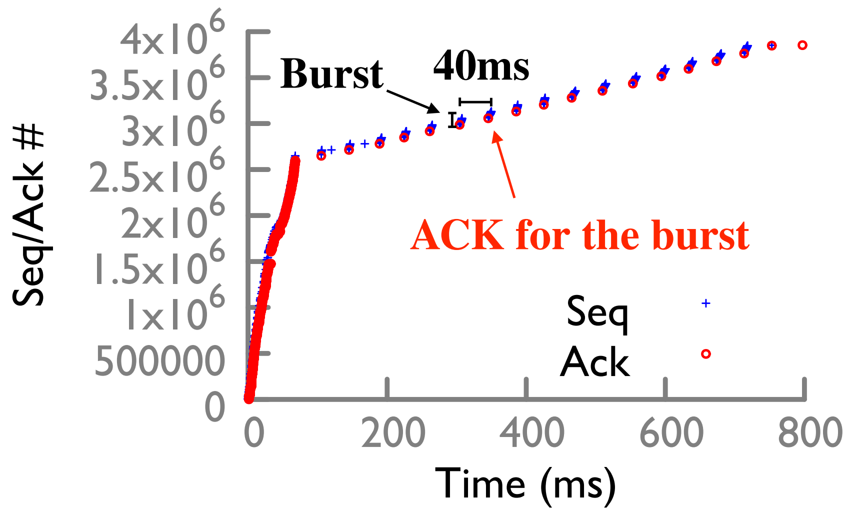


Figure 6.7: A flow with delayed ACKs.

ful because in datacenters it is often unclear where the performance bottleneck is, and blaming the network is often the first reaction[61]. Unlike previous systems that infer the bottleneck[137,61], DETER can help us quantify the duration of different bottlenecks without instrumenting the applications.

Overhead. DETER records a total of 200.6MB data in the runtime. For comparison, Tcpdump uses 22.4GB to record only the IP and TCP headers and timestamps. Even if we apply the state-of-the-art resource-efficient solution for packet compression[98] to Tcpdump, the storage will reduce to 6.5GB[‡‡], compared to which DETER storage is still only 3.1%.

If we keep using DETER to monitor a datacenter that continuously runs such Spark jobs, DETER storage overhead translates to 2.8GB/host/day. We can delete the data every day if we do not see performance problems.

We also use Linux perf[29] to evaluate the CPU overhead of DETER recording. DETER uses 0.094% of total CPU time.

[‡‡]Since we don't have NetSight's code, we calculated based NetSight's reported value. The same calculation is used in other experiments.

Table 6.3: Reasons for 99.9-th percentile latency for flows of different sizes in datacenter workload.

Flow size (MB)	<0.1	[0.1,1]	[1,10]	>10
Congestion	149	35	25	2
Late FR	29	27	0	0
ACK drops	0	2	0	0
Tail drops	4	1	0	0
RTO	2	1	2	0

6.5.2 Diagnosis in Datacenter Workload

Evaluation setting. We now diagnose TCP tail latency problems under empirical workloads modeled after traffic patterns that have been observed in production datacenters. We run a client-server RPC call software[26] in the same 20-node Amazon EC2 testbed. The clients set up a persistent TCP connection to each server, and request flows according to Poisson process from a random server. We set the flow sizes following the distribution observed in a production datacenter running web search applications[57]. We also add incast traffic pattern, by having the client simultaneously request 10 random servers, so the 10 servers respond synchronously causing incast. We set the average request rate to have an 80% network load, and 20% of the load is incast traffic. We generate a total of 280K requests over 380 persistent connections. All 20 nodes run both client and server.

Diagnosing tail latency. In Table 6.3, we classifies the root causes into five categories: congestion (i.e., low throughput), the fast retransmission happens very late (late FR), ACK drops (so the sender gets stuck), tail drops (so the packets at the end of a flow get dropped), and RTO.

We analyze the short flows (100KB-1MB) with latency above 99.9-th percentile as an example. At the 99.9-th percentile, flows experience 173.8 slow down of completion time compared to the case of running the flow alone. We make the following interesting observations:

RTO is not the main root cause of tail latency. A widely discussed reason for tail latency is RTO[71,172]. But actually RTO is rare in this experiment. The reason is that when there are multiple requests in the same connection, later requests can help recover the packet losses of previous requests (by triggering fast retransmission), so TCP loss recovery is effective in this scenario.

Fast retransmission (FR) is delayed for 10s of milliseconds. When these flows experience loss, the senders start FR after 10s of duplicate ACKs (dupACKs). This is unexpected because the normal threshold for FR is 3 dupACKs. And this is bad because short flows usually do not have so many dupACKs. In fact, most (22 out of 27) of these flows do not have enough dupACKs on their own; their FR starts 10s of milliseconds later when another request in the same connection starts and triggers more dupACKs.

With DETER, we can replay repeatedly and gain more insight into the problem. For one of the connections that experiences the late FR problem, we use TCP Probe[34] to print out the threshold for dupACKs (tp->reordering) on every ACK's arrival. We find that this threshold starts at 3, but later increases (and never decreases), so when the flow that experiences late FR arrive, the threshold is 45. We search in the TCP code, and find the threshold only increases when TCP detects reordering.

A quick fix is to set the upper bound of this threshold (net.ipv4.tcp_max_reordering) lower, but it risks spurious retransmission in the presence of reordering. A potential optimization to TCP may be regularly reducing the threshold.

Overhead. Tcpdump records 16.8GB, or 4.9GB if with compression. DETER records a total of 103.8MB, which is 2.1% of compressed size.

The CPU overhead is 1.49%. The overhead is higher than in Spark, because the client-server software only uses CPU to send and receive data, without any data processing. In fact, it spends 99.78% of its CPU time in the networking stack (including DETER). So 1.49% is very close to the lower bound of DETER CPU overhead.

6.5.3 Diagnosing RTO in a Testbed

RTO usually has a large impact on the latency. However, there are many different causes of RTO, and often involves different parameters. In §6.2.1 we have shown one case. Here we show two other causes for RTO that we see in our testbed. In all cases, TCP counters can only be the first step—knowing that timeouts and packet losses happen. But it is very hard to realize the relationship between the timeout and other events. With DETER, we can replay the connection to get the packet traces and trace the TCP execution to dig out the root cause.

Evaluation setting. We use 3 hosts connected through a single switch via 10Gbps links. We pick two of the hosts as senders and the rest one as the receiver. Each of the senders sends one long flow (10MB) to the receiver. One of the senders also sends a short flow (10KB) to the receiver.

Root cause 1: not enough dupACKs. In this case, the short flow experiences RTO. We can use DETER to replay the connections and capture the packet trace. The trace shows that the short flow sends 7 packets in the first round, and the 5-th packet gets dropped. Thus, although the 6-th and the 7-th packets trigger dupACKs, the number of dupACKs is not enough to trigger fast retransmission.

Root cause 2: setting large TCP receive buffer size. The receive buffer size is a frequently tuned parameter for networks with different bandwidth-delay products. For example, an inter-datacenter connection with 100ms RTT and 1Gbps bandwidth need 12.5MB buffer size. Unfortunately, a large receive buffer can cause RTO issues. Here we show the diagnosis in an example with 10MB receive buffer. The SACK is not turned on.

In the packet trace, we find after a packet loss, there are more than 3 dupACKs, but the sender does not fast retransmit the lost packet. This is unexpected because just 3 dupACKs should trigger fast retransmission. We first suspect that this may be the late FR case that we show in §6.5.2, but it is not the case.

To dig out the root cause, we use Ftrace to get the function call graph of handling each ACK.

Surprisingly, We find that TCP does not go to the dupACK branch. This means TCP even does not treat them as dupACKs. So we use TCP Probe to print the variables that are used to classify ACKs as duplicates. The flag variable reveals the reason: TCP does not treat the ACKs as duplicate because the flag's WIN_UPDATE bit is set[30]. This means each of these ACKs carries a different window size. We confirm this in the packet trace: each ACK carries a larger window size.

The direct cause for this problem is that the receive buffer size is very large. The receive window starts with a small size, and increases two MSS per received data packet until reaching configured receive buffer size. Thus, the window size keeps growing throughout the lifetime of this connection. However, this also suggests a potential optimization to TCP that it should have a smarter classification for dupACKs.

6.5.4 Evaluating Switch Queue Replay

Now we evaluate the accuracy of replaying switch queues in our testbed and simulation. We first run traffic in our testbed, and replay the queue to evaluate the accuracy. Then to understand how the switch queue replay works under more switches and more congestions, we run empirical traffic in a large scale simulation, and replay the queues.

6.5.4.1 Evaluation with testbed

Evaluation setting: The testbed comprises 3 hosts. To get the groundtruth of the queue content, we use a NetFPGA switch and program it to send out the queue content through the unused port. The switch has a total of 393KB buffer shared across 3 ports[§§]. The MTU is 1500B. The host clocks are synchronized with 100s of nanoseconds precision by Precision Time Protocol[20].

Because congestion is the most challenging scenario to replay, we set traffic to have severe con-

[§§]We use the commonly used dynamic threshold[75] with $\alpha = 4$.

gestion. We use 2 hosts as senders and the rest one as a receiver. Each of the two senders generates 2 long flows (10MB each) to the receiver simultaneously. Each sender also generates 4 short flows (10KB each) to the receiver, one every 5ms. So there are a total of 4 long flows and 8 short flows.

During the runtime, we use DETER to collect data, and also collect the content of the congested queue. Then we first replay each connection to get the packet trace, and replay the queue. We replay the queue in both the original testbed, and in a simulation. The simulation has the same topology, and simulates the same link throughput, latency, and buffer setting as the NetFPGA switch.

Accuracy: The metric we use is *queue content difference*: the difference between the runtime queue q_{run} and the replay queue q_{rep} that each packet sees. Formally, we define

$qdiff = \sum_{f \in q_{run} \cup q_{rep}} f.size_{run} - f.size_{rep}$

where $f.size_{run}$ means the bytes of flow f in the queue during the runtime and $f.size_{rep}$ is for the replay.

On average the queue content difference is 0.57 MTU in the testbed, and 1.0 MTU in the simulation. On the 99-th percentile, the difference is 4.83 MTU in the testbed, and 3.85 MTU in the simulation, both of which are very low compared to the buffer size. Replay in the testbed has a slightly higher tail difference because timing variations (e.g., thread scheduling) exist in the testbed, but not in the simulation.

6.5.4.2 Evaluation in large-scale simulation

Our testbed evaluation shows that the replay is effective for one switch. In production, there are more hosts, multiple layers of switches, and more congestions across the switches. So we use simulation to evaluate a larger-scale network.

Evaluation setting: We run the simulation in ns-3[50], with 320 switches and 1024 hosts connected through a K=16 Fat-Tree with 10Gbps links. Each switch has 2MB buffer, shared by all its

16 ports[¶¶]. To simulate the clock synchronization error, we add a delta to each host's clock, with a uniform distribution between 0 and 5 us[‖‖].

The traffic includes both empirical background traffic that follows the flow size distribution of a web search workload[57], and incast traffic. The source and the destination of each background flow are chosen uniformly random. The flow arrival rate follows a Poisson process, and we vary the flow arrival rate to achieve different levels of traffic load, from 10% to 80%. We also generate the incast traffic by having the client simultaneously requests 40 servers, each of which sends back 250KB response (10MB total response size). We generate 2400 incast per-second.

To understand how the sampling affects the accuracy, we sample the sending times with different threshold of error: 2μs, 5μs, and 10μs. We then replay the queues.

Accuracy: Figure 6.8 shows the queue content difference of all queues in the network. The difference increases mildly with higher load, both on average and at the 99-th percentile. For example, with 5μs threshold of error, at 30% load, the maximum load of most datacenters in practice[156,36], the differences are 0.78 MTU on average and 5.7 MTU at 99 percentile. At 80% load, an extremely high load, the differences are 1.7 MTU on average and 9.7 MTU at the tail. It also shows that a 5μs threshold achieves relatively good accuracy: it only increases less than 0.3 MTU (on average) and less than 1.8 MTU (at tail) difference compared to 2μs.

We also compare the packet drop error with and without the drop accuracy improvement. The drop error is $\frac{\#false\_accept + \#false\_drop + \#drop\_wrong\_location}{\#packets\_dropped\_in\_either\_runtime\_or\_replay}$. Our evaluation shows that the drop error reduces significantly. For example, for 5μs sampling threshold at 30% load, the error reduces from 58.3% to 2.87%.

The drop error is low under various loads, from 2.52% at the 10% load, to 3.81% at the 80% load.

[¶¶]For the buffer sharing policy, we use the commonly used dynamic threshold[75] with $\alpha = 4$.

[‖‖]PTP in LAN can achieve submicrosecond accuracy, and under 3.2μs in WAN most of the time[32]. More advanced clock synchronizations[124,91] guarantee submicrosecond accuracy. We choose 5μs to be conservative.

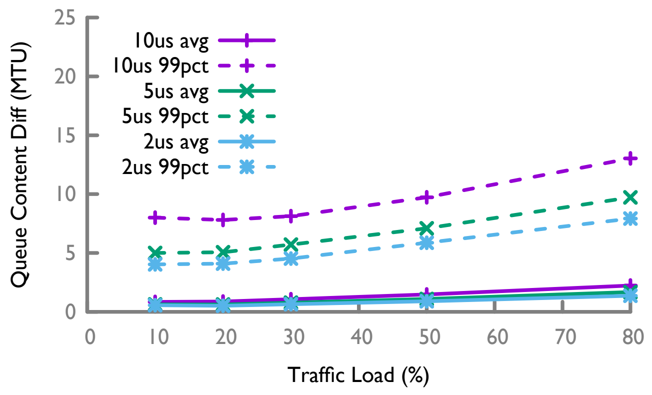


Figure 6.8: The queue content differences of replay in simulation.

There is no false-drop, as the simulation can avoid this (§6.3.4). Most errors are false-accepts. Only less than 0.37% of the drops show up at wrong locations, which means we can trust the drops in the replay with high confidence, because only 0.37% of them give wrong locations. Since 80% load is extremely high and we also added incast traffic, we believe most datacenters would not stress the network at this level, so we believe the drop error rate is low in general.

6.5.4.3 Diagnosing RTO using queue information

Sometimes RTO can be caused by the queuing mechanisms of switches. We run the traffic in a 4 host (A, B, C, D) testbed. B and C respectively send 5 long flows (500MB each) to A. In the middle of the long flow transmission, A, C and D respectively send 5 short flows (100KB each) to B simultaneously. Two of the long flows from C to A experience RTO. Using DETER to replay them, we find that they both drop a whole window of packets, at the same time. But this time we cannot find any problem in the TCP stack. So we use the packet traces for all the connections to replay switch queues in an ns-3 simulator.

During the replay, we collect all the enqueue and drop events at the switch. The packets are dropped at queue 0 of the switch. Figure 6.9 shows the length and the cumulative drop count of

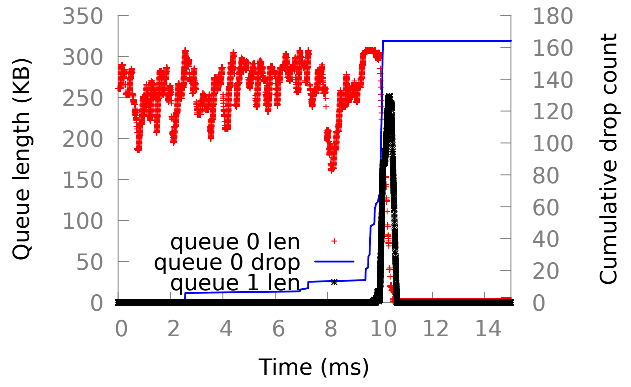


Figure 6.9: The lengths of two queues that share the buffer.

queue 0. At around 10ms, there is a sudden increase in drops. Unexpectedly, the queue length is decreasing at the same time. We suspect that the switch buffer sharing[75] causes this problem, because the threshold of a queue decreases when the total buffer utilization of the switch grows.

So we replay again and monitor other queues of the switch. We find that a burst of packets builds up queue 1 at the time of queue 0 drops packets. This confirms our hypothesis.

This problem could also happen in datacenters because most datacenter switches use shared memory across different queues. The threshold of any queue is proportional to the total free buffer size. If the switch buffer utilization suddenly increases, the threshold shrinks, which causes temporary blackhole at the almost-full queues (e.g., queue 0 in Figure 6.9). The sudden increase in switch buffer utilization can happen because of incast, which is common in datacenters.

6.6 Related Work

Replay systems. There are many replay systems for kernel, multicore applications and distributed systems[165,115,89,151,60,97,37]. They record the input and interaction of the target of replay (a subset of components of the entire system) with the rest of the system to isolate the target, and then make sure the target itself replays accurately. There are two ways to directly adopt such replay techniques for TCP: (1) Replay each host's TCP stack separately. This means we should record every packet as they are the input to the stack, which is a significant overhead. (2) Replay the whole network altogether, including all connections and switches, which is very expensive and hard to get right as shown in §6.2.2. DETER customizes replay techniques for TCP: we replay each connection (a pair of TCP stacks), and only record the mutations to the packet stream in between (drop/ECN) to reduce the overhead of recording every packet, while avoiding replay the whole network together. We also introduce customized solutions to reduce the overhead of recording non-deterministic variables inside the TCP stack.

Monitoring tools in datacenters. Per-packet monitoring tools[39,33,98] and TCP execution tracing tools[34] provide detailed information for diagnosis, but running them continuously is too expensive. To reduce overhead, people collect coarser-grained information such as TCP counters[137,61] or per-flow stats on the host[166] or switches[8,129]. There are also query systems (e.g., Everflow[190], Trumpet[145], Marple[147]) that allow operators to specify the packets and events to capture in a network. DETER is complementary to these works in that it enables deterministic replay for debugging the same performance problem iteratively. DETER requires low recording overhead at runtime and allow operators to use all kinds of monitoring tools during the replay.

Other network-related replay. OFRewind[180] replays the switch control plane, while DETER replays TCP and the switch data plane. Monkey[73] and Swing[173] are tools that synthesize testing traffic based on the runtime recorded traffic pattern, while DETER focus on replay for diagnosis.

6.7 Discussion

Extension to other network transport features: Here are a few examples of transport features that may affect the replay.

Generic receive offload (GRO): If GRO[21] is enabled, we also need to record the way it merges packets. It just requires recording the number of packets being merged into one segment, which is available in the skb metadata and just costs 6 bits per merge. Usually each merging contains 10s of packets, so the overhead is low. During the replay, the packet corrector should also merge the incoming packets as recorded.

Delay-based congestion control (CC): Our current prototype is based on loss-based CC. To extend our solution to delayed-based CC, we need to record the timestamps that used for updating CC states. We can compress them a lot, because consecutive timestamps differ by a few microseconds most of the time, so we just need a few bits to record the delta.

RED in switch: RED randomly drop packets. Replaying the queues and drops may have a large error in this case, but replaying TCP connections is not affected. This shows the benefit of our design decision: decoupling the replay of each individual connection, so that it does not depend on switches.

Use cases of DETER. DETER is designed for ease of use. The only requirement is that the user turns on DETER on both endpoints of the connection, which is often the case for network operators and cloud tenants. Internet application developers can also use DETER for performance testing. Datacenter network operators may also benefit from replaying the switch queues, because they may have the network topology and switch data plane simulators.

Host stack changes. If the host stack changes, DETER may need to change accordingly, but it is not hard. First, Linux already abstracts CC out of basic TCP framework, so changes to CC does not need to recode DETER in the basic framework, which contains most of the recording. Besides, we have principles for what to record and replay (Table 6.1 and §6.4), so it would be easy to identify the required changes to DETER.

We expect the recording overhead would not change much with stack changes, because most of the overhead comes from socket calls and lock acquisitions, both of which are not sensitive to stack changes: socket call is determined by the applications, and most lock acquisitions are for receiving packets whose amount is determined by traffic volumes. The overhead associated with kernel variables is very small with our technique of recording their updates or influences, and we believe this benefit remains in the future.

Generality to other transport protocols. We believe the replay technique is general across different protocols. Basically, what other transport protocols do are not very different from TCP: reads from/writes to applications, sends/receives packets, and possibly controls sending rate based on packet measurement. Similar to TCP, we just need to record the interaction with the application and the network, and then make sure we handle the concurrency inside the protocol.

Network failures. Network failures (e.g., routing fluctuations or blackholes) do not affect DETER replaying the connections, but do affect DETER replaying the switch queues which assumes that the routing states are stable. However, network failures are themselves bigger problems than the problems related to switch queueing, and there are many other works focus on addressing such issues[190,96,132,179,129]. DETER is complementary to these works, because it helps to understand how TCP reacts to such conditions.

Storage overhead of socket calls. Usually the number of socket calls is much smaller than the number of packets. Production datacenter survey[57,94] shows that most network bytes are from large flows (>1MB), which usually mean large send/receive sizes. Moreover, even if an application has many short messages, the developers tend to batch them into a large one to reduce the CPU overhead. If some network does only have applications that generate small socket calls, recording every socket calls may be high overhead.

6.8 Summary

Deter enables deterministic TCP replay, which can reproduce performance problems, provide packet traces and support tracing of TCP executions. Deter eliminates the butterfly effect by replaying individual TCP connections separately and capture all the interactions between a TCP connection with the application and the network in a lightweight fashion. We demonstrate that Deter can be very effective in diagnosing a variety of TCP performance problems.

7

Conclusion

7.1 The Overall Picture

To provide high application performance, network control and telemetry play important roles. We design control schemes that can robustly provide high performance under dynamics, and telemetry systems that are both precise and fine-grained to aid in performance troubleshooting.

Applications need high-speed data transfer and fine-grained coordination across servers to achieve high performance. To this end, we design HPCC to robustly provide high throughput and speed-of-light latency, even under highly dynamic traffic. With HPCC, applications have more freedom in sending messages and scaling out without worrying about the performance. We also design Sundial to robustly provide $\sim$100ns precision bound under various types of failures in production. With Sundial, many distributed systems and databases can achieve high throughput and low latency of transactions, with strong consistency.

To troubleshoot performance issues related to networking, we design precise and fine-grained telemetry systems for both switches and host networking stack. We design FlowRadar and LossRadar to expose precise per-flow and per-loss information, so that we can pinpoint the location and culprits of many performance problems. We also design DETER to expose precise per-packet per-line-of-code information from the TCP stack, which helps debug many performance issues caused by TCP and its interaction with the network. We can continuously run the resource-efficient run-time monitoring of these systems, and easily zoom into the problems when necessary.

7.2 Broader Impact

HPCC has wide and high impacts. Alibaba cloud has already deployed HPCC. Beyond that, various switch and NIC vendors (Intel, Mellanox, Broadcom, Cisco, Innovium, Marvell, etc.) have supported HPCC in their recent products. Moreover, major cloud providers, switch vendors, and NIC vendors are pushing the standardization of HPCC: a recent IETF draft of HPCC was jointly written by Alibaba, Intel, and Mellanox, with the most recent version in September 2020.

We have already built a Sundial prototype in Google and deployed it on a median-scale (>500 servers) test cluster, although Sundial was recently designed in mid 2020. Additionally, we have shown huge performance improvements in Spanner[77] and in Swift[118] brought by Sundial.

Our design of FlowRadar and LossRadar resulted in a joint patent with Barefoot/Intel, and Alibaba is very interested in using the technique for loss detection.

DETER's Linux kernel-based implementation is open-sourced, and we revealed several TCP problems when running large distributed systems such as Spark and RPC. In addition, many software engineers are very interested in leveraging techniques of DETER to diagnose transport problems in their cloud.

7.3 Summary of Contributions

We summarize the contributions here.

New objectives for network control and telemetry. By observing the challenges faced today, we propose qualitatively better objectives than existing approaches:

(1) Adding robustness to performance-critical control tasks.

(2) Designing telemetry systems that are both precise and fine-grained.

The high impacts of systems in this dissertation reflect the importance of these new objectives.

Guided by the new objectives, we design systems based on the opportunities for codesigning different hardware and software components. We have the following three design principles that back the design.

Closing the gap between observation and control to make control precise and timely. In HPCC, we propose to close the gap by using inflight bytes for both observation and control, so it can converge in just 1 RTT to high throughput and zero queueing delay in most cases. In Sundial, we propose to close the gap by having the controller precompute a backup plan, based on which each device can make a quick decision with local observation.

Designing new algorithms and data structures to make effective use of different devices' capabilities. In Sundial, we design a new algorithm for finding the generic backup plan based on graph theory, so it is simple enough to fit in devices' limited capability. In FlowRadar and LossRadar, the data structures in switches embrace hash collisions by using XOR-based encoding. Finally, in DETER, we design a unique record and replay technique customized to TCP to enable extremely lightweight recording in the hosts.

Rethinking the division of labor among switches, hosts, and the controller with a paradigm shift away from the self-contained design model. In HPCC, the observation of total inflight bytes is divided into switch providing raw states and hosts calculating total inflight bytes. In Sundial, the failure recovery is divided into controller computing a backup plan and device-local reaction. In FlowRadar and LossRadar, we decouple the maintenance of hash tables into switch-local simple encoding and the in-controller cross-switch decoding. In DETER, the detailed information is retained by lightweight recording in the host and replaying in the controller.

7.4 Concluding Remarks

As the performance requirement continues to grow in the post-Moore era, hardware-software codesign is an exciting area and will become the norm in the future, where software is used as the glue between specialized hardware and application needs. However, unlike software engineering of which we have more than 50 years of experience[1], hardware-software codesign is just beginning. In this dissertation, we design and build several concrete systems, in the context of networking, as an initial exploration of hardware-software codesign.

In the future, we hope to continue investigating hardware-software codesign, in both networking and the broader system context, and from both performance perspective and other perspectives (e.g., reliability). And we hope to distill principles of developing new systems in the context of hardware-software codesign.

References

[1] 50 years of se, icse 2018. https://www.icse2018.org/info/50-years-of-se.

[2] Broadcom: Timing over Packet (ToP) Processor for Precision Timing Applications. https://www.broadcom.com/products/embedded-and-networking-processors/communications/bcm53903.

[3] Deterlab. https://www.isi.deterlab.net/.

[4] Flowradar implementation in p4. https://github.com/USC-NSL/FlowRadar-P4.

[5] Hpcc++: Enhanced high precision congestion control. https://tools.ietf.org/id/draft-pan-tsvwg-hpccplus-02.html.

[6] HPCC Group. http://hpcc-group.github.io/.

[7] Introduction to cisco ios netflow - a technical overview. http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[8] NetFlow. https://www.ietf.org/rfc/rfc3954.txt.

[9] The network is reliable. https://aphyr.com/posts/288-the-network-is-reliable.

[10] Open vSwitch. http://openvswitch.org/.

[11] Operand forwarding. https://en.wikipedia.org/wiki/Operand_forwarding.

[12] P4 behavioral model. https://github.com/p4lang/behavioral-model/blob/master/targets/simple_switch/simple_switch.cpp.

[13] P4 language consortium. p4.org.

[14] P4 simulator. https://github.com/p4lang.

[15] Packet loss impact on tcp throughput in esnet. http://fasterdata.es.net/network-tuning/tcp-issues-explained/packet-loss/.

[16] Solving the mystery of link imbalance a metastable failure state at scale. https://code.facebook.com/posts/1499322996995183/.

[17] IEEE 802.1D Work Group, IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Bridges, 2004.

[18] CockroachDB, 2008. https://github.com/cockroachdb/.

[19] ftrace, 2008. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[20] IEEE Standard 1588-2008, 2008. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757.

[21] Generic receive offload, 2009. https://lwn.net/Articles/358910/.

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

[23] Amazon us-east cloud goes down. http://www.eweek.com/cloud/amazon-us-east-cloud-goes-down.html, 2013.

[24] Router overhead when enabling netflow. http://blog.tmcnet.com/advanced-netflow-traffic-analysis/2013/05/router-overhead-when-enabling-netflow.html, 2013.

[25] Broadcom moves from simulation to emulation with Mentor, 2014. https://www.electronicsweekly.com/uncategorised/broadcom-moves-from-simulation-to-emulation-with-mentor-2014-01/.

[26] Empirical Traffic Generator, 2014. https://github.com/datacenter/empirical-traffic-gen.

[27] NetFilter, 2014. http://www.netfilter.org/.

[28] Redundancy N+1, N+2 vs. 2N vs. 2N+1, 2014. https://www.datacenters.com/news/redundancy-n-1-n-2-vs-2n-vs-2n-1.

[29] Linux perf, 2015. https://perf.wiki.kernel.org/index.php/Main_Page.

[30] TCP window updates combined with dup acks sent in response to packet loss, 2015. https://www.ietf.org/mail-archive/web/tcpm/current/msg09480.html.

[31] Cisco Packet Tracer, 2016. https://learningnetwork.cisco.com/docs/DOC-29644.

[32] IEEE 1588 PTP clock synchronization over a WAN backbone, 2016. https://www.endace.com/ptp-timing-whitepaper.pdf.

[33] In-band Network Telemetry, 2016. http://p4.org/p4/inband-network-telemetry.

[34] TCP Probe, 2016. https://wiki.linuxfoundation.org/networking/tcpprobe.

[35] IEEE 1588 PTP and Analytics on the Cisco Nexus 3548 Switch, 2017. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white-paper-c11-731501.html.

[36] Microsoft Keynote at SIGCOMM 2017, 2017. http://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf.

[37] Mozilla RR, 2017. https://rr-project.org/.

[38] RACK: a time-based fast loss detection algorithm for TCP, 2017. https://tools.ietf.org/html/draft-ietf-tcpm-rack-02.

[39] Tcpdump, 2017. http://www.tcpdump.org/tcpdump_man.html.

[40] Amazon EC2, 2018. https://aws.amazon.com/ec2/.

[41] Boson NetSim, 2018. http://www.boson.com/netsim-cisco-network-simulator.

[42] Clock Oscillators Surface Mount Type KC3225L-P2/ KC3225L-P3 Series, 2018. https://global.kyocera.com/prdct/electro/pdf/kc3225l_p2p3_e.pdf.

[43] DCTCP Bug, 2018. https://github.com/torvalds/linux/commit/27cde44a259c380a3c09066fc4b42de7dde9b1ad.

[44] Research Highlights In HPC, HPDA-AI, Cloud Computing, Quantum Computing, The Global Exascale Race, and Innovation Award Winners. https://hyperionresearch.com/wp-content/uploads/2019/02/Hyperion-Research-SC18-Breakfast-Presentation.pdf, 2018.

[45] Spark TeraSort, 2018. https://github.com/ehiggs/spark-terasort.

[46] In-band Network Telemetry in Barefoot Tofino. https://www.opencompute.org/files/INT-In-Band-Network-Telemetry-A-Powerful-Analytics-Framework-for-your-Data-Center-OCP-Final3.pdf, 2019.

[47] In-band Network Telemetry in Broadcom Tomahawk 3. https://www.broadcom.com/company/news/product-releases/2372840, 2019.

[48] In-band Network Telemetry in Broadcom Trident3. https://www.broadcom.com/blog/new-trident-3-switch-delivers-smarter-programmability-for-enterprise-and-service-provider-datacent
2019.

[49] Linux TCP Github, 2019. https://github.com/torvalds/linux/tree/master/net/ipv4.

[50] Network Simulator 3, 2019. https://www.nsnam.org/.

[51] Plotting a Course to a Continued Moore's Law. `https://youtu.be/6wq6g_vi6yw`, 2019.

[52] Switch Implementation with P4, 2019. `https://github.com/p4lang/switch/blob/master/p4src/switch.p4`.

[53] Juniper Precision Time Protocol Overview, 2020. `https://www.juniper.net/documentation/en_US/junos/topics/concept/ptp-overview.html`.

[54] Mellanox Precision Time Protocol, 2020. `https://docs.mellanox.com/display/ONYXv381174/Precision+Time+Protocol`.

[55] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM, 2008*.

[56] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[57] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[58] 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 *NSDI, 2012*.

[59] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM, 2013*.

[60] Gautam Altekar and Ion Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[61] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[62] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1), 1999.

[63] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI, 2015*.

[64] Hamsa Balakrishnan, Nandita Dukkipati, Nick McKeown, and Claire J Tomlin. Stability analysis of explicit congestion control protocols. *IEEE Communications Letters*, 11(10):823–825, 2007.

[65] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the ACM SIGCOMM*, pages 662–680, 2020.

[66] Thomas Bonald, Laurent Massoulié, Alexandre Proutière, and Jorma Virtamo. A queueing analysis of max-min fairness, proportional fairness and balanced fairness. *Queueing Systems*, 53(1-2):65–84, 2006.

[67] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: From approximate membership checks to approximate state machines. In *SIGCOMM*, 2006.

[68] Flavio Bonomi, Michael Mitzenmacher, Rina Panigraphy, Sushil Singh, and George Varghese. Bloom filters via d-left hashing and dynamic bit reassignment extended abstract. In *Forty-Fourth Annual Allerton Conf., Illinois, USA*, pages 877–883, 2006.

[69] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1985.

[70] Weifeng Chen, Yong Huang, Bruno F. Ribeiro, Kyoungwon Suh, Honggang Zhang, Edmundo de Souza e Silva, Jim Kurose, and Don Towsley. Exploiting the ipid field to infer network path and end-system characteristics. In *PAM*, 2005.

[71] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, 2009.

[72] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data centers. In *NSDI*, 2014.

[73] Yu-Chung Cheng, Urs Holzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *Usenix*, 2004.

[74] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM, 2017*.

[75] Abhijit K. Choudhury and Ellen L. Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Trans. Netw.*, 1998.

[76] Cisco. Netflow performance analysis. *White paper*, 2005.

[77] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[78] Donald Davies. The control of congestion in packet-switching networks. *IEEE Transactions on Communications*, 20(3):546–550, 1972.

[79] Jeff Dean. Designs, lessons and advice from building large distributed systems. In *LADIS keynote*, 2009.

[80] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communication of the ACM*, 2013.

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

[82] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *ACM SIGCOMM*, 2003.

[83] Nandita Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Stanford University, 2007.

[84] D. Eppstein, M. Goodrich, F. Uyeda, and G. Varghese. What's the difference? efficient set difference without prior context. In *SIGCOMM*, 2011.

[85] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better netflow. *ACM SIGCOMM*, 2004.

[86] Cristian Estan and George Varghese. Data streaming in computer networking. In *Workshop on Management and Processing of Data Streams*, 2003.

[87] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI, 2016.*

[88] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT, 2015.*

[89] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, 2006.

[90] Dániel Géhberger, Dávid Balla, Markosz Maliosz, and Csaba Simon. Performance evaluation of low latency communication alternatives in a containerized cloud environment. In *IEEE CLOUD, 2018*.

[91] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, 2018.

[92] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *arXiv:1101.2245v2*, 2011.

[93] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. Abc: A simple explicit congestion control protocol for wireless networks. *arXiv:1905.03429*, 2019.

[94] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM*, 2009.

[95] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM, 2016*.

[96] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[97] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[98] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[99] 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 *SIGCOMM, 2017*.

[100] New BPDU Handling. Understanding rapid spanning tree protocol (802.1 w). *Catalyst*, 2948(L3/4908G):L3.

[101] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *CoNEXT, 2017*.

[102] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the ACM SIGCOMM*, pages 113–126, 2017.

[103] Van Jacobson. Congestion avoidance and control. In *SIGCOMM, 1988*.

[104] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[105] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[106] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. A distributed algorithm to calculate max-min fair rates without per-flow state. In *SIGMETRICS, 2019*.

[107] Glenn Judd. Attaining the promise and avoiding the pitfalls of tcp in the datacenter. In *NSDI, 2015*.

[108] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2), 2007.

[109] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM, 2002*.

[110] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

[111] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[112] Frank Kelly, Gaurav Raina, and Thomas Voice. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM Computer Communication Review*, 38(3):51–62, 2008.

[113] Frank Kelly and Elena Yudovina. *Stochastic networks*, volume 2. Cambridge University Press, 2014.

[114] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[115] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[116] R. Kompella, K. Levchenko, A. Snoeren, and G. Varghese. Every microsecond counts: Tracking fine-grain latencies with a loss difference aggregator. In *SIGCOMM*, 2009.

[117] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, 1987.

[118] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the ACM SIGCOMM*, 2020.

[119] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants). In *SIGCOMM*, 2003.

[120] Maciej Kuzniar, Peter Pere, and Dejan Kosti. What you need to know about sdn flow tables. In *PAM*, 2015.

[121] Leslie Lamport. *Synchronizing time servers*. Digital, Systems Research Center, 1987.

[122] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[123] Leslie Lamport and Peter M Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 68–74, 1984.

[124] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, 2016.

[125] Wlodzimierz Lewandowski, Jacques Azoubib, and William J Klepczynski. Gps: Primary tool for time transfer. *Proceedings of the IEEE*, 87(1):163–172, 1999.

[126] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[127] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock-synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, 2020. USENIX Association.

[128] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. Deter: Deterministic tcp replay for performance diagnosis. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 437–452, 2019.

[129] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[130] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, 2016.

[131] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[132] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[133] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM SIGCOMM*, pages 334–350, 2019.

[134] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter Braids: A novel counter architecture for per-flow measurement. In *SIGMETRICS*, 2010.

[135] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *APNET, 2017*.

[136] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 165–176, New York, NY, USA, 2006. ACM.

[137] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[138] Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 295–305, 1983.

[139] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. In *SIGCOMM Comput. Commun. Rev.*, 1997.

[140] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2), 2008.

[141] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[142] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM, 2015*.

[143] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *SIGCOMM, 2018*.

[144] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM, 2018*.

[145] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *SIGCOMM*, 2016.

[146] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *NSDI*, 2013.

[147] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[148] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[149] Ilkka Norros, James W. Roberts, Alain Simonian, and Jorma T. Virtamo. The superposition of variable bit rate sources in an atm multiplexer. *IEEE Journal on Selected Areas in Communications*, 9(3):378–387, 1991.

[150] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms — ESA 2001. Lecture Notes in Computer Science 2161*.

[151] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[152] Pawan Prakash, Advait Dixit, Y. Charlie Hu, and Ramana Kompella. The TCP outcast problem: Exposing unfairness in data center networks. In *NSDI*, 2012.

[153] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *SIGCOMM*, 2014.

[154] Renato Recio, Bernard Metzler, Paul Culley, Jeff Hilland, and Dave Garcia. A remote direct memory access protocol specification (RFC5040). Technical report, 2007.

[155] James W. Roberts, editor. *Performance Evaluation and Design of Multiservice Networks*. Commission of the European Communities, 1992.

[156] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[157] Luca Schenato and Federico Fiorentin. Average timesynch: A consensus-based protocol for clock synchronization in wireless sensor networks. *Automatica*, 47(9):1878–1886, 2011.

[158] Ulrich Schmid. Synchronized utc for distributed real-time systems. *Annual Review in Automatic Programming*, 18:101–107, 1994.

[159] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojeviundefined, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. SIGMOD '19, 2019.

[160] Naveen Kr Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, 2017.

[161] Alex Shpiner, Eitan Zahavi, Vladimir Zdornov, Tal Anker, and Matty Kadosh. Unlocking credit loop deadlocks. In *HotNets, 2016*.

[162] 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 *SIGCOMM, 2015*.

[163] Roberto Solis, Vivek S Borkar, and PR Kumar. A new distributed time synchronization protocol for multihop wireless networks. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 2734–2739. IEEE, 2006.

[164] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended Bloom filter: An aid to network processing. In *SIGCOMM*, 2005.

[165] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, 2004.

[166] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.

[167] Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[168] D. E Taylor and J. S Turner. ClassBench: A Packet Classification Benchmark. *Transactions on Networking*, 15(3), 2007.

[169] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 283–297, 2018.

[170] Robert Underwood, Jason Anderson, and Amy Apon. Measuring network latency variation impacts to high performance computing application performance. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 68–79, 2018.

[171] Frank Uyeda, Luca Foschini, Fred Baker, Subhash Suri, and George Varghese. Efficiently Measuring Bandwidth at All Time Scales. In *NSDI*, 2011.

[172] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *SIGCOMM, 2009*.

[173] Kashi Venkatesh Vishwanath and Amin Vahdat. Realistic and responsive network traffic generation. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2006.

[174] Berthold Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4), 2003.

[175] W. Vogels. Performance and scalability. http://www.allthingsdistributed.com/2006/04/performance_and_scalability.html, 2009.

[176] Thomas Voice and Gaurav Raina. Stability analysis of a max-min fair rate control protocol (rcp) in a small buffer regime. *IEEE Transactions on Automatic Control*, 54(8):1908–1913, 2009.

[177] Mea Wang, Baochun Li, and Zongpeng Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. *Distributed Computing Systems, International Conference on*, 0:628–635, 2004.

[178] Geoffrey Werner-Allen, Geetika Tewari, Ankit Patel, Matt Welsh, and Radhika Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, 2005.

[179] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. In *SIGCOMM*, 2012.

[180] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, 2011.

[181] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.

[182] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI*, 2011.

[183] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[184] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. In *SIGCOMM, 2012.*

[185] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[186] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[187] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Scaling up clustered network appliances with ScaleBricks. In *SIGCOMM*, 2015.

[188] 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 *SIGCOMM, 2015*.

[189] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *CoNEXT, 2016*.

[190] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.