ORDERING VERSUS BOUNDED DELAY: TIMELY CONSENSUS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Katherine Rose Gioioso

August 2021

ABSTRACT

Traditionally, consensus protocols have been designed under the best-effort delivery communication model. Motivated by the fact that datacenter networks enable co-designing network communication protocols and consensus protocols, an exciting recent line of research has demonstrated that it is possible to design high-performance consensus and state machine replication protocols by enabling a stronger communication model, one where the network provides message ordering.

This thesis explores a very different communication model that can be enabled by datacenter networks—bounded message delays—and demonstrates that, under this model, it is possible to design consensus protocols that guarantee strong properties, including safety and termination even when all but one replica fails. Our contributions are three-fold. First, we present a network communication protocol that enables a novel Timely Unreliable Multicast (TUM) primitive; TUM guarantees that, for every message sent by a sender: (1) all receivers that receive the message receive it within a bounded amount of time that is known a priori; and (2) messages may be lost due to hardware failures only, and not due to buffer overflows. Second, we design Timely consensus and state machine replication protocols that build upon the TUM primitive to guarantee safety and termination, even with all but one replica failures (under the crash recovery failure model). Finally, using an end-to-end implementation of the TUM primitive and the Timely protocols, we demonstrate that Timely protocols enable these properties while achieving high performance even under worst-case network hardware failures.

BIOGRAPHICAL SKETCH

Katherine (Katie) Gioioso grew up in Milton, Massachusetts. She graduated from Cornell University in 2019 with Bachelors of Science in Computer Science. She then received a Masters of Science in Computer Science from Cornell, advised by Professor Rachit Agarwal. In addition to her studies, Katie is a strong advocate for women in Computer Science and a passionate teacher. During her time at Cornell, she served as Co-Director of Outreach for Women in Computing at Cornell (WICC), organizing workshops for local middle and high school students. She then served as co-president of WICC. Katie began her teaching career as a facilitator for Cornell's Academic Excellence Workshops, a supplementary course for students in Introduction to Computer Science. In addition to teaching technical concepts, she enjoyed introducing students to the joys and excitement of CS. She then served as a Teaching Assistant for the Computer Science department for 6 semesters. Katie is the recipient of the Jonathan Marx Award for Leadership in Computer Science and multiple outstanding teaching assistant awards.

ACKNOWLEDGEMENTS

There are many people to thank.

First, my advisor Rachit Agarwal and my minor advisor Eva Tardos. Rachit takes the role of advisor to heart. He is keenly aware that we are all works-in-progress and takes personal responsibility for that progress. I am honored and grateful to be a recipient of his enthusiasm and his patience. Rachit taught me how to break down complex problems and to not be afraid to think big. He made it clear that he believes in me; sentiment I value because it comes from someone I respect and admire. Eva served officially as my minor advisor but mostly as a trusted mentor throughout my Masters (and even before that). I look up to her because she has mastered the ultimate trifecta of skills: research, teaching, and administration. Her dedication to her students, to diversity efforts, and to the department makes Cornell CS what it is today. She is a valued source of advice and inspiration. Both Rachit and Eva have been instrumental to my success. Thank you.

Three friends were particularly influential in my Cornell experience. Michelle Li was my mentor from the very beginning of my CS career, before I knew anything or anyone. She consistently pointed me in the right direction, provided encouragement, and set an example I could look up to. I credit much of my success to her. Samantha Dimmer was my buddy for most classes at Cornell. We endured many projects, problem sets, and exams together. We shared a love of problem solving, particularly of solving problems together, and a mutual respect for each other's skills and intelligence. I consistently leaned on her companionship and support. Saksham Agarwal was my go-to person throughout my Masters. He was and is a valued resource for all things research, grad school, and life. All three continue to be trusted friends.

It's hard to express the immense sense of respect and gratitude I have for the Cornell CS community at large. Cornell CS is responsible for the entirety of my CS education, from my very first introductory course all the way up to this thesis. It has been my home base for the last six years and gave me an environment where I could thrive. First, the professors. Cornell CS professors teach and live with purpose. Anne Bracy, Michael Clarkson, Walker White, Bobby Kleinberg, John Hopcroft, and Lorenzo Alvisi were particularly influential in my discovery of CS and growth as a person. The department functions thanks to the support of its staff members. I benefited most from the help of Vanessa Maley, Nicole Roy, and Becky Stewart. Finally, it's an honor to be part of a dedicated community of students. Special thanks to the mighty women of WICC: my co-presidents Avani and Nandita, as well as Vy, Elizabeth, Janice, Nina, May, Amanda, Melody, Agi, and Emma, to name just a few. These women taught me how to talk across differences, how to adapt to new environments, and how to lead. I navigate the world of tech with confidence thanks to them. Beyond WICC, the CS student organizations made the major feel like more than just a set of classes. The endless meetings, workshops, and socials (and free food!) helped me believe I belong in CS. This community would not be possible without the tireless dedication of its student leaders: Avani Bhargava, Nandita Mohan, Irene Yoon, Danny Qiu, Rishi Bommasani[1], and many more.

Throughout my time at Cornell, I found belonging among the residents of High Rise 5, members of Pi Phi, AEW students and facilitators, and other TAs. Sydney, Gillian, Melissa, Cara, Ellen, and Leah are my rocks and my dearest friends. And finally, I would not be here without my family. Steph is always just a few steps ahead of me, paving the way and providing a map for how to live. Christina leads her life with joy. She taught me how to connect with others

[1]This section was inspired by Rishi's Masters thesis

and bring happiness to the people around me. My parents actively nourished my curiosity and intellect and subtly pointed me towards engineering. They taught me to work hard, take initiative, and act deliberately. I grow thanks to their continued support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Traditionally, distributed systems have been designed under the assumption that the network supports a weak communication model—it can deliver messages in an order that is arbitrarily different from the order in which messages were sent, can deliver messages with arbitrary delays, can deliver duplicated messages, and can even drop messages due to buffer overflows. Under such a weak communication model, applications must embrace weak semantics, high complexity, and/or poor performance. A classical example is that of consensus protocols. It is well-known that, under the above communication model, applications must not only give up on semantics (safety or termination) [11], but also use protocols that have poor worst-case performance [23].

An exciting recent line of research [37, 25] has explored the benefits of assuming stronger communication models. The motivation comes from the fact that a large class of distributed applications today run in datacenter networks. Unlike the Internet context, datacenter networks are capable of supporting stronger network semantics: they are controlled by a single entity and can thus enable network changes; they already deploy customized or programmable network hardware and can thus enable offloading functionality to network hardware; and, they have structured topologies and can thus enable design of customized protocols (*e.g.*, multicast or broadcast). For instance, two state-of-the-art protocols—SpecPaxos [37] and NOPaxos [25]—demonstrate that it is possible to design high-performance consensus and state machine replication protocols by assuming one particular communication model from the network: *message ordering*.

This paper explores a very different communication model: *bounded message delays*. Unlike the message ordering semantics that ensure a failure-free network delivering messages in the same order as sent by the sender, our bounded message delay semantics ensure a failure-free network delivering messages from the sender to the receiver within a (known) bounded amount of time. Our model leads to protocols that have significantly different properties: while the message ordering communication model can enable consensus protocols that guarantee safety with $(n-1)/2$ replica failures, our bounded message delay communication model enables much more powerful consensus protocols—those that guarantee both safety and termination, even with $(n-1)$ replica failures.

In exploring consensus and state machine replication protocols under the bounded message delay communication model, this paper makes three contributions. First, we present a network design that enables *Timely Unreliable Multicast (TUM)* primitive, that guarantees two properties for every message sent by a sender: (1) all receivers that receive the message receive it within a bounded amount of time that is known a priori; and (2) messages may be lost due to hardware failures only, and *not* due to buffer overflows. Both of these are powerful properties. The first one ensures that each message observes (a tiny) bounded amount of queueing delay in the network; thus, in scenarios when network hardware does not fail, the message is delivered to all the receivers within a bounded amount of time. As we will see, this enables an extremely simple consensus protocol that can handle $(n-1)$ replica failures while guaranteeing safety and termination for the case of no network hardware failures. The second property ensures that, unlike all previous communication models where network semantics can be violated by both buffer overflows and hardware failures, our communication model ensures that a lost message *must* be

due to hardware failures. As we will see, this property makes network failures insignificant: with minor modification, the consensus protocol for the case of no network hardware failures continues to work for the case of failures, while providing the same guarantees.

Our second contribution is design of a Timely consensus protocol, and a Timely state machine replication protocol that exploit the TUM network primitive to guarantee safety and termination, even with $(n-1)$ replica failures, under the crash recovery failure model and under a connectivity condition that is akin to no network partitions. We provide formal proofs for all Timely protocol guarantees.

Our third contribution is an end-to-end implementation of the TUM primitive and Timely protocols. Evaluation on a small-scale testbed demonstrates Timely protocols enable new operating points compared to existing consensus protocols.

CHAPTER 2

TIMELY CONSENSUS OVERVIEW

This chapter starts by outlining the network, end-host, and failure models (§2.1), and the definitions (§2.2) used throughout this paper. We then briefly recap existing results under our model (§2.3), and then summarize our results (§2.3.1).

2.1 Network, end host, and failure models

This paper, similar to [37, 25], is motivated by the fact that datacenter networks offer a very different operating environment for applications when compared to the classical case of Internet. Thus, it is useful to precisely define our network model, end-host model, and failure models. For our formal proofs, we will assume the existence of a global clock. Network and end-host elements are unaware of this global clock, and operate on their local clock. End-host clocks may vary arbitrarily from the global clock. Switch local clocks may be arbitrarily slower than the global clock but not faster.

Network model. We assume that the datacenter network is organized around what are typically referred to as hierarchical network topologies (FatTree [7], VL2 [15], or leaf-spine [8]); all datacenter networks today use such topologies. For ease of exposition, we will describe the mechanisms using a two-tier leaf-spine topology [7, 8]—for switches with k ports, this topology has $k/2$ *core* switches, k *aggregate* switches, and $N = k^2/2$ hosts (an example for $k = 4$ is shown in figure 2.1). All our results naturally generalize to three-tier topologies.

We exploit two functionalities offered by all commodity switches. First, we

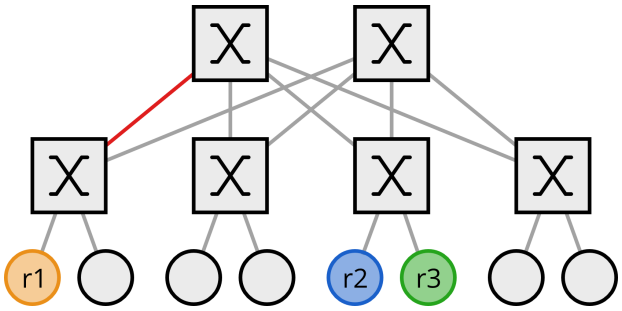


Figure 2.1: **Leaf-spine datacenter network topology with k = 4, and an example for connected and completely-connected replicas**. The red link is a crashed link. Replica $r1$ is connected because it has a failure-free path to one core switch. Replicas $r2$ and $r3$ are completely-connected.

will use the standard isolation techniques (*e.g.*, priority queues [9, 16]) for ensuring that all messages used in our protocol are sent at highest priority, and do not observe any interference from other traffic. Second, we will also exploit the fact that each switch processes and sends packets at the same rate, as required by the IEEE standard [5]. Our design, similar to [37, 25], also offloads some of the functionality to the network hardware. To that end, we will exploit two functionalities offered by programmable switches: (1) switches can detect adjacent link failures within a bounded amount of time and generate failure notification packets in response to link failures [39, 10, 17]; and (2) network switches have the ability to mirror packets—a switch can forward a message along multiple of its outgoing ports [37, 25, 38].

End-host model. Each host executes according to its local clock. Each host maintains an *input buffer* and an *output buffer*, each of a fixed size, on a per-application/replica basis; these buffers will be used to store messages required by the protocol. In a failure-free scenario, we know the maximum amount of time that any host takes to process all the messages in an input buffer and an output buffer. More formally, there is a known upper bound ϕ such that the following holds: for an application running at the host to start processing the

buffers at time t, it can process all messages in the input buffer and put the response message (if any) in the output buffer for that application by time $t + \phi$.

We make two important observations here. First, safety and termination guarantees provided by our design are independent of the value of ϕ—all our results hold for any $\phi \geq 0$; only the "observed" performance depends on the precise value of ϕ. Second, modern end-host network hardware (network interface cards and on-board PCIe devices) introduce little to no variance in processing latencies, with tail latency being of the order of single-digit microseconds [31, 18, 20, 29]; in addition, with kernel-bypass techniques [35, 14, 34, 27, 20, 19], it is possible to perform application-layer scheduling within single-digit microseconds. As a result, for modern datacenter networks, ϕ (that captures the tail latency both in end-host network hardware and application processing) is in single-digit microseconds, a small fraction of datacenter network fabric speed-of-light latency (tens of microseconds [28, 40]). We will demonstrate in our evaluation that, even at extremely high loads, this holds true.

Failure model. We consider the crash recovery failure model for end hosts and network elements (links, switches, etc.) [24, 6, 32]. In particular, end hosts and network elements can crash, and can recover at arbitrary times. All end hosts and network elements are assumed to be bug-free.

2.2 Definitions

We will use the following definitions throughout the paper.

	End-host failures		Link failures		Switch failures	
	Safety	Termination	Safety	Termination	Safety	Termination
Paxos + variants	$(n-1)/2$	✗	✓	✗	✓	✗
SpecPaxos [37]	$(n-1)/2$	✗	✓	✗	✓	✗
NOPaxos [25]	$(n-1)/2$	✗	✓	✗	✓	✗
Timely	$n-1$	$n-1$	✓	✓	✓	✓

Table 2.1: **Comparison of properties guaranteed by existing consensus protocols and by Timely.** Timely safety guarantees hold as long as one of the replicas is completely-connected; Timely termination guarantees hold as long as there is no replica partition.

Definition 1 Failure-free path: *A path between two hosts is said to be failure-free if and only if the path does not have any crashed links and/or switches.*

Definition 2 (Completely-)Connected host: *An end host p is said to be connected if and only if there exists a failure-free path between p and at least one of the non-faulty core switches. The host is said to be completely-connected if and only if it has a failure-free path to all non-faulty core switches.*

Definition 3 Non-faulty host: *An end host is said to be non-faulty if and only if it does not crash and it is connected.*

Definition 4 Replica partition: *A set of replicas is partitioned if the non-faulty replicas can be divided into subsets A and B such that there does not exist a failure free path from any replica in subset A to any replica in subset B.*

Definition 5 Safety: *A consensus protocol guarantees safety if and only if the following three conditions hold for all non-faulty replicas: (i) every replica decides the same value; (ii) if every replica proposes v, every replica decides v; and, if a replica decides v, some replica proposed v.*

Definition 6 Termination: *A consensus protocol guarantees termination if and only if all non-faulty replicas eventually decide a value.*

Definition 7 t-Resilient consensus: *A consensus protocol is said to be t-resilient if and only if it guarantees safety and termination in presence of t or fewer faulty replicas.*

2.3 Motivation: Consensus in Datacenter Networks

The properties enabled by a consensus protocol, as well as its performance, depend on the communication model assumed from the underlying network. Indeed, when the network is assumed to provide best-effort (possibly arbitrary message delays, message drops, message reordering, etc.) communication model, it is known that even 1-resilient consensus is impossible [13]. We provide a detailed discussion on related work in §6, but note that classical consensus protocols (that are used in practice) and their variants assume best-effort communication model. However, as argued eloquently in [37], many applications today run in datacenter network environments that are quite different from the classical Internet context—these networks deploy geographically colocated hosts (thus offering low-latency round trip times when the network is unloaded), they are controlled by a single entity, they already deploy customized hardware—and thus, provide the opportunity to enable stronger communication models from the network.

Building upon the above motivation, SpecPaxos [37] and NOPaxos [25] protocols show that if the network can guarantee that messages are delivered in the same order as they were sent, it is not only possible to achieve properties sim-

ilar to the classical Paxos protocol but also avoid the worst-case performance in many scenarios. For instance, SpecPaxos—in best-case scenarios—avoids the overheads by assuming "Mostly-Ordered Multicast" (best-effort message ordering) communication model. NOPaxos further expands the envelope of best-case scenarios by enabling an "Ordered Unreliable Multicast" communication model using a centralized sequencer (that can be implemented either at one of the network switches, or at one of the end hosts).

This paper explores a different communication model—Timely Unreliable Multicast—that guarantees that each message sent by a sender is delivered within a bounded amount of time. We demonstrate that this communication model enables new operating points for datacenter consensus protocols. We are motivated by two additional opportunities offered by modern datacenter networks: (1) recent adoption of high-bandwidth links (100Gbps and beyond) means that the round trip times in an unloaded network is at most a few tens of microseconds [28, 17, 29]; and (2) recent adoption of kernel-bypass techniques and μs-scale application and network processing capabilities [34, 14, 20] reduces the variance in end host processing delays to single-digit microseconds, making them quite predictable. For such networks, the real source of "unpredictable" message latencies is queueing delay, and—this is the crux of the paper—queueing delays are a property of the communication protocol; thus, if we are able to design a communication protocol that enables small, bounded, queueing delays, it is possible to explore new operating points for datacenter consensus protocols by co-designing the consensus protocol with the communication protocol.

2.3.1 Timely Results

This paper has two core results. First, we present a new communication protocol design that enables the Timely Unreliable Multicast primitive. We then use this primitive to design Timely consensus and state machine replication protocols. The core result is:

Theorem 1 *Timely Consensus is $N-1$ resilient if there is one non-faulty, completely-connected, replica. Safety is guaranteed as long as there is one non-faulty replica and no replica partition. Termination is guaranteed as long as there is one non-faulty replica that remains completely-connected for the entirety of the consensus protocol.*

We note that our result does not violate any impossibility result—indeed, as outlined in [11], our communication and failure models from §2.1 are sufficient for the properties enabled by Timely protocols. Table 2.1 compares the properties of Timely and existing datacenter consensus protocols.

CHAPTER 3

TIMELY UNRELIABLE MULTICAST

In this section, we present the Timely network design that enables our Timely Unreliable Multicast communication model, providing the following semantics (see Figure 3.1): *if a host multicasts a message to a set $\mathcal{S}$ of hosts at time t on the global clock, and at least one host in $\mathcal{S}$ receives the message, then all non-faulty hosts in $\mathcal{S}$ either receive the message by time $t + \delta$ on the global clock, or receive a failure notification by time $t + \delta + \delta_f$ on the global clock, for fixed known δ and δ_f.* We will provide the exact values of δ and δ_f in §3.4.1 once we have enough context.

We now present a communication protocol design that enables Timely Unreliable Multicast. The key insight behind the Timely network design is as follows. There are five sources of message delays in any communication network—transmission delay, propagation delay, switching delay, queueing delay, and end-host delay (network interface card, PCIe, and stack processing delay). Consider the case of a single fixed-size message in the network: for a datacenter network, the first three delays are not only fixed, but are also known in advance; the only source of variability is due to some part of the system being overloaded—high queueing delays are possible when switches or NICs are overloaded (in the worst case, the message may be dropped), and PCIe and stack processing delays could be high when the host is overloaded. If we can somehow ensure that

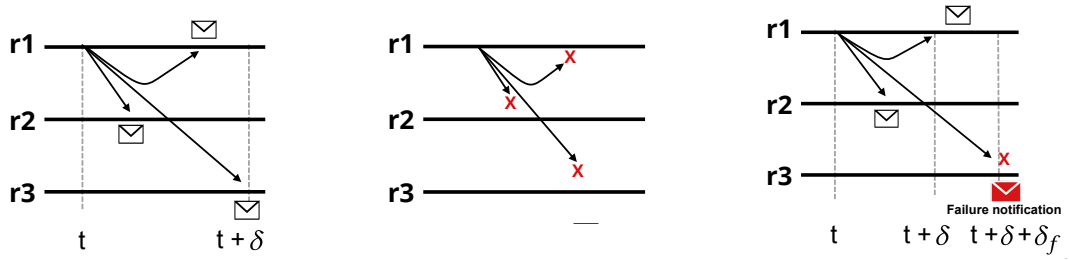


Figure 3.1: **Illustration of Timely Unreliable Multicast semantics.**

switches and NICs observe minimal or no queueing for consensus-related messages, the only source of message delay variability will be PCIe and host stack processing; the former is already a small fraction of propagation and transmission delays, and the latter can be near-zero by using kernel-bypass techniques and state-of-the-art schedulers (or dedicated cores at the hosts). Thus, the whole problem of enabling Timely Unreliable Multicast primitive boils down to (1) designing a communication protocol that enables minimal or zero queueing at switches and NICs; (2) enabling network-supported multicast; and (3) enabling failure notifications. Our Timely design, building upon several recent works, achieves precisely these goals.

We first describe the three components of Timely design—bounded queueing via admission control (§3.1), multicast at core switches (§3.2), and failure notification (§3.3)—assuming zero host processing delays. We then extend the Timely design to handle host processing delays (§3.4).

3.1 Minimal queueing using *admission control*

In typical networks, end hosts make two decisions at any time instant: (1) whether to transmit a packet or not; and (2) if yes, which packet to transmit. Delegating both these responsibilities to hosts causes problems in the network—at any network element (NIC or switches), if packets arrive faster than they can be forwarded, queue occupancy will grow; as a result, packets will experience queueing delays. Worse, packets may be dropped if a queue gets full.

Building upon ideas from recent work on admission control based transport protocols in datacenter networks [17, 29], Timely use network-driven admission

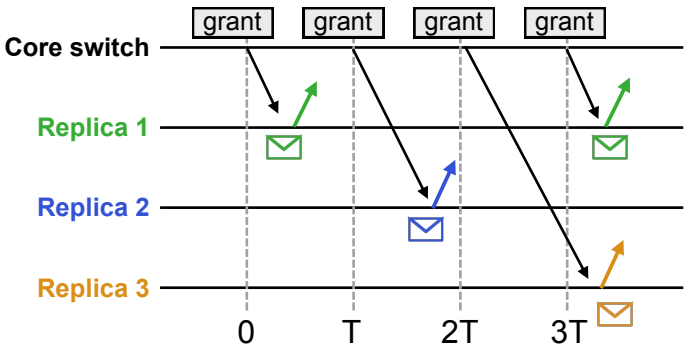


Figure 3.2: **An illustration of Timely admission control mechanism.** The core switch sends a `grant` once per unit time (T, as defined in §3.1), rotating round robin among replicas. When a replica receives a `grant`, it sends a data packet.

control to delegate the first decision from the hosts to the network: core switches decide when hosts send packets. Hosts are still responsible for deciding which packets to send (based on Timely consensus protocol), but a host cannot send a packet until it receives a `grant` from one of the core switches. Such a network-based admission control allows regulating the rate at which packets enter the network, thus limiting the number of packets that can arrive at any NIC and/or switch at any given time; as we will see, this is key to guaranteeing bounded queue occupancy and bounded message latency in all failure-free scenarios.

Warm up: the simple case of a single application and a single core switch. Let T be the transmission time of two fixed-size packets (that we will define with Timely consensus protocol): one data packet and one `grant` packet. The core switch sends a `grant` to one replica every T time, rotating round robin through each replica. When a replica receives a `grant`, it is allowed to transmit a data packet.

Figures 3.2 demonstrates this mechanism. For the case of a single application and a single core switch, it is easy to see that the admission control mechanism ensures that each network element receives data at the rate at which it can trans-

mit, thus ensuring zero queueing delays.

The case of multiple core switches. It is easy to generalize the above design to multiple core switches: if the network has c core switches, each core switch now sends one `grant` per cT time, where T is as defined above. Each core switch again rotates round robin through each replica. When a replica receives a `grant` from some core switch, it sends a data packet to that core switch.

We provide formal proofs in Appendix 3, but informally, the design extension for the case of multiple core switches guarantees zero persistent queueing and bounded transient queueing at each element. Specifically, transient queueing is possible because core switches issue `grants` without coordination and since their local clocks may not be synchronized; thus, all core switches could issue a `grant` at the exact same global time, and queue occupancy could grow to a maximum size of c since c packets could arrive at a switch at the exact same global time. However, if such were the case, each core switch will wait for cT amount of local time before issuing another `grant`—by design, this is enough time for the queue to drain. Put another way, even with this transient queueing there is no persistent queueing because a `grant` is issued at the same average rate as in the simple case above: once per T.

The case of multiple core switches and multiple colocated applications. To generalize to multiple core switches and multiple colocated applications, we decompose the network into multiple virtual networks: one per application. Each virtual network is assigned an equal bandwidth along each link such that the sum of the virtual bandwidths on any link does not exceed the bandwidth of that link. An example decomposition and bandwidth allocation is demonstrated in Figure 3.3. For any individual virtual network, let T_v be the trans-

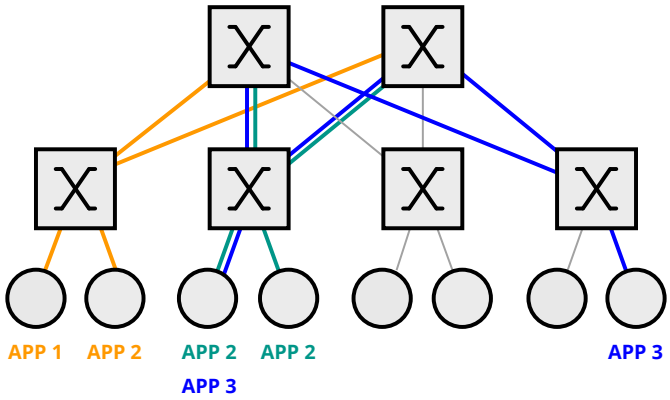


Figure 3.3: **Illustration of topology decomposition idea in Timely network design for colocated applications.** Application 1 replicas do not share hosts with other applications; two replicas from applications 2 and 3 are on the same host. The topology is decomposed into three virtual networks, one per application. Application 1 can use full bandwidth, and applications 2 and 3 each get 1/2 bandwidth.

mission time of one data and one `grant` packet using the bandwidth assigned to that virtual network. Each core switch now issues a `grant` to each virtual network once every cT_v time, rotating round robin through each replica in that virtual network.

Similar to the previous case, this design guarantees zero persistent and bounded transient queueing by ensuring that the average rate `grant` packets are issued for all virtual networks never exceeds the rate at which a link can transmit. As long as each core switch never issues a `grant` to two applications that share an end-host within any span of cT time, transient queueing is still bounded by the number of core switches. A formal proof is provided in Appendix 3.

3.2 Bounded-delay Multicast *at core switches*

Most network switches already support in-network multicast—a switch can receive one packet and send the packet to multiple destinations by mirroring the packet to multiple outgoing links. Similar to prior techniques [37, 25, 38], we use this functionality to enable multicast but with a simple modification—unlike prior mechanisms that use shortest path routing for multicast packets (that is, packets are potentially mirrored at multiple network switches such that each copy takes the shortest path to reach its destination), Timely design performs multicast only at core switches.

Intuitively, with shortest path routing, a multicast packet takes a wide variety of paths to reach all destinations, and accordingly, the fate of each multicast copy varies. For instance, consider the case of a multicast packet that has one destination on the sender's aggregate switch and multiple other destinations on a separate aggregate switch. When the packet reaches the sender's aggregate switch, the aggregate switch may mirror the packet to some downlinks (directly to a destination host) and some uplinks (to core switches). If the aggregate switch is disconnected from the core switches due to link failures, then the destinations on the sender's aggregate switch will receive the packet but other destinations will not.

Timely design exploits the structure in datacenter topologies to ensure that all copies of a multicast packet share the same fate. In particular, all packets are routed first to a core switch, and then downstream to destination replicas. This allows us to maintain a simple invariant that will be useful in our Timely consensus protocol—if the path between the sender and the core switch is faulty,

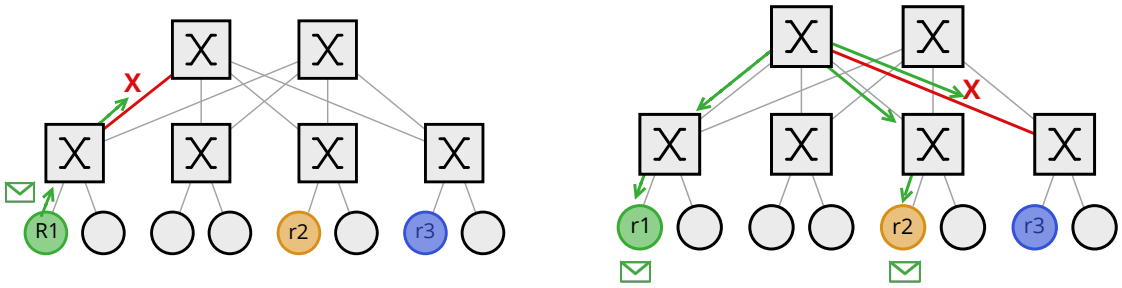


Figure 3.4: **Illustration of packet mirroring at core switches.** (left) If a message is dropped before it reaches the a core switch, no replica receives the message; (right) If a message reaches the core switch, all replicas connected to that core switch receive the message.

none of the replicas receive a copy; if the path between the sender and the core switch is failure-free, any replica connected to the core switch (including the sender) receives a copy. Indeed, the only replicas that do not receive the packet are those that are not connected to the core switch. Figure 3.4 presents an example.

The design of Timely multicast is thus as follows. When a host receives a `grant`, it sends a multicast data packet. The multicast packet is routed to the core switch that sent the original `grant`. The core switch then mirrors the packet to the aggregate switches corresponding to the destination replicas. The aggregate switches then mirror the packet to all replicas, including the sender. Figure 3.5 shows a data packet routed from a sender to three replicas. It is easy to see that Timely multicast achieves the same bounded message latency guarantees as discussed in the previous subsection (again, formally proved in Appendix 3).

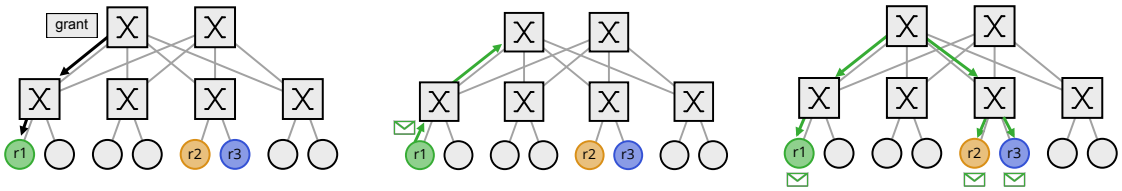


Figure 3.5: **Illustration of the `grant` based protocol for Timely Unreliable Multicast in Timely.** (top left) a core switch generates a `grant` and sends it to a replica; (top right) the replica sends a data packet back to the core switch; (bottom) the network broadcasts the data packet to all replicas by mirroring the packet to each link.

3.3 Timely Unreliable Multicast: Bounded-delay multicast with link failure notification

Switches detect adjacent link failures and send a failure notification to downstream hosts. This, however, provides failure (and subsequent potential packet drop) information only to hosts downstream of the failed link. In most networks, this information is not useful because multicast packets take a wide variety of paths to reach their destinations; thus, a host cannot reason about the fate of packets sent to other destinations.

Timely uses its multicast design, and in particular the fact that all packets are routed via one of the core switches, to implement the final piece of Timely Unreliable Multicast: FAILURE notifications. When a switch detects a link failure, it sends a FAILURE notification to its downstream hosts. Since every packet is routed through one of the core switches, if the packet is dropped on its way to the core switch, no host will receive the failure notification. However, if the message reaches the core switch, every replica that has a failure-free path to the core switch will receive the message; more importantly, each replica that has the failed link along the path to the core switch will necessarily receive the FAILURE notification. By assumption, switches detect link failures within a bounded

amount of time; in addition, bounded queues guarantee the notification will arrive at the replicas within a bounded amount of time. Thus, replicas receive a FAILURE notification within a bounded amount of time after the failure.

3.4 Incorporating host processing delay

Recall that when a host receives a data packet, it add the data packet to an input buffer. When the host receives a `grant`, it processes the input buffer, executes logic, and then sends a data packet. As discussed in §2, this processing delay is bounded by some known value ϕ. Up until this point, we have considered the case of $\phi = 0$; replicas send data packets instantaneously upon receiving a `grant`. We now discuss how Timely design incorporates the case of $\phi \geq 0$. We first show the Timely Unreliable Multicast property holds even with non-zero processing delay. Finally, we provide end-to-end delay guarantees enabled by Timely Unreliable Multicast.

Timely Unreliable Multicast with $\phi \geq 0$

By using admission control, Timely guarantees zero persistent queueing and bounded transient queueing for the case of zero host processing delays. The challenge with non-zero processing delays is that, even if `grants` are issued exactly once per T, hosts may respond to `grants` after varying amounts of time. As a result, multiple hosts may send a data packet at the exact same time introducing queueing at the switches.

Our insight here is that host processing delays can introduce persistent

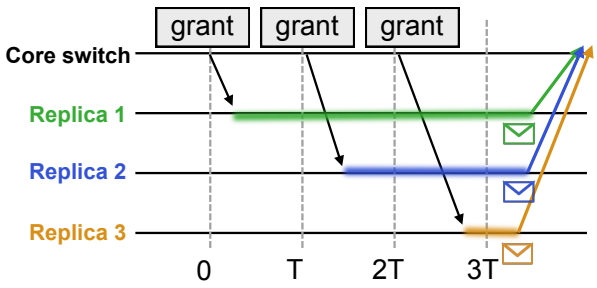


Figure 3.6: **Understanding the challenge due to non zero processing delay.** Processing delays are highlighted. Even though variable processing delay causes the three hosts to send data packets at the exact same time, the rate at which data packets enter over any period of nT time is limited by the admission control mechanism.

queueing but only by a bounded amount—Figure 3.6 demonstrates this. Intuitively, consider any period of time nT for any $n \geq 1$; during this time, due to Timely's admission control, the maximum number of data packets that any host can send into the network is $\lceil \phi/T \rceil + n$ since ϕ amount of processing delay could at most accumulate $\lceil \phi/T \rceil$ `grants` that the host can respond to in a burst. Out of these, n data packets will be dequeued since they were explicitly admitted during this time. Thus, during any arbitrary period of time, a host can inject at most $\lceil \phi/T \rceil$ "extra" data packets into the network, which the network may not be able to drain. As a result, network elements may now see persistent queueing; however, persistent queueing will still be bounded by the transmission time for $\lceil \phi/T \rceil$ data packets—and hence, bounded message delay guarantees are not violated (as long as data packets are not too large and ϕ/T is not too large; in practice, as discussed in §2, both of these conditions hold). We prove this formally in Appendix 3.

3.4.1 Timely message processing guarantees

The Timely Unreliable Multicast primitive guarantees that a message sent by a sender arrives at all replicas within a bounded amount of time (or receives a FAILURE notification). Combined with bounded host processing times, Timely provides an even stronger property: if a host finishes processing at some time t, it guarantees that the host is now "up-to-date" with all messages sent by all other hosts until certain time in the past. Formally, there exists some fixed known Δ such that if a hosts finishes processing at some time t, it has either: (i) processed a FAILURE notification; or (ii) processed all messages sent before $t - \Delta$. Let c be the number of core switches, let `RTT` be the round trip time for a data packet along the longest path when there is no other packet in the network, and let T be the transmission time for a data packet and for a `grant` request. Then, for the Timely design, we prove in Appendix 3 that $\Delta = \delta + \delta_f + \phi$, where δ_f is the failure notification time, and

$$\delta = 4[(\lceil \phi/T \rceil + c) \cdot T) + \texttt{RTT}/2]$$

is the message delay bound. Note that each of δ, δ_f and Δ is a fixed constant (that depends only on network hardware).

CHAPTER 4

CONSENSUS PROTOCOL

In this section, we describe the Timely consensus and state machine replication protocols that build upon the Timely Unreliable Multicast primitive.

4.1 The power of Timely Unreliable Multicast

We demonstrate the power of the Timely network design by designing two additional primitives on top of its Timely Unreliable Multicast primitive: untimely reliable broadcast, and message timers. We will use these primitives as a black-box in our consensus and state machine replication protocols in the next subsection.

4.1.1 Untimely reliable multicast

The Untimely reliable multicast primitive that we design in this subsection satisfies two properties:

- If a host sends a message to a set of hosts $\mathcal{S}$ and at least one non-faulty host in $\mathcal{S}$ receives the message, then all non-faulty hosts in $\mathcal{S}$ *eventually* receive the message;
- If a non-faulty host sends a message to a set of hosts $\mathcal{S}$ then all non-faulty hosts in $\mathcal{S}$ *eventually* receive the message[1].

[1]These two properties are equivalent to validity and agreement in the reliable broadcast problem.

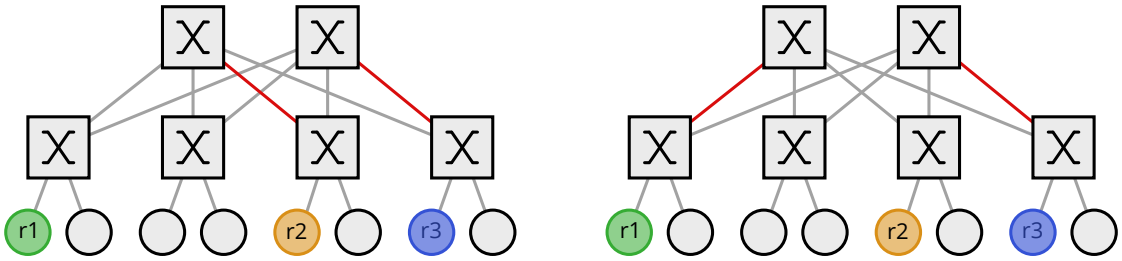


Figure 4.1: **An example to motivate the mechanisms in Untimely reliable multicast**. Links in red are crashed. (left) any message routed through the left core switch will not arrive at $r2$. Similarly, any message routed through the right core switch will not arrive at $r3$. Thus, to achieve untimely reliable multicast in this network, each message must be routed via all core switches. (right) it is impossible for $r1$ to send a message through each core switch; if it wants to send a message to each of $r1$ and $r2$, it needs to first send the message to $r2$, and then $r2$ can send the message via each core switch.

Timely uses the structure of datacenter network topologies to achieve untimely reliable multicast. We first show that to achieve untimely reliable multicast, a message must be routed via each core switch. Figure 4.1(left) demonstrates the necessity of this. One way to achieve this is for the sender to send one copy of the message through each core switch using the Timely Unreliable Multicast primitive from the previous section. However, as shown in Figure 4.1(right), simply sending a replicated copy of the message via all core switches is also not sufficient. In general, when a replica receives an untimely reliable multicast message, it "echoes" the message to all the core switches it is connected to. As long as there is no replica partition, every connected non-faulty replica will eventually receive the message. Note that since the correctness condition only requires the message to be delivered eventually, no bounded message delay properties are needed, and it is okay for the message to be queued at other replicas.

The protocol is as follows. Each host maintains multiple output buffers, one per core switch. To send a message, the sender adds a copy of the message to

Algorithm 1 Untimely reliable multicast protocol

```
Each host maintains one output buffer per core switch
procedure UNTIMELYRELIABLEMULTICAST(message)
    add message to the output buffer of each core switch
procedure RECEIVE(packet)
    if packet is a grant from core switch c then
        send a message from c's output buffer
    else if packet is a new U.R.Multicast message then
        add packet to the output buffer of each core switch
```

each output buffer. When a `grant` is received from a core switch, the sender sends a message from the corresponding output buffer. When a host receives the message, it responds by "echoing" the message, by adding a copy of the message to its own output buffers. Again, when a `grant` is received from a core switch, this host sends a message from the corresponding output buffer. As an optimization, a receiving host can keep track of which copies of the message it has received, and only echo the message through the remaining core switches. The pseudocode for the untimely reliable multicast protocol is given in Algorithm 1.

4.1.2 Message timers

Timely Unreliable Multicast also enables what we call message timers—each end host uses the number of messages received to "estimate" a *lower bound* on the amount of time passed on the global clock since the start of the timer. If a message timer reads x then at least x, possibly more but definitely not less, time has passed on the global clock since the start of the timer.

Packets are meaningful measurements of time in the Timely network for two reasons. First, core switches send `grants` at a designated pull frequency; thus,

the local clocks of individual core switches becomes a meaningful entity: one `grant` is one tick of the clock. This clock tick is guaranteed to be slower than global clock tick because, as discussed in §2, core switch clocks cannot be faster than the global clock. Next, `grants` and their corresponding data packets are delivered within a bounded amount of time. This preserves the spacing between clock ticks at individual core switches—if a core switch sends `grants` at most once per cT (with respect to switch's local clock), at most $\lceil \frac{\phi}{cT} \rceil + n + 1$ data packets can be received by any host from a single core switch within any cnT period of global time.

Message timers only require a lower bound on the amount of time passed. This definition is important for two reasons. First, this makes the correctness of message timers resilient to link failures, and the resulting packet drops. If a packet is dropped in the network, the message timer will not increase; as a result, the host will underestimate the amount of time that has passed which is correct since only a lower bound is required. Second, this definition makes message timers correct even when a host does not know the exact frequency at which each core switch sends `grants`. Recall that pull frequency of a virtual network may change as applications are added and removed from the network. Thus, the pull frequency may change without the knowledge of a host. If pull frequency changes, `grants` will be issued to a single application less frequently than once per T. In this case, the timer will increase less frequently than global time and correctness is maintained.

To implement a message timer, each replica maintains a packet count per core switch. For every data packet received from a core switch, the replica increases the corresponding packet counter by 1. To reset the timer, all packet

counters are set to 0. To read the time since the start of the timer, the replica converts each packet counter to time. The conversion from packets to time is based on the pull frequency of the network. We know at most $\lceil \frac{\phi}{cT} \rceil + n + 1$ data packets can be received from a single core switch in cnT time. Rearranging terms, if the packet counter reads n, at least $(n - 1 - \lceil \frac{\phi}{cT} \rceil) \cdot c \cdot T$ time has passed since the start of the timer. Once each packet counter is converted to time, the message timer returns the maximum time read by any packet counter. The packet counter with the highest value is the most accurate: it is the largest underestimate among many correct underestimates.

Because Timely message timers depend on data packets to increment their time, it is advantageous to send a data packet each time a `grant` is received. Thus, if a replica receives a `grant` but does not have a data packet to send, it sends a dummy packet instead. This improves the efficiency of the message timers but is not needed for correctness; Timely message timers are correct even if the number of data packets transmitted is less than the number of `grant`s sent.

4.2 Timely Consensus

In this section, we present the core of our state machine replication protocol. In state machine replication, clients send requests to replicas. Replicas commit the request to a slot in the log and then send a response to the original client. Replicas need to agree which request goes in which slot of the log. We focus on replicas agreeing on one slot in the log. For this, each replica has a value v_i that it would like to propose for the slot. This can easily be extended to a full state machine replication protocol (Appendix C).

We use all the four properties from the Timely design so far: (1) all messages sent using Timely Unreliable Multicast arrive at all replicas in a bounded amount of time; (2) when a replica finishes processing at some time t, it has processed all messages sent before $t - \Delta$; (3) untimely reliable multicast; and, (4) message timers. We first describe the case of no network hardware failure case, and then provide two simple extensions to handle failures.

Timely consensus for the case of no network hardware failures. The core of this algorithm, originally introduced in [11], is that with bounded message delays, replicas can use "wait times" to determine the status of other replicas. If a replica goes at least 2Δ time without hearing from another replica, it can provably determine that it has received the same set of messages as any other replicas that have also waited 2Δ time. If it has received the same set of messages, consensus can be reached. The challenge is to ensure that the algorithm continues to provide guarantees in presence of network hardware failures.

Specifically, the algorithm begins with each replica proposing its value v_i by sending a PROPOSE packet. When a replica receives a PROPOSE packet, it adds it to a set of all received values. If at least 2Δ time passes without receiving a message, the replica sends a DECIDE containing the minimum of all received values. Any time a replica processes a DECIDE packet with a value v, it decides v, regardless of the values received.

This algorithm is given in Algorithm 2, ignoring the colored text. An example execution of this protocol is provided in Figure 4.2. There are three replicas: $r1$, $r2$ and $r3$. $r1$ sends PROPOSE(1) and that message is received by all other replicas. $r2$ receives that message at time t. Right before $t + 2\Delta$, $r3$ receives a `grant` from core switch 1 and sends PROPOSE(0). Shortly after $t + 2\Delta$, $r2$ re-

Algorithm 2 consensus with network failures

```
proposedValues;
isActive = true
// process input buffer
for i = 0...inputBufferSize do
    pop a message from the input buffer
    if message is FAILURE then
        set isActive to false
    else if message is DECIDE(v) or ECHO(v) then
        if not decided then
            send ECHO(decision) using U.R.Multicast
            decide decision
    else if message is PROPOSE(v) then
        add v to proposedValues
        reset messageTimer
// generate and send data packet
if this replica has not proposed then
    send PROPOSE(v_i) using T.U.Multicast
else if messageTimer ¿ 2Δ and isActive then
    decision = min(proposedValues)
    send DECIDE(decision) using T.U.Mulicast
```

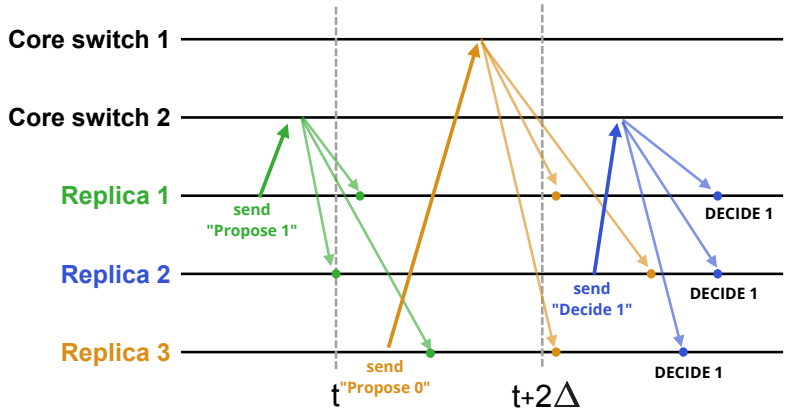


Figure 4.2: **Illustration of Timely Consensus protocol in a failure-free scenario.** See §4.2 for detailed discussion.

ceives a `grant` from core switch 2. $r2$ did not receive PROPOSE(0) yet so it sends DECIDE(1). Because replicas must wait 2Δ before sending DECIDE, each replica receives DECIDE(1).

We prove the safety and termination guarantees for the above protocol in

Appendix B. Of course, consensus is not guaranteed in this algorithm in case of network hardware failures. If a proposal for some value v is received by some replica but not others it is possible that one replica decides v while a different replica decides some other value v', violating safety. If DECIDE packets are consistently dropped by the network due to failures, it is even possible that a replica never decides a value, violating termination. Next, we explain how Timely adjusts this algorithm to guarantee safety and termination in the presence of network hardware failures. We introduce two ideas: witness replicas and ECHO packets, that will help guarantee safety and termination, respectively.

Guaranteeing safety during network failures with witness replicas. To guarantee safety during network failures, we designate replicas as *active* and *witness* replicas. A replica is active until it receives a FAILURE notification, when it becomes a witness. Active replicas participate in consensus as specified above. Witness replicas participate in consensus, but cannot send DECIDE packets.

To see why this works, notice that safety is violated if a replica does not receive information that other replicas do receive. Replicas that have received a FAILURE notification have "incomplete" information; they may have missed proposals and decisions that other replicas received. These witness replicas thus cannot be trusted to make a correct decision and should not send DECIDE packets. More formally, we know from the case of no network failures that replicas will never send contradictory DECIDE packets if each replica receives the same set of messages. Using Timely Unreliable Multicast, active replicas all receive the same set of messages. Safety is thus preserved among active replicas and these replicas can participate in consensus fully.

We make two notes here. First, if every replica becomes a witness then

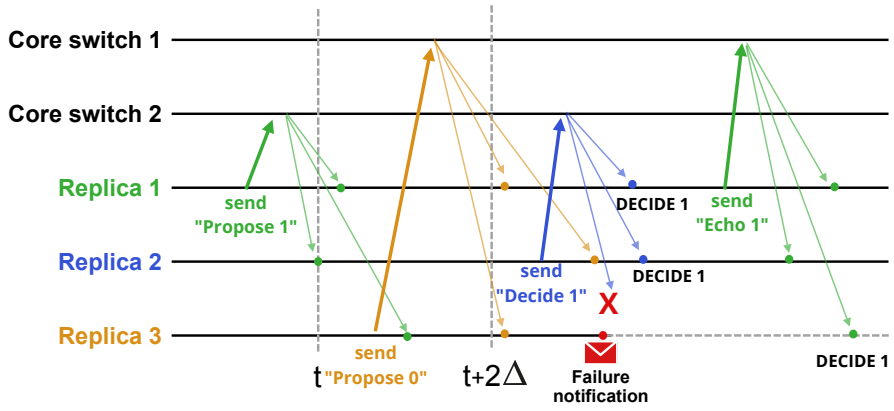


Figure 4.3: **Illustration of Timely Consensus during failures.**

no replica will ever send a DECIDE, violating termination. As a reminder, we guarantee termination as long as there is at least one non-faulty, completely-connected, replica. This replica will never become a witness. Second, we have a race condition—a replica may think it is active, send a DECIDE, and then receive a delayed FAILURE notification. This race condition is handled by the defintion of Δ.

These changes are presented in algorithm 2 in purple.

Guaranteeing termination with ECHOs. The termination property requires that every non-faulty replica eventually decide a value. The main challenge in guaranteeing termination in the presence of network failures is that DECIDE packets may be dropped. To guarantee termination, we use the untimely reliable multicast primitive that guarantees every replica will eventually receive the message. If we send DECIDE packets using untimely reliable multicast, all replicas will eventually receive a decision. Since packets sent using untimely reliable multicast are not guaranteed to arrive in a bounded amount of time, we use two types of decision packets: DECIDE packets and ECHO packets. DECIDE packets are sent using Timely Unreliable Multicast, and ECHO packets are sent using

untimely reliable multicast. When a replica is ready to decide, it first sends a DECIDE packet. This initial DECIDE packet ensures all replicas receive the decision or a FAILURE notification within a bounded amount of time. If a replica receives DECIDE(v), it sends ECHO(v) using untimely reliable multicast. If a replica receives ECHO(v), it continues the untimely reliable multicast protocol. It is safe for a replica to decide v once a copy of the ECHO has been sent through each core switch that the replica is connected to. These changes are presented in algorithm 2 in blue. An example execution of algorithm 2 of this is given in Figure 4.3.

State machine replication optimization. As an optimizaton, replicas can decide on a *set* of values per slot. If each replica decides the same set, and values can be ordered consistently within a set, correctness is maintained. To do this, after 2Δ time passes without receiving a PROPOSE for the slot, a replica sends a DECIDE containing the set of proposed values received so far. There is a bound on the size of the decided set because each replica still proposes at most one value per slot.

CHAPTER 5

EVALUATION

This section presents evaluation results for an end-to-end implementation of Timely design, the Timely Unreliable Multicast communication model, and the Timely consensus and state machine replication protocols. Over a small-scale testbed, we compare Timely with five existing state machine replication protocols—NOPaxos, SpecPaxos, FastPaxos, MultiPaxos, and Multi-Paxos+Batching on our testbed (using the implementation provided by [25]).

We present three sets of results: (1) evaluation of all protocols in terms of their latency-throughput curves; (2) comparison of Timely protocol with NOPaxos (that improves upon SpecPaxos) during "imperfect" network conditions; and (3) sensitivity analysis of Timely using simulations. Before diving deeper into our evaluation results, we make an important note. For the first set of results, our goal is not to improve upon existing protocols in terms of latency-throughput curve—in fact, since Timely provides stronger guarantees than all existing protocols (in terms of number of replica failures for which safety can be guaranteed, etc.), we do not expect Timely to significantly outperform Spec-Paxos and NOPaxos; the benefits Timely gets in our evaluation for the first set of results are merely an experimental artifact, due to different implementations using different network stacks. Perhaps more importantly, our second set of results are indeed fundamental: we will demonstrate that when the network behaves imperfectly (hardware failures and/or packet drops), Timely protocol is able to maintain much higher performance than existing protocols. We will provide more intuition inline.

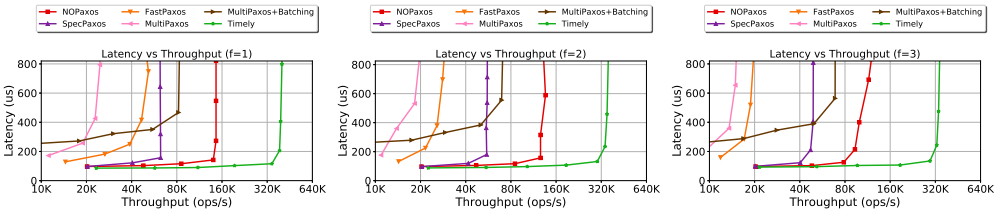
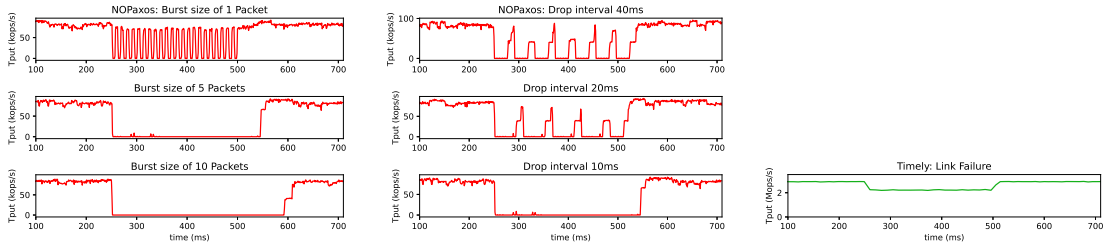


Figure 5.1: **Performance of all protocol our an 8-server testbed for varying number of tolerable failures (**$1, 2$ **and** 3 **from left to right).** Discussion in §5.2.



(a) NOPaxos: Increasing Burst of drops

(b) NOPaxos: Increasing Drop Frequency

(c) Timely: Link failure

Figure 5.2: **NOPaxos observes much worse performance impact due to network imperfection (packet drops due buffer overflows and/or failures) than Timely (packet drops due to failures).** Discussion in §5.2.

5.1 Evaluation Setup

Timely implementation and testbed. We have implemented Timely using ~2000 lines of code on top of DPDK [3]. Using DPDK allows Timely packets to bypass the kernel, thus avoiding unpredictable delays[1]. We run our experiments using on a testbed with 8 hosts connected to a single switch; each server has 3.4GHz Intel Xeon 3680 CPUs, and 256GB RAM. We use a `grant` frequency of 1.32μs for Timely, Δ = 60μs, 50B control packets and 100B data packets.

Performance metrics. We measure the latency for each completed request (that is, the time duration between the request generation and response arrival at the

[1]In our experiments using DPDK, we observed packet round-trip tail latencies to be bounded by 30μs.

client), and the overall throughput for the application (that is, the total number of successful requests per unit time). We present the latency-throughput curves for different setups, each having enough replicas to tolerate $1, 2$, and 3 failures (this requires $2, 3$, and 4 replicas for Timely, $3, 5$, and 7 replicas for all other protocols).

Simulation setup. To dive deeper into Timely performance at larger scales, we have also implemented Timely on top a packet level simulator. We use a standard two-tier full-bisection bandwidth leaf-spine topology [9, 29, 17] consisting of 144 hosts, 9 aggregate switches and 4 core switches. The hosts use a 10Gbps access link bandwidth and the aggregate to core switch links are 40Gbps. The link propagation and switching delays are set to be 200ns and 450ns, respectively, and the switches are assumed to have 16MB buffer capacity [4, 1, 2]. The host processing delays are set to be 10μs and the maximum one-way latency in this network with the Timely queue bounds is 6.5μ. Taking into account processing delays, we set $\Delta = 33\mu$s.

5.2 Testbed Results

Figure 5.1 shows the latency vs throughput performance for evaluated replication protocols on the testbed. Timely is able to provide a maximum throughput of $\sim$ 340k ops/s and latency of $\sim 88\mu$s. For all the other schemes, the relative trends are same as in the NOPaxos paper [25]. FastPaxos, MultiPaxos, and MultiPaxos+Batching are leader-based schemes, and are bottlenecked by the leader on throughput, resulting in much lower maximum achievable throughput. SpecPaxos and NOPaxos however do not require a leader for the aver-

age case and thus can sustain higher throughput. As discussed earlier, the performance benefits of Timely over NOPaxos are merely an experimental artifact here—one uses kernel network stack, while the other uses kernel bypass.

Failures and packet drops. We use microbenchmarks to evaluate Timely and NOPaxos in presence of failures and packet drops. We could not evaluate link failures on our testbed: a link failure with one switch makes the affected host completely disconnected from the network. For NOPaxos, we instead drop packets in bursts. In NOPaxos, packets may be dropped due to congestion in the network. If this occurs, a burst of packets will be dropped until congestion reduces. By design, the Timely network does not drop packets due to buffer overflows. Thus, we evaluate Timely in the presence of link failures in our simulator (§5.3).

For NOPaxos, we drop packets sent to the leader in bursts and vary the burst frequency and the number of packets dropped in a burst. In these experiments, the bursts begin at 250ms and end at 500ms. Figure 5.2(left) shows NOPaxos throughput when dropping 1, 5, and 10 packets every 10ms. In NOPaxos, when a packet is received with a sequence number higher than the next expected sequence number, a timeout is set for 2ms. If the expected sequence number does not come before the timeout, a gap commit protocol is initiated to commit a gap to the log. When 1 packet is dropped, throughput drops to zero for the length of the timeout plus the duration of the gap commit protocol. Throughput then recovers until another packet is dropped 10ms later. When 5 and 10 packets are dropped, the sum of the timeouts for all dropped packets is 10ms and 20ms, respectively, which is larger than the burst frequency. The leader does not have enough time to recover from the first burst before another begins and

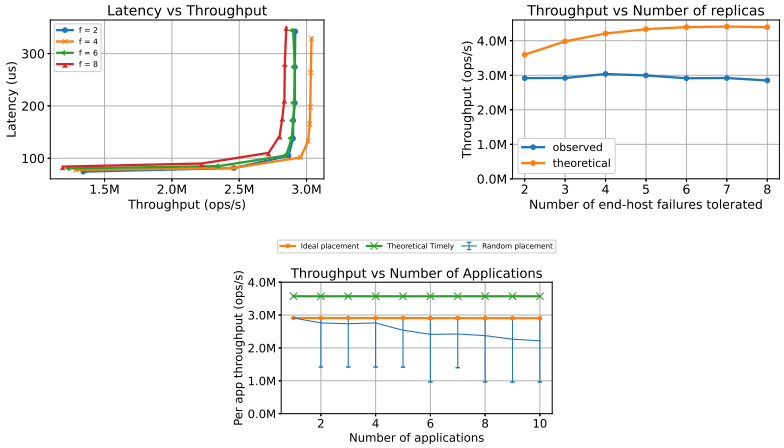


Figure 5.3: **Understanding Timely performance over a larger-scale topology.** Latency with varying number of replicas (left), Throughput with varying number of replicas (center) and throughput with varying number of applications (right).

hence throughput is close to zero. The effects of these bursts are long-lasting: in both cases zero throughput lasts for at least 50ms after the bursts end. Figure 5.2(center) shows the throughput with bursts of size 5 dropped every 40, 20, and 10ms. With drop intervals of 40ms and 20ms, throughput is zero during the burst, recovers, and then goes back down to zero during the next burst. These results are highly dependent on the packet drop timeout value. This timeout, however, has a fundamental tradeoff. With a high timeout, replicas unnecessarily wait before initiating the gap request protocol. With a low timeout, a gap request protocol may be initiated while an out-of-order packet is still in flight.

Timely, on the other hand, guarantees to achieve throughput (1-$\frac{\texttt{\#failed\_replicas}}{\texttt{\#replicas}}$) even under failures. For this setup, the crashed link was connected to one replica (out of three) and 2 client hosts (out of 10). When the link crashes, throughput drops from 2.9M ops/s to 2.2M ops/s. This throughput drop comes from two sources: (1) The core switch connected to the crashed link issues unanswered `grants` to the affected replica and thus fewer total data

packets are sent and (2) client requests and responses are dropped and need to be retried. The throughput recovers fully within 10ms of the link recovery.

5.3 Timely sensitivity analysis

We run large-scale simulations to understand Timely performance at scale. The performance trends even for large datacenter topology remain similar to the testbed results discussed previously, and the sensitivity analysis shows us that Timely is robust in terms of obtainable throughput with varying setup parameters. For this subsection, unless specified otherwise following baseline configuration is used—144 node topology, 3 replicas, single application, and no failures.

Baseline (Figure 5.3(left)) . Simulation results showcase latency vs throughput trends similar to the testbed results. The maximum sustainable throughput is $\sim$ 2.9M. and the latency with a small number of clients is 75μs. The biggest sources of delay are (1) Waiting for 2Δ time and (2) replica processing delays for REQUEST, PROPOSE, and DECIDE packets.

Increasing number of replicas. Figure 5.3(center) shows the observed throughput for increasing number of replicas (and thus, increasing fault tolerance) for the same application. We compare the observed throughput to the Timely theoretical maximum throughput: the Timely throughput assuming each slot decides as many values as possible in as few packets as possible. The theoretical throughput increases as the number of replicas increases: more values can be committed per slot, data packet sizes grow, and pulls are sent less frequently and take a smaller percentage of the bandwidth. The observed throughput re-

mains stable due to timing inefficienies. With more replicas, it is more likely that multiple DECIDE packets for the same slot are sent at the same time.

Increasing number of apps. Figure 5.3(right) shows maximum throughput with an increasing number of concurrent applications (1 to 9). We show three cases (1) maximum theoretical throughput (2) maximum throughput with each replica on a different hosts (ideal placement) (3) maximum throughput with random replica placement. If replicas are ideally placed, the per app maxmimum throughput remains stable; applications do not need to share bandwidth and thus the pull frequency of each application is unaffected by other applications. If replicas are distributed randomly across all hosts, throughput degrades with an increasing number of applications. With a random distribution, it is possible replicas from two applications share an end-host. These applications must share bandwidth, reducing per app throughput. With increasing applications, the chance two replicas share an end-host increases, resulting in decreasing per app throughput.

CHAPTER 6

RELATED WORK

Consensus. The theoretical bounds of consensus are well understood. The famous Fischer, Lynch, and Paterson impossibility result [13] proves asynchronous consensus is not possible with even one faulty end-host. [23] explores lower bounds needed to achieve asynchronous consensus. [11] outlines four sychronous environments in which consensus is possible. This is the basis of our Timely consensus algorithm. [12] explores the possibility of consensus with various definitions of partial synchrony.

State Machine replication. There is a long line of work that improves state machine replication in practice. Viewstamped Replication [32] [26], FastPaxos [22] and EPaxos [30] are all variants of the classic Paxos [21] protocol. Raft [33] presents simplified state machine replication protocol. Each of these protocols attempts to improve performance of Paxos while maintaining the same guarantees: 2f + 1 replicas are needed where f is the number of faulty end-hosts.

Recent work focuses on the role of network ordering in state machine replication. [37] designs a new datacenter network primitive called Mostly Ordered Multicast and uses that primitive to design a new state machine replication protocol SpecPaxos. In SpecPaxos, replicas speculatively commit values to a log. Application support is needed to roll back speculative commits. The NOPaxos [25] state machine replication protocol is designed using another datacenter network primitive called Ordered Unreliable Multicast. Ordered Unreliable Multicast sends requests through a single network sequencer and then multicasts sequenced requests to each replica. Ordered Unreliable Multicast is not resilient

to packet drops and thus 2f + 1 replicas are needed to ensure safety despite packet drops. The NOPaxos sequencer is a single point of failure.

Bounded-queue networks. Most existing bounded queue networks pause packet transmissions at a link-by-link granularity to guarantee switch buffers do not overflow. These designs prevent packet drops due to buffer overflow but do not provide bounded message latencies: a packet may be stuck at a switch for an unbounded amount of time while its outgoing link is on pause. Fastpass [36] provides a zero-queue network with bounded packet latencies using a centralized arbiter for admission control. This arbiter can quickly become a bottleneck. Timely provides bounded packet latencies without a centralized arbiter.

CHAPTER 7

CONCLUSION

Traditionally, consensus protocols have been designed under the best-effort delivery communication model. Recent research has demonstrated that datacenter networks can enable a stronger communication model, one where the network provides message ordering, leading to high-performance consensus and state machine replication protocols. This paper explores a different communication model that can be enabled by datacenter networks—bounded message delays—and demonstrates that this model enables design of consensus protocols that guarantee strong properties, including safety and termination even when all but one replica fails. Using analytical and empirical results, we have demonstrated that Timely protocols enable operating points that were previously unachievable.

APPENDIX A

TIMELY UNRELIABLE MULTICAST PROOF

In this section, we prove *if a host multicasts a message to a set $\mathcal{S}$ of hosts at time t on the global clock, and at least one host in $\mathcal{S}$ receives the message, then all non-faulty hosts in $\mathcal{S}$ either receive the message by time $t + \delta$ on the global clock, or receive a failure notification by time $t + \delta + \delta_f$ on the global clock, for fixed known δ and δ_f.*

To do this, we prove switch queue sizes are bounded. We then restate our defintions our of, and prove correctness of, δ and δ_f

A.1 Bounded switch queues

To prove bounded switch queues, we prove that at most $(\lceil \phi/T \rceil + n + c)$ data packets can enter the network in any span of nT time.

First, consider the case of c core switches and processing delay 0 and one application. Each core switch issues a `grant` at most once per cT time. The worst case scenario is if core switches do not coordinate and issue c pulls at the exact same time. If this happens, another burst of c `grants` will not be sent until cT time later. Thus in any span of cT time, at most 2c `grants` can be sent (c at time 0 and c at time cT). Extending to NT time, for some N, at most N `grants` can be sent in NT time. Thus at most (N + c) `grants` are received by replicas in NT time.

Now, we extend to how many data packets can enter the network in any span of time. Data packets are only sent in response to `grants`. In any span of

nT time, a replica can respond to grants received over a period of $\phi + nT$ time: grants received during the span of nT time plus grants received up to ϕ before that time. Rearranging terms, in any span of nT time, replicas can respond to grants received over a period of $(\lceil \phi/T \rceil + n)T$ time. Substituting for $(\lceil \phi/T \rceil + n)$ for N above, at most $(\lceil \phi/T \rceil + n + c)$ grants can be received in $\phi + nT$ time. Thus at most $(\lceil \phi/T \rceil + n + c)$ data packets enter the network in any span of nT time.

We this, we claim queues grow to size at most $(\lceil \phi/T \rceil + c)$. At most $(\lceil \phi/T \rceil + n + c)$ data packets can arrive aggregate switches in a period of nT time (either all to one aggregate switch or distributed among aggregate switches). At least n can be drained in nT time and thus upstream aggregate switch queues are bounded by $(\lceil \phi/T \rceil + c)$. Thus at most $(\lceil \phi/T \rceil + n + c)$ can arrive at core switches in nT time, leading to the same switch queue bound. The same logic applies to downstream aggregate switch queues and we have bounded queueing at every switch.

Extending to multiple applications. The bottleneck link in the Timely design is the downstream link between aggregate switches and hosts. Core switch links only transmit a portion of the packets - those corresponding to that core switch. The downstream links between aggregate switches and hosts transmit all packets from all core switches. Additionally, since packets are multicast, these links transmit strictly more messages than the upstream links (if there are multiple host senders).

Our results hold as long as for each of these links, grants are issued on average once per T. If an application does not share a host with any other applications, and that application is issued on average one grant per T, the downstream aggregate switch links transmit on average one data packet per T, satisfying the

queue bounds above. If an application shares a host with x applications, and each application is issued on average one `grant` per xT, the shared link still transmits a data packet once per T. As long as each core issues `grants` to the different applications on the same host at different times, which can be easily controlled by a core switch, the results hold.

A.2 Timely Unreliable Multicast guarantees

Definition of δ. If RTT is the round trip time for a data packet along the longest path when there is no other packet in the network, then all packets that are not dropped by the network will arrive at their destination by RTT/2 plus the queuing delay. If each queue is bounded by $(\lceil \phi/T \rceil + c)$ then the maximum queueing delay for any packet on a single switch is $(\lceil \phi/T \rceil + c)T$. In a two tier leaf-spine topology there are four hops, and thus the maximum queueing delay is $4(\lceil \phi/T \rceil + c)T$. Thus if a packet is sent at time t and is not dropped by the network, it will arrive at its destination at time $t + \text{RTT}/2 + 4(\lceil \phi/T \rceil + c)T = t + \delta$.

Definition of δ_f. Let δ_f be the link failure notification delay: if a link crashes at time t_f an aggregate switch can detect an adjacent link failure, generate FAILURE notification, and send the notifications and affected end-hosts will receive the nofication by time $t_f + \delta_f$.

Timely Unreliable Multicast correctness. According to the design of Timely core switch based multicast, if a packet is sent at time t and is dropped before reaching the core switch, no end-hosts will receive the packet and the Timely Unreliable Multicast property is guaranteed. If it reaches the core switch, all

end-hosts connected to that core receive the packet. By the definition of δ, this packet will be received by $t + \delta$. If an end-host is not connected to the core switch due to a link failure and the packet is dropped, the link must fail before the packet reaches the end-host; the link must fail before $t + \delta$. In this case, affected end-hosts will receive a notification by time $t + \delta + \delta_f$

Untimely reliable multicast.

If a message goes through each core switch then every message connected to a core switch will receive the message and untimely reliable multicast is satisfied.

If the sender is non-faulty, it will send a message through every core switch it is connected to. Either the sender is completely connected, or it is not. If the sender is completely connected, it will send a message through every core switch and untimely reliable multicast is satisfied. If the sender is not completely connected, it sends a message through a subset of core switches C. Let H be the set of end-hosts connected to at least one core switch in C. If H contains every non-faulty end-host, untimely reliable multicast is satisfied. If there exists a set of hosts H' that are not in H, there exists some non-faulty host h in H that can communicate with at least one host h' in H' (by the definition of no replica partition). In this case h with echo the message and h' will receive it. After this echo, add all core switches that the message has been sent through to C and all hosts that have received the message to H. The logic repeats until H' is empty and all non-faulty end-hosts have received the message.

If the sender is faulty, it could send a message through some of the core switches it is connected to but not all. Either some end-host receives the mes-

sage or some end-host does not. If no end-hosts receive the message, untimely reliable multicast is vacuously satisfied. Otherwise, let H_f be the set of hosts that receive the message. If there is least one non-faulty end-host in H_f, that non-faulty end-host will send the message through each core switch it is connected to, and the logic follows the same logic as the non-faulty sender logic above. If all of the end-hosts in H_f, either one will manage to send it to a new host or all will crash before the message is received by a new host. In the latter case, none of the non-faulty end-hosts received the message and untimely reliable broadcast is vacuously satisfied. In the former, the size of H_f grows by at least one. Repeat the logic above until either a non-faulty host is in H_f or all hosts in H_f have crashed.

APPENDIX B

CONSENSUS PROOF

We first prove safety and termination for one slot in the log by making small modifications to the proof from [11]. We first prove a corallary of the defintion of Δ provided in 3.4:

Claim: For $\Delta = \delta + \delta_f + \phi$, if an end-host sends a DECIDE at some time t it processed all messages sent before $t - \Delta$ that were received by at least one end-host.

Proof: If an end-host sends a DECIDE at t, it has processed all packets and FAILURE notifications received before $t - \phi$. Further, since it has sent a DECIDE, it did not receive a FAILURE notification before time $t - \phi$. Accounting for the failure notification delay, there was no network failure that affected the end-host before time $t - \phi - \delta_f$. If there are no relevant network failures at that time, the end-host has received all messages received by at least one end-host sent before the maximum message latency. Thus is has received all messages received by at least one end-host sent before $t - \phi - \delta_f - \delta = t - \Delta$.

Consensus safety: Assuming no replica partition and at least one non-faulty end-host, every end-host that decides a value decides the same value.

Proof: If a replica decides a value, it has already sent ECHO(v) using Untimely Reliable Multicast. By the correctness of Untimely Reliable Multicast with no replica partition and a non-faulty sender, every end-host will receive the ECHO(v). Thus an replica cannot decide a value and crash before informing other end-hosts of the decision.

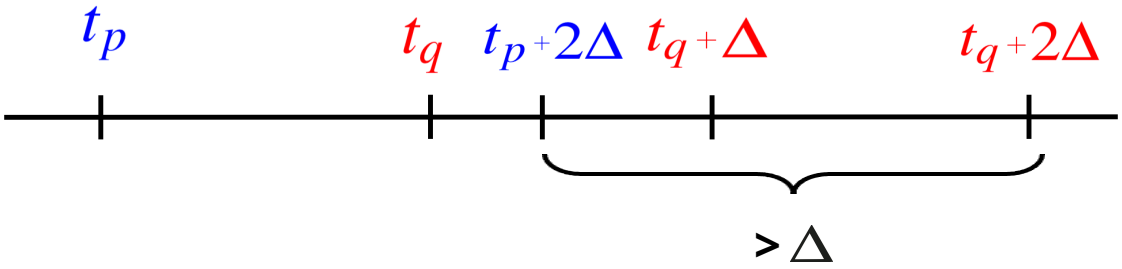
A replica only decides v if it receives a DECIDE(v) or ECHO(v) packet. An ECHO(v) is only sent if a replica receives a DECIDE(v). Thus to prove safety it suffices to prove that if a replica recieves DECIDE(v), no end-host will ever receive DECIDE(v') for some $v \neq v'$.

Assume two end-hosts receive different DECIDE packets. These DECIDE packets were sent by different end-hosts. Let p be the first end-host to transmit a DECIDE(v) that is received by at least one end-host. Let q be the first end-host to transmit a DECIDE(v') (for $v \neq v'$) that is received by at least one end-host. If there is a tie, assign p and q arbitrarily. Let t_p be the global time of p's last update and t_q be the time of q's last update. By our implementation of message clocks, p and q send their decision packets no earlier than $t_p + 2\Delta$ and $t_q + 2\Delta$ in global time, respectively.

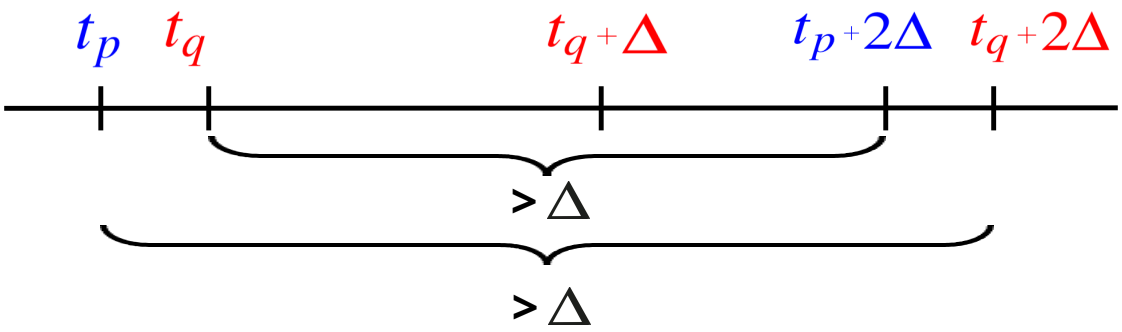
If $t_q + \Delta \geq t_p + 2\Delta$ then q will process DECIDE(v) before $t_q + 2\Delta$ [using the invariant above] and q cannot send DECIDE(v'). Thus $t_q + \Delta < t_p + 2\Delta$. If $t_q + \Delta < t_p + 2\Delta$ then $t_q < t_p + 2\Delta$ and q must have processed everything at time t_q that p processed at $t_p + 2\Delta$. Similarly, $t_p + 2\Delta < t_q + 2\Delta$ so $t_p < t_q + 2\Delta$ so p has processed everything at t_p that q processed at $t_q + 2\Delta$. If both p and q send DECIDE, they have not received a FAILURE notification so they have both received every successful message sent and no other messages. Thus q at $t_q + 2\Delta$ has processed the same set of values as p at $t_p + 2\Delta$ and q cannot send DECIDE(v'). See figure B.1.

Termination: If there is at least one non-faulty completely connecte end-host then every non-faulty end-host eventually decides a value.

Proof: First, we show that a DECIDE packet eventually reaches at least one



(a) If $t_q + \Delta \geq t_p + 2\Delta$ then $t_q + 2\Delta$ is at least Δ time after $t_p + 2\Delta$. Thus when p sends a DECIDE at time $t_p + 2\Delta$, the DECIDE will arrive to q before $t_q + 2\Delta$.



(b) If $t_q + \Delta < t_p + 2\Delta$ then there is at least Δ time between t_q and $t_p + 2\Delta$. Thus q at t_q has processed the same set of messages that p has at $t_p + 2\Delta$. Similarly, there is at least Δ time between t_p and $t_q + 2\Delta$ and p at t_p has received the same set of messages as q at $t_q + 2\Delta$.

Figure B.1

core switch. Every end-host can attempt to propose at most once. Thus there exists a time when completely connected end-hosts stop receiving proposals. After that, 2Δ time passes and as long as there is at least one completely connected end-host then a DECIDE will be sent. Since the sender is completely connected, that DECIDE will reach the core switch. If the DECIDE packet reaches a core switch then one non-faulty completely connected end-host will receive that DECIDE and successfully send an ECHO through each core switch, ensuring every connected end-host receives an ECHO and eventually decides.

APPENDIX C

EXTENDING TO STATE MACHINE REPLICATION

For state machine replication, replicas use the consensus protocol 2 to agree on each slot in the log. For a individual replica, a slot is "open" if the replica has not received a proposal for that slot. Replicas maintain a pointer to the lowest open slot in the log. When a replica receives a request from a client, it proposes the request to the lowest open slot. If at least 2Δ time has passed without receiving a proposal for any slot in the log that is not open, the replica can send a DECIDE for that slot. The untimely reliable multicast and decision protocols follow as in algorithm 2.

As is standard practice, replicas keep track of the last request decided for each client to ensure duplicate requests are not committed twice.

Safety follows directly from the correctness of the consensus protocol. As long as there is one completely connected non-faulty end-host, termination is guaranteed; that end-host will continue proposing to and deciding slots in the log.

Algorithm 3 state machine replication

```
isActive = true
unproposedValues
proposedValues
procedure PROCESSINPUTBUFFER
    for i = 0...inputBufferSize do
        pop a message from the input buffer
        if the message is FAILURE then
            set isActive to false
        else if the message is DECIDE(v, s) or ECHO(v, s) then
            COMMIT(v,s)
        else if the message is PROPOSE(v, s) then
            add v to proposedValues[s]
            reset messageClock[s]
        else if the message is REQUEST(v) then
            add v to unproposedValues
procedure CONSENSUSLOGIC(coreId)
    if there is a slot s such that the messageClock[s] reads 2Delta and isActive
then
        decision = min(proposedValues[s])
        send DECIDE(decision, s) using T.U.Multicast
    else if unproposedValues is not empty then
        v = a value from unproposedValues
        firstEmptySlot is the lowest slot that has not received a propose
        send PROPOSE(v, firstEmptySlot) using T.U.Multicast
procedure COMMIT(decision, slot)
    if log[slot] == null then
        add decision to log[slot]
        send ECHO(decision, slot)) using U.R.Multicast
        if this replica received the original request for decision then
            send response to the original client
```

APPENDIX D

RECOVERIES

When an end-host or network component recovers, end-hosts need to coordinate on state before continuing with the protocol. We define "affected" end-hosts as the end-hosts affected by the failure. For end-host recoveries, it is the recovering end-host. For link recoveries, it is the end-hosts connected to the aggregate switch of the failed link. For switch failures, it is all the end-hosts.

The key challenge in managing failures and recoveries is thus determining which slots are safe for any end-host to decide. If a single end-host is recovering (either due to an end-host recovery or link recovery) the affected end-hosts ask other end-hosts which slots are in-progress and which slots have not received any proposals. The affected end-hosts can partipicate in slots for which proposals have not been received; all other end-hosts can continue working on the in-progress slots. If all end-hosts are recovering (due to a switch recovery), any in-progress slot is unsafe. For this, end-hosts coordinate to determine which slots are in-progress. A No-op is committed to in-progress slots and end-hosts can commit values to open slots.

The link, end-host, and switch recovery protocols are outlined below. These protocols assume switch local clocks do not drift arbitrarily from one another.

D.1 Link Recoveries

When a link recovers, the aggregate switch sends a Recovery notification to all downstream end-hosts. Affected end-hosts then run a link recovery protocol.

There's three link recovery cases: (1) the end-host was connected before and becomes completely connected (2) the end-host was disconnected before and becomes connected (3) the end-host was connected before and is now more connected. If the end-host was disconnected before, it was faulty. This case will be handled in end-host crash recoveries. If the end-host was connected and stays connected, nothing changes; the end-host still needs to act as a witness. We focus on the case where the end-host transitions from connected to completely connected.

When an end-host becomes completely connected, it first has to wait out any race conditions in the network. It waits δ time to ensure any in-flight packets are received by all non-faulty end-hosts. It then sends a LRECOVERY message using Timely Unreliable Multicast. When a completely connected end-host receives a LRECOVERY message, it replies with its highest slot for which a value has been proposed. The recovering end-host waits for a response from any completely connected end-host. Competely connected end-hosts receive all messages and thus its highest in-progress slot is the global highest in-progress slot. The recovering end-host acts as a witness for slots up to the in-progress slot and participates fully in consensus for slots higher than the max in-progress slot.

Algorithm 4 Link Recoveries

```
if now completely connected then
    wait δ + δ_f time
    send LRECOVERY message using T.U.Multicast
    wait for a reply containing max in-progress slot s
    for slots < s do
        act as a witness
    for slots ≥ s do
        participate in consensus fully
if receive LRECOVERY message then
    if completely connected and not recovering then
        send a response containing max in-progress slot s using T.U.Multicast
```

D.2 End-Host Recoveries

To recover, an end-host needs to determine two things. First, it needs to learn all the values that were committed while it was crashed. Next, the end-host needs to determine which slots it can partipate in. If the end-host is connected but not completely connected, it acts as a witness for all slots. If it is completely connected, it needs to learn the highest in-progress slot. As in the link recovery case, it can partipate in consensus fully for slots larger than the max in-progress slot.

The recovery protocol is as follows. First, the recovering end-host waits out any race conditions by waiting $\delta + \delta_f$ time. It then sends a ERECOVERY message to using Timely Unreliable Multicast. The ERECOVERY message contains the last value committed before the end-host crashed. When a completely connected end-host receives an ERECOVERY message, it does two things. First, it replies with the highest in-progress slot using Timely Unreliable Multicast. Next, it executes a log transfer, sending every committed value that the recovering end-host missed. As the recovering end-host receives the log transfer, it commits values. If it is completely connected, tt can begin participating in consensus fully for slots higher than the highest in-progress slot.

D.3 Switch Failures and Recoveries

When a switch fails, all end-hosts receive a FAILURE notification and become a witness. If each end-host is a witness, end-hosts never make decisions. This is safe but not ideal; throughput would be zero for the duration of the switch fail-

Algorithm 5 End-host Recoveries

```
if recovering then
    wait δ + δ_f time
    send ERECOVERY(highest decided slot) using T.U.Multicast
    wait for a reply containing max in-progress slot s
    wait for a log transfer
    commit values received during the log transfer
    if completely connected then
        for slots < s do
            act as a witness
        for slots ≥ s do
            participate in consensus fully
    else
        act as a witness for all slots
if receive a ERECOVERY(highest decided slot) then
    if completely connected then
        reply with max in-progress slot using T.U.Multicast
        send log from highest decided slot to max in-progress slot
```

ure and recovery protocol. To resolve this, end-hosts agree to "ignore" switches and modify their definition of completely connected accordingly. An end-host is completely connected if it has a failure free path to all switches that are not currently ignored.

More formally, each end-host maintains a list of active switches. Our switch failure and recovery protocol guarantees that if there is no ongoing switch failure or recovery protocol, each non-faulty end-host has the same list of active switches. An end-host is completely connected if there is no ongoing switch failure or recovery protocol and the end-host has a failure free path to each switch in active switches. The list of active switches serves as an imperfect failure detector. If all end-hosts agree that a switch is faulty, they can ignore that switch, regardless of whether or not the switch is actually faulty. Thus, if an end-host receives a `grant` from a switch not in active switches, it does not respond to the `grant`. Accordingly, if an end-host receives a data packet from a switch not in

active switches, it drops the data packet before processing it.

Maintaining correctness in the face of ignored switches has two challenges. First, end-hosts need to agree on which switches to ignore. Second, end-hosts need to ensure safety each time a switch reconfiguration occurs.

To agree on which switches to ignore, an end-host sends its proposed switch configuration change to all end-hosts. It then waits for replies. When an end-host receives a proposed change, it replies whether it agrees or not (again using reliable broadcast). If all end-hosts agree, the new switch configuration is used. If any end-host disagrees, the old configuration persists. All proposed changes and replies are sent using reliable broadcast, ensuring each end-host knows of each proposal and whether it succeeds.

For this to work, each end-host needs to know when it has received all replies. It cannot wait for n replies; an end-host may be faulty and never reply. For this, we lean on the fact that reliable broadcast takes a bounded amount of time as long as there is a bounded number of concurrent reliable broadcast messages. We bound the number of concurrent reliable broadcast messages. When an end-host proposes its change or replies to a change, it sents a timer for the max reliable broadcast time bound. After that, it can be sure it has received all replies. We will go into the details of the reliable broadcast bound in the next subsection. For now, it suffices to know that any timer is set for reliable broadcast time bound.

To ensure safety each time a switch reconfiguration occurs, end-hosts agree on which slots are in-progress and which slots are open. A no-op is committed to each in-progress slot.

In the following sections, we explain how to agree on switch reconfigurations and maintain safety while doing so. We first consider the case of removing a switch. We then consider the case of adding a switch. Finally, we show these protocols maintain a bounded number of concurrent reliable broadcast messages.

D.3.1 Removing switches

To handle the case of removing switches, we need to know when to attempt to remove a switch, how to agree on removing a switch, and how to maintain safety upon removing a switch. To decide when to ignore a switch, we use the fact that liveness is violated if a switch is crashed.

If significatnly more than 2Δ time passes without a decision, it is likely that none of the end-hosts are completely connected and there could be a switch failure. For some timeout $> 2\Delta$, if timeout time passes without receiving any decisions, an end-host sends a SWITCHREMOVE message containing the list of switches it is connected to. Any time an end-host receives a SWITCHREMOVE message, it replies with the list of switches it is connected to. After an end-host receives all SWITCHREMOVE messages for a given reconfiguration attempt, it decides whether to remove a switch. If there exists a switch that is not connected to any end-host, the end-hosts ignore that switch.

Once a switch is ignored, end-hosts need to agree on which slots are safe to commit. Any slot that is not decided by any end-host and has recived a PROPOSE is not safe. Each end-host reliably broadcasts its highest decided slot and its max in-progress slot. End-hosts commit a no-op to all slots between the

globally highest decided slot and the global max in-progress slot. Consensus can continue on slots higher than the max in-progress slot.

To reduce the number of concurrent reliable broadcast message, we combine these two step. Pseudocode is provided below.

Algorithm 6 Switch failure detector

```
if TIMEOUT time has passed without receiving a DECIDE then
    initiate the remove switch protocol
```

Algorithm 7 Switch Remove

```
myMaxDecide is the highest slot the replica has received a DECIDE for
myMaxPropose is the highest slot the replica has received a PROPOSE for
if a reconfiguation is in progress then
    set a timer and then try again later
else
    send SWITCHREMOVE (activeSwitches, myMaxDecide, myMaxPropose,
recoveryID)
    set a timer
    after timer time has passed, check all packets received for recoveryID
    if there is a switch disconnected from everyone then
        remove switch from activeSwitches
        maxDecide is the maximum maxDecide from all packets
        maxPropose is the maximum maxPropose from all packets
        commit a no-op for slots between maxDecide and maxPropose
        continue consensus for slots higher than maxPropose
when receive a switchRemove(switches, maxDecide, maxPropose, recov-
eryID)
if recoveryID is a new id then
    send SWITCHREMOVE (activeSwitches, myMaxDecide, myMaxPropose,
recoveryID)
```

D.3.2 Adding Switches

When an end-host becomes connected to a switch currently being ignored, the end-host initiates a protocol to add that switch. The end-host sends a

SWITCHADD message. The message is reliably broadcast to all end-hosts. When an end-host receives a SWITCHADD message, it adds the switch back to its list of active switches and updates its definition of completely connected accordingly. Once the reliable broadcast routine terminates, any end-host connected to the newly added switch can use that switch freely.

Algorithm 8 Switch Add

```
if a reconfiguration is in progress then
    set a timer and try again later
else
    send SwitchAdd(switch, recoveryID)
    set a timer
    when timer goes off: add switch to activeSwitches
when receive a switchAdd(switch, recoveryID)
add switch
set a timer
when timer goes off: add switch to activeSwitches
```

D.3.3 Untimely reliable multicast bound

A replica only initiates a reconfiguration if it does not know of any other reconfigurations in progress. Thus at most n reconfigurations can occur concurrently (if each replica starts a reconfiguration at the exact same time). Each time a switch is removed, at most n messages are sent using untimely reliable multicast: one per replica. The switch add protocol uses just one untimely reliable multicast message. Thus at most r^2 untimely reliable multicast messages are sent concurrently.

If there are r^2 concurrenly reliable broadcast messages, each message can take at most $r^2(r+1)T_vc\Delta$ time. To see why, let a "chain"[1] be the number of

[1] this concept closely follows the idea of a dangerous chain in the reliable broadcast problem

hosts a message needs to travel through for it to reach some destination host h. The maximum possible chain is c (the number of core switches). To see why, consider that each host in the chain must increase the number of core switches the message has been sent through by at least 1. Otherwise, all hosts are faulty or there is a network partition. For each transmission, either a host expands the number of core switches. There are at most c core switches so the chain can be of length at most c.

It takes at most $(k + 1)rT_v\Delta$ time to expand the length of a chain, where r is the number of replicas in the virtual network and T_v is the transmission time of a data packet using the bandwidth assigned to that virtual network and k is the number of concurrent untimely reliable multicast messages. A message may wait in a receiver's input buffer for at most rT_v time before the host receives a `grant` and processes the message. Once it processes the message, copies of the message are added to the output buffer. If there are most k concurrent messages, the output buffer is of size at most k. Thus the message will be sent after at most krT_v time in the output buffer.

BIBLIOGRAPHY

[1] Barefoot Networks. "https://www.barefootnetworks.com/".

[2] Cisco Switches. "https://www.cisco.com/c/en/us/products/switches/".

[3] Data Plane Development Kit. "https://dpdk.org/".

[4] Ethernet Switches and Switch Fabric Devices. "https://www.broadcom.com/products/ethernet-connectivity/switching/".

[5] Ieee standard for ethernet–amendment 8:physical layer specifications and management parameters for 2.5 gb/s, 5 gb/s, and 10 gb/s automotive electrical ethernet. *IEEE Std 802.3ch-2020 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018, IEEE Std 802.3bt-2018, IEEE Std 802.3cd-2018, IEEE Std 802.3cn-2019, IEEE Std 802.3cg-2019, IEEE Std 802.3cq-2020, and IEEE Std 802.3cm-2020)*, page 161, 2020.

[6] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *International Symposium on Distributed Computing*, pages 231–245, 1998.

[7] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[8] Mohammad Alizadeh and Tom Edsall. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st annual symposium on high-performance interconnects*, pages 71–74. IEEE, 2013.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*, 2013.

[10] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 17–28, 2014.

[11] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal

synchronism needed for distributed consensus. *Journal of the ACM*, 34(1), 1987.

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[14] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 281–297, 2020.

[15] Albert Greenberg, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, Dave Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *ACM SIGCOMM*, 2009.

[16] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can {JUMP} them! In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 1–14, 2015.

[17] 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 *ACM SIGCOMM*, 2017.

[18] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The case for a network fast path to the cpu. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 52–59, 2019.

[19] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 489–502, 2014.

[20] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[21] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[22] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2), 2006.

[23] Leslie Lamport. Lower Bounds for Asynchronous Consensus. *Distributed Computing*, 19(2), Oct. 2006.

[24] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. June 1979.

[25] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 467–483, 2016.

[26] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. In *Technical Report MIT-CSAIL-TR-2012-021*. MIT Computer Science and Artifical Intelligence Laboratory, 2012.

[27] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[29] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*, 2018.

[30] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[31] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie perfor-

mance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[32] Brian M. Oki and Barbara Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of distributed computing (PODC)*, pages 8–17, 1988.

[33] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, 2014.

[34] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 361–378, 2019.

[35] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[36] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *ACM SIGCOMM*, 2014.

[37] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–57, 2015.

[38] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source routed multicast for public clouds. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 458–471. 2019.

[39] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. Fastlane: making short flows shorter with agile drop notification. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 84–96, 2015.

[40] 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. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.