Predictable Performance and Low Cost for Geo-Distributed Applications

by

Muhammed Uluyol

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Associate Professor Harsha V. Madhyastha, Chair
Assistant Professor Mosharaf Kabir Chowdhury
Assistant Professor Manos Kapritsos
Assistant Professor Neda Masoud

Muhammed Uluyol
uluyol@umich.edu
ORCID iD: 0000-0002-7712-7136

To my family.

ACKNOWLEDGMENTS

All praise is due to Allah, the Lord of the worlds, the Benevolent, and the Merciful. With His aid and guidance, I have been able to complete this task, and I hope that some benefit comes from it.

I have been very fortunate with the mentors I have had over the past seven years. First and foremost, my advisor, Harsha, pulled me into his group when I didn't really understand what I was doing. He patiently guided and molded me until I became somewhat capable of performing research independently. He connected with me many wonderful collaborators and has always been available when I needed help from him. Working with Harsha has been one of the highlights of my PhD.

I thank the members of my dissertation committee: Manos, Mosharaf, and Neda for their valuable feedback. I am especially grateful to Mosharaf, whom I have worked closely with on multiple projects and from whom I have sought advice on multiple occasions. His humor and straightforward thinking is always appreciated.

I would also like to thank my peers who I saw regularly at the office. My fellow members of BBB 4929 (Andrew Quinn, Ayush Goel, David Devecsary, Jingyuan Li, Joseph Lee, Vaspol Ruamviboonsuk, and Zhe Wu), Andrew Kwong, Ashkan Nikravesh, Jie You, and Shichang Xu have been helpful for bouncing ideas around, walking back and forth to the water cooler, and making life easier.

Two people got their hands dirty to help push my research projects past the finish line. Anthony Huang integrated EPaxos into PANDO's experimental setup, identified GitLab's workload properties, and worked on PANDO's TLA+ specification. These contributions really helped provide a solid story for PANDO's performance claims and correctness. Ayush analyzed some data for PANDO and – despite having other things on his plate – found time to debug some tricky issues with the scalability evaluation for HEYP. I appreciate all of their help.

I would also to thank the long list collaborators I have had at Nutanix and Google. At Nutanix, Karan Gupta was very open to discussing ideas and encouraged me to come up with wacky ideas. At Google, I had a big circle of excellent collaborators including Ben Zhang, Chi-Yao Hong, Dina Papagiannaki, Jonathan Zolla, Kirill Mendelev, and Sankalp Singh. I would also like to thank Amin Vahdat and Subhasree Mandal for their support, along with Amin (again), David Wetherall, Jeff Mogul, Morley Mao, and David Culler for the valuable feedback.

Finally, I would like to thank my family, who has supported me patiently. My parents, who put up with me as a child, teenager, and young adult have been a continuous source of support and care. My wife has put up with more moves living with me than she has during the rest of her life. I thank her for her unwavering kindness and hope this madness will end soon. Towards the end of my PhD, I was blessed with my daughter, Ameena, who is a continuous source of happiness in my family.

This marathon is now over. I am glad I did it, but I don't think I'll be doing it again!

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF APPENDICES

LIST OF ALGORITHMS

ALGORITHM

ABSTRACT

Applications that run in the cloud must be geo-distributed to achieve high reliability, minimize delays, and follow data localization laws. However, geo-distributed applications face many challenges in achieving these goals due to a key bottleneck: the wide-area network (WAN). Without enough, cheaply-available WAN bandwidth, applications are forced to degrade the service they provide or shut down altogether. Additionally, the latency between data centers limits how quickly an application can serve user requests.

My dissertation makes two contributions to better serve providers of geo-distributed applications. First, I present a new architecture, HEYP, for sharing a private WAN across many tenants. State-of-the-art WANs maximize efficiency by trying make use of every bit of spare capacity on the network. However, in doing so, they risk introducing interference between tenants. In contrast, HEYP offers strong isolation guarantees between tenants, but without sacrificing the efficiency of existing shared WANs. Next, I characterize the impact that latency has on read and write operations when manipulating data that is stored in multiple data centers, and how there exists a three-way tradeoff between optimizing for read latency, write latency, and cost. I study existing approaches and identify inefficiencies that cause them to require significantly higher cost than necessary to meet a set of latency targets. I then describe the design of PANDO, a system that I have developed to offer a near-optimal tradeoff between read latency, write latency, and cost.

Both PANDO and HEYP offer substantially improved tradeoffs between performance and cost that can benefit geo-distributed applications. More generally, however, my work shows that current geo-distributed systems offer tradeoffs that are suboptimal in important ways, and that focusing on the constraints of WANs will be a key part of improving them.

CHAPTER 1

Introduction

Two trends – the increasingly global reach of the Internet and our growing dependence on online services – are changing the considerations application providers need to account for when designing their software. First, as Internet access improves in more parts of the world, these applications (i.e., online services) are serving users that are spread across larger geographic distances from one another. However, because long-distance data transfers over the Internet can be slow, it is often challenging for application providers to deliver good performance. Furthermore, applications that are present in multiple countries must be adapted to comply with the regulations for data localization that are being enacted around the world [51, 61]. These laws restrict where providers of online services can access and store data. Second, our society's increasing reliance on these services means that providers need ensure that applications remain online. We require online services to work [74, 73, 157], shop [1, 7], and teach [58, 3], and we take notice when these services are down [83, 173, 97, 102].

To meet these demands, providers are spreading application data and logic across multiple data centers that are each located in a different geographic region. Figure 1.1 shows how a hypothetical video conferencing application might adopt such a *geo-distributed* architecture. Each user interacts with a nearby instance of the service. When users call one another, the application relays the video streams between the instances used by each user.

This architecture offers several benefits.

- By storing each user's data in nearby data centers, the application can quickly authenticate users and comply with any data regulations.
- When individual data centers fail, the application can remain online by redirecting traffic to the remaining healthy instances.

However, in order to adopt a geo-distributed architecture, an application provider must address a number of challenging problems. These problems include how to share the available resources across services, how to maintain a consistent view of the data when multiple instances of the application write in parallel, and where to store data.

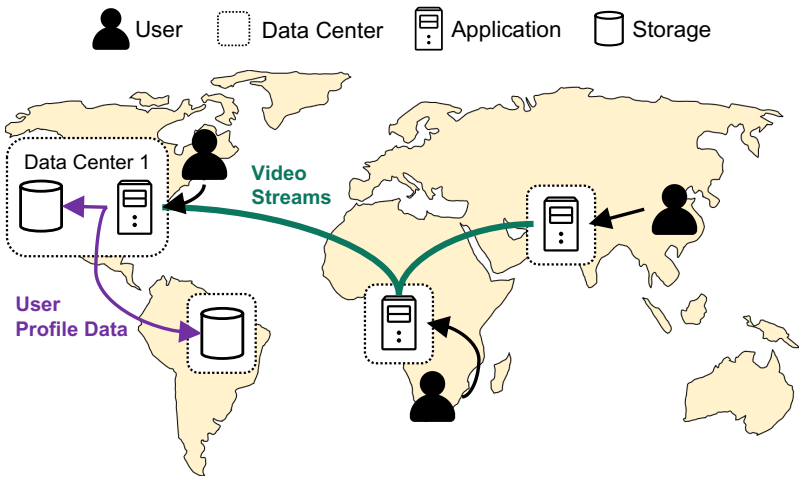


Figure 1.1: Architecture of a hypothetical geo-distributed video conferencing application.

These problems are not new – applications that use more than one machine within a data center must address them as well. Because of efforts made in both academia [178, 87, 140] and industry [16, 4, 149, 2, 125, 21], application providers can leverage widely-available infrastructure to address many of these challenges when distributing an application across machines that are located within a single data center.

However, one cannot simply transplant systems that operate within a single data center to run across data centers and hope that they will perform well. Compared to applications that run within a single data center, geo-distributed applications face unique constraints.

1.1 WAN is a key bottleneck

The wide-area networks (WANs) that connect data centers to one another have fundamentally different properties compared to the networks that are used within data centers.

- **Latency is high and non-uniform.** Within a data center, the round trip time (RTT) between two machines is limited to a few hundred microseconds [28, 129, 181]. In contrast, WAN RTTs can be tens to thousands of times longer, depending on the distance between data centers and the specific routes used to send data. It is therefore important to pay attention to not only the *number* of rounds of communication that take place, but also *where* the communication occurs (i.e., between which pairs of data centers).

  To understand the impact this has on system design, consider a replicated storage system. The system maintains multiple copies of the data for fault tolerance, but it needs some mechanism to keep the copies synchronized. This synchronization can be achieved by funneling all requests through a single leader, and electing a new leader upon failure. Within a data center, the over-

head of using a leader may be acceptable since the network RTT may be small compared to queuing and storage delays (e.g., Microsoft's CosmosDB targets a latency bound of 10ms for read requests [125], or about 100× higher than the RTT). However, across data centers, this approach risks incurring high latency for requests that are sent by data centers that are distant to the leader.

Over time, this discrepancy between latency for data center networks and WANs will persist. Any improvements to network RTT are fundamentally limited by the speed of light, and data center networks span smaller physical distances than WANs.

- **Bandwidth is expensive.** WANs that are used to connect data centers to one another can cost hundreds of millions of dollars [89]. It is therefore prudent to minimize the amount of the WAN (bandwidth) capacity required if one aims to minimize cost or support higher volumes of load in a given topology [89, 98, 105, 176, 103].

  In contrast, typical data center networks offer large volumes of bandwidth between all pairs of machines [156, 32, 151, 181] – i.e., data center networks have low oversubscription ratios – which has led some use cases to focus on responsiveness over efficiency [63, 29, 99, 117].

These limitations suggest that distributed systems that support geo-distributed applications should be designed to make the best use of the WAN. Of course, this must be balanced against other desirable properties. For example, for a storage workload that has few writes but many reads, a WAN-optimal strategy may be to replicate all data to all data centers. However, the storage cost of this approach may be prohibitive, especially if only a small subset of data is being read.

For most problems, a tradeoff will exist between WAN optimization and the other goals at hand. The question is, how much can these tradeoffs be improved?

1.2 Thesis and Contributions

I have explored this question in the designs of two systems that support most, if not all, geo-distributed applications: inter-data center WANs and storage systems. Despite the effort that has been poured into these problem areas, the results of my research, which I present in this dissertation, support the following thesis: ***it is practical to improve performance versus cost tradeoffs in geo-distributed applications.***

1. Reliable delivery of WAN bandwidth with HEYP. Returning to our example application in Figure 1.1, consider what happens when less bandwidth is available than needed to transfer full-quality video streams between data centers: the application will be forced to reduce the video quality and possibly reject a subset of calls to avoid overloading the network.

To minimize the likelihood of this, the application provider must provide sufficient capacity

between data centers to support its expected peak bandwidth. Since the Internet does not offer performance guarantees, the provider must allocate the bandwidth in a private WAN.

Building a private WAN requires large capital expenses [89], but cloud providers amortize these costs by sharing a single WAN across a large number of tenants. Although this provides savings through economies of scale, further savings can be realized. A large portion of WAN capacity is built to handle atypical circumstances – e.g., link failures – so state-of-the-art WANs continuously reconfigure the network to utilize this spare capacity to serve excess, opportunistic traffic.

However, the high efficiency of existing WAN architectures comes at the cost of predictability. Although WAN controllers distinguish between excess traffic and traffic that is within each tenant's guarantee, no distinction is made when the traffic is being forwarded between network switches. The result is that by providing additional bandwidth to admit excess traffic, state-of-the-art WANs risk delivering less bandwidth to tenants than promised.

I propose a new architecture, HEYP (for Highly Efficient, Yet Predictable), that matches the efficiency of state-of-the-art WANs while providing substantially improved predictability. The key to HEYP's performance is that it explicitly treats the excess traffic of each tenant separately from the portion of traffic that is within the tenant's guarantee. This enables HEYP to use more robust, but wasteful, mechanisms to ensure that bandwidth guarantees are met while leveraging highly efficient mechanisms to maximize efficiency.

To match the efficiency of state-of-the-art WANs, HEYP needs to partition the flows of each tenant into separate bins and route each bin over its own set of paths. This presents two main challenges. First, existing applications assume that each flow's traffic will be sent over a stable path. Therefore, HEYP must not only partition each tenant's flows to maintain a desired traffic volume in each bin, but it must also maximize the stability of the assignment. Second, many applications dynamically route requests to avoid bottlenecks in the network. While this improves application performance, it can interact poorly with HEYP's partitioning of flows by changing the volume of traffic in each bin. HEYP mitigates both issues; the goals of these mitigations are to preserve isolation between tenants and to ensure that applications can make use of the available bandwidth with few, if any, changes.

My experiments suggest that HEYP offers the highest predictability for satisfying bandwidth guarantees and matches the efficiency of state-of-the-art WANs. These results hold even under a sensitivity analysis where I alter the characteristics of both the WAN and the examined workloads. In a testbed that hosts an application workload with excess traffic, I find that the throughput offered by HEYP is within 12% of an optimal approach.

2. Near-optimal latency versus cost tradeoffs for storage with PANDO. Having established how cloud providers can deliver predictable network performance, I can turn my attention to designing systems that use the network without worrying about high performance variability. I focus

on the three-way tradeoff between read latency, write latency, and storage cost that affects geo-distributed storage systems.

Replicating data across data centers enables web services to serve users with low latency and tolerate the unavailability of any one data center. A web server close to a user can serve the user's requests by accessing nearby copies of relevant data.

There are several constraints that limit the read latency, write latency, and storage cost that can be achieved when one aims to maintain a single logical view of the data. First, writers must synchronize both with each other and with any readers. This is achieved by ensuring that, under all possible circumstances, a data center written to by the writer will be accessed by future reads and writes. By writing to data centers that are closer to readers, one can improve read latency, at the cost of increased write latency. Second, in order to offer low read latency, readers must be able to fetch data from a nearby set of data centers. By increasing the number of data centers (and therefore cost), one can improve read latency.

I study how existing approaches perform in this three-dimensional tradeoff, and find that the performance they achieve in one dimension (e.g., read latency) given constraints in the other two (e.g., write latency and storage cost) are suboptimal. Approaches that are optimized for use across a WAN store a full copy of each object at every site, and can therefore only offer low latency at high cost. In contrast, approaches that store only a portion of the data at every site (using erasure coding) offer low cost but at high read and write latency. The slowness is a result of two inefficiencies: writing data using multiple rounds of communication over the WAN, and contacting a larger number of sites than an optimal approach. Even when considering a combination of existing approaches, I find that the resulting latency–cost tradeoffs have significant room for improvement.

I introduce PANDO, our system design that achieves a near-optimal tradeoff between read latency, write latency, and storage cost. PANDO leverages erasure coding to optimize cost, but it addresses both sources of inefficiency with existing approaches. First, PANDO eliminates most of the latency cost of two-phase writes by rethinking how to execute them in WAN environment. Second, PANDO waits for responses from a minimal number of sites when there are no conflicting operations, and only waits for larger set otherwise.

Using a combination of measurement-driven analysis and a prototype deployment, I compare PANDO against state-of-the-art approaches. In the latency–cost tradeoff space, I find that PANDO reduces by 88% the median gap between achievable tradeoffs and the best theoretically feasible tradeoffs. Depending on the application, these benefits can offer both cost savings and performance improvements.

CHAPTER 2

HEYP: Highly Available Bandwidth Guarantees on Highly Utilized Cloud WANs

The public Internet offers no performance guarantees. Therefore, many large cloud providers have deployed their own private wide-area networks (WANs) [98, 89, 101, 31], wherein they provision appropriate network capacity to offer predictable wide-area bandwidth and latency to tenants under a range of failure scenarios and communication patterns. Additionally, via admission control [105] and judicious routing [98, 89], the cloud provider can limit bandwidth interference among tenants.

Since services do not always send traffic at their peak rate, statically configuring a WAN to reserve the bandwidth promised to each tenant and preventing tenants from sending at a higher rate will result in poor network utilization. Cloud providers instead leverage their centralized control of their WANs to dynamically reconfigure routes and admission rate limits in reaction to changes in traffic demands [98, 105, 89]. Based on its global view, a central controller can ensure that any unused capacity that remains after admitting guaranteed demands for bandwidth is shared among tenants' surplus demands as per its business policy. Our simulations using data from Google's large global, private WAN show that such an approach can satisfy 50% more of the traffic demands on average compared to static approaches.

Current WAN architectures for improving network utilization in this manner, however, significantly hamper predictability. For example, in the above-mentioned simulations, dynamically allocating bandwidth offers 99% or higher availability to 10$\times$ fewer bandwidth guarantees, as compared to static reservation. A key cause for this dramatically lower predictability is that, to use the bandwidth promised to it, a tenant has to often wait for the central global controller to throttle previously admitted surplus demands of other tenants and reconfigure routes. This is problematic because the speed with which a central controller can react to demand changes is fundamentally limited by two factors: 1) the extremely large scale of global WANs [98, 105, 103, 82, 126], and 2) the need to sequentially apply routing changes in order to prevent inconsistency in routing configurations across switches in the network [89]. These sources of delay will only worsen over time since cloud providers are constantly expanding the number of sites in their WAN [152, 90, 127, 104],

and it often takes multiple rounds of reconfigurations for the global controller to correctly estimate and accommodate a tenant's true demand.

To remove this dependence on the global controller for ensuring predictable performance, we argue that any tenant's surplus demands should explicitly be treated differently, and admitted at a lower quality-of-service (QoS) level. When a tenant ramps up its bandwidth usage while staying within its bandwidth guarantee, we can then rely on switches to prioritize its traffic, instead of having to reduce the admission for other tenants utilizing spare capacity. Consequently, global control delays no longer affect the cloud provider's ability to satisfy bandwidth guarantees. With this approach, a tenant bears the risk that its excess traffic is more susceptible to congestion. But, it is, after all, utilizing more bandwidth than was promised to it.

We realize this promise of using QoS downgrade with HEYP (for Highly Efficient, Yet Predictable), our new control plane architecture for private WANs. Our design addresses three challenges that are unique to large cloud provider WANs compared to prior work which has used this approach to offer bandwidth guarantees on the public Internet [57, 56].

First, we show that the use of QoS downgrade calls for a change in how the global controller computes routes, compared to the status quo [98, 89]. A single tenant's demand is often large enough that capacity from multiple routes must be dedicated to its traffic. Spreading a tenant's high and low priority traffic in the same proportion among all the routes for this tenant constrains which routes can be used to carry low priority traffic, consequently limiting network utilization. Therefore, HEYP installs separate paths for each tenant's promised and surplus bandwidth: stable paths for the former on which capacity is guaranteed irrespective of other tenants' demands, and periodically recomputed paths for the latter to opportunistically utilize unused capacity.

Second, the consequence of using separate paths for high and low priority traffic is that the subset of a tenant's flows which are downgraded cannot be independently determined in each control period. Since latency varies across routes, TCP's congestion control will degrade the performance of any flow which keeps flip-flopping between QoS levels. But, pinning each flow to a specific QoS for a set amount of time limits our ability to respond to demand changes. Instead, we introduce caterpillar hashing, a flow selection mechanism designed to maximize QoS stability. Whenever we need to decrease the fraction of a tenant's traffic that is downgraded, we do so by upgrading the last-downgraded flow, and vice versa to increase the fraction downgraded.

Lastly, in contrast to when every tenant is capped at the bandwidth promised to it, an application may respond to QoS downgrade of its surplus traffic by shifting load towards that subset of its tasks which offer better performance. Since these tasks are more likely to be the ones permitted to send high priority traffic, the net result will be the application sending more high priority traffic than allowed. In response, we can change the QoS assignment, but the application will again react to this change. To converge to a stable QoS assignment for any tenant's traffic, HEYP attempts to

identify that subset of the tenant's flows which, if admitted at high priority, cannot ramp up any further due to bottlenecks other than WAN link capacity (e.g., host CPU or NIC).

We evaluate HEYP using testbed experiments and simulations. In our simulations driven by traces obtained from Google, HEYP matches the efficiency obtainable with dynamic bandwidth allocation, and it delivers the availability of bandwidth guarantees afforded by static approaches. We observe similar results when we apply our prototype to an application workload. Tenants that are within their bandwidth guarantees are unaffected by those who have excess traffic, and applications which utilize spare capacity achieve throughput that is within 12% of an optimal approach.

2.1 Setting and Motivation

We focus on settings where a WAN administered by a single organization is shared by many tenants. In this setting, we aim to satisfy four goals.

1. Provide predictability by satisfying bandwidth approvals. Based on every tenant's anticipated needs, the provider approves a certain level of bandwidth per tenant between source and destination data centers; we refer to each $(tenant, src, dst)$ tuple as a flowgroup. An *approval* per flowgroup enables more judicious capacity planning compared to guaranteeing every tenant bandwidth in and out of each data center irrespective of its communication pattern [65].

Every approval comes with an associated SLO for the availability of the approved bandwidth, and optionally with guarantees on the length of paths used to route it. A higher availability SLO calls for more redundant bandwidth on the appropriate links to cope with failures; but, in this dissertation, we consider all approvals as having the same SLO, and we discuss support for multiple SLO levels in §2.5. We assume approvals are not oversubscribed, so all approvals will be satisfiable as long as the capacity lost due to failures is within the bounds that the network provider wishes to tolerate.

While there exists prior work for making bandwidth approvals resilient to failures [118, 41, 161], we focus on the more commonly occurring risk: rapidly-changing traffic demands.

2. Improve network efficiency by accommodating opportunistic transfers. Our secondary objective is to admit as much of each flowgroup's demand as feasible; a flowgroup's *demand* is the bandwidth it will consume given infinite WAN capacity. The network should typically be able to admit some above-approval demands as it must have spare capacity to tolerate failures, and tenants do not always fully utilize their approvals. Admitting above-approval demands also reduces the risk associated with under-estimation of desired bandwidth. To prevent tenants from becoming dependent on work-conserving bandwidth, they can either be charged for its use [130] or every flowgroup can occasionally be capped at its approval even if there is spare capacity.

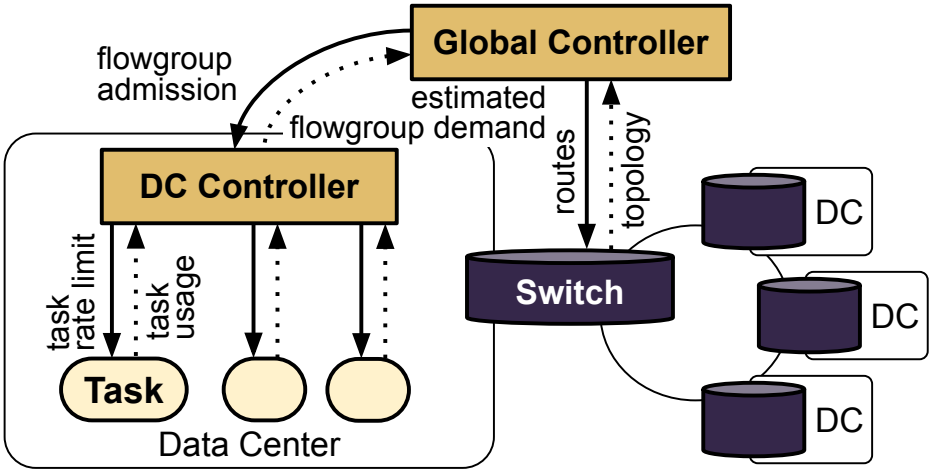


Figure 2.1: Architecture of a software-defined WAN.

3. Support flexible traffic engineering and bandwidth sharing policies. Network policies are rich [105], vary across providers [107, 98, 89, 78, 118], and evolve over time [90]. For example, traffic engineering policies face a tension between optimizing for latency or balanced load, and providers have to choose tradeoffs that are appropriate for their workloads and topology. Rather than dictating particular traffic engineering or bandwidth sharing policies, we aim to be flexible outside of the goals set forth in this section.

4. Maintain compatibility with TCP. As it is the most widely-used transport protocol, maintaining compatibility with TCP is necessary to avoid breaking applications. This restriction rules out certain design choices, e.g., because TCP requires most packets to be delivered in order, we cannot spray packets that belong to the same connection over multiple paths which differ in end-to-end latency.

2.1.1 Dynamic control across and in data centers

To meet these goals, current WANs are architected as shown in Figure 2.1. Within each data center, a DC controller collects bandwidth *usage* statistics for each flowgroup, aggregating measurements across *tasks*; we consider each instance of an application (i.e., a container or virtual machine) as a set of tasks, where each task sends traffic to a specific DC. From these usage statistics, the controllers *estimate* each flowgroup's demand, e.g., by taking the maximum usage across the past 90 seconds and inflating it by 10% [105]. Using these demand estimates along with the current topology, a global controller periodically adjusts routes to better satisfy demands and determines per-flowgroup *admissions* [105, 137] (i.e., how much aggregate bandwidth to admit for each flowgroup); in practice, separate global controllers may be used for routing and admission control [98, 105]. The DC controllers divide any flowgroup's admission across its tasks and con-

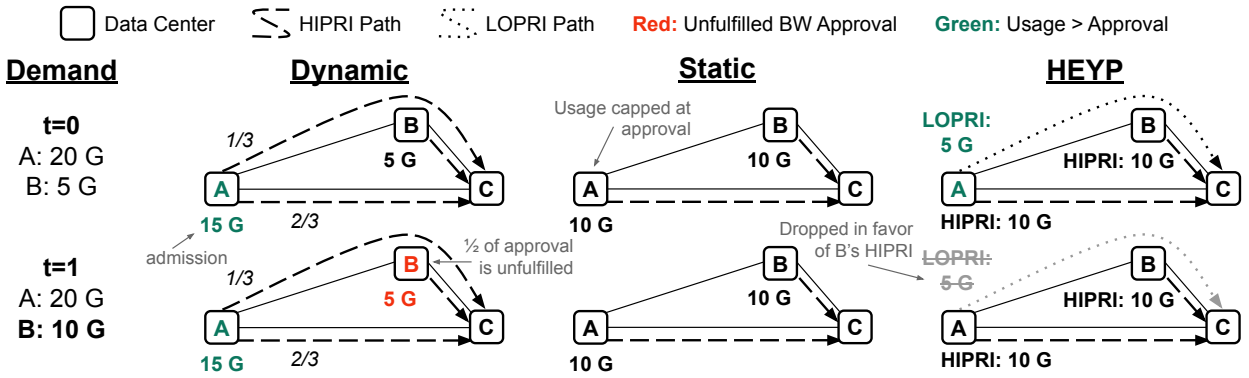


Figure 2.2: An example where dynamic control prematurely throttles traffic and static allocation fails to fully use network capacity. All links have 10 Gbps capacity. Approvals for both A→C and B→C are 10 Gbps. Initial demands (top) are 20 Gbps for A→C and 5 Gbps for B→C. After limiting it to send at most 15 Gbps, Dynamic sends 33% of A→C's traffic through B and the remaining 67% directly to C. This allows it to fully utilize the network. Later (bottom), when B→C's demand rises to 10 Gbps, Static admits the increased within-approval demand but Dynamic does not. HEYP provides the best of both: it fully utilizes the network with the initial demands and later accommodates the increased demand for B→C.

tinuously revise this split as demands change. Each task paces its sending rate so as to stay within the programmed rate limit [105, 89, 153].

A key advantage of this approach, compared to distributed approaches such as RSVP-TE [36], comes from the central controller's global visibility. When the network is unable to completely satisfy all demands, the global controller can easily enforce any bandwidth sharing policy desired by the network provider. These include, but are not limited to, allocating bandwidth to tenants in proportion to their payments, max-min fairness, and maximizing throughput.

2.1.2 Global control delays are a key bottleneck

To understand the sensitivity of dynamic control to delay, consider the example in Figure 2.2. For simplicity, we ignore failures and route each flowgroup's traffic over the shortest path. If that does not provide enough capacity, we recursively add the next shortest path to the flowgroup's routes.

With Dynamic (left), the global controller first satisfies within-approval demands; it allocates the shortest path for each approval and sets the admissions to 10 Gbps for A→C and 5 Gbps for B→C. It then allocates leftover capacity to surplus demands; it installs a second route for A→C to utilize the spare capacity between B and C, and increases A→C's admission to 15 Gbps.

Although the configuration chosen by Dynamic maximizes demand satisfaction, it risks violating B→C's approval if its demand rises above 5 Gbps. Existing systems which use Dynamic's approach (such as B4 [98, 105] and SWAN [89]), therefore, project demand to be higher than the

current usage, e.g., by inflating the usage by 10% [105]. However, this is insufficient to prevent approval violations when demands rise sharply between global reconfigurations (e.g., when B→C's demand increases from 5 to 10 Gbps). Multiple iterations of global reconfiguration are necessary in such cases, since each one only increases the admission by 10%.

One could mitigate the risk of approval violations by improving the responsiveness of Dynamic's global controller, but achieving this is an uphill battle.

- First, there is the issue of scale. The speed with which a global controller can act is fundamentally limited by the scale of a global WAN [105] and the need to sequence routing updates to avoid congestion [89], e.g., routing changes may require tens of seconds to minutes to complete [118]. In addition, the input size to the global controller is rapidly growing. Several large content and cloud providers have added 50–100% more nodes to their WAN over the last 2–6 years [104, 76, 152, 98, 90], and the number of flowgroups grows quadratically in relation to this.
- Second, when the global controller is unavailable, the remaining network components continue to use the last known state [68, 98, 105, 89, 101]. Data from production networks suggests that failures can lead to frequent and long delays. An analysis [75] of over 100 failures in Google's networks attributes 9% of failures to unavailability of the WAN control plane. In 2019, an especially long incident [11] brought down the control plane of Google's backbone network for *over four hours* and caused up to 100% packet loss on certain links.

An alternative approach that eliminates the need for a responsive controller entirely is to use a static configuration that only aims to satisfy approvals. In our example, Static (Figure 2.2 center) will satisfy the approvals by setting the admissions for both flowgroups to 10 Gbps and configuring each to use only their direct path. Although Static does not take into account either flowgroup's demand at the global level, the DC controller within each data center must dynamically redistribute the admission for each flowgroup across its tasks. The DC controller can react more quickly than the global controller (§2.4.2.4, see also [105, 89]), and is not a significant source of approval violations. As a result, Static ensures that approvals are satisfied regardless of the demand matrix, but it does not admit any above-approval demand.

In §2.4.1, we quantify the tradeoff between approval and demand satisfaction using the two approaches. The results match the intuition presented here. Static achieves high approval satisfaction and Dynamic provides high demand satisfaction, but each performs poorly in the other metric.

2.2 Approach and Challenges

To balance the satisfaction of both approvals and demands, the question at hand is: how to retain the benefits of centralized WAN control (i.e., better network utilization and support for flexible bandwidth sharing policies) while addressing its adverse impact on satisfying bandwidth approvals?

Our high level insight is that modifying routes and admissions are not the only measures available in our toolkit for reacting to congestion. In addition, we can leverage support within the network data plane to prioritize the delivery of packets marked with a higher QoS value. This feature can be used to satisfy approvals without involving the global controller, except to handle failures.

A natural approach for using this capability would work as follows. In the common case, the global controller will set the admission for every flowgroup to at least be its approval. Any additional traffic admitted onto the network (to utilize spare capacity) will have its QoS reduced to a lower priority (LOPRI). Any flowgroup's ability to increase its within-approval demand will then not depend on the global controller's ability to react. Instead, network switches will strictly prioritize the delivery of its higher-priority (HIPRI) traffic over any competing above-approval, LOPRI traffic (we discuss other prioritization policies in §2.5). We would rate limit LOPRI traffic to avoid excessive loss and to ensure that distribution of spare bandwidth is as per business policy.

While this approach shows promise, we need to address three challenges: one on global control and two on control within each data center.

Sharing routes limits efficiency. Each flowgroup's traffic is divided across the routes installed for it in proportion to their weights. When the global controller wants to add an additional path to support, say one-fourth of the above-approval demand, the new path must also admit a quarter of the approval. How should we allocate routes so that this restriction does not limit the efficiency of the network?

QoS churn interacts poorly with congestion control. Each time HEYP migrates a particular TCP flow from HIPRI to LOPRI (or vice versa), it risks changing the RTT for that connection. Such changes, if they occur frequently, will hamper TCP's ability to accurately estimate the bandwidth-delay product, thereby preventing it from fully utilizing available network bandwidth. Therefore, in determining what fraction of a flowgroup's traffic to downgrade, how can the DC controller minimize QoS churn for individual flows?

Uneven bandwidth distribution can lead to harmful app–DC controller interactions. When HEYP downgrades the QoS for part of a flowgroup, the application may, due to congestion, observe worse throughput on its LOPRI flows compared to its HIPRI ones. The application could react by directing more load to its tasks which provide faster responses. As a result, the flowgroup might send more HIPRI traffic than its approval allows and potentially interfere with the approvals

of other flowgroups. The DC controller will react by downgrading a different subset of the flowgroup's traffic, but of course, the application can again respond by shifting load. To avoid adverse impact on both the flowgroup in question (unnecessary QoS churn) and other flowgroups (approval violations), how do we ensure that the DC controller converges quickly to a stable QoS assignment that admits only the approval at HIPRI?

2.3 Design

In this section, we explain how HEYP addresses each of the above-mentioned concerns. HEYP's design is tailored to the needs of large cloud providers. In aiming to satisfy the goals set out in §2.1, it provides the following key properties.

- Under planned failure scenarios, each flowgroup can ramp up its usage to its approval without any reaction from the global or DC controllers. Within-approval traffic will use paths that meet the specified latency SLO.
- Once a flowgroup exceeds its approval, HEYP will downgrade the flowgroup's excess traffic to LOPRI and rate limit it. The LOPRI routes and admissions are determined using dynamic global control to maximize efficiency.
- To avoid degrading the performance of applications that have part of their traffic downgraded to LOPRI, HEYP maximizes the minimum time each task spends at a particular QoS. Applications that want to make the best use of the available LOPRI bandwidth should internally divert work away from bottleneck tasks. Many existing applications – e.g., HTTP proxies [14, 19], bulk data copies [33], and others [18, 8, 15, 17] – have this capability.
- HEYP's DC controller is biased to over-admit HIPRI traffic when usage is concentrated across a small number of tasks. Network operators can account for this by provisioning additional headroom (§2.3.3). For cloud WANs, we expect that approvals will be large enough for the required headroom to be low.

2.3.1 Separate HIPRI and LOPRI routes for efficiency

To appreciate why the use of QoS downgrade necessitates a change in the global controller's routing strategy, consider the example from Figure 2.2. To accommodate 15 Gbps of A→C's demand, existing 'Dynamic' controllers would compute and install two routes: one along the direct path and one along the indirect path via B, with the former set to carry one-third of the flowgroup's traffic and the latter two-thirds. If we admit 10 Gbps of A→C's traffic on HIPRI, since that is its approval, then 6.6 Gbps of A→C's HIPRI traffic would go over A-C and 3.3 Gbps over A-B-C.

Inputs: Approvals and demands per flowgroup
Topology annotated with link capacities
Outputs (per flowgroup):
Set of HIPRI routes and set of LOPRI routes
HIPRI admission and LOPRI admission

---

1. Compute HIPRI routes and admissions to satisfy *approvals*
2. Compute unused link capacity by deducting any link capacity consumed by *within-approval demands*
3. Compute LOPRI routes and admissions to satisfy *above-approval demands*

Algorithm 2.1: Global computation of routes and admissions. Steps 1 and 3 follow provider's allocation policy.

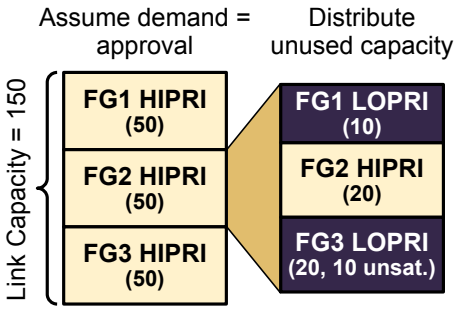
When B→C's within-approval usage increases, we risk violating its approval as it will compete for capacity with A→C's HIPRI traffic.

The problem here is that, if both within- and above-approval traffic are split in the same proportion across paths, we cannot simultaneously satisfy the two properties we want: 1) within-approval demands must be met irrespective of other flowgroups' demands, e.g., A→C should not route within-approval traffic over A-B-C, and 2) above-approval traffic should be able to use any link capacity that is unused by other flowgroups; this constraint is opposite to the previous one: A→C's above-approval traffic must go over A-B-C.

To resolve this issue, in HEYP, we compute *multiple sets* of paths per flowgroup that each meet one of these objectives. We route each flowgroup's within-approval traffic in a manner that statically guarantees no interference with other within-approval traffic. Additionally, we ensure that the routes for above-approval traffic make use of any spare capacity on the network. In our example, we send 10 Gbps of A→C's HIPRI traffic over A-C, and 5 Gbps of A→C's LOPRI traffic over A-B-C. When B→C's demand rises to 10 Gbps, it takes priority over A→C's LOPRI above-approval traffic. Since A→C now sends HIPRI traffic only over the direct link to C, both approvals are satisfied.

Global allocation framework. HEYP determines the sets of paths and admissions for each flowgroup as follows. To support many traffic engineering bandwidth sharing policies, HEYP uses existing algorithms as black box functions to provide capacity to within-approval (HIPRI) or above-approval (LOPRI) traffic. Within a particular QoS level, these functions are free to enforce their own policies.

In existing systems, the global WAN controller [98, 89] computes routes for a particular traffic demand matrix in two phases: 1) fit all within-approval demands, and 2) based on the provider's bandwidth sharing policy, accommodate as much above-approval demands as feasible given the capacity that remains. The routes for every flowgroup comprise the union of the routes computed



(a) Allocation in each phase

	HIPRI	LOPRI
FG1	50 Gbps	10 Gbps
FG2	50 Gbps	0 Gbps
FG3	50 Gbps	20 Gbps

(b) Final admissions

Figure 2.3: Allocation for three flowgroups from data centers A to B over a direct link of capacity 150 Gbps. Each has a 50 Gbps approval, and demands (in Gbps) are [FG1: 60, FG2: 20, FG3: 80]. In Phase 1, all three approvals fit. Phase 2 distributes 30 Gbps of FG2's unused allocation fairly between FG1 and FG3. The rest of FG3's demand is left unsatisfied.

in the two phases and the admission is the sum of capacity allocated on each route.

HEYP's global controller similarly executes in two phases, but both phases differ (Algorithm 2.1) and the outputs of each phase apply separately to either HIPRI (i.e., within-approval) or LOPRI (i.e., above-approval) traffic.

- **Phase 1: Match Static's approval satisfaction.** First, to ensure that flowgroups can burst up to their approvals, we compute HIPRI routes to fit all *approvals*, not just within-approval demands, while ensuring that path lengths are within guaranteed bounds (§2.1). In the unlikely scenario that more capacity is lost due to failures and maintenance than what the provider planned for, capacity is shared according to policy (e.g., max-min fairness).
- **Phase 2: Match Dynamic's demand satisfaction.** Next, we determine additional routes based on observed demand. The key is to compute the capacity consumed by Phase 1 based on *within-approval demands*, not approvals. With this, HEYP admits the same volume of above-approval demands as Dynamic. Moreover, when Phase 1 is unable to fit all approvals into the network, HEYP can admit additional within-approval traffic in this second phase, thereby surpassing Static with respect to approval satisfaction.

In achieving these desirable properties, we are oversubscribing link capacities: HEYP allocates routes based on approvals in Phase 1 but computes the capacity consumed by these routes based on within-approval demands. However, when a link's capacity is oversubscribed, HIPRI traffic will be preferentially delivered, thus ensuring that congestion has no impact on approval satisfaction. HEYP never oversubscribes link capacity in Phase 1 to ensure that an increase in one flowgroup's within-approval demand does not impact the ability to satisfy approvals for other flowgroups. Figure 2.3 illustrates how the HEYP controller separately computes HIPRI and LOPRI admissions.

Mitigating switch limitations. The degree to which HEYP oversubscribes link capacities is

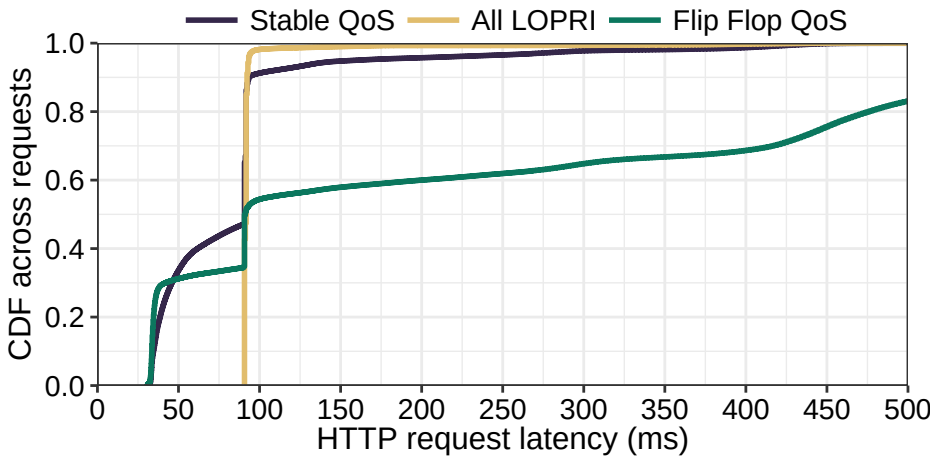


Figure 2.4: Latency with 8 tasks in three scenarios: all use LOPRI, half use LOPRI (tasks never change QoS), and half use LOPRI (each task flips its QoS once every 5 seconds). HIPRI tasks are rate limited to one-eighth of the demand (see §2.3.3); if this is removed "Stable QoS" outperforms "All LOPRI".

configurable: in Phase 2, the available capacity on each link can be set such that the sum of HIPRI and LOPRI admissions do not exceed a configurable multiple of the link's capacity. For network switches that share buffers between per-QoS queues, this can be used to reduce HIPRI packet drops under a flood of LOPRI traffic.

2.3.2 Minimizing QoS churn with caterpillar hashing

Once the global controller has determined the HIPRI and LOPRI admissions for a particular flow-group, the DC controller must assign a QoS level for each of the flowgroup's flows, i.e., each (src IP, src port, dst IP, dst port, protocol) 5-tuple. For this, it first needs to measure the total usage of the flowgroup, and then identify a subset of flows to downgrade such that the sum usage of the remaining flows equals the approval.

Need to minimize QoS churn. A straightforward approach for picking flows to downgrade would be to use a knapsack solver to identify a set of flows whose aggregate usage is closest to the flowgroup's current usage minus the HIPRI admission. However, knapsack solvers make no effort to maintain stable QoS assignments across multiple runs, harming application performance. Every time the QoS assigned to a flow is changed, its bandwidth and latency characteristics change as well. If TCP's congestion control is unable to adapt quickly enough, application performance suffers.

Figure 2.4 demonstrates the impact of frequently changing QoS between backend servers in one data center and an HTTP proxy in another (see §2.4.2 for details). Both the latency (90%ile

of 600 ms vs 95 ms) and throughput (mean 19K req/s vs 21K req/s) seen by the clients suffer, when compared to scenarios where every flow is pinned to a specific QoS level. The reason for this degradation is that BBR [50], the congestion control used in the experiment, is not able to send data at a rate high enough to avoid large queuing delays. BBR actively probes for new round-trip time (RTT) measurements at most once every 10 seconds (more frequent probes would sacrifice throughput) [50]. So, when the DC controller changes QoS every 5 seconds, BBR incorrectly estimates that 99% of LOPRI flows have the RTT of the HIPRI path, and maintains fewer bytes in flight as a result (average congestion window is over 55% smaller), adding queuing delay. If we change the control period to be one minute, the difference between "Stable QoS" and "Flip Flop" disappears.

Challenges in minimizing QoS churn. To maintain QoS stability, one could try to 'pin' each flow to its QoS for some minimum threshold of time. However, doing so would impact the accuracy with which the DC controller can downgrade the desired fraction of a flowgroup's traffic, since only a subset of flows would be eligible for QoS changes.

Alternatively, one could hash every flow's identifier and downgrade the traffic of those flows whose hashed identifier falls below a threshold. The DC controller can assign more (less) of a flowgroup's traffic to LOPRI by increasing (decreasing) this threshold. The problem, however, is the *order* in which flows are downgraded and upgraded. When the threshold rises to downgrade additional flows, and later drops to upgrade flows, the most recently downgraded flows would be upgraded; vice-versa when the threshold subsequently is increased again. This behavior maximizes the worst-case QoS churn for individual flows.

Rethinking hashing-based QoS downgrade. In HEYP, we introduce caterpillar hashing as a flow selection mechanism that minimizes QoS churn. Caterpillar hashing chooses which flows to downgrade using a range of the hash space, rather than a threshold. As illustrated by Figure 2.5, when we need to increase (decrease) the fraction of flows that are downgraded, we grow (shrink) the range by moving the upper (lower) threshold. This behavior upgrades the flows that were downgraded earliest, and therefore, maximizes the minimum time each flow spends at a particular QoS.

Hashing-based approaches randomly select flows for downgrade, and therefore have lower accuracy compared to using a knapsack solver. However, in the following section, we explain how HEYP's DC controller leverages feedback control, and this largely mitigates any concerns about accuracy.

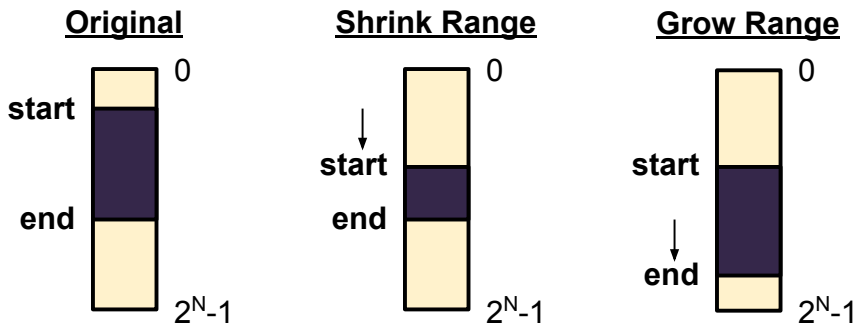


Figure 2.5: Caterpillar hashing shrinking and growing the subset of flows which are downgraded.

2.3.3 Mitigating harmful app–controller interactions

Since existing DC controllers configure tasks (§2.1.1), they could be extended to measure what fraction of usage is above approval, apply caterpillar hashing to select tasks for downgrade, and then compute rate limits for LOPRI tasks. The controller could use caterpillar hashing to ensure that the fraction of a flowgroup's traffic which is downgraded equals 1 - (total usage)/(HIPRI admission), where HIPRI admission is equal to approval, except under extreme failure scenarios. However, this approach can lead to harmful interactions between the DC controller and applications.

Consider the HTTP workload used to generate Figure 2.4. When LOPRI flows experience congestion, the HTTP proxy would observe longer queues for LOPRI tasks compared to HIPRI tasks, and shift more of its load to the HIPRI tasks. This would cause the flowgroup to have HIPRI usage greater than its approval, since the set of HIPRI tasks is now transmitting bandwidth that used to be spread across a larger set of tasks. However, the DC controller would not react because the fraction of usage above approval is unchanged; after all, the load has simply shifted between tasks. Had the DC controller instead used a knapsack solver, it would have seen that the flowgroup's HIPRI usage is higher than intended and selected a different subset of tasks to downgrade. But, the application will again react by shifting its usage around. To prevent this cat-and-mouse game, which will result in high QoS churn and put approval satisfaction for other flowgroups at risk, let us first consider two strawman approaches.

Strawman 1: Downgrade jobs as a unit. Most cluster management systems have some notion of a job that is used to deploy applications [167, 16, 87]. To downgrade a portion of any flowgroup, if we were to downgrade at the granularity of jobs, the application would be unable to respond in the above manner. However, some applications are composed of multiple jobs, and since the DC controller has no knowledge of which jobs are critical for the application, downgrading an entire job may degrade the user experience.

Strawman 2: Rate limit HIPRI traffic. Alternatively, one could use rate limiting to prevent an

application from sending more HIPRI traffic than its approval, as we did in Figure 2.4. However, despite its use in production WANs, distributed rate limiting suffers from inaccuracy and risks throttling tasks unnecessarily [147] (see also §2.4.2.3). For LOPRI traffic, we believe the costs of rate limiting are worth the benefit: it enables policy-based sharing and avoids high loss on fully-loaded links. In contrast, for HIPRI traffic, we seek to prevent problematic interactions between applications and the DC controller without rate limiting.

Search for application bottleneck. To avoid the downsides of these strawman approaches, we use the following observation: applications can respond to QoS downgrade by shifting around load only because their HIPRI tasks are able to handle additional load. Eventually, each task becomes limited by some resource other than WAN link capacity, e.g., the machine's network card. Therefore, if we ensure that all HIPRI tasks are saturated, the application will not shift additional load to HIPRI tasks.

To search for this operating point – where the HIPRI tasks are saturated enough that the application does not shift additional load over from LOPRI tasks – HEYP employs feedback control. Although the DC controller does not know exactly when tasks become saturated, it can iteratively increase the fraction of tasks that are downgraded. We assume that no individual task can saturate an approval, and hence, HIPRI tasks will eventually become saturated.

In each control period, HEYP's DC controller revises the fraction of downgraded tasks in proportion to the relative error in enforcing a flowgroup's HIPRI admission. Using caterpillar hashing, the controller increases (or decreases) the fraction that is downgraded in proportion to (HIPRI usage - HIPRI admission) / flowgroup's overall usage. This simple form of control [168] mitigates the harmful interaction. As an added benefit, it improves the accuracy of the DC controller's selection of tasks to downgrade. If the downgraded tasks combined have too much or too little usage, the feedback controller will observe this error and try to eliminate it.

There remain two concerns that need to be addressed.

- First, when usage is below the HIPRI admission, we do not know what fraction of the usage should be upgraded to HIPRI. It could be that the flowgroup's demand is below the HIPRI admission; in this case, the correct response would be to upgrade all tasks. On the other hand, it could be that the controller has downgraded too much traffic and should simply upgrade a small portion of it. HEYP tries to balance its behavior for these different cases by always upgrading 20% of traffic. This provides a slower, but hopefully acceptable response to the first case (five control periods are needed to upgrade the entire flowgroup) and reduced QoS churn in the second case.
- Second, HEYP ignores excess HIPRI usage in two cases. The first case is when the HIPRI usage is within measurement noise of the HIPRI admission. This threshold can be determined using an online estimator or offline analysis. For simplicity, our prototype uses a static value.

Config: $upgradeInc = 0.2$ (fixed frac. to upgrade)
$propGain = 0.5$ (proportional gain)
$errNoise = 0.05$ (noise in usage measurement)
$kCoarse = 2$ (task err multipler)
Output: fraction of usage to downgrade (upgrade if < 0)

---

if *total usage* $<$ *HIPRI admission* **then**
  return $upgradeInc$
end
$err \leftarrow$ (HIPRI usage - HIPRI admission) $\div$ total usage
$coarseness \leftarrow kCoarse \times$ max task usage $\div$ total usage
if $0 < err < \max(errNoise, coarseness)$ **then**
  return 0
end
return $propGain \times err$

Algorithm 2.2: Feedback control determines what fraction of usage to downgrade. The configuration parameters were tuned against a range of simulated workloads (§2.4.3).

The second case is when the HIPRI usage exceeds the HIPRI admission by a small multiple of the maximum task usage. The intuition is that task usages may be too coarse to achieve the desired split, and the maximum task usage serves as an overestimate of the coarseness of all task usages. To prevent the resulting excess HIPRI usage from causing approval violations, the network provider should provision enough headroom to accommodate both cases. As noted at the start of §2.3, we expect the required headroom for cloud WANs to be low.

Algorithm 2.2 presents HEYP's final control logic for revising the fraction of tasks to downgrade. In §2.4.3, we empirically show that HEYP's DC controller provides low QoS churn and quickly converges to an accurate split under a variety of workloads.

2.4 Evaluation

We evaluate HEYP's performance in three parts. First, using production traces from Google's WAN and a discrete-event simulator, we evaluate the benefits of HEYP's global controller for satisfying both approvals and demands across data centers. Then, we deploy a prototype of HEYP's DC controller on CloudLab [66] and evaluate its ability to enforce HIPRI admissions (using QoS downgrade) and its utility on an application workload. Finally, we use monte carlo simulation to evaluate HEYP's DC controller across a larger set of workloads than we can evaluate in a testbed setting. The primary takeaways from our evaluation are as follows:

- HEYP offers the best combination of approval and demand satisfaction: 99% availability of approved bandwidth for 87–99% of flowgroups (better than even static approval-based allocation)

	% of flowgroups with ≥ 99.9% approval satisfaction			% of flowgroups with ≥ 99% approval satisfaction			Mean demand satisfaction (%)		
	Week 1	Week 2	Week 3	Week 1	Week 2	Week 3	Week 1	Week 2	Week 3
Static	37	41	46	81	97	94	55	44	58
RA+Dynamic	34	34	46	80	97	94	70	74	73
Dynamic+JA	3	2	3	9	13	9	81	79	84
Dynamic	3	2	4	7	9	7	86	88	89
HEYP	37	43	51	87	99	97	86	83	90
Legend	0–20 20–40 40–60 60–80 80–100			0–20 20–40 40–60 60–80 80–100			0–60 60–70 70–80 80–90 90–100		

Table 2.1: Simulation results for Google network traces across three weeks.

while offering similar demand satisfaction as dynamic allocation, which is able to satisfy only 7–9% of approvals 99% of the time.

- In a sensitivity analysis, we find that dynamic allocation falls short of the approval satisfaction offered by HEYP even if control plane delays are cut by 5× and demands change slowly. In addition, HEYP delivers high approval satisfaction even if demands change twice as fast as in Google's WAN while the approval satisfaction of dynamic allocation is further reduced.
- When applied to an HTTP workload, HEYP offers the best combination of isolation and performance. When competing against a flowgroup with excess traffic, the latency and throughput of a within-approval flowgroup are unchanged compared to static, approval-based rate limiting. In addition, the additional bandwidth HEYP provides to the above-approval flowgroup improves throughput to within 12% of the theoretical max.

2.4.1 Predictability and efficiency across DCs

To evaluate HEYP's impact on sharing bandwidth between flowgroups spread across many data centers, we use a custom, discrete-event simulator. Simulation enables us to evaluate designs that are impossible or difficult to realize (e.g., we consider a hypothetical control plane that acts 5× faster than the state-of-the-art). We use a custom simulator because existing software has high overhead for evaluating WAN control planes [85, 9, 10, 62], licensing concerns [12], or focuses only on traffic engineering [106].

2.4.1.1 Inter-DC network simulator

As in prior work [106], our simulator models the topology at a data center level and applies max-min fair sharing of link bandwidth across flows. In addition, our simulator captures several features that govern the behavior of software-defined WANs.

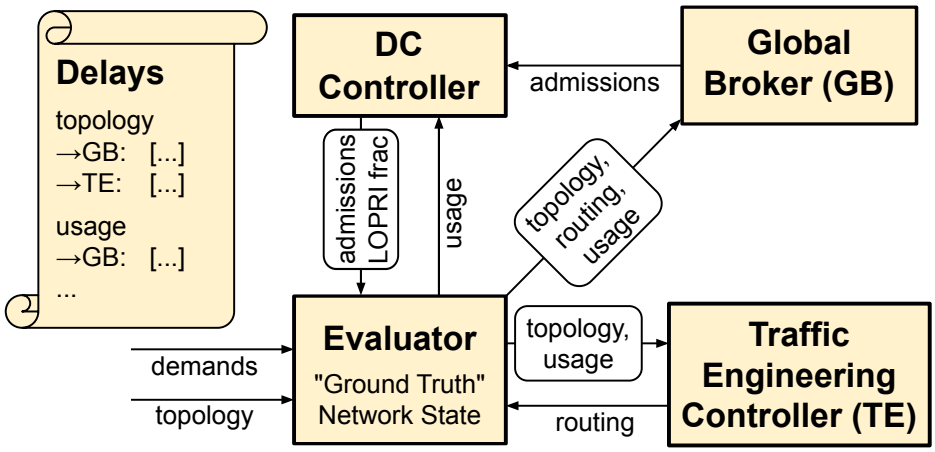


Figure 2.6: Architecture of our inter-DC simulator.

Network controllers. As in B4 [98, 105], our simulator employs separate global controllers to select routes and admissions: the Traffic Engineering (TE) Controller and the Global Broker, respectively. Since we model traffic at the data center level, the simulated DC Controllers cannot configure individual tasks. Instead, we try to capture the impact that partitioning traffic into HIPRI and LOPRI has on approval and demand satisfaction. For example, if 30% of demand is marked LOPRI and usage drops to the approval, 30% of demand will remain LOPRI until the DC Controller revises the split. Appendix A.1 contains the logic for each controller.

Modeling delays and inconsistency of state. We model the WAN as a set of processes that share no state. Processes send messages to a each other, scheduling their arrival at a future time. This model captures both the delays in controller response and any inconsistency of state across controllers.

Capturing demand uncertainty. An Evaluator process (see Figure 2.6) tracks the network state and computes metrics (e.g., demand satisfaction). The Evaluator broadcasts changes in any flowgroup's *usage* to all controllers. As a result, controllers may not observe rapid increases in a flowgroup's *demand* until several control periods have passed.

Validation. To confirm that the data output by our simulator is meaningful, we compare the mean, hourly demand satisfaction reported by Google's production system against our simulated adaptation of it (see Dynamic in §2.4.1). Figure 2.7 shows that there is a statistically significant, positive correlation between the demand satisfaction observed in our simulation and in production. While the production system contains additional heuristics to improve performance, our simulation is a reasonably good predictor for the demand satisfaction seen in production: time frames in which the production system has higher (lower) demand satisfaction are also times in which the simulator performs well (poorly).

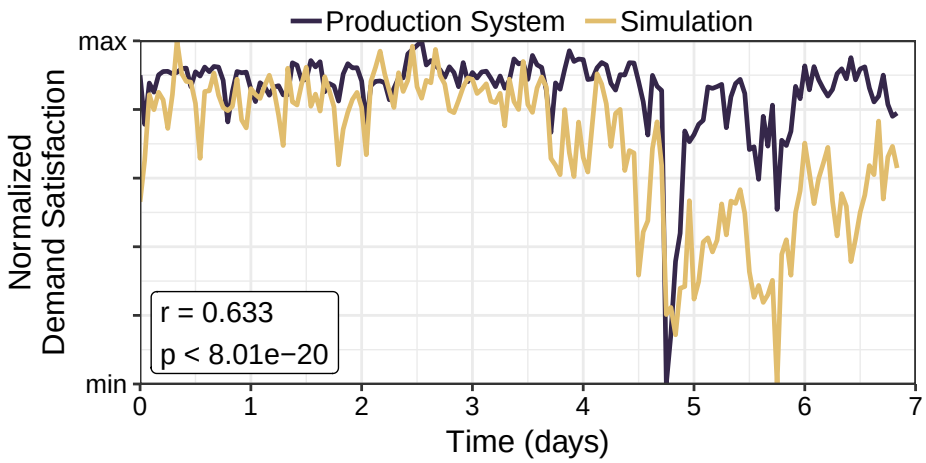


Figure 2.7: Demand satisfaction (normalized to have the same minimum and maximum values) in production versus simulation. The correlation coefficient (r) and p-value are noted.

2.4.1.2 Trace data, allocation algorithms, and metrics

We use traces obtained from Google's WAN containing data for three separate weeks in 2019. Each trace contains snapshots of the topology and demand between data centers – as estimated by the production system – measured once a minute. Bandwidth approvals were derived from production data collected from the Google WAN and adjusted to account for differences between the simulation and production environments. Google ensures that (given fast controller response) approvals can be met under appropriate failure scenarios. For each type of control plane delay (e.g., time taken to install a new set of routes), our simulations mimic the distribution seen in production.

We implement and compare HEYP against the following approaches in our simulator.

- **Static** allocation policy is oblivious to demands. When the Global Broker observes a new topology or routing (resp., when the Traffic Engineering (TE) Controller observes a new topology), it computes new admissions (resp., new routes) given the approvals as demand.
- **Dynamic** policy approximates the behavior of B4 [98, 105] and SWAN [89]. Unlike Static, it reacts not only to topology changes but also when demands change, in order to allocate above-approval traffic after satisfying within-approval demands. For any flowgroup, all traffic traverses one set of paths and uses the same QoS level.
- We also consider two variants of Dynamic which strike intermediate tradeoffs between approval and demand satisfaction. **RA+Dynamic** (for Reserve Approval + Dynamic) assumes that demand = max(approval, demand). Hence, it will allocate at least as much capacity as Static but will attempt to accommodate above-approval traffic when demands change. **Dynamic+JA** (for Dynamic + Jump to Approval) assumes that the demand for any flowgroup being throttled is equal to its approval, thereby preempting the need for multiple iterations of global reconfigura-

tion for within-approval demand to ramp up. The throttling signal is propagated together with the usage information.

Allocation algorithms. In all approaches, the Global Broker and TE Controller first allocate bandwidth to satisfy within-approval demands, then use residual capacity to satisfy above-approval demands. In either phase, they enforce max-min fair sharing across flowgroups. To compute routes, the TE Controller selects the shortest available path for each flowgroup and computes a max-min fair allocation of bandwidth across these paths. This process loops until either all demands are satisfied or all links are saturated. Traffic for a flowgroup is split across the routes allocated for it in the ratio of the admission computed for each route. When a link fails, flowgroups may experience traffic loss until the controller installs new routes. For more details, see Appendix A.2.

Metrics. We examine the approval and demand satisfaction of each approach. We consider a flowgroup's approval to be satisfied whenever its usage is $\geq 0.95 \times$ min(approval, demand). We compute demand satisfaction as the sum of per-flowgroup usages divided by the sum of their demands. We use this metric – as opposed to link utilization – to measure efficiency because a higher value directly corresponds to a better use of network resources.

2.4.1.3 Results

Table 2.1 presents the results for each of the three week-long traces; we consider two commonly studied [90, 41, 184] (99% and 99.9%) availability targets.

Approval satisfaction for HEYP and Static are similar, as expected, since Static's allocation is the same as that used in HEYP's HIPRI allocation. However, Dynamic satisfies up to twice the demand of Static, a result of it's allocation being demand aware. In most cases, HEYP achieves similar demand satisfaction to Dynamic, but HEYP consistently offers significantly higher availability of approved bandwidth. One reason for Dynamic's poor availability is the duration of approval violations: 20% of violations are resolved only after multiple iterations of global control.

Dynamic+JA and RA+Dynamic hit intermediate tradeoffs in between Static and Dynamic. The reason for this is that Dynamic+JA and RA+Dynamic reserve bandwidth based on approvals, even when demands are lower than approvals. Since RA+Dynamic does so always, it offers lower demand satisfaction like Static; whereas, since Dynamic+JA allocates for approval only once a flowgroup is throttled, it offers low approval satisfaction like Dynamic.

Impact of tail latency. In our traces, the time from when the DC Controller detects a change in demand until new admissions (routes) are installed is $3\times$ ($1.5\times$) larger at the 99th percentile than at the median. To investigate whether high tail latency is negatively impacting tail approval satisfaction, we simulate Week 2 with all control delays limited to the 45–55th percentile range of the distribution observed in production. We see little increase in Dynamic's approval satisfaction;

Legend: 0–20 | 20–40 | 40–60 | 60–80 | 80–100

Control Plane Speed	Rate of Demand Change (larger is faster)								
	Dynamic			Static			HEYP		
	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×
5× Faster	59	25	15	88	87	87	96	94	93
Normal	15	9	8	87	87	86	93	91	90
5× Slower	4	4	4	60	60	59	61	59	59

(a) Percent of flowgroups with ≥ 99% approval satisfaction

Legend: 0–20 | 20–40 | 40–60 | 60–80 | 80–100

Control Plane Speed	Rate of Demand Change (larger is faster)								
	Dynamic			Static			HEYP		
	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×
5× Faster	12	7	4	76	72	67	91	87	85
Normal	6	5	3	46	45	43	48	46	45
5× Slower	3	2	2	36	34	35	38	36	36

(b) Percent of flowgroups with ≥ 99.9% approval satisfaction

Legend: 0–60 | 60–70 | 70–80 | 80–90 | 90–100

Control Plane Speed	Rate of Demand Change (larger is faster)								
	Dynamic			Static			HEYP		
	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×	0.5×	1.0×	2.0×
5× Faster	95	94	93	55	55	55	88	88	87
Normal	89	86	84	55	55	55	87	86	85
5× Slower	78	75	75	55	55	55	84	83	82

(c) Mean demand satisfaction (%)

Table 2.2: Performance when varying both the speed at which controllers react and the rate at which demands change.

only 13% (4%) of flowgroups have 99% (99.9%) approval satisfaction. We conclude that even the median global control delays in such a heavily engineered WAN are too high to accommodate the churn in demand.

Sensitivity to changes in workload and setting. To evaluate each approach in a broader range of settings, we vary the inputs from Google along two dimensions: the rate at which demands change and the control plane's speed (both in the delays incurred and the frequency with which controllers run). Table 2.2 compares Dynamic, Static, and HEYP on a 48-hour trace during Week 1. We make several observations regarding approval satisfaction:

- Dynamic is highly dependent on timely responses to demand changes. Between the easiest scenario (fast control plane and slow-changing demands) and the hardest (slow control plane and fast-changing demands), fraction of approvals satisfied with Dynamic drop by over 10×.
- Regardless of the control plane's speed, approval satisfaction with HEYP and Static is indepen-

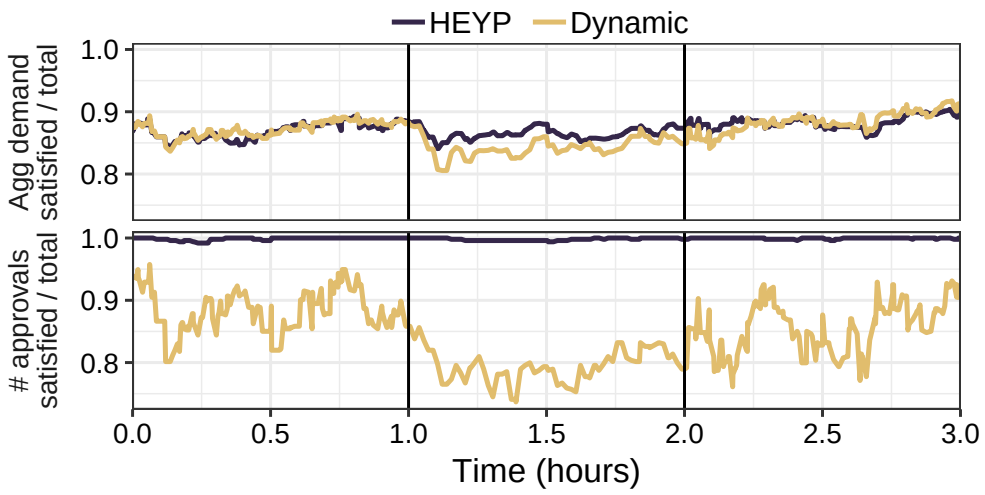


Figure 2.8: Performance under a global controller outage (starts after 1 hour and lasts for 60 min).

dent of the rate of demand change. Whereas, Dynamic significantly suffers when demands ramp up faster; even with a fast control plane, the fraction of approvals receiving 99% availability with Dynamic drops by 4× when going from a slow to a fast rate of demand change.

With respect to demand satisfaction, Static is poor across the board since it does not react to changes in demand or admit above-approval traffic. In contrast, a slower control plane significantly decreases demand satisfaction with Dynamic but has no impact on HEYP; by allocating HIPRI routes and admissions based on approvals, not within-approval demands, HEYP allows within-approval usage to ramp up without any action by the global controller. With a faster control plane, both Dynamic and HEYP more quickly adapt to accommodate changing above-approval demands.

Performance under a global controller outage. An extreme case of a slow control plane is when the global controller is down. To evaluate performance under such a scenario, we select a 3-hour window from Week 1 and simulate a failure of both the Global Broker and the TE Controller. No data plane failures take place during the outage.

Figure 2.8 shows that HEYP consistently satisfies nearly all approvals during the outage, whereas approval satisfaction with Dynamic drops shortly after the control plane outage begins. While both approaches have degraded demand satisfaction during the outage, HEYP satisfies more demand than Dynamic because it pre-allocates capacity to satisfy any increase in within-approval demands. At other times, HEYP and Dynamic provide similar demand satisfaction.

2.4.2 Testbed evaluation

Next, we deploy a prototype of HEYP's DC controller and study its ability to accurately enforce HIPRI admissions (using QoS downgrade) with low QoS churn. In addition, we examine the

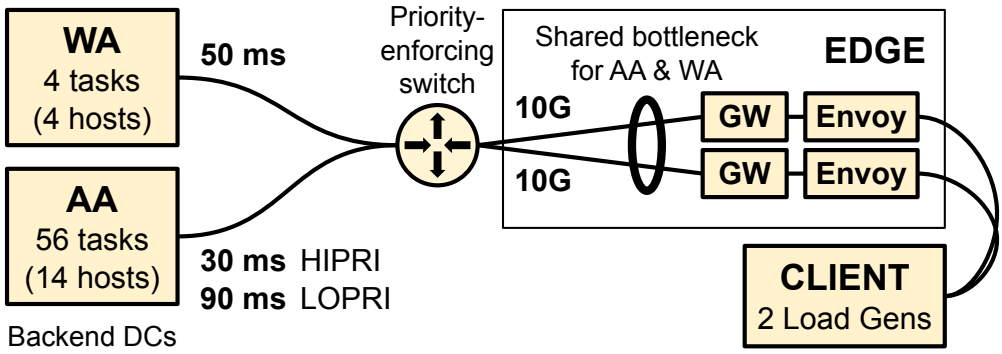


Figure 2.9: Testbed setup for HTTP workload. We run a separate DC controller for each data center (not shown).

impact of QoS downgrade on an application workload, both from the perspective of isolating any within-approval flowgroup and maximizing an above-approval flowgroup's ability to use spare capacity.

2.4.2.1 Application workload and setup

Since web services are highly sensitive to latency inflation and bandwidth shortages, we evaluate HEYP against HTTP workloads and emulate the architecture of production web services. As shown in Figure 2.9, clients (which use Fortio [20], a load generator) issue requests in an open loop to EDGE, where one of two Envoy [14] proxies examines which backend the request is for and directs it to an appropriate backend server. Upon receiving the proxied request, the backend generates a response that is then forwarded by the proxy back to the client. Each backend task is registered with the local DC controller, and enforces QoS downgrade and rate limiting policies via standard Linux facilities.

We deploy backends onto CloudLab's xl170 machines (10 cores) connected via 10 Gbps links to a Dell S4048-ON switch. The switch is configured to enforce strict priority queuing between HIPRI and LOPRI traffic. The Envoy proxies and Fortio clients run on dedicated c6525-25g machines (16 cores) and are connected to each other via a 25 Gbps network. Each Envoy proxy reaches the backend servers via its own gateway server (xl170) that is connected to both networks. Following existing systems [105], we set the DC controller to compute new QoS assignments and rate limits once every second (we show that this rate is feasible with millions of tasks in §2.4.2.4).

We run two backend services, logically separated into two "data centers": AA (for above approval) and WA (for within approval). We simulate latency between them and EDGE using netem [84]. The approvals for AA→EDGE and WA→EDGE are 2 and 12 Gbps, respectively. WA's approval was chosen to exceed half of the bottleneck link's capacity (20 Gbps, 10 Gbps for

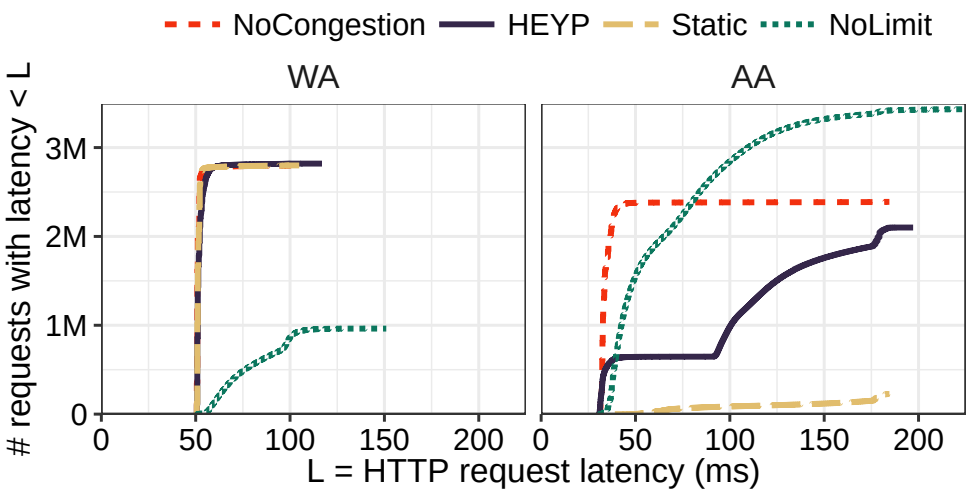


Figure 2.10: Latency vs number of requests served with that latency for each flowgroup.

each EDGE proxy) so that a max-min fair distribution of the capacity would violate the approval.

We compare HEYP to the following approaches:

- **NoCongestion.** This approach estimates the best throughput and latency that AA can achieve irrespective of whether QoS downgrade or rate limiting is employed, i.e., when bandwidth is the only constraint. We obtain this estimate by reducing WA's sending rate so that the sum of AA and WA demands are satisfied without overloading any links; empirically, we have determined that the bottleneck link can sustain up to 90% utilization.
- **Static.** By rate limiting each flowgroup to its approval, this approach prioritizes providing isolation for WA at the cost of AA's demand satisfaction. This serves as a baseline for comparing QoS downgrade and rate limiting as admission control mechanisms. Our implementation follows BwE's Job Enforcer [105]; in particular, we have implemented both dynamic oversubscription (based on workload burstiness) and static oversubscription (scale up capacity by 1.25×).
- **KnapDown.** To study the utility of feedback control and caterpillar hashing, we downgrade QoS using a knapsack solver, the initial approach described in §2.3.2.
- **NoLimit.** To demonstrate that some form of control is needed to satisfy approvals, we consider the effects of using neither QoS downgrade nor rate limiting.

When studying AA's performance, we focus on scenarios where the tenant has configured the application to degrade gracefully under overload, and therefore enable load shedding. To ensure that any requests that are served maintain reasonable latency [40, 13], Envoy routes requests to the least-loaded backend server and eagerly rejects requests for WA when the corresponding backend servers become overloaded. When studying an approach's ability to isolate WA's traffic from a

noisy neighbor, we disable load shedding for AA's traffic to ensure that we are not measuring the effects of AA's load shedding, but the network isolation mechanism.

2.4.2.2 Performance under gradual workload change

We start by examining the performance of a workload that changes gradually, e.g., user traffic increasing over a day. We set AA's demand to 12 Gbps and ramp up WA's demand at a constant rate from 6 to 12 Gbps over 2 minutes. Figure 2.10 presents the latency and throughput for both backends.

Performance isolation for WA. Static and HEYP both provide strong isolation for WA; latency matches both NoCongestion and the case where AA has no above-approval traffic. With NoLimit, bandwidth is shared based on the behavior of congestion control, not on approvals. As a result, WA's performance degrades when AA, which contains 14× as many tasks as WA, captures more bandwidth than it.

Benefit of above-approval bandwidth for AA. Of the approaches that satisfy WA's approval, we see that HEYP offers the best combination of latency and throughput (throughput is 88% of NoCongestion and 1.8M requests complete within 150 ms) compared to Static (throughput is 9% of NoCongestion and only 120K requests complete in 150 ms). The 12% gap in throughput between HEYP and NoCongestion is due to load shedding; once disabled, AA's throughput with HEYP matches NoCongestion, albeit at an even higher latency (above 400 ms).

Note that the low latency that NoLimit and NoCongestion offer to AA is an artifact of our experimental setup. We inject 60 ms of additional propagation delay for LOPRI traffic to emulate the case where it traverses a longer path than HIPRI traffic. In practice, this should only occur when the global controller observes high utilization on a bottleneck link for HIPRI traffic. In this case, NoLimit and NoCongestion would also need to use the longer path for a portion of their traffic, but our testbed is unable to capture this.

Utility of feedback control on limiting harmful app–controller interactions. For the same workload as Figure 2.10, Figure 2.11 shows that the gap between the approval and HIPRI usage for AA is consistently higher when using KnapDown than HEYP. In each instance where KnapDown is able to eliminate all excess HIPRI, we see that AA quickly returns to using more HIPRI than its approval. When the DC controller runs again, KnapDown makes no attempt to maintain stable QoS assignments unlike HEYP, which leverages caterpillar hashing and feedback control. As a result, KnapDown performs 27× the number of QoS changes as HEYP. Of the three approaches shown, HEYP provides the lowest mean absolute error: for the top case, it is within 9% of the approval vs 14% using Static and 45% using KnapDown.

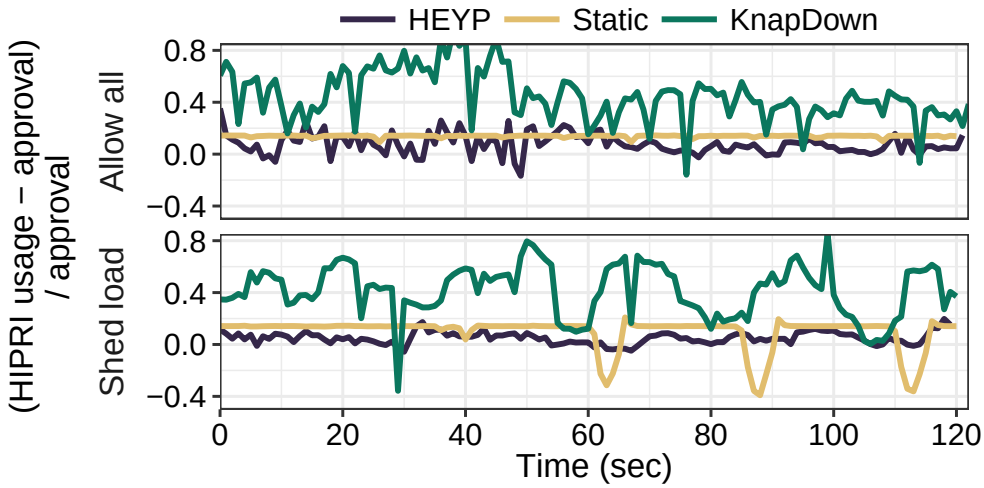


Figure 2.11: Accuracy of HIPRI admission control. Unlike HEYP and Static, KnapDown admits excess HIPRI traffic when applications redistribute load.

2.4.2.3 Performance under sudden workload change

Next, we stress test HEYP's ability to keep HIPRI usage near the approval under sudden workload changes. We keep WA's demand static at 12 Gbps and configure AA's demand to rise sharply after 20 seconds from 3 to 9 Gbps, then drop after a minute to 3 Gbps, and rise one last time after another 60 seconds to 9 Gbps.

Figure 2.12 presents the HIPRI error for AA when Envoy sheds load, and when it admits all requests. Focusing on Figure 2.12(top), we see that Static consistently admits 14% more HIPRI usage than approval allows. The excess HIPRI usage is due to Static's oversubscription of bandwidth. With better tuning, Static's accuracy may improve, but if we enable load shedding for AA (see Figure 2.12(bottom)), we see that Static frequently throttles AA→EDGE below its approval. This illustrates the difficulty in tuning approaches that leverage rate limiting: if we configure Static to oversubscribe the network less, than it may perform better in the former case, but it would throttle even more aggressively in the latter case.

In contrast, HEYP's controller adapts without tuning to the two workloads. The inaccuracy of its rate limiting only impacts how much LOPRI capacity HEYP delivers, not its approval satisfaction. HEYP will satisfy the approval even when it downgrades too much traffic, except when LOPRI is sufficiently congested or throttled. In Figure 2.12, HEYP satisfies AA→EDGE's approval 96% of the time in the bottom case, and Static satisfies the approval only 80% of the time (both have 100% satisfaction under no load shedding).

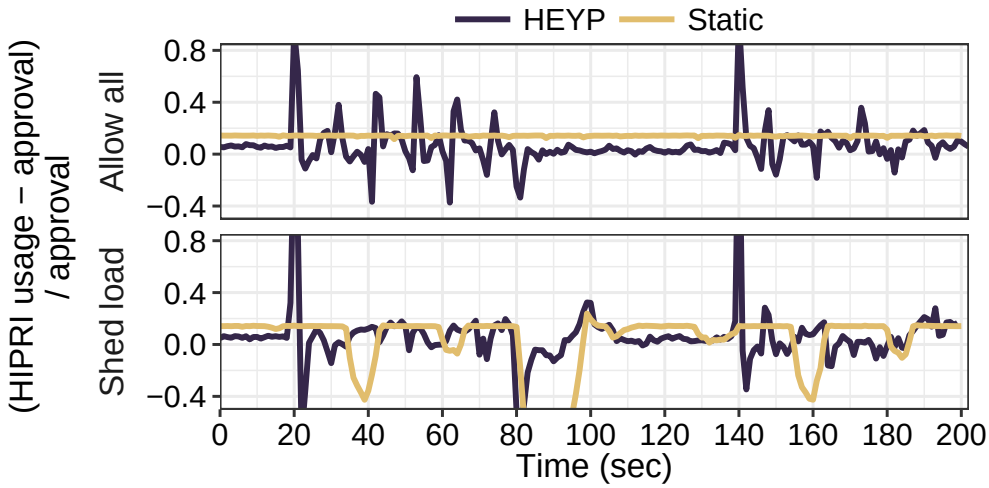


Figure 2.12: The accuracy of HEYP and Static when trying to keep AA→EDGE's HIPRI usage near its approval under sudden demand changes (at 20s, 80s, and 140s).

2.4.2.4 Scalability of the DC controller

The faster HEYP's DC controller can react to changes in demand, the more accurately it can enforce admissions and avoid throttling. However, in today's clouds, individual tenants may run millions of tasks [175]. For this reason, it is important to optimize the reaction time of the DC controller.

The main scalability bottlenecks of the DC controller are the collection of task-level usage and the broadcasting of task-level QoS. The partitioning of traffic into HIPRI and LOPRI, done via caterpillar hashing and feedback control, is constant time, and therefore is not a significant bottleneck. To optimize the collection of usage data, our prototype estimates each flowgroup's usage using threshold sampling [64] – a sampling mechanism that provides accurate estimates even under skewed workloads. To optimize the time needed to broadcast QoS assignments, our prototype DC controller only broadcasts QoS assignments that have changed. This result of this optimization is that the broadcasting work required over a time period is dependent only on the rate of flowgroup-level demand changes and the number of tasks, not the length of a control period. For example, if a flowgroup's usage gradually doubles from the approval over the course of 5 seconds, then the controller will need to downgrade approximately half of the tasks within that time frame. A controller with a short control period would simply downgrade a smaller number of tasks each period compared to one using a longer control period.

Evaluation Setup. We focus on measuring how quickly our prototype DC controller can react to changes in demand for a single flowgroup, so we feed it usage data for 1 million simulated tasks (the usage data is transmitted via RPCs to a real DC controller, but no such tasks exist). The

Figure 2.13: Delay between tasks sending usage to the DC controller and receiving a QoS assignment.

flowgroup's demand cycles between 50% of its approval to 150% of its approval and back every 10 seconds. Each task carries one one-millionth of the flowgroup's overall demand, and we configure the system to sample usage from approximately 1.6% of tasks. The DC controller runs once every 400 ms.

Results. Figure 2.13 shows that HEYP's DC controller is able to respond to changes within 500 ms. If the DC controller perfectly eliminated over usage (or under usage) of HIPRI every iteration, this would imply that the DC controller would bound HIPRI usage to within 10% of the approval (for workloads which grow or shrink their demand by 10% of their approval every 500 ms). However, HEYP's feedback control requires more than one iteration to obtain perfect accuracy, as each iteration attempts to eliminate half of the error, and so the error may persist for several seconds.

The gradually increasing slope is a result of smearing the arrival times of usage data. In our prototype, each task sleeps for a random period before transmitting their initial usage data to the DC controller. This prevents sudden spikes in load by spreading the load out over time.

2.4.3 Large-scale simulation of HEYP's DC controller

In our testbed (§2.4.2) we were only able to study the behavior of HEYP's DC controller in a limited set of workloads. In this section, we examine its behavior under a wider range of workloads using monte carlo simulation.

Setup. In each run, we generate a static set of per-task demands according to a desired distribution and repeatedly invoke the DC controller against it to either downgrade traffic (if all tasks are HIPRI) or upgrade traffic (if all tasks are LOPRI). We focus on cases when downgrade is performed and

Demand Dist.	EM-5%		EXP		FB15		UNI	
Init. % Downgraded	0	100	0	100	0	100	0	100
Convergence Time (#periods)	4.72	8.8	12.38	14.89	7.83	10.07	17.13	14.27
No. of QoS changes undone	3.89	5.2	6.56	7.05	3.98	4.77	9.58	5.39
No. of Oscillations	0.25	0.23	1.88	1.61	0.81	0.66	2.59	1.63
Final Overage (%)	15	11	4	4	6	4	3	3
Final Shortage (%)	0	0	0	0	0	0	0	0
Intermediate Overage (%)	23	2	10	2	14	2	7	2
Intermediate Shortage (%)	1	13	0	7	0	10	0	7

Table 2.3: Performance of HEYP's DC controller when downgrading part of a flowgroup across a range of simulated settings. To compute the value of each cell, we take the mean value across 100 monte carlo runs. Convergence time is measured in control periods. Overage and shortage are measured both once the controller has converged ("Final Overage") and during the period before convergence ("Intermediate Overage"). Compare with the number of QoS changes that are unintended against the number of QoS changes expected for each case (100, since we expect to downgrade or upgrade roughly half of the tasks in each case).

LOPRI is congested, as these are the most difficult to handle for the controller. To do so, we set the approval to one-half of the expected demand and set the available LOPRI bandwidth to 25% of the aggregate demand.

Demand distributions. We simulate 200 tasks and distribute demand across tasks according to one of the following distributions (all have a mean usage per task of 2 Gbps):

- UNI: The demand of each task is chosen between 0 and 4 Gbps uniformly at random.
- EM-5%: The top 5% of tasks have demand chosen between 30 and 34 Gbps uniformly at random. The remaining tasks have demand between 0 and 842 Mbps, also chosen uniformly at random.
- EXP: The demand of each task is chosen from an exponential distribution and capped to 40 Gbps.
- FB15: We generate demands for the four types of WAN-using applications at Facebook [151], and scale them so that the distribution mean is the desired 2 Gbps. We assume the fraction of tasks belonging to each application is proportion to its demand, and evenly spread each application's demand across its tasks with a random value of 5% noise added.

Convergence time and QoS churn. First, we examine the time required for the DC controller to converge on a stable set of LOPRI tasks (and therefore stop changing the configuration of the tenant's tasks). In all of the tested cases, HEYP's DC controller eventually converges, and 95% of the time, the controller converges in fewer than 18 control periods.

Before converging, the DC controller oscillates between downgrading and upgrading tasks an average of 0.23–2.59 times. During each oscillation, the controller is undoing some of the downgrade (or upgrade) decisions it previously made. On average, however, the oscillations result in little churn; the workloads being tested consist of 200 tasks, but less than 10 QoS assignments are undone by HEYP's DC controller.

Overage and Shortage. For EXP, FB15, and UNI, overage (excess HIPRI usage) was approximately 5% of the approval and no shortage (volume of demand we failed to admit at HIPRI) remained once the controller converged. However, intermediate states exhibited higher overage (mean up to 14%) and shortage (mean up to 19%. The higher amounts of overage compared to shortage are a consequence of HEYP's bias to prefer it (§2.3.3). EM-5% exhibits more overage than workload types – after converging the mean is up to 15% – due to the coarseness of demands. Each "elephant" task carries approximately 8% of the demand, and so the DC controller stops reacting once overage is twice this value.

2.5 Discussion

Weighted fair queuing. HEYP's global allocation can be adapted for networks that share bandwidth across QoS levels using weighted fair queuing, rather than strict prioritization. The key is to account for the reservation of bandwidth to LOPRI traffic. For example, if the ratio of weights for HIPRI:LOPRI QoS is 8:2, then LOPRI traffic can use 20% of the link's bandwidth regardless of the HIPRI usage. In this case, we would scale down the link capacities in Phase 1 (§2.3.1) to 80% of the original values, so that HIPRI traffic always receives its full admission.

Multiple approval SLOs. In this dissertation, we aim to maximize the satisfaction of a single, high-priority class of approvals. However, HEYP can support multiple levels of prioritized approvals by iteratively allocating routes and admissions for each class, with lower classes using the residual capacity left over from higher classes. The relative importance of a high-priority flowgroup's above-approval traffic compared to a lower-priority flowgroup's within-approval traffic depends on the cloud provider's business policy. For example, if cloud provider wanted to offer two bandwidth approval SLOs on a network with three QoS levels – HIPRI, MEDPRI, and LOPRI – the provider could choose to treat above-approval traffic for higher SLO approvals as equivalent to within-approval traffic for lower SLO approvals, marking both as MEDPRI.

2.6 Related Work

Software-defined WANs. The rising demand for network bandwidth across data centers has led to the development of many global private WANs, e.g., by Microsoft [89], Google [98], and Facebook [101]. These networks use centralized demand monitoring and traffic engineering to cost-efficiently transfer large volumes of data, though scaling them presents challenges [90, 72, 22]. While HEYP builds on these systems and shares a similar software-defined architecture, it aims to satisfy bandwidth approvals as a primary objective without sacrificing network utilization.

Bandwidth isolation between cloud tenants. Many prior systems aim to guarantee bandwidth between virtual machines in the data center, ranging from approaches that simply isolate tenants from one other [38, 77, 115], to others which provide work conservation [99, 144], to ones that enforce rich notions of fairness across tenants [143]. Adapting these approaches to the WAN setting is not straightforward. Providers have less flexibility with regards to application placement, control plane delays are significantly larger, and bandwidth guarantees are at the granularity of flowgroups, each of which spans a large number of hosts.

BwE [105] and SWAN [89] provide WAN bandwidth isolation by dynamically controlling the sending rates of tenants. HEYP differs from these approaches by combining static and dynamic allocation through the use of QoS downgrade.

Fault-tolerant routing. Many approaches have been proposed to quickly restore network connectivity following a failure [141, 119, 172, 183], and fault-tolerant traffic engineering approaches [118, 161, 41, 100] further aim to ensure that the remaining paths after a failure can support the admitted traffic. These approaches can be used together with HEYP to ensure high approval satisfaction without relying on the global controller reacting to either demand or topology changes.

QoS downgrade. Prior work has used QoS downgrade to provide statistical assurances of capacity to end users of the Internet [57, 56]. Unlike HEYP, these approaches do not scale to the large flowgroups present in data centers. HEYP accounts for the fact that no individual gateway can process all of the traffic belonging to a tenant, and it enables individual flowgroups to consume large quantities of capacity by allowing for any single flowgroup's traffic to be sent along multiple routes. Our use of separate routes for HIPRI and LOPRI traffic, however, introduces the need to maintain QoS stability, which HEYP explicitly aims to provide.

Cargo shipments. There are many similarities between the management of private WANs and shipping companies. Both perform forecasting [42], routing [98, 89, 171], manage disruptions [44, 118, 52], and overbook capacity [105, 71, 145], so one might wonder whether techniques used in HEYP might be applicable to shipping, or vice versa. However, there are a couple of factors that make this unlikely. First, unlike on a private WAN – where senders can always retransmit packets

– shipping companies cannot simply destroy low priority cargo as it belongs to their customers. Second, applications that use a WAN will react to changes in the network within hundreds of milliseconds. This makes them highly sensitive to small changes in delay (see §2.3.2). In contrast, cargo is shipped over timescales that are orders of magnitude larger; therefore, any disruptions would have to introduce correspondingly longer delays to be noticed.

Application-layer WAN optimization. Applications often have flexibility in where they can run tasks, e.g., when planning the execution of analytical jobs [169, 146, 108, 94, 180] or choosing which copies of data to read when multiple exist [164, 132, 112, 45]. This flexibility enables applications to reduce latency and avoid bandwidth bottlenecks in the network. However, as these techniques cannot prevent the network from becoming overloaded, a shared WAN still requires a separate mechanism to isolate tenants from one another.

2.7 Summary

Existing control plane architectures for global-scale private WANs are unable to offer highly available bandwidth guarantees at high utilization. A key cause is their dependence on a fundamentally slow central controller to reconfigure the network in response to changing traffic demands. In this work, we showed how to remove any reliance on the global controller for satisfying bandwidth guarantees by leveraging the data plane's ability to prioritize traffic based on QoS levels. Our HEYP WAN architecture uses the central controller only to maximize efficiency and handle topology changes, and we account for interactions with other layers of the network stack that result from admitting surplus traffic at a lower QoS along a separate set of routes. We showed that HEYP is able to simultaneously offer predictability and efficiency across a range of workloads and settings.

CHAPTER 3

PANDO: Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage

Replicating data across data centers is important for a web service to tolerate the unavailability of some data centers [39] and to serve users with low latency [88]. A front-end web server close to a user can serve the user's requests by accessing nearby copies of relevant data (see Figure 3.1). Even in collaborative services such as Google Docs and ShareLaTeX, accessing a majority of replicas suffices for a front-end to read or update shared data while preserving consistency.

However, it is challenging to keep data spread across the globe strongly consistent as no single design can simultaneously minimize read latency, write latency, and cost.

- To preserve consistency, *any* subset of sites which are accessed to serve a read must overlap with *all* subsets used for writes. Therefore, allowing a front-end to read from nearby data sites forces other front-ends to write to distant data sites, thus increasing write latency.
- Similarly, providing low read latency requires having at least one data site near each front-end, thereby increasing the total number of data sites. This inflates expenses incurred both for storage and for data transfers to synchronize data sites.

Given these tradeoffs, service providers must determine how to meet their desired latency goals at minimum cost. Or, correspondingly, how to minimize read and write latencies given a cost budget? We make the following contributions towards addressing these questions.

1. We show that existing solutions for enabling strongly consistent distributed storage are far from optimal in trading off latency versus cost. The cost necessary to satisfy bounds on read and write latencies is often significantly higher than the lowest cost theoretically feasible. For example, across a range of access patterns and latency bounds, the state-of-the-art geo-replication protocol EPaxos [132] imposes on average 30% higher storage cost than is optimal (§3.4.1.2). This sub-optimality also inflates the minimum latency bounds satisfiable within a cost budget.

2. We demonstrate the feasibility of achieving near-optimal latency versus cost tradeoffs in strongly consistent geo-distributed storage. In other words, we do not merely improve upon

the status quo, but show that there remains little room for improvement over the tradeoffs enabled by PANDO, our new approach for consensus across the wide-area network. PANDO exploits the property that, from any data center's perspective, some data centers are more proximate than others in a geo-distributed deployment. Therefore, beyond reducing the *number of round-trips* of wide-area communication when executing reads and writes (as has typically been the goal in prior work [123, 112, 132]), it is equally important to reduce the *magnitude of delay incurred on every round-trip*. We apply this principle in two ways.

2a. We show how to erasure-code objects across data sites without reads incurring higher wide-area latencies compared to replicated data. By splitting each object's data and storing one split (instead of one replica) per data site, a service can use its cost budget to spread each object's data across more data centers than is feasible with replication. To leverage this increased geographic spread for minimizing latencies, PANDO separates out two typically intertwined aspects of consensus: discovering whether the last write completed, and determining how to resolve any associated uncertainty. Since writes seldom fail in typical web service deployments, we enable a client to read an object by first communicating with a small subset of nearby data sites; only in the rare case when it is uncertain whether the last write completed does the client incur a latency penalty to discover how to resolve the uncertainty.

2b. In the wide-area setting, we show how to reach consensus in two rounds, yet approximate a one-round protocol's latency. Executing writes in two rounds simplifies compatibility with erasure-coded data, and we ensure that this approach has little impact on latency. First, PANDO requires clients to contact a smaller, more proximate subset of data sites in the first round than in the second round. Second, after a client initiates the first round, it delegates initiation of the second round to a more central data center, which receives all responses from the first round. By combining these two measures, messaging delays incurred in the first phase of a write help reduce the latency incurred in the second phase, instead of adding to it.

3. We compare PANDO to state-of-the-art consensus protocols via extensive measurement-driven analyses and in deployments on Azure. In the latency–cost tradeoff space, we find that PANDO reduces by 88% the median gap between achievable tradeoffs and the best theoretically feasible tradeoffs. Moreover, PANDO can cut dollar costs to meet the same latency goals by 46% and lower 95^{th} percentile read latency by up to 62% at the same storage overhead.

3.1 Setting and Motivation

We begin by describing our target setting, the approach we use for enabling globally consistent reads and writes, and the shortcomings of existing solutions that use this approach.

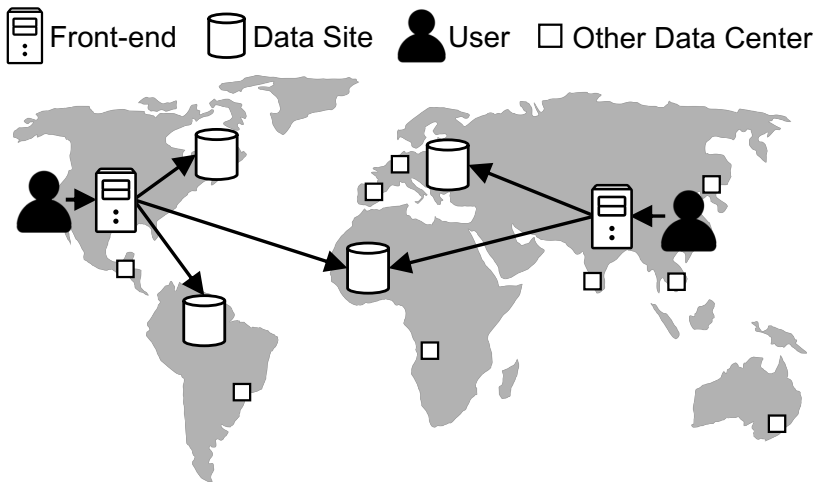


Figure 3.1: Users issue requests to their nearest front-end servers which in turn access geo-distributed storage.

3.1.1 System model, goals, and assumptions

We seek to meet the storage needs of globally deployed applications, such as Google Docs [74] and ShareLaTeX [6], in which low latency and high availability are critical, yet weak data consistency (such as eventual or causal) is not an option. In particular, we focus on enabling a geo-distributed object/key-value store which a service's front-end servers read from and write to when serving requests from users. We aim to support GETs and conditional-PUTs on any individual key; we defer support for multi-key transactions to future work. In contrast to PUTs (which blindly overwrite the value for a key), conditional-PUTs attempt to write to a specific version of a key and can succeed only if that version does not already have a committed value. This is essential in services such as Google Docs and ShareLaTeX to ensure that a client cannot overwrite an update that it has not seen.

In enabling such a geo-distributed key-value store, we are guided by the following objectives:

- **Strong consistency:** Ensure all reads and writes on any key are linearizable; i.e., all writes are totally ordered and every read returns the last successful write.
- **Low latency:** Satisfy service provider's SLOs[1] (service-level objectives) for bounds on read and write latencies, so as to ensure a minimum quality-of-service for all users. We focus on the wide-area latency incurred when serving reads or writes, assuming appropriate capacity planning and load balancing to bound queuing delays.
- **Low cost:** Minimize cost (sum of dollar costs for storage, data transfers, storage operations, and compute) necessary to satisfy latency goals. Since cost for storage operations and data

[1]Unlike SLAs, violations of SLOs are acceptable, but need to be minimized.

transfers grows with more copies stored, in parts of the dissertation, we use storage overhead (i.e., number of copies stored of every data item) as a proxy for cost. This frees us from making any assumptions about pricing policy or the workload (e.g., read-to-write ratio).

- **Fault-tolerance:** Serve requests on any key as long as fewer than f data centers are unavailable.

We focus on satisfying input latency bounds in the absence of conflicts and failures—both of which occur rarely in practice [47, 48, 122, 60]—but seek to minimize performance degradation when they do occur (§3.2.5 and §3.4.1.2). In addition, we build upon state-of-the-art cloud services which offer low latency variance between their data centers [81] and within their intra-data center storage services (e.g., Azure's CosmosDB provides a 10 ms tail read latency SLA [128]).

Note that, in order to satisfy desired latency SLOs at minimum cost (or to minimize latencies given a cost budget), a service cannot select the data sites for an object at random. Instead, as we describe later in Section 3.3, any service must utilize its knowledge of an object's workload (e.g., locations of the users among whom the object is shared) in doing so.

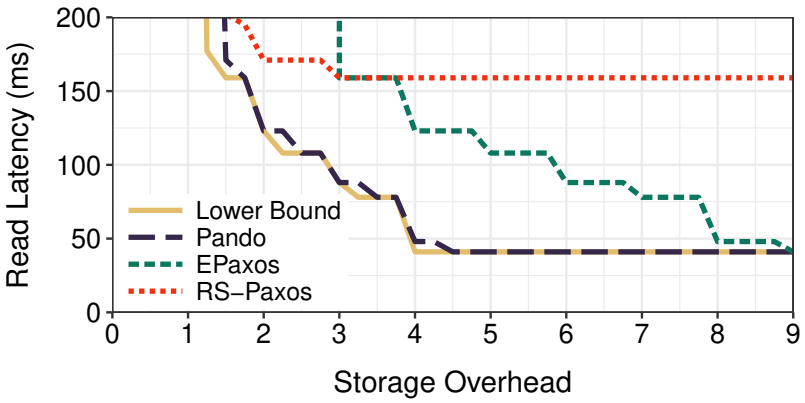
3.1.2 Approach

One can ensure linearizability in distributed storage by serializing all writes through a leader and rely on it for reads, e.g., primary-backup [30], chain replication [166], and Raft [140]. A single leader, however, cannot be close to all front-ends across the globe. Front-ends which are distant from the leader will have to suffer high latencies.

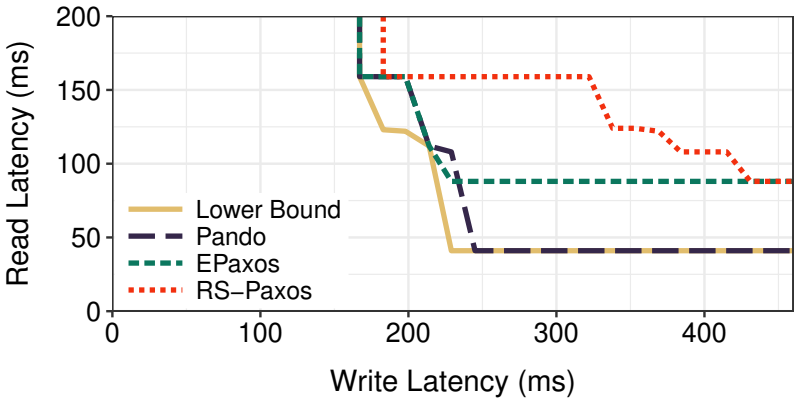
To reduce the need to contact a distant leader, one could use read leases [37, 133] and migrate the leader based on the current workload, e.g., choose as the leader the replica closest to the front-end currently issuing reads and writes. However, unless the workload exhibits very high locality, tail latency will be dominated by the latency overheads incurred during leader migration and lease acquisition.

To keep read and write latencies within specified bounds irrespective of the level of locality, we pursue a leaderless approach. Among the leaderless protocols which allow every front-end to read and write data from a subset of nearby data sites (a read or write quorum), we consider those based on Paxos because it enables consensus. Other quorum-based approaches [35] which only enable atomic register semantics (i.e., PUT and GET) are incapable of supporting conditional updates [86]. While there exist many variants of Paxos, in all cases, we can optimize latencies in two ways.

First, instead of executing Paxos, a front-end can read an object by simply fetching the object's data from a read quorum. To enable this, a successful writer asynchronously marks the version it wrote as committed at all data sites. In the common case, when there are no failures or conflicts, a read is complete in one round trip if the highest version seen across a read quorum is marked as



(a) Write latency $\leq$ 300 ms



(b) Storage overhead $\leq 6\times$

Figure 3.2: Slices of the three-dimensional tradeoff space where we compare latency estimates for replication-based EPaxos [132], erasure coding-based RS-Paxos [134], and our solution PANDO against a lower bound. Front-ends are in Azure's Australia East, Central India, East Asia, East US, and Korea South data centers, whereas data sites are chosen from all Azure data centers.

committed [114].

Second, instead of every front-end itself executing reads and writes, we allow for it to relay its operations through a delegate in another data center. The flexibility of utilizing a delegate can be leveraged to reduce latency when, compared to the front-end, that delegate is more centrally placed relative to the data sites of the object being accessed.

3.1.3 Sub-optimality of existing solutions

The state-of-the-art Paxos variant for geo-replicated data is EPaxos [132], as we show in Section 3.4. For typical replication factors (i.e., 3 or 5), EPaxos enables any front-end to read/write with one round of wide-area communication with the nearest majority of replicas. If lower read latencies than feasible with $2f + 1$ replicas are desired, then one can use a higher replication factor

N, set the size R of read quorums to be $\geq f + 1$ (to ensure overlap with write quorums even in the face of f failures) and set the size W of write quorums to $N - R + 1$ (to preserve consistency).

Figure 3.2 shows the tradeoffs enabled by EPaxos for an example access pattern. For each read latency bound, these graphs respectively plot the minimum storage overhead and write latency bounds that are satisfiable. As we discuss in §3.3, we compute these bounds by solving protocol-specific mixed integer programs which take as input the expected access pattern and latency measurements between all pairs of data centers (§3.4.1). We show two two-dimensional slices of the three-dimensional read latency–write latency–storage overhead tradeoff space.

To gauge the optimality of the tradeoffs achievable with EPaxos, we compare it against a lower bound. Given a bound on read latency, the minimum storage overhead necessary and the minimum write latency bound that can be satisfied cannot be lower than those determined by our lower bound. Though the lower bound may be unachievable by any existing consensus protocol, we compute it by solving a mixed integer program which assumes that reads and writes can be executed in a single round and enforces the following properties that *any* quorum-based approach must respect:

- *Tolerate unavailability of $\leq f$ data centers:* All data sites in at least one read and one write quorum must be available in the event that $\leq f$ data centers fail.
- *Prevent data loss:* At least one copy of data must remain in any write quorum when any f data sites are unavailable.
- *Serve reads:* The data sites in any read quorum must collectively contain at least one copy of the object.
- *Preserve strong consistency:* All read–write and write–write quorum pairs must have a non-empty intersection.

Equally important are constraints that we do *not* impose: all read quorums (same for write quorums) need not be of the same size, and an arbitrary fraction of an object's data can be stored at any data site.

Figure 3.2 shows that EPaxos is sub-optimal in two ways. First, to meet any particular bound on read latency, EPaxos imposes a significant cost overhead; in Figure 3.2(a), EPaxos requires at least 9 replicas to satisfy the lowest feasible read latency bound (40 ms), whereas the lower bound storage overhead is 4x. Recall that, greater the number of copies of data stored, higher the data transfer costs when reading and writing. Second, given a cost budget, the read latencies achievable with EPaxos are significantly higher than the lower bound; in Figure 3.2(b), where storage overhead is capped at 6x, we see that the minimum read latency achievable with EPaxos (80 ms) is twice the lower bound (40 ms).

Of course, a lower bound is just that; some of the tradeoffs that it deems feasible may potentially be unachievable. However, for the example in Figure 3.2 and across a wide range of configurations

in Section 3.4, we show that PANDO comes close to matching the lower bound. We describe how next.

3.2 Design

The fundamental source of EPaxos's sub-optimality in trading off cost and latency is its reliance on replication. Replication-based approaches inflate the cost necessary to meet read latency goals because spreading an object's data across more sites entails storing an additional *full copy* at each of these sites. To enable latency versus cost tradeoffs that are closer to optimal, the key is to store a *portion* of an object's data at each data site, like in the lower bound.

Therefore, we leverage erasure coding, a data-agnostic approach which enables such flexible data placement while matching replication's fault-tolerance at lower cost [174]. For example, to tolerate $f = 1$ failures, instead of requiring at least $2f + 1 = 3$ replicas, one could use Reed-Solomon coding [150] to partition an object into $k = 2$ splits, generate $r = 2$ parity splits, and store one split each at $k+r = 4$ sites; any k splits suffice to reconstruct the object's data. Compared to replication, this reduces storage overhead to $2\times$, thus also reducing the number of copies of data transferred over the wide-area when reading or writing.

State-of-the-art implementations of erasure coding [96] require only hundreds of nanoseconds to encode or decode kilobyte-sized objects. This latency is negligible compared to wide-area latencies, which range from tens to hundreds of milliseconds. Moreover, the computational costs for encoding and decoding pale in comparison to costs for data transfers and storage operations (§3.4.1.3).

3.2.1 Impact of erasure coding on wide-area latency

While there exist a number of protocols which preserve linearizability on erasure-coded data [25, 49], they largely focus on supporting PUT/GET semantics. To support conditional updates, we consider how to enable consensus on erasure-coded data with a leaderless approach such as Paxos. We have one of two options.

One approach would be to extend one of several one-round variants of Paxos to work on erasure-coded data. However, most of these protocols require large quorums (e.g., a write would have to be applied to a super-majority [112] or even all [123] data sites), rendering them significantly worse than the lower bound. Whereas, extending EPaxos [132], which requires small quorums despite needing a single round, to be compatible with erasure-coded data is far from trivial given the complex mechanisms that it employs for failure recovery.

Therefore, we build upon the classic two-phase version of Paxos [110] and address associated

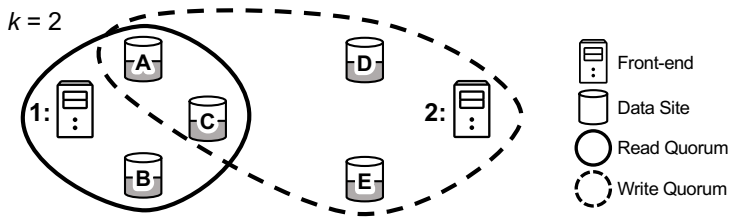
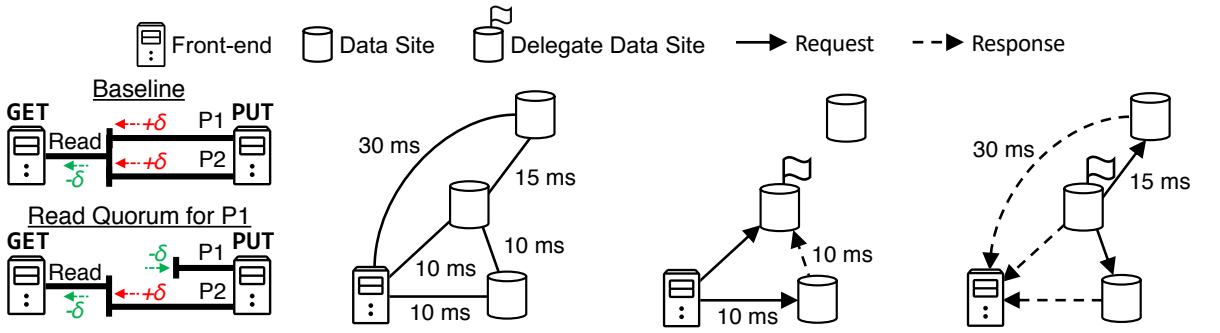


Figure 3.3: Example execution of RS-Paxos on an erasure-coded object, whose data is partitioned into $k = 2$ splits. For all readers and writers to be able to reconstruct the last successful write, any write quorum must have an overlap of k or more data sites with every read and every write quorum.

latency overheads. In either phase, a writer (a front-end or its delegate) communicates with all the data sites of an object and waits for responses from a write quorum. In Phase 1, the writer discovers whether there already is a value for the version it is attempting to write and attempts to elect itself leader for this version. In Phase 2, it sends its write to all data sites. A write to a version succeeds only if, prior to its completion of both phases, no other writer has been elected the leader. If the leader fails during Phase 2 but the write succeeds at a quorum of data sites, subsequent leaders will adopt the existing value and use it as part of their Phase 2, ensuring that the value for any specific version never changes once chosen.

This natural application of Paxos on erasure-coded data, called RS-Paxos [134], is inefficient in three ways.

- **Two rounds of wide-area communication.** Any reduction in read latency achieved by enabling every front-end to read from a more proximate read quorum has twice the adverse effect on write latency. In Figure 3.2(b), we see that when the read latency bound is stringent (e.g., $\leq$ 100 ms), the minimum write latency bound satisfiable with RS-Paxos is twice that achievable with EPaxos. When the read latency bound is loose (e.g., $\geq$ 150 ms), write latency inflation with RS-Paxos is lower because the data sites are close to each other and front-ends benefit from delegation.
- **Increased impact of conflicts.** Executing writes in two rounds makes them more prone to performance degradation when conflicts arise. When multiple writes to the same key execute concurrently, none of the writes may succeed within two rounds. Either round of each write may fail at more than a quorum of data sites if other writes complete one of their rounds at those sites.
- **Larger intersections between quorums.** As we see in Figure 3.2(a), at storage overheads of 4x or more, the minimum read latency bound satisfiable with RS-Paxos is significantly higher than that achievable with EPaxos. This arises because, when an object's data is partitioned into k splits, every read quorum must have an overlap of at least k sites with every write quorum (see



(a) Small Phase 1 Quorums (b) Inter-DC Latencies (c) Phase 1 by Front-end (d) Phase 2 by Delegate

Figure 3.4: PANDO's techniques for optimizing write latency. (a) Reusing read quorums in Phase 1 of writes enables reduction in read latency without impacting (Phase 1 + Phase 2) latency for writes. (b) Example deployment with one-way delays between relevant pairs of data centers shown. Phase 1 quorum size is 2 and Phase 2 quorum size is 3. If same (Phase 2) quorum were used in both phases of a write, like in RS-Paxos, write latency would be 120 ms. (c) and (d) By directing Phase 1 responses to a delegate and having it initiate Phase 2, PANDO reduces write latency to 65 ms (20 ms in Phase 1 + 45 ms in Phase 2), close to the 60 ms latency feasible with one-round writes.

Figure 3.3). Thus, erasure coding's utility in helping spread an object's data across more sites (than feasible with replication for the same storage overhead) is nullified.

3.2.2 Overview of PANDO

What if these inefficiencies did not exist when executing Paxos on erasure-coded data? To identify the latency versus cost tradeoffs that would be achievable in this case, we consider a hypothetical ideal execution of Paxos on erasure-coded data: one which requires a single round of communication and can make do with an overlap of only one site between read–write and write–write quorum pairs. For the example used in Figure 3.2, this hypothetical ideal (not shown in the figure) comes close to matching the lower bound.

Encouraged by this promising result, we design PANDO to approximate this ideal execution of Paxos on erasure-coded data. First, we describe how to execute Paxos in two rounds on geo-distributed data, yet come close to matching the messaging delays incurred with one-round protocols. Second, leveraging the rarity of conflicts and failures in typical web service workloads, we describe how to make do with a single data site overlap between quorums in the common case. Finally, we discuss how to minimize performance degradation when conflicts do arise. In our description, we assume an object's data is partitioned into k splits.

3.2.3 Mitigating write latency

We reduce the latency overhead of executing Paxos in two rounds by revisiting the idea of delegation (§3.1.2): a front-end sends its write request to a stateless delegate, which executes Paxos and returns the response. When data sites are spread out (to enable low read latencies), two round-trips to a write quorum incurs comparable delay from the front-end versus from the delegate. The round-trip from the front-end to the delegate proves to be an overhead.

To mitigate this overhead, what if 1) transmission of the message from the front-end to the delegate overlaps with Phase 1 of Paxos, and 2) transmission of the response back overlaps with Phase 2? The latency for a front-end to execute the two-phase version of Paxos would then be roughly equivalent to one round-trip between the front-end and the delegate, thus matching the latency feasible with a one-round protocol. We show how to make this feasible in two steps.

3.2.3.1 Shrinking Phase 1 quorums

First, we revisit the property of classic Paxos that a writer needs responses from the same number of data sites in both phases of Paxos: the size of a write quorum. To ensure that a writer discovers any previously committed value, Paxos only requires that any Phase 1 quorum intersect with every Phase 2 quorum; Phase 1 quorums need not overlap [93]. In PANDO, we take advantage of this freedom to use a smaller quorum in the first phase of Paxos than in the second phase.

We observe that the intersection requirements imposed on Phase 1 and Phase 2 quorums are precisely the properties required of read and write quorums: any read quorum must intersect with every write quorum, whereas no overlap between read quorums is required. Therefore, when executing Phase 1 of Paxos to write to an object, it suffices to get responses from a read quorum, thus allowing improvements in read latency to also benefit leader election. A writer (a front-end or its delegate) needs responses from a write quorum only when executing Phase 2.

Figure 3.4(a) illustrates the corresponding improvements in write latency. When a quorum of the same size is used in both phases of a write, a reduction of δ in the read latency bound results in a 2δ increase in the minimum satisfiable write latency bound (because of the need for read and write quorums to overlap). In contrast, our reuse of read quorums in the Phase 1 of writes ensures that spreading out data sites to enable lower read latencies has (roughly speaking) no impact on write latency; when read quorums are shrunk to reduce the read latency bound by δ, the increase of δ in Phase 2 latency (to preserve overlap between quorums) is offset by the decrease of δ in Phase 1 latency.

3.2.3.2 Partially delegating write logic

While our reuse of read quorums in Phase 1 of a write helps reduce write latency, Phase 2 latency remains comparable to a one-round write protocol. Therefore, the total write latency remains significantly higher than that feasible with one-round protocols.

PANDO addresses this problem via *partial* use of delegation. Rather than having a front-end executing a write either do all the work of executing Paxos itself or offload all of this work to a delegate, we offload *some* of it to a delegate.

Figures 3.4(c–d) show how this works in PANDO. A front-end initiates Phase 1 of Paxos by sending requests to data sites of the object it is writing to, asking them to send their responses to a chosen delegate. In parallel, the front-end sends the value it wants to write directly to the delegate. Once the delegate receives enough responses (i.e., the size of a read quorum), it will either inform the front-end that Phase 1 failed (the rare case) or initiate Phase 2 (the common case), sending the value to be written to all data sites for the object. Those data sites in turn send their responses directly back to the front-end, which considers the write complete once it receives responses from a write quorum.

Note that partial delegation preserves Paxos's fault tolerance guarantees. To see why, consider the case where a end-user's client sends the same request to two front-ends—perhaps due to suspecting that the first front-end has failed—and both front-ends execute the request. Paxos guarantees that at most one of these writes will succeed. Similarly, with partial delegation, in the rare case when the front-end suspects that the delegate is unavailable, it can simply re-execute both phases on its own. Paxos will resolve any conflicts and at most one of the two writes (one executed via the delegate and the other executed by the front-end) will succeed.

Thanks to the heterogeneity of latencies across different pairs of data centers, the use of small Phase 1 quorums combined with the delegation of Phase 2 eliminates most of the latency overhead of two-phase writes. In Figure 3.4(b-d), the two techniques reduce write latency down from 120 ms with classic Paxos to 65 ms with PANDO, only 5 ms higher than what can be achieved with a one-round protocol. The remaining overhead results from the fact that there still has to be some point of convergence between the two phases.

3.2.4 Enabling smaller quorums

The techniques we have described thus far lower the minimum write latency SLO that is satisfiable given an SLO for read latency. However, as we have seen in Figure 3.2(a), erasure coding inflates the minimum read latency SLO achievable given a cost budget (e.g., a bound on storage overhead). As discussed in Section 3.2.1, this is due to the need for larger intersections between quorums when data is erasure-coded, as compared to when replicated.

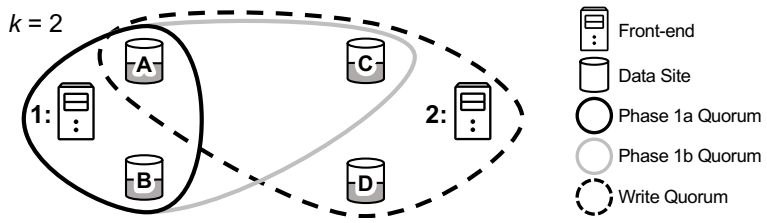


Figure 3.5: For an object partitioned into $k = 2$ splits, PANDO requires an overlap of only one site between any Phase 1a and Phase 2 quorum. Responses from the larger Phase 1b quorum are needed only in the case of failure or conflict.

Recall that the need for an intersection of k data sites between any pair of read and write quorums exists so that any read on an object will be able to reconstruct the last value written; at least k splits written during the last successful write will be part of any read quorum. Thus, linearizability is preserved even in the worst case when a write completes at the minimum number of data sites necessary to be successful: a write quorum. However, since concurrent writes are uncommon [122, 60] and data sites are rarely unavailable in typical cloud deployments [47, 48], most writes will be applied to all data sites. Therefore, in the common case, all data sites in any read quorum will reflect the latest write.

In PANDO, we leverage this distinction between the common case and the worst case to optimize read latency (and equivalently any write's Phase 1 latency, given that PANDO uses the same quorum size in both cases) as follows.

Read from smaller quorum in the common case. After issuing read requests to all data sites, a reader initially waits for responses from a subset which is 1) at least of size k and 2) has an intersection of at least one site with every write quorum; we refer to this as a Phase 1a quorum. In the common case, all k splits have the same version and at least one of them is marked committed; the read is complete in this case. An overlap of only one site with every write quorum suffices for the reader to discover the latest version of the object; at least one of the splits received so far by the reader will be one written by the last successful write to this object.

Read from larger quorum if failure or conflict. At this juncture, if the last successful write has not yet been applied to all data sites, the reader may only know the latest version of the object but not the value of that version. To reconstruct that value, the reader must wait for responses from more data sites until the subset it has heard from has an overlap of k sites or more with every write quorum; this is a Phase 1b quorum. As a result, a reader must incur the latency penalty of waiting for responses from farther data sites only if the last successful write was executed when either some data sites were unavailable or a conflicting write was in progress.

In the example in Figure 3.5, Front-end 1 can complete reading based on responses from sites A and B in the common case since two splits suffice to reconstruct the object. If the last write

was from Front-end 2 and this write completed only at a subset of sites, there are two cases to consider:

- If Front-end 2's write has been applied to a write quorum (say A, C, and D), then the response from site A will help Front-end 1 discover the existence of this write. Front-end 1 needs an additional response from C in this case to be able to reconstruct the value written by Front-end 2.
- If Front-end 2's write has been applied to less than a write quorum (say, A and D), then Front-end 1 may be unable to find k splits for this version even from a Phase 1b quorum (A, B, and C). In this case, that value could not have been committed to *any* Phase 2 quorum. Therefore, the reader falls back to the previous version. PANDO garbage collects the value for a version only once a value has been committed for the next version (§3.3). The overhead of storing multiple versions of a key will be short-lived in our target setting where failures and write conflicts are rare.

Phase 1a and 1b quorums can also be used as described above during the first round of a write. The only difference in the case of writes is that responses from data sites can be potentially directed to a delegate at a different data center than the one which initiates Phase 1.

To preserve correctness of both reads and writes, the minimum size of Phase 1a quorums must be $\max(k, f+1)$, and Phase 1b and Phase 2 quorums must contain at least $f+k$ data sites. These quorum sizes are inter-dependent because any Phase 1a quorum must have a non-empty overlap with every Phase 2 quorum and any Phase 1b or Phase 2 quorum must have an overlap of at least k sites with every Phase 2 quorum. For each of the three quorum types, all quorums of that type are of the same size and any subset of data sites of that size represent a valid quorum of that type.

Note that, if further reductions in common-case read latency are desired, one could use timed read leases as follows [37, 133]. Instead of using the normal read path, a front-end that holds a lease for a key could cache the value or fetch it from k nearby data sites to avoid the latency of communicating with a complete Phase 1a quorum. However, this approach would not benefit tail latency for reads and may increase latency for writes.

3.2.5 Reducing impact of conflicting writes

Lastly, we discuss how PANDO mitigates performance degradation when conflicts arise. As mentioned in §3.1, since conflicts rarely occur in practice [122, 60], we allow for violations of input latency bounds when multiple writes to a key execute concurrently. However, we ensure that the latency of concurrent writes is not arbitrarily degraded.

Our high-level idea is to select one of every key's data sites as the leader and to make use of this leader *only when conflicts arise*. PANDO's leaderless approach helps satisfy lower latency bounds

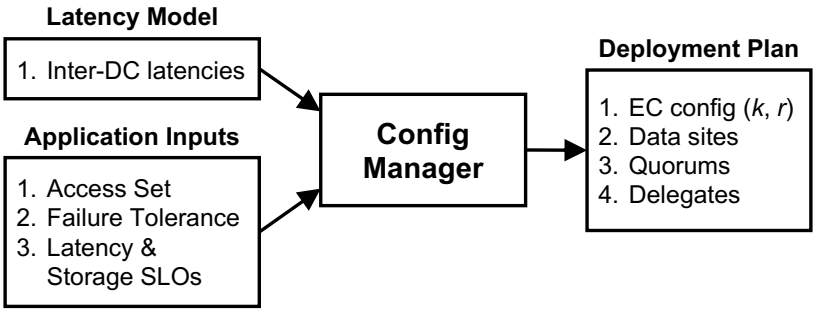


Figure 3.6: Selecting a deployment plan with ConfigManager.

by eliminating the need for any front-end to contact a potentially distant leader. However, when a front-end's attempted write fails and it is uncertain whether a value has already been committed for this version, the front-end forwards its write to the leader. In contrast to the front-end retrying the write on its own, relying on the leader can ensure that the write completes within at most two rounds.

To make this work, we ensure that any write executed by a key's leader always supersedes writes to that key being attempted in parallel by front-ends. For this, we exploit the fact that front-ends always retry writes via the leader, i.e., any front-end will attempt to directly execute a write at most once. Therefore, when executing Paxos, we permit any front-end to use proposal numbers of the form (0, front-end's ID) but only allow the leader to set the first component to values greater than or equal to 1, so that its writes take precedence at every data site.

Note that, since we consider it okay to violate the write latency bound in the rare cases when conflicts occur, we do not require the leader to be close to any specific front-end. Therefore, leader election can happen in the background (using any of a number of approaches [46, 24]) whenever the current one fails. If conflicting writes are attempted precisely when the leader is unavailable, these writes will block until a new leader is elected. Like prior work [132, 110], PANDO cannot bound worst-case write latency when conflicts *and* data center failures occur simultaneously.

A proof of PANDO's correctness and a TLA+ specification are in Appendices B and C.

3.3 Implementation

To empirically compare the manner in which different consensus approaches trade off read latency against write latency and cost, we implemented a key-value store which optimizes the selection of data sites for an object based on knowledge of how the object will be accessed.

ConfigManager. Central to this key-value store is the ConfigManager, which sits off the data path (thus not blocking reads and writes) and identifies *deployment plans*, one per access pattern.

As shown in Figure 3.6, a deployment plan determines the number of splits k that the key's value is partitioned into, the number of redundant splits r, and the $k + r$ data sites at which these splits are stored; $k = 1$ corresponds to replication, and Reed-Solomon coding [150] is used when $k > 1$. The deployment plan also specifies the sizes of different quorum types and the choice of delegates (if any).

To make this determination, in addition to the application's latency, cost, and fault-tolerance goals, ConfigManager relies on the application to specify every key's *access set*: data centers from which front-ends are expected to issue requests for the key. An application can determine an object's access set based on its knowledge of the set of users who will access that object, e.g., in Google Docs, the access set for a document is the set of data centers from which the service will serve users sharing the document. When uncertain (e.g., for a public document), the access set can be specified as comprising all data centers hosting its front-ends; this uncertainty will translate to higher latencies and cost.

The ConfigManager selects deployment plans by solving a mixed integer program, which accounts for the particular consensus approach being used. For example, PANDO's ConfigManager selects a delegate and preferred quorums per front-end, using RTT measurements to predict latencies incurred. Given bounds on any two dimensions of the tradeoff space, the ConfigManager can optimize the third (e.g. minimize max read latency across front-ends given write latency and storage cost SLOs). Given the stability of latencies observed between data centers in the cloud both in prior work [81] and in our measurements,[2] and since our current implementation assumes an object's access set is unchanged after it is created, we defer reconfiguration of an object's data sites [34] to future work.

Executing reads and writes. Unlike typical applications of Paxos, our use of erasure coding prevents servers from processing the contents of Paxos logs. Instead of separating application and Paxos state, we maintain one Paxos log for every key and aggressively prune old log entries. In order to execute a write request, a Proxy VM initiates Phase 1 of Paxos and waits for the delegate to run Phase 2. If the operation times out, the Proxy VM assumes the delegate has failed and executes both phases itself. Once Phase 2 successfully completes, the Proxy VM notifies the client and asynchronously informs learners so that they may commit their local state and garbage collect old log entries. The read path is simpler: a Proxy VM fetches the associated Paxos state and reconstructs the latest value before returning to the client. If the latest state happens to be uncommitted, then the Proxy VM issues a write-back to guarantee consistency.

[2] In six months of latency measurements between all pairs of Azure data centers, we observe less than 6% change in median latency from month to month for any data center pair and less than 10% difference between 90^{th} percentile and median latency within each month for most pairs.

3.4 Evaluation

We evaluate PANDO in two parts. First, in a measurement-based analysis, we estimate PANDO's benefits over prior solutions for enabling strongly consistent distributed storage. We quantify these benefits not only with respect to latency and cost separately, but also the extent to which PANDO helps bridge the gap to the lower bound in the latency–cost tradeoff space (§3.1). Second, we deploy our prototype key-value store and compare latency and throughput characteristics under microbenchmarks and an application workload. The primary takeaways from our evaluation are:

- Compared to the union of the best available replication- and erasure coding-based approaches, PANDO reduces the median gap to the lower bound by 88% in the read latency–write latency–storage overhead tradeoff space.
- Compared to EPaxos, given bounds on any two of storage overhead, read latency, and write latency, PANDO can improve read latency by 12–31% and reduce dollar costs (for storage, compute, and data transfers) by 6–46%, while degrading write latency by at most 3%.
- In a geo-distributed deployment on Azure, PANDO offers 18–62% lower read latencies than EPaxos and can reduce 95th percentile latency for two GitLab operations by 19–60% over EPaxos and RS-Paxos.

3.4.1 Measurement-based analysis

Setup. Our analysis uses network latencies between all pairs of 25 Microsoft Azure data centers. We categorize access sets (the subset of data centers from which an object is accessed) into four types: North America (NA), North America & Europe (NA-EU), North America & Asia (NA-AS), and Global (GL). For NA and NA-EU, we use 200 access sets chosen randomly. For NA-AS and GL, we first filter front-end data centers so that they are at least 20 ms apart, and then sample 200 random access sets. In all cases, we consider all 25 Azure data centers as potential data sites.

We compare PANDO to four replication-based approaches (EPaxos [132], Fast Paxos [112], Mencius [123], and Multi-Paxos [110]) and the only prior approach which can enable conditional updates on erasure-coded data (RS-Paxos [134]). We refer to the union of EPaxos and RS-Paxos (i.e., use either approach to satisfy the desired SLOs) as EP $\cup$ RSP.

Metrics. Our analysis looks at three types of metrics: 1) read and write latency (in either case, we estimate the max latency seen by any front-end in the access set) and storage overhead (size of the data stored divided by size of user data); 2) *GapVolume*, a metric which captures the gap in the three-dimensional read latency–write latency–storage overhead tradeoff space between the lower bound (described in §3.1) and the approach in question; and 3) total dollar cost as the sum

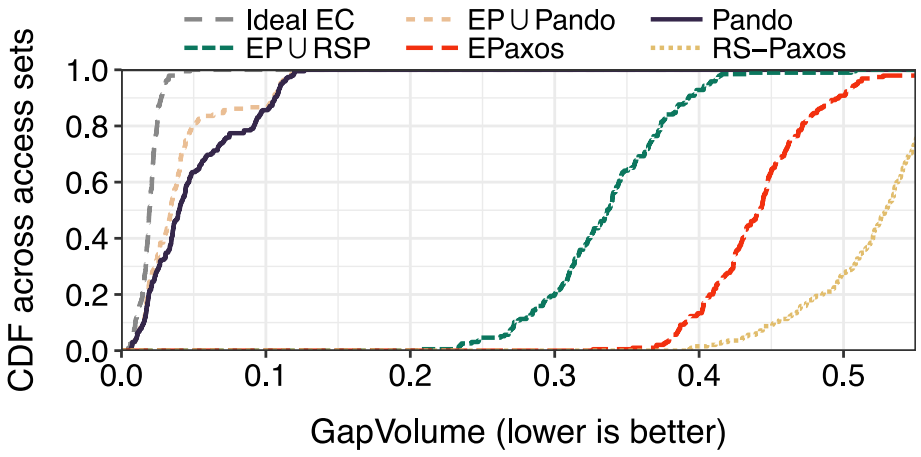


Figure 3.7: For NA-AS access sets, comparison of GapVolume with PANDO to EPaxos and RS-Paxos individually and their union (EP ∪ RSP). In addition, we evaluate EP ∪ PANDO (the union of EPaxos and PANDO) and Ideal EC (a hypothetical Paxos variant that supports erasure coding, one-round writes, and 1-split intersection across quorums).

of compute, storage, data transfer, and operation costs necessary to support reads and writes.

3.4.1.1 Impact on Achievable Tradeoffs

We use GapVolume to evaluate how close each approach is to the lower bound (§3.1.3). For any access set, we compute GapVolume with a specific consensus approach as the gap in the (read, write, storage) tradeoff space between the surfaces represented by the lower bound and by tradeoffs achievable with this consensus approach. We normalize this gap relative to the volume of the entire theoretically feasible tradeoff space, i.e., the portion of the tradeoff space above the lower bound surface. For every access set, we cap read and write latencies at values that are achievable with all approaches, and we limit storage overhead to a maximum of 7 as higher values are unlikely to be tenable in practice.

Proximity to lower bound. Figure 3.7 shows that PANDO significantly reduces GapVolume compared to EPaxos and RS-Paxos for access sets of type NA-AS. We do not show results for other replication-based approaches because they are subsumed by EPaxos, i.e., every combination of SLOs that is achievable with Mencius, Fast Paxos, and Multi-Paxos is also achievable with EPaxos. PANDO lowers median GapVolume to 4%, compared to 53% with RS-Paxos and 44% with EPaxos. Even with EP ∪ RSP (i.e., use two significantly different designs to realize different tradeoffs), median GapVolume remains at 34%. Table 3.1 shows similar benefits for NA, NA-EU, and GL access sets.

Moreover, EP ∪ PANDO (i.e., SLO combinations achievable with any of EPaxos or PANDO) is

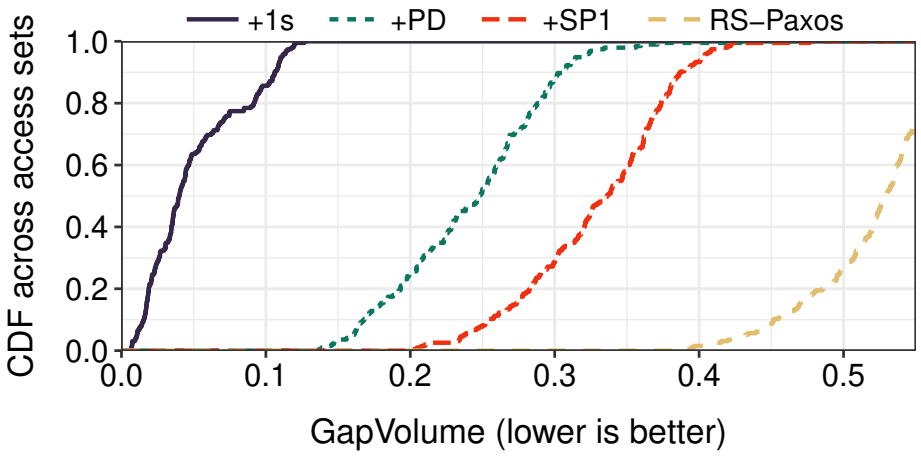


Figure 3.8: For access sets of type NA-AS, contributions of each of PANDO's techniques in reducing GapVolume. SP1 = small Phase 1, PD = partial delegation, 1s = 1-split overlap.

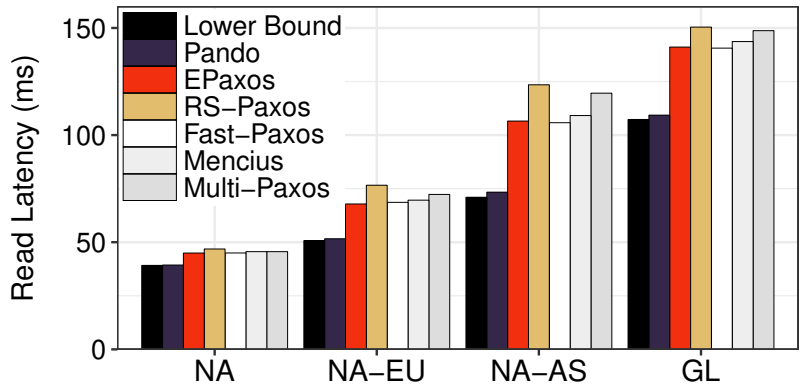
GapVolume	NA	NA-EU	NA-AS	GL
PANDO	0.06	0.07	0.04	0.07
EP ∪ RSP	0.37	0.40	0.34	0.34
EPaxos	0.44	0.48	0.44	0.49
RS-Paxos	0.52	0.59	0.53	0.48

Table 3.1: GapVolume for median access set of various types. Lower values are better; imply closer to the lower bound.

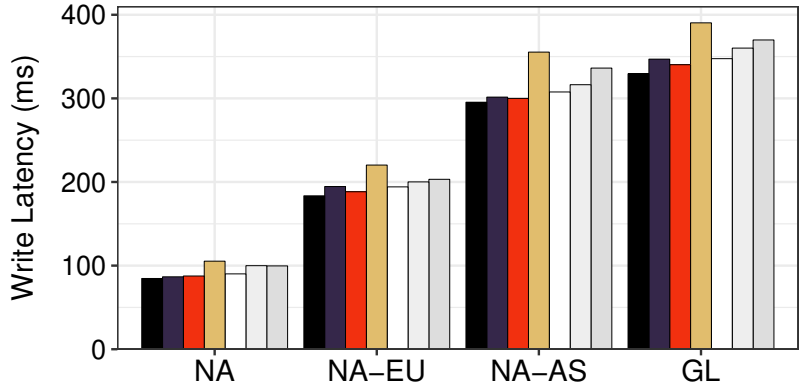
only marginally closer to the lower bound (i.e., has lower GapVolume) than PANDO, and that too only for some access sets. The few SLO combinations that EPaxos can achieve but not PANDO all have low write latency SLOs, in which case no choice of delegate can help PANDO overcome the overheads of two-round writes.

Utility of individual techniques. Figure 3.8 shows that each of the techniques used in PANDO contribute to the GapVolume reductions. For the median access set, using small Phase 1 quorums reduces GapVolume over RS-Paxos by 36%, adding partial delegation reduces GapVolume by a further 16%, and finally incorporating 1-split intersection reduces GapVolume by an additional 39%. When examining the improvements for each access set, we observe that both small Phase 1 quorums and 1-split intersection help across all access sets by reducing quorum size requirements. Similarly, we find that partial delegation typically improves GapVolume, indicating that some data sites are often closer to Phase 1 and Phase 2 quorums than the front-end.

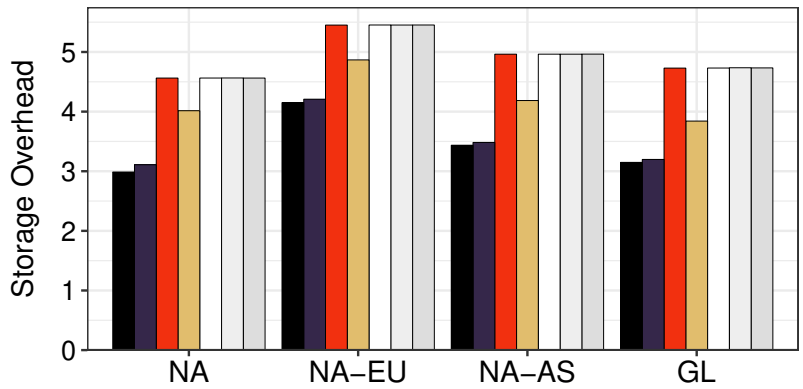
Obstacles to matching the lower bound. From the gap between PANDO and Ideal EC in Figure 3.7, we surmise that most of the remaining gap between PANDO and the lower bound could



(a) Read Latency



(b) Write Latency



(c) Storage Overhead

Figure 3.9: Average performance across different metrics. Lower is better in all plots. For each metric, we pick SLO combinations for the other two metrics that are achievable with all approaches. For each such SLO pair, we estimate the minimum value of the metric achievable with each approach. We then take the geometric mean across all access sets and SLO pairs.

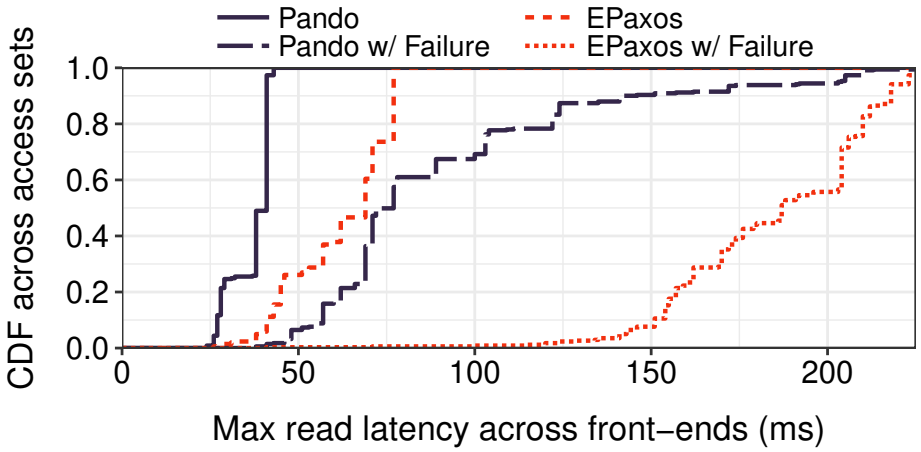


Figure 3.10: For access sets of type NA-AS, impact of data center failures on read latency for PANDO and EPaxos (300 ms write SLO, 5× overhead storage SLO).

be closed if one-round writes on erasure-coded data were feasible. Addressing any potential sub-optimality thereafter likely requires realizing the lower bound's flexibility with regards to varying the fraction of an object's data across sites (e.g., by using a different erasure coding strategy than Reed-Solomon coding) and varying quorum sizes across front-ends.

3.4.1.2 Latency and Storage Improvements

Figure 3.9 examines improvements in each of read latency, write latency, and storage overhead independently. To do this for read latency, we first identify all (write, storage) SLO pairs that are achievable by all candidate approaches. For each such pair, we then estimate the lowest read latency bound that is satisfiable with each approach. We take the geometric mean [70] across all feasible (write, storage) SLO pairs for all access sets to compare PANDO's performance relative to other approaches. We perform similar computations for write latency and storage overhead.

We find that PANDO achieves 12–31% lower read latency, 0–3% higher write latency, and 22–32% lower storage overhead than EPaxos across all types of access sets. Although PANDO executes writes in two phases, the use of small Phase 1 quorums plus partial delegation provides similar write latency as EPaxos. In all cases, EPaxos outperforms Fast Paxos, Mencius, and Multi-Paxos. Compared to RS-Paxos, PANDO reduces read latency by 15–40%, write latency by 11–17%, and storage overhead by 13–22%.

Latency under failures. Figure 3.10 compares the read latency bounds satisfiable with PANDO and EPaxos when any one data center is unavailable. During failures, a front-end may need to contact more distant data sites in order to read or write data. In this case, for the median access set, we observe that PANDO supports a read latency bound which is 110 ms lower than EPaxos. Since

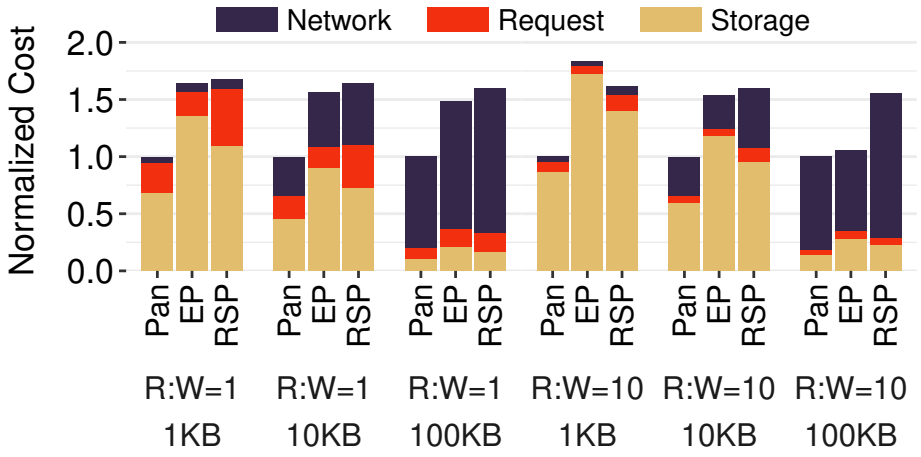


Figure 3.11: Comparison of cost for a month in NA-AS to store 10 TB of data and execute 600M requests/month. In all cases, the costs of Proxy VMs (not shown) were negligible, and read and write latency SLOs were set to 100 ms and 375 ms.

erasure coding spreads data more widely than replication for the same storage overhead, there are more nearby sites to fall back on when a failure occurs.

However, erasure coding is not universally helpful in failure scenarios. Upon detecting the loss of its write delegate, a PANDO front-end will locally identify a new one that minimizes latency at the front-end. Still, across NA-AS access sets, median write latency with PANDO is 10% higher than EPaxos when any one data site is unavailable, despite the two approaches having similar latency in the failure-free case. In addition, under permanently data loss, bringing up a replacement data site requires decoding the data of k separate sites instead of fetching the same volume of data from one replica.

3.4.1.3 Cost

Beyond storage, public cloud providers also charge users for wide-area data transfers, PUT/GET requests to storage, and for virtual machines used to execute RPCs and encode/decode data. These overheads have driven production systems to adopt two key optimizations. First, replication-based systems execute reads by fetching version numbers from remote sites, not data. Second, Paxos-based systems issue writes only to a quorum (or for PANDO, a superset of a write quorum that intersects with all Phase 1a quorums likely to be used). Taking these optimizations into consideration, we now account for these other sources of cost and evaluate PANDO's utility in reducing total cost.

We considered 200 access sets of type NA-AS and set latency SLOs that both RS-Paxos and EPaxos are capable of meeting: 100 ms for read latency and 375 ms for write latency. We derived the CPU cost of Proxy VMs by measuring the throughput achieved in deployments of our prototype

system. Using pricing data from Azure CosmosDB [125], we estimated the cost necessary to store 10 TB of data and issue 600M requests, averaged across all access sets; these parameters are based on a popular web service's workload [5] and a poll of typical MySQL deployment sizes [179].

Across several values for mean object size and read-to-write ratio, Figure 3.11 shows that PANDO reduces overall costs by 6–46% over EPaxos and 35–40% over RS-Paxos. When objects are large, PANDO's cost savings primarily stem from the reduction in the data transferred over the wide-area network. Note that even though EPaxos uses replication, it still requires reading remote data when a copy is not stored at the front-end data center. Whereas, when objects are small, storage fees dominate and PANDO reduces cost primarily due to the lower storage overhead that it imposes. Though erasure coding increases the number of requests to storage compared to replication, ConfigManager opts to erasure-code data only when the corresponding decrease in storage and data transfer costs help reduce overall cost. Unlike write requests, which have to first write metadata to storage before transferring and writing the data itself, read requests only issue storage operations to fetch data. This leads to greater cost reductions for read-dominated workloads.

3.4.2 Prototype deployment

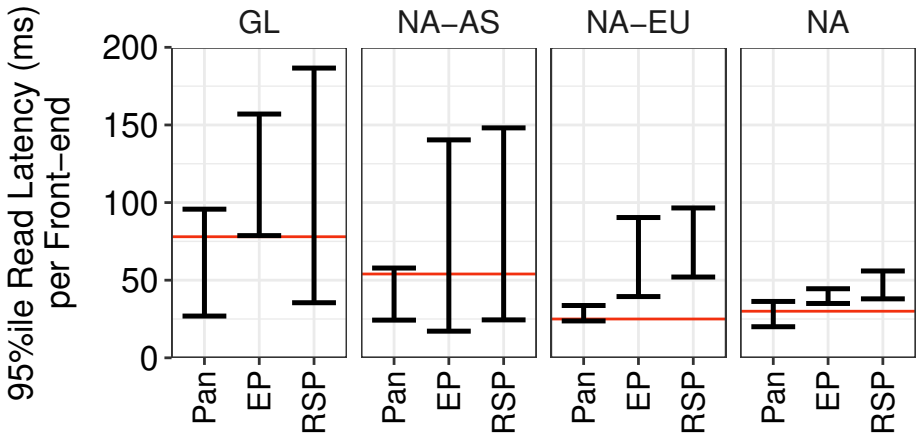
Next, via deployments on Azure, we experimentally compare PANDO versus EPaxos and RS-Paxos. We use our implementations of PANDO and RS-Paxos and the open-source implementation of EPaxos [131]. This experimental comparison helps account for factors missing from our analysis, such as latency variance and contention between requests. We consider one access set of each of our 4 types:

- NA: Central US, East US, North Central US, West US
- NA-EU: Canada East, Central US, North Europe, West Europe
- NA-AS: Central US, Japan West, Korea South
- GL: Australia East, North Europe, SE Asia, West US

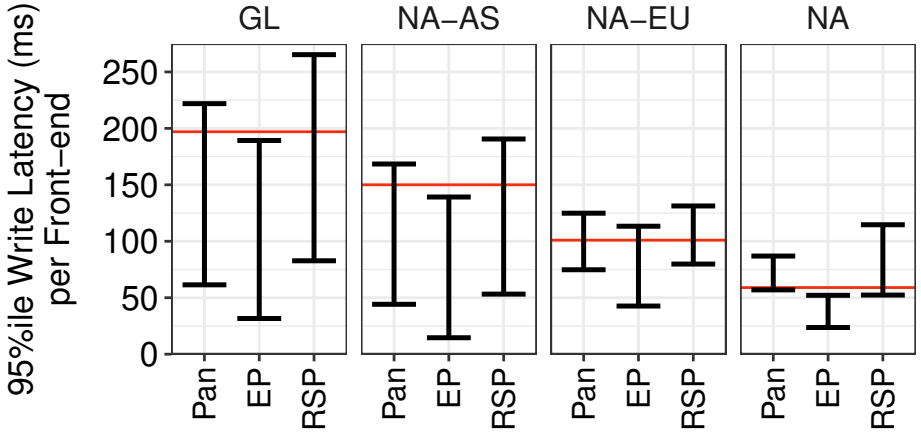
Informed by prior studies of production web service workloads [60, 43], we read and write objects between 1–100 KB in size. Unless stated otherwise, we use A1v2 (1 CPU, 2 GB memory) virtual machines and issue requests using YCSB [59]—a key-value store benchmark.

3.4.2.1 Microbenchmarks

Tail latency across front-ends. Figure 3.12 shows 95th percentile read and write latencies for each of the four access sets when running a low contention (zipfian coefficient $= 0.1$) workload with 1 KB values and a read:write ratio of 1. In all cases, when using PANDO, we observe that the slowest front-end performs similarly to the read latency SLO deemed feasible by ConfigManager.



(a) Read Latency when Write SLO = 300, 300, 150, and 100 ms for GL, NA-AS, NA-EU, and NA, respectively



(b) Write Latency when Read SLO = 200, 150, 125, and 75 ms for GL, NA-AS, NA-EU, and NA, respectively

Figure 3.12: Latency comparison with a low-contention workload under a storage SLO of 3× overhead. Red lines represent the lowest latency SLO that ConfigManager identifies as feasible with PANDO. With every approach, in each access set, we measure 95th %ile latency at every front-end and plot the min and max of this value across front-ends. Pan = Pando, EP = EPaxos, and RSP = RS-Paxos.

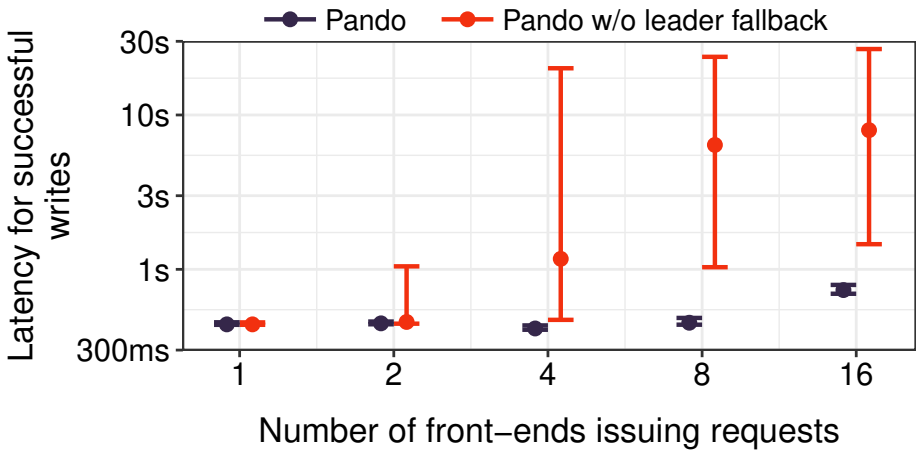


Figure 3.13: Write latency comparison under contention using a fully leaderless approach and the leader-based fallback (§3.2.5). 5th percentile, median, and 95th percentile across 1000 writes are shown. Note logscale on y-axis.

This confirms the low latency variance in the CosmosDB instance at each data center and on the network paths between them. While all approaches achieve sub-55 ms read latency in NA, only PANDO can provide sub-100 ms latency in all regions. In GL, NA-AS, and NA-EU, PANDO improves read latency for the slowest front-end by 39–62% compared to EPaxos. PANDO falls short of the write latency offered by EPaxos but comes close.

Latency under high conflict rates. Although our focus is on workloads with few write conflicts, we seek to bound performance degradation when conflicts occur. To evaluate this, we setup front-ends in 16 Azure data centers spread across five continents. We mimic conflicts by synchronizing a subset of front-ends (assuming low clock skew) to issue writes on the same key and version simultaneously. We show latency for successful conditional writes since other writes will learn the committed value and terminate shortly afterward.

Figure 3.13 shows that PANDO is effective at bounding latency for writes in the presence of conflicts. Without a leader-based fallback, writes in PANDO may need to be tried many times before succeeding, resulting in unbounded latency growth, e.g., with four concurrent writers, we observe more than 15 proposals for particular (key, version) pairs. In contrast, falling back to a leader ensures that a write succeeds within two write attempts.

Read and write throughput. While erasure coding can decrease bandwidth usage compared to replication, it requires additional computation in the form of coding/decoding and messaging overhead. We quantify the inflection point at which CPU overheads dominate by deploying PANDO in a single data center and measuring the achievable throughput with all data in memory. Each server, which stored 1 split or 1 replica, had two Xeon Silver 4114 processors and 192 GB of memory. All servers were connected over a 10 Gbps network with full bisection bandwidth. Across

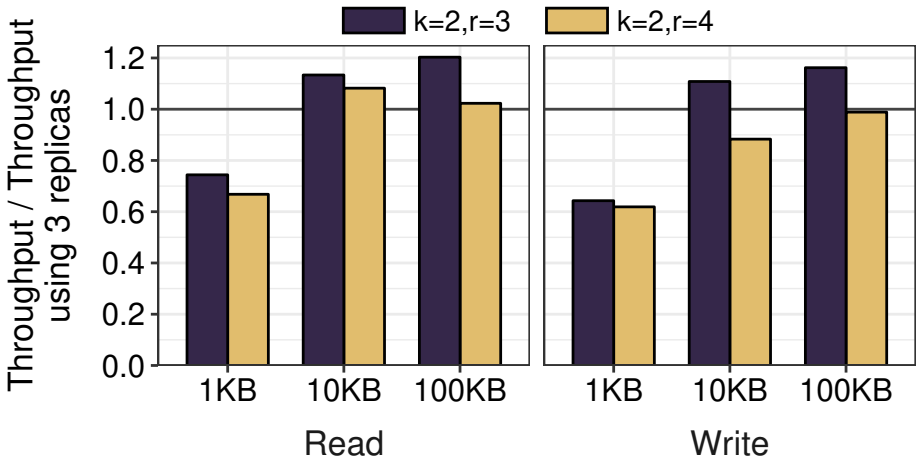


Figure 3.14: Per-machine throughput of different erasure coding configurations compared to using 3 replicas.

multiple value sizes, we measured the per-server throughput of filling the system with over 20 GB of data and reading it back.

Figure 3.14 compares the per-machine throughput achieved with 3 replicas to two erasure coding configurations, one with the same storage overhead and another with lower storage overhead. When objects are 10 KB or larger, we find that bandwidth is the primary bottleneck. Because it has identical bandwidth demands as replication, the $(k = 2, r = 4)$ configuration achieves similar read throughput and 0.9–1$\times$ the write throughput of replication for objects larger than 10 KB. Whereas, due to its lower bandwidth consumption, the $(k = 2, r = 3)$ configuration offers 1.1-1.2$\times$ the throughput of replication for 10 KB–100 KB sized objects. All configurations are CPU-bound with value sizes of 1 KB or smaller. Since replication requires exchanging fewer messages per request than erasure coding, it has lower CPU overhead and can thus achieve higher throughput.

3.4.2.2 Application Workload

Lastly, we evaluate the utility of PANDO on a geo-distributed deployment of GitLab [73], a software development application that provides source code management, issue tracking, and continuous integration.

Operations and setup. We evaluate the performance of two GitLab operations: listing issues targeting a development milestone (GetIssues) and (un-)protecting a branch from changes (ProtectBranch). GetIssues fetches a list of issues for the requested milestone and then fetches 20 issues in parallel to display on a page. ProtectBranch reads the current branch metadata then updates its protection status.

We deployed front-ends and storage backends in the NA-AS access set on A2v2 (2 CPU, 4 GB

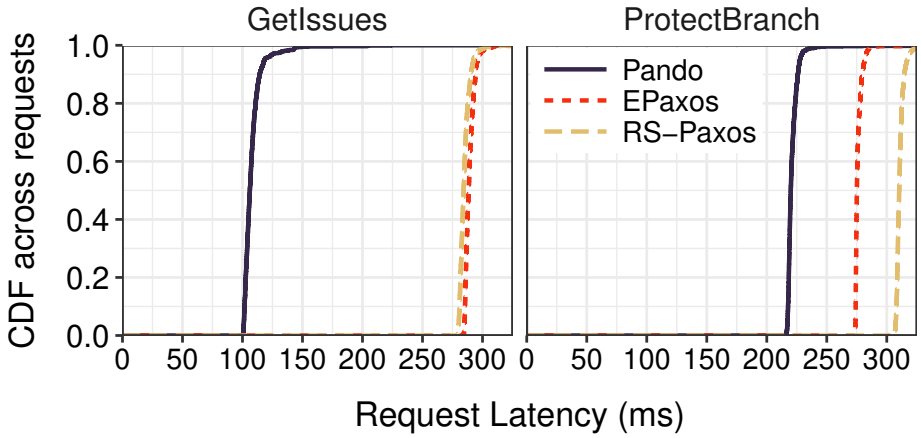


Figure 3.15: Latencies for GitLab requests in Central US.

memory) virtual machines, and preloaded the system with 100 projects, each with 20 branches, 10 milestones, and 100 issues. We used a $3\times$ bound on storage overhead and set the write latency SLO to 175 ms. Every front-end executed 1000 GetIssues and ProtectBranch requests in an open loop and selected items using a uniform key distribution.

Performance. Figure 3.15 shows the latency distribution observed for both operations by the front-end in Central US. PANDO reduces 95th percentile GetIssues latency by over 59% compared to both EPaxos and RS-Paxos. Because ProtectBranch consists of a write in addition to a read operation, it incurs higher latency compared to GetIssues, which consists solely of read operations. Despite this, PANDO is able to lower 95th percentile ProtectBranch latency by 19% over EPaxos and 28% over RS-Paxos.

3.5 Related work

Geo-distributed storage. While some prior geo-distributed storage systems [120, 116, 121, 163] weaken consistency semantics to minimize latencies and unavailability, PANDO follows others [60, 159, 182, 37] in serving the needs of applications that cannot make do with weak consistency. Compared to efforts focused solely on minimizing latency with any specific replication factor [132, 123, 112, 45], PANDO aims to also minimize the cost necessary to meet latency goals. Unlike systems [23, 176, 170, 139] which focus only on judiciously placing data to minimize cost, we also leverage erasure coding and rethink how to enable consensus on erasure-coded data.

Partial delegation in PANDO is akin to the chaining of RPCs [158] to eliminate wide-area delays. We show that combining this technique with the use of smaller quorums in Phase 1 of Paxos helps a two-round execution approximate the latencies achievable with one-round protocols in a geo-distributed setting.

Erasure-coded storage. Erasure coding has been widely used for protecting data from failures [174], most notably in RAID [142]. While PANDO leverages Reed–Solomon codes [150] for storage across data centers, other codes have been used to correct errors in DRAM [80], transmit data over networks [155], and efficiently reconstruct data in cloud storage [165, 95, 154]. New codes [124] can reduce cost over Reed–Solomon for operations that go beyond a basic GET/PUT interface. In contrast to the typical use of erasure coding for immutable and/or cold data [135, 79, 148, 136, 55], PANDO supports the storage of hot, mutable objects.

Previous protocols [25, 49] that support strong consistency with erasure-coded data provide only atomic register semantics or require two rounds of communication [134]. We show how to enable consensus on geo-distributed erasure-coded data without sacrificing latency. Some systems [54, 55] support strong consistency by erasure coding data but replicating metadata. We chose to not pursue this route to avoid the complexity of keeping the two in sync, as well as to minimize latency and metadata overhead.

Paxos variants. Many variants of Paxos [110] have been proposed over the years [134, 93, 113, 111], including several [132, 112, 123, 67] which enable low latency geo-distributed storage. Compared to Paxos variants that reduce the number of wide-area round trips [132, 123, 112, 67], PANDO lowers latency by reducing the magnitude of delay in each round trip.

Flexible Paxos [93] was the first to observe that Paxos only requires overlap between every Phase 1–Phase 2 quorum pair, and others [26, 138, 92] have leveraged this observation since. All of these approaches make *Phase 2 quorums smaller*, so as to improve throughput and common case latency in settings with high spatial locality. In PANDO, we instead *reduce the size of Phase 1 quorums and reuse these quorums for reads*, thereby enabling previously unachievable tradeoffs between read and write latency bounds in a workload-agnostic manner.

Compression. Data compression is often used to lower the cost of storing data [91, 160, 53] or transferring it over a network [69]. In contrast to erasure coding, the effectiveness of compression depends on both the choice of compression algorithm used as well as the input data [27]. Compression and erasure coding are complementary as data can be compressed and then erasure-coded or vice-versa.

3.6 Summary

Today, not only do minimizing cost and minimizing latency call for radically different designs of geo-distributed storage, but many latency versus cost tradeoffs are unachievable in practice. Our approach for enabling consensus, PANDO, addresses these shortcomings with a single configurable design. PANDO adapts the use of the Paxos consensus protocol in the wide-area to (1) allow readers to fetch data from a smaller set of data sites when erasure coding data and (2) mitigate the latency

of executing writes in two phases. We evaluated PANDO across a wide range of scenarios and showed that the latency versus cost tradeoffs achievable with PANDO are close to optimal.

CHAPTER 4

Conclusions

Every type of system faces bottlenecks that govern its performance. This dissertation shows how to offer significantly improved tradeoffs in both geo-distributed networking and storage, and supports my thesis that there is still room to improve over state-of-the-art designs.

In WAN networks, I show the importance of control delays on the predictability of the WAN and develop a WAN architecture that offers desirable predictability properties when controllers are slow or down. My main insight is to have the network treat opportunistic traffic differently from traffic that is within a tenant's guarantee. I then mitigate the negative performance impacts this behavior can introduce to existing applications.

I also show that no combination of geo-distributed storage systems today can always achieve an optimal read–write latency tradeoff given a constraint on storage cost, and I show how one can offer a near-optimal tradeoff. The key is to leverage erasure coding, rather than replication, to distribute data across data centers. By carefully adapting how to apply existing consensus protocols to erasure coded storage, I mitigate the performance issues with existing approaches and unlock near-optimal performance.

4.1 Future Work

A backwards-compatibility layer for HEYP. HEYP marks the portion of traffic that is above a tenant's approval with a lower QoS level than it intended. When a tenant's demand exceeds the available HIPRI and LOPRI capacity, the flows marked as LOPRI will experience more congestion than the HIPRI flows. Although many applications should perform reasonably under this behavior (see §2.3), some may respond poorly since it is different to what existing WANs provide. For these applications, it would be useful to have a DC controller that could provide fairness across all of a tenant's flows (both HIPRI and LOPRI) by rate limiting the HIPRI traffic and balancing the volume of traffic marked HIPRI and LOPRI. The main challenge would be in identifying the available LOPRI capacity – recall that LOPRI capacity can be 'stolen' by a flowgroup that is within

its approval. We did not pursue this approach for HEYP because rate limiting HIPRI can introduce unnecessary throttling that may introduce approval violations (see §2.4.2 and [147]). However, it may be useful as a compatibility mechanism for older applications.

Sharing WAN bandwidth according to policy without demand estimation. A WAN that uses HEYP must still employ BwE [105], or similar, to control how LOPRI capacity is shared between flowgroups. One limitation of BwE is its reliance on accurate demand estimations. If BwE substantially overestimates demand, it wastes network capacity. On the other hand, when BwE underestimates demand, it can take multiple global control periods before BwE converges to a fair allocation of bandwidth across flowgroups. Eliminating this dependence on demand estimation could potentially reduce the amount of wasted bandwidth. More importantly, it may be enough to ensure that BwE always restores fairness within one global control period.

Supporting richer manipulation of erasure-coded data. PANDO supports a simple key-value store interface – the bare minimum API that is useful to users. The reason for this is that PANDO erasure-codes individual objects, so only clients, not data sites, can see the entire object.

One step towards providing a richer interface would be efficient support for updates. Currently, updating part of a value in PANDO requires updating the data at all data sites. However, by leveraging a code that supports efficient updates [124], it should be possible to support more fine-grained operations, and perhaps, change the granularity at which data is erasure coded.

Failure domains in geo-distributed storage systems. Geo-distributed storage systems can choose to have their failure domain be that of a single machine [60, 162] or an entire data center [121, 177, 55]. The former approach requires lower overhead since there is no redundancy within the data center. However, the latter offers faster failure recovery, since health checks can identify failures more quickly within a data center than across data centers. A study that compares high-quality implementations of both approaches would be useful to identify under what conditions each is appropriate.

4.2 Final Remarks

The infrastructure available to help build geo-distributed applications is still in its infancy. Most of the services that are currently offered by cloud providers focus on large, single data center applications. Naively deploying these services across data centers is unlikely to provide optimal tradeoffs to users, especially when we consider the improvements my work has shown over existing WAN-optimized systems. Instead, geo-distributed applications need systems that are optimized specifically for the WAN, and that offer a variety of tradeoffs to support different use cases. This dissertation shows how to do this for problems in both networking and storage systems, and I hope the results inform future system designs.

APPENDIX A

Details of simulation environment used to evaluate HEYP

A.1 Discrete-event simulation control logic

Our simulation in §2.4.1 diverges from the design described in §2.3 in two ways. First, route and admission computation are performed by two separate controllers (Algorithms A.1 and A.2) similar to B4 [98, 105]). Second, because the the network traces lack per-task data, the simulated DC controller can only partition traffic based on usage (Algorithm A.3).

Inputs:
Approvals and demands per flowgroup
Topology annotated with link capacities
Route allocation function (see Algorithm A.4)
Outputs (per flowgroup): HIPRI and LOPRI routes

1. HIPRI routes, HIPRI admissions $\leftarrow$ $AllocateRoutes$(approvals, approvals, link capacities)
2. Compute unused link capacity by deducting any link capacity consumed by the volume of each flowgroup's *demand that is under the HIPRI admission*
3. HIPRI routes, _ $\leftarrow$ $AllocateRoutes$(demands - HIPRI admissions, approvals - HIPRI admissions, link capacities from Step 2)

Algorithm A.1: Allocating routes separately from admissions while accounting for any over-subscription caused by failures.

A.2 Route allocation algorithm used in §2.4.1

Algorithm A.4 provides a detailed description of the routing algorithm used by all approaches in §2.4.1. It is similar to the greedy algorithm used by B4 [98] and prioritizes satisfying any within-approval demands before above-approval ones.

Inputs:

Approvals and demands per flowgroup
HIPRI and LOPRI routes per flowgroup
Topology annotated with link capacities

Outputs (per flowgroup): HIPRI and LOPRI admissions

---

1. Set the HIPRI admissions to a max-min fair allocation of bandwidth to satisfy *approvals* using the HIPRI routes
2. Compute unused link capacity by deducting any link capacity consumed by the volume of each flowgroup's *demand that is under the HIPRI admission*
3. Set the LOPRI admissions to a max-min fair allocation of bandwidth to satisfy any residual demand using the LOPRI routes

Algorithm A.2: Allocating admissions separately from routes while accounting for any over-subscription caused by failures.

Inputs:

Usage and admission per (flowgroup, QoS)
Demand per flowgroup
Current fraction of demand marked as HIPRI per flowgroup

Outputs (per flowgroup):

New fraction of demand to mark as HIPRI

---

foreach *flowgroup* **do**
  $t \leftarrow \min(\text{demand}, \text{HIPRI + LOPRI admissions})$
  Set new HIPRI fraction to $\min(1, \text{HIPRI limit}/t)$
end

Algorithm A.3: Splitting traffic into QoS levels.

Configuration parameters:

Maximum number of paths (i.e. path budget) per flowgroup

Inputs: Approvals and demands per flowgroup

Topology annotated with link capacities

Outputs (per flowgroup): Routes and admissions

// Initialization

1 $PathAdmissions \leftarrow \{\}$

// Start by satisfying within-approval demands

2 For all flowgroups f, $D_f \leftarrow \min(\text{demand}_f, \text{approvals}_f)$

// Main path allocation loop

3 **while** *some flowgroup has positive demand and some link has positive capacity* **do**

4 $\quad CurPaths \leftarrow \{\}$

5 $\quad$ **foreach** *flowgroup* f *with* $D_f > 0$ **do**

6 $\quad\quad p \leftarrow$ next shortest path that avoids links with no capacity

7 $\quad\quad$ **if** *no such* p *exists or if adding* p *to* $Routes_f$ *exceeds the path budget* **then**

8 $\quad\quad\quad D_f \leftarrow 0$

9 $\quad\quad$ **else**

10 $\quad\quad\quad CurPaths_f \leftarrow p$

11 $\quad$ **end**

12 $\quad$ Compute a max-min fair allocation of link capacity to satisfy D using $CurPaths$

13 $\quad$ Add allocations to admissions and subtract from D_f

14 $\quad PathAdmissions_{f,p} \leftarrow PathAdmissions_{f,p} +$ admission

15 **end**

// We have satisfied any within-approval demands (if possible), try to satisfy above-approval demands

16 Set $D_f \leftarrow \text{demand}_f - \sum_p PathAdmissions_{f,p}$

17 Repeat loop on Lines 3–15

18 **foreach** *flowgroup* f **do**

19 $\quad$ **if** $demand_f = 0$ **then**

20 $\quad\quad$ Use the shortest route and set admission to 0

21 $\quad$ **else**

22 $\quad\quad$ Use routes in $PathAdmissions_f$ with each getting a share of traffic proportionate to the path admission

23 $\quad\quad$ Set admission to $\sum_p PathAdmissions_{f,p}$

24 **end**

Algorithm A.4: Route computation algorithm.

APPENDIX B

The Pando write protocol: specification and proof of correctness

In this section, we focus on how PANDO achieves consensus on a *single value* and prove that it matches the guarantees provided by Paxos. Other functionality is layered on top of this base as follows:

- **Mutating values.** As with Multi-Paxos, we build a distributed log of values and run PANDO on each entry of the log. We only ever attempt a write for version i if we know that $i - 1$ has already been chosen. This invariant ensures that the log is contiguous, and that all but possibly the latest version have been decided.

- **Partial delegation of writes.** One of the key optimizations used in PANDO is to execute Phase 1 and Phase 2 on different nodes (§3.2.3.2). We achieve this without sacrificing fault tolerance as follows. Each proposer is assigned an id (used for Lamport clocks), but we additionally assign a proposer id to each (proposer, delegate) pair. When executing a write using partial delegation, we simply direct responses accordingly, and have the proposer inform the delegate about which value to propose (unless one was recovered, in which case the delegate has to inform the proposer about the change). In case the delegate fails, a proposer can always choose to execute a write operation normally, and because it uses a different proposer id in this case, it will look as though the write from the proposer and the write from the (proposer, delegate) pair are writes from two separate nodes. We already prove (§B.1) that PANDO maintains consistency in this case.

- **One round reads.** As with other consensus protocols, we support (common-case) one-round reads by adding a third, asynchronous phase to writes that broadcasts which value was chosen and caches this information at each acceptor. Upon executing a read at a Phase 1a quorum, we check to see if any acceptor knows whether a value has already been chosen. If we find such a value, we try to reconstruct it and fall back on the larger Phase 1b quorum

$A.ppn$	Promised proposal no. stored at acceptor A
$A.apn$	Accepted proposal no. stored at acceptor A
$A.vid$	Accepted value id stored at acceptor A
$A.vlen$	Accepted value length stored at acceptor A
$A.vsplit$	Accepted value split stored at acceptor A
vid_v	Unique id for v, typically a hash or random number
$vlen_v$	Length of v (to remove padding)
$\text{Split}(v, A)$	(Computed on proposers) The erasure-coded split associated with acceptor A

Figure B.1: Summary of notation.

in case there are not enough splits present in the Phase 1a quorum. Otherwise, we follow the write path, but propose a value only if we were able to recover one (else none have been chosen). We maintain linearizability with this approach because the task of resolving uncertainty is done via the write path.

- **Fallback to leader.** In PANDO, front-ends directly execute writes unless a conflict is observed, in which case they defer the request to a leader (§3.2.5). From the perspective of the consensus protocol, the leader is just another proposer, so no consistency issues may arise even if multiple leaders exist. However, PANDO prevents non-leader front-ends from attempting writes more than once which can lead to unavailability if the leader fails. It is up to the leader election mechanism to quickly elect a new leader when the the current one fails.

PANDO's consistency and liveness properties rely on certain quorum constraints being met. We describe the constraints below under the assumption that data is partitioned into k splits (Constraint 3 needed only if Phase 1a quorums are used for reads):

1. The intersection of any Phase 1a and Phase 2 quorums contains at least 1 split.
2. The intersection of any Phase 1b and Phase 2 quorums contains at least k splits.
3. A Phase 1a quorum must contain at least k splits.
4. After f nodes fail, at least one Phase 1b and Phase 2 quorum must consist of nodes that are available.

Below is pseudocode for the PANDO write protocol.

Phase 1 (Prepare-Promise)

Proposer P initiates a write for value v:

1. Select a unique proposal number p (typically done using Lamport clocks).
2. Broadcast $\mathrm{Prepare}(p)$ messages to all acceptors.

Acceptor A, upon receiving $\mathrm{Prepare}(p)$ message from Proposer P:

3. If $p > A.ppn$ then set $A.ppn \leftarrow p$ and reply $\mathrm{Promise}(A.apn, A.vid, A.vlen, A.vsplit)$.
4. Else reply NACK.

Proposer P, upon receiving $\mathrm{Promise}$ messages from a Phase 1a quorum:

5. If the values in all $\mathrm{Promise}$ responses are NULL, then skip to Phase 2 with $v' \leftarrow v$.

Proposer P, upon receiving $\mathrm{Promise}$ messages from a Phase 1b quorum:

6. Iterate over all $\mathrm{Promise}$ responses sorted in decreasing order of their apn.
   (a) If there are at least k splits for value w associated with apn, recover the value w (using the associated $vlen$ and $vsplit$s) and continue to Phase 2 with $v' \leftarrow w$.
7. If no value was recovered, continue to Phase 2 with $v' \leftarrow v$.

Phase 2 (Propose-Accept)

Proposer P, initiating Phase 2 to write value v' with proposal number p:

8. If no value was recovered in Phase 1, set $vid_{v'} = hash(v)$ (or some other unique number, see Figure B.1). If a value was recovered, use the existing $vid_{v'}$.
9. Broadcast $\mathrm{Propose}(p, vid_{v'}, vlen_{v'}, \mathrm{Split}(v', A))$ messages to all acceptors.

Acceptor A, upon receiving $\mathrm{Propose}(p, vid, vlen, vsplit)$ from a Proposer P:

10. If $p < A.ppn$ reply NACK
11. $A.ppn \leftarrow p$
12. $A.apn \leftarrow p$
13. $A.vid \leftarrow vid$
14. $A.vlen \leftarrow vlen$
15. $A.vsplit \leftarrow vsplit$
16. Reply $\mathrm{Accept}(p)$

Proposer P, upon receiving $\mathrm{Accept}(p)$ messages from a Phase 2 quorum:

17. P now knows that v' was chosen, and can check whether the chosen value v' differs from the initial value v or not.

B.1 Proof of correctness

Definitions. We let $\mathcal{A}$ refer to the set of all acceptors and use $\mathcal{Q}_a$, $\mathcal{Q}_b$, and $\mathcal{Q}_2$ refer to the sets of Phase 1a, Phase 1b, and Phase 2 quorums, respectively. Using this notation, we restate our quorum assumptions:

$$Q \subseteq \mathcal{A} \qquad \forall Q \in \mathcal{Q}_a \cup \mathcal{Q}_b \cup \mathcal{Q}_2 \tag{1}$$

$$|Q_a \cap Q_2| \geq 1 \qquad \forall Q_a \in \mathcal{Q}_a, Q_2 \in \mathcal{Q}_2 \tag{2}$$

$$|Q_b \cap Q_2| \geq k \qquad \forall Q_b \in \mathcal{Q}_b, Q_2 \in \mathcal{Q}_2 \tag{3}$$

Definition 1. *A value is **chosen** if there exists a Phase 2 quorum of acceptors that all agree on the identity of the value and store splits corresponding to that value.*

We now show that the PANDO write protocol provides the same guarantees as Paxos:

- **Nontriviality.** Any chosen value must have been proposed by a proposer.
- **Liveness.** A value will eventually be chosen provided that RPCs complete before timing out and all acceptors in at least one Phase 1b and Phase 2 quorum are available.
- **Consistency.** At most one value can be chosen.
- **Stability.** Once a value is chosen, no other value may be chosen.

Theorem 1. *(Nontriviality)* PANDO *will only choose values that have been proposed.*

Proof. By definition, a value can only be chosen if it is present at a Phase 2 quorum of acceptors. Values are only stored at acceptors in response to Propose messages initiated by proposers. □

Theorem 2. *(Liveness)* PANDO *will choose a a value provided that RPCs complete before timing out and all acceptors in at least one Phase 1b and Phase 2 quorum are available.*

Proof. Let t refer to the (maximum) network and execution latency for an RPC. Since PANDO has two rounds of execution, a write can complete within $2t$ as long as a requested is uncontended. If all proposers retry RPCs using randomized exponential backoff, a time window of length $\geq 2t$ will eventually open where only a Proposer P is executing. Since no other proposer is sending any RPCs during this time, both Phase 1 and Phase 2 will succeed for Proposer P. □

Following the precedence of [93], we will show that PANDO provides both consistency and stability by proving that it provides a stronger guarantee.

Lemma 1. *If a value v is chosen with proposal number p, then for any proposal with proposal number $p' > p$ and value v', $v' = v$.*

Proof. Recall that PANDO proposers use globally unique proposal numbers (Line 1); this makes it impossible for two different proposals to share a proposal number p. Therefore, if two proposals are both chosen, they must have different proposal numbers. If $v' = v$ then we trivially have the desired property. Therefore, assume $v' \neq v$.

Without loss of generality, we will consider the smallest p' such that $p' > p$ and $v' \neq v$ (*minimality assumption*). We will show that this case always results in a contradiction: either the Prepare messages for p' will fail (and thus no Propose messages will ever be sent) or the proposer will adopt and re-propose value v.

Let $Q_{2,p}$ be the Phase 2 quorum used for proposal number p, and $Q_{a,p'}$ be the Phase 1a quorum used for p'. By Quorum Property 2, we know that $|Q_{2,p} \cap Q_{a,p'}|$ is non-empty. We will now look at the possible ordering of events at each acceptor A in the intersection of these two quorums ($Q_{2,p}$ and $Q_{a,p'}$):

- Case 1: A receives $\mathrm{Prepare}(p')$ before $\mathrm{Propose}(p, \ldots)$.

  The highest proposal number at A would be $p' > p$ by the time $\mathrm{Propose}(p, \ldots)$ was processed, and so A would reject $\mathrm{Propose}(p, \ldots)$. However, we know that this is not the case since $A \in Q_{2,p}$, so this is a contradiction.

- Case 2: A receives $\mathrm{Propose}(p, \ldots)$ before $\mathrm{Prepare}(p')$.

  The last promised proposal number at A is q such that $p \leq q < p'$ ($q > p'$ would be a contradiction since $\mathrm{Prepare}(p')$ would fail even though $A \in Q_{a,p'}$). By our minimality assumption, we know that all proposals z such that $p \leq z < p'$ fail or re-propose v. Therefore, the acceptor A responds with $\mathrm{Promise}(q, vid_v, \ldots)$.

At this point, the proposer has received at least one $\mathrm{Promise}$ message with a non-empty value. Therefore, it does not take the Phase 1 fast path and waits until it has heard from a Phase 1b quorum (denoted $Q_{b,p'}$). Using the same logic as above, the proposer for p' will receive a minimum of k $\mathrm{Promise}$ messages each referencing value v since there are k acceptors in $Q_{b,p'} \cap Q_{2,p}$ (Quorum Property 3). Since the proposer has a minimum of k responses for v, it can reconstruct value v. Let q denote the highest proposal number among all k responses.

Besides those in $Q_{b,p'} \cap Q_{2,p}$, other acceptors in $Q_{b,p'}$ may return values that differ from v. We consider the proposal number q' for each of these accepted values:

- Case 1: $q' < q$. The proposer for p' will ignore the value for q' since it uses the highest proposal number for which it has k splits.

- Case 2: $p' < q'$. Not possible since $\mathrm{Prepare}(p')$ would have failed.
- Case 3: $p < q' < p'$. This implies that a $\mathrm{Propose}(q', v'')$ was issued where $v'' \neq v$. This violates our minimality assumption.

Therefore, the proposer will adopt value v since it can reconstruct it (the proposer has k splits from the acceptors in $Q_{b,p'} \cap Q_{2,p}$ alone) and the highest returned proposal number references it. This contradicts our assumption that $v' \neq v$. □

Theorem 3. *(Consistency)* PANDO *will choose at most one value.*

Proof. Assume that two different proposals with proposal numbers p and q are chosen. Since proposers use globally unique proposal numbers, $p \neq q$. This implies that one of the proposal numbers is greater than the other, assume that $q > p$. By Lemma 1, the two proposals write the same value. □

Theorem 4. *(Stability) Once a value is chosen by* PANDO*, no other value may be chosen.*

Proof. The proposal numbers used for any two chosen proposals will not be equal. Thus, with the additional assumption that acceptors store their state in durable storage, this follows immediately from Lemma 1. □

APPENDIX C

TLA+ specification for Pando reads and writes

In addition to our proof of correctness for PANDO's write path, we have model checked PANDO's correctness using TLA+ [109]. The purpose of this exercise was to *mechanically verify* PANDO's safety guarantees under a number of scenarios.

We checked the following invariants: consistency and stability for writes, that any value marked chosen at an acceptor was indeed chosen, and that successful reads only ever returned chosen values. The configurations modeled used 2–3 proposers (and readers) that could write (read) 2–3 values to (from) 4–6 acceptors when splitting the data into 2–4 splits. We set up 2–3 quorums of each type (Phase 1a, Phase 1b, and Phase 2).

The TLA+ model checker considers all possible histories including those with message reordering and arbitrary (or infinite) delay in delivering messages. When run on the specification for PANDO (below) and the configurations listed earlier, no invariant violations were found.

MODULE $Pando$

EXTENDS $Integers$, TLC, $FiniteSets$

CONSTANTS $Acceptors$, $Ballots$, $Values$,
$Quorum1a$, $Quorum1b$, $Quorum2$, K

ASSUME $QuorumAssumption \triangleq$
$\land\ Quorum1a \subseteq \text{SUBSET } Acceptors$
$\land\ Quorum1b \subseteq \text{SUBSET } Acceptors$
$\land\ Quorum2 \subseteq \text{SUBSET } Acceptors$
Overlap of 1
$\land \forall\, QA \in Quorum1a :$
$\forall\, Q2 \in Quorum2 :$
$Cardinality(QA \cap Q2) \geq 1$
Overlap of K
$\land \forall\, QB \in Quorum1b :$

$$\forall\, Q2 \in Quorum2 :$$
$$\quad Cardinality(QB \cap Q2) \geq K$$

VARIABLES $msgs$, The set of messages that have been sent

$maxPBal$, $maxPBal[a]$ is the highest promised ballot (proposal number) at acceptor a

$maxABal$, $maxABal[a]$ is the highest accepted ballot (proposal number) at acceptor a

$maxVal$, $maxVal[a]$ is the value for $maxABal[a]$ at acceptor a

$chosen$, $chosen[a]$ is the value that acceptor a heard was chosen (or else is $None$)

$readLog$ $readLog[b]$ is the value that was read during ballot b

$$vars \triangleq \langle msgs,\ maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$$
$$None \triangleq \text{CHOOSE } v : v \notin Values$$

Type invariants.

$$\begin{aligned} Messages \triangleq\ & [type : \{\text{“prepare”}\},\ bal : Ballots] \\ \cup\ & [type : \{\text{“promise”}\},\ bal : Ballots,\ maxABal : Ballots \cup \{-1\}, \\ & maxVal : Values \cup \{None\},\ acc : Acceptors, \\ & chosen\ : Values \cup \{None\}] \\ \cup\ & [type : \{\text{“propose”}\},\ bal : Ballots,\ val : Values \cup \{None\}, \\ & op : \{\text{“R”},\ \text{“W”}\}] \\ \cup\ & [type : \{\text{“accept”}\},\ bal : Ballots,\ val : Values,\ acc : Acceptors, \\ & op : \{\text{“R”},\ \text{“W”}\}] \\ \cup\ & [type : \{\text{“learn”}\},\ bal : Ballots,\ val : Values] \end{aligned}$$

$$\begin{aligned} TypeOK \triangleq\ & \wedge msgs \in \text{SUBSET } Messages \\ & \wedge maxABal \in [Acceptors \to Ballots \cup \{-1\}] \\ & \wedge maxPBal \in [Acceptors \to Ballots \cup \{-1\}] \\ & \wedge maxVal \in [Acceptors \to Values \cup \{None\}] \\ & \wedge chosen \in [Acceptors \to Values \cup \{None\}] \\ & \wedge readLog \in [Ballots \to Values \cup \{None\}] \\ & \wedge \forall\, a \in Acceptors : maxPBal[a] \geq maxABal[a] \end{aligned}$$

Initial state.

$$\begin{aligned} Init \triangleq\ & \wedge msgs = \{\} \\ & \wedge maxPBal = [a \in Acceptors \mapsto -1] \\ & \wedge maxABal = [a \in Acceptors \mapsto -1] \\ & \wedge maxVal = [a \in Acceptors \mapsto None] \\ & \wedge chosen = [a \in Acceptors \mapsto None] \end{aligned}$$

$\land\ readLog \quad = [b \in Ballots \mapsto None]$

Send message m.

$Send(m) \triangleq msgs' = msgs \cup \{m\}$

Prepare: The proposer chooses a ballot id and broadcasts prepare requests to all acceptors.
All writes start here.

$$\begin{aligned} Prepare(b) \triangleq\ & \land \neg\exists\, m \in msgs : (m.type = \text{“prepare”}) \land (m.bal = b) \\ & \land Send([type \mapsto \text{“prepare”},\ bal \mapsto b]) \\ & \land \text{UNCHANGED } \langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle \end{aligned}$$

Promise: If an acceptor receives a prepare request with ballot id greater than that of any prepare request which it has already responded to, then it responds to the request with a promise. The promise reply contains the proposal (if any) with the highest ballot id that it has accepted.

$$\begin{aligned} & Promise(a) \triangleq \\ & \quad \exists\, m \in msgs : \\ & \qquad \land\ m.type = \text{“prepare”} \\ & \qquad \land\ m.bal > maxPBal[a] \\ & \qquad \land\ Send([type \mapsto \text{“promise”},\ acc \mapsto a,\ bal \mapsto m.bal, \\ & \qquad\qquad maxABal \mapsto maxABal[a],\ maxVal \mapsto maxVal[a], \\ & \qquad\qquad chosen \mapsto chosen[a]]) \\ & \qquad \land\ maxPBal' = [maxPBal \text{ EXCEPT } ![a] = m.bal] \\ & \qquad \land \text{UNCHANGED } \langle maxABal,\ maxVal,\ chosen,\ readLog\rangle \end{aligned}$$

Propose (fast path): The proposer waits until it collects promises from a Phase 1a quorum of acceptors. If no previous value is found, then the proposer can skip to Phase 2 with its own value.

$$\begin{aligned} & ProposeA(b) \triangleq \\ & \quad \land \neg\exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b) \\ & \quad \land \exists\, v \in Values\ : \\ & \qquad \land \exists\, Q \in Quorum1a : \\ & \qquad\quad \text{LET } Q1Msgs \triangleq \{m \in msgs :\ \land\ m.type = \text{“promise”} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \land\ m.bal\ \ = b \\ & \qquad\qquad\qquad\qquad\qquad\qquad \land\ m.acc\ \in Q\} \\ & \qquad\quad \text{IN} \\ & \qquad\qquad \text{Check for promises from all acceptors in Q} \\ & \qquad\qquad \land \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a \\ & \qquad\qquad \text{Make sure no previous vals have been returned in promises} \\ & \qquad\qquad \land \forall\, m \in Q1Msgs : m.maxABal = -1 \\ & \quad \land Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ val \mapsto v,\ op \mapsto \text{“W”}]) \end{aligned}$$

$\land$ UNCHANGED $\langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$

Propose (slow path): The proposer waits for promises from a Phase 1b quorum of acceptors. If no value is found accepted, then the proposer can pick its own value for the next phase. If any accepted coded split is found in one of the promises, the proposer detects whether there are at least K splits (for the particular value) in these promises. Next, the proposer picks up the recoverable value with the highest ballot, and uses it for next phase.

$ProposeB(b) \triangleq$

$\land \neg\exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b)$

$\land \exists\, Q \in Quorum1b :$

LET $Q1Msgs \triangleq \{m \in msgs : \land m.type = \text{“promise”}$

$\land m.bal = b$

$\land m.acc \in Q\}$

$Q1Vals \triangleq [v \in Values \cup \{None\} \mapsto$

$\{m \in Q1Msgs : m.maxVal = v\}]$

IN

Check that all acceptors from Q responded

$\land \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a$

$\land \exists\, v \in Values :$

$\land$ No recoverable value, use anything

$\lor \forall\, vv \in Values : Cardinality(Q1Vals[vv]) < K$

Check if v is recoverable and of highest ballot

$\lor$ Use previous value if K splits exist

$\land Cardinality(Q1Vals[v]) \geq K$

$\land \exists\, m \in Q1Vals[v] :$

Ensure no other recoverable value has a higher ballot

$\land \forall\, mm \in Q1Msgs :$

$\lor m.bal \geq mm.bal$

$\lor Cardinality(Q1Vals[mm.maxVal]) < K$

$\land Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ val \mapsto v,\ op \mapsto \text{“W”}])$

$\land$ UNCHANGED $\langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$

Phase 2: If an acceptor receives an accept request with ballot i, it accepts the proposal unless it has already responded to a prepare request having a ballot greater than it does.

$Accept(a) \triangleq$

$\land \exists\, m \in msgs :$

$\land m.type = \text{“propose”}$

$\land m.bal \geq maxPBal[a]$

$\land maxABal' = [maxABal \text{ EXCEPT } ![a] = m.bal]$

$\land\ maxPBal' = [maxPBal \text{ EXCEPT } ![a] = m.bal]$
$\land\ maxVal' \ = [maxVal \ \text{ EXCEPT } ![a] \ = m.val]$
$\land\ Send([type \mapsto \text{“accept”},\ bal \mapsto m.bal,\ acc \mapsto a,\ val \mapsto m.val,$
$\qquad op \mapsto m.op])$
$\land$ UNCHANGED $\langle chosen,\ readLog\rangle$

ProposerEnd: If the proposer receives acknowledgements from a Phase 2 quorum, then it knows that the value was chosen and broadcasts this.

$ProposerEnd(b) \ \triangleq$
$\land \exists\, v \in Values :$
$\land \exists\, Q \in Quorum2 :$
LET $Q2msgs \ \triangleq \ \{m \in msgs : \ \land\ m.type = \text{“accept”}$
$\land\ m.bal \ = b$
$\land\ m.val \ = v$
$\land\ m.acc \in Q\}$
IN
Check for accept messages from all members of Q
$\land \forall\, a \in Q : \exists\, m \in Q2msgs : m.acc = a$
If this was in response to a read, log the result
$\land$ Read: log the result
$\lor\ \land \exists\, m \in Q2msgs : m.op = \text{“R”}$
$\land\ readLog' = [readLog \text{ EXCEPT } ![b] = v]$
Write: don't log the result
$\lor\ (\forall\, m \in Q2msgs : m.op = \text{“W”} \land$ UNCHANGED $\langle readLog\rangle)$
$\land\ Send([type \mapsto \text{“learn”},\ bal \mapsto b,\ val \mapsto v])$
$\land$ UNCHANGED $\langle maxABal,\ maxPBal,\ maxVal,\ chosen\rangle$

Learn: A proposer has announced that value v is chosen.

$Learn(a) \ \triangleq$
$\land\ \ \exists\, m \in msgs :$
$\land\ m.type = \text{“learn”}$
Process accept before learn, needed for ReadInv, not the protocol
$\land\ maxABal[a] \geq m.bal$
$\land\ chosen' = [chosen \text{ EXCEPT } ![a] = m.val]$
$\land$ UNCHANGED $\langle msgs,\ maxPBal,\ maxABal,\ maxVal,\ readLog\rangle$

Count how many splits of v we have received.

$CountSplitsOf(resps,\ v) \ \triangleq \ Cardinality(\{m \in resps : m.maxVal = v\})$

FastRead: Check if any value returned from a Phase 1a quorum was chosen. If we have enough splits to reconstruct that value, then return immediately. If not, wait for Phase 1b quorum. If we have a value that was marked chosen, return. Otherwise, perform a write-back.

$$
\begin{array}{l}
FastRead(b) \triangleq \\
\quad \land \neg \exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b) \\
\quad \land \\
\qquad \text{Fastest path: Phase 1a quorum has } k \text{ splits and the value is chosen} \\
\qquad \lor \land \exists\, Q \in Quorum1a : \\
\qquad\quad \text{LET } RMsgs \triangleq \{m \in msgs : \land m.type = \text{“promise”} \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.bal \;\; = b \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.acc \in Q\} \\
\qquad\quad \text{IN} \quad \text{Check that all acceptors from } Q \text{ responded} \\
\qquad\qquad \land \forall\, a \in Q : \exists\, m \in RMsgs : m.acc = a \\
\qquad\qquad \text{Check that we have } k \text{ splits of a chosen value} \\
\qquad\qquad \land \exists\, m \in RMsgs : \\
\qquad\qquad\quad \land m.chosen \neq None \\
\qquad\qquad\quad \land CountSplitsOf(RMsgs,\, m.chosen) \geq K \\
\qquad\qquad\quad \land readLog' = [readLog \text{ EXCEPT } ![b] = m.chosen] \\
\qquad \land \text{UNCHANGED } \langle msgs,\, maxPBal,\, maxABal,\, maxVal,\, chosen \rangle \\
\qquad \text{Fast path: Phase 1b quorum has } k \text{ splits and the value is chosen} \\
\qquad \lor \land \exists\, Q \in Quorum1b : \\
\qquad\quad \text{LET } RMsgs \triangleq \{m \in msgs : \land m.type = \text{“promise”} \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.bal \;\; = b \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.acc \in Q\} \\
\qquad\quad \text{IN} \quad \text{Check that all acceptors from } Q \text{ responded} \\
\qquad\qquad \land \forall\, a \in Q : \exists\, m \in RMsgs : m.acc = a \\
\qquad\qquad \text{Check that we have } k \text{ splits of a chosen value} \\
\qquad\qquad \land \exists\, m \in RMsgs : \\
\qquad\qquad\quad \land m.chosen \neq None \\
\qquad\qquad\quad \land CountSplitsOf(RMsgs,\, m.chosen) \geq K \\
\qquad\qquad\quad \land readLog' = [readLog \text{ EXCEPT } ![b] = m.chosen] \\
\qquad \land \text{UNCHANGED } \langle msgs,\, maxPBal,\, maxABal,\, maxVal,\, chosen \rangle \\
\qquad \text{Slow path: Phase 1b recovery and write back} \\
\qquad \lor \land \exists\, Q \in Quorum1b : \\
\qquad\quad \text{LET } Q1Msgs \triangleq \{m \in msgs : \land m.type = \text{“promise”} \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.bal \;\; = b
\end{array}
$$

$$\wedge\ m.acc\ \in Q\}$$

$$Q1Vals\ \triangleq\ [v\ \in Values \cup \{None\} \mapsto \{m \in Q1Msgs : m.maxVal = v\}]$$

IN

Check that all acceptors from Q responded

$\wedge \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a$

$\wedge \exists\, v \in Values :$

Check if v is recoverable and of highest ballot

Use previous value if K splits exist

$\wedge\ Cardinality(Q1Vals[v]) \geq K$

$\wedge \exists\, m \in Q1Vals[v] :$

Ensure no other recoverable value has a higher ballot

$\wedge \forall\, mm \in Q1Msgs :$

$\vee\ m.bal \geq mm.bal$

$\vee\ Cardinality(Q1Vals[mm.maxVal]) < K$

readLog will be updated in ProposerEnd

$\wedge\ Send([type \mapsto$ "propose", $bal \mapsto b,\ val \mapsto v,$

$op \mapsto$ "R"$])$

$\wedge$ UNCHANGED $\langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$

No value recovered: Return *None*

$\vee\ \wedge\ readLog' = [readLog$ EXCEPT $![b] = None]$

$\wedge$ UNCHANGED $\langle msgs,\ maxPBal,\ maxABal,\ maxVal,\ chosen\rangle$

Next state.

$$Next\ \triangleq\ \vee \exists\, b \in Ballots\ :\ \vee Prepare(b) \vee ProposeA(b) \vee ProposeB(b) \vee ProposerEnd(b) \vee FastRead(b)$$

$$\vee \exists\, a \in Acceptors\ :\ Promise(a) \vee Accept(a) \vee Learn(a)$$

$$Spec\ \triangleq\ Init \wedge \Box[Next]_{vars}$$

Invariant helpers.

$AllChosenWereAcceptedByPhase2\ \triangleq$

$\forall\, a \in Acceptors :$

$\vee\ chosen[a] = None$

$$
\begin{array}{l}
\quad \lor \exists\, Q \in Quorum2 : \\
\qquad \forall\, a2 \in Q : \\
\qquad\quad \exists\, m \in msgs : \land\, m.type = \text{“accept”} \\
\qquad\qquad\qquad\qquad \land\, m.acc \;\; = a2 \\
\qquad\qquad\qquad\qquad \land\, m.val \;\; = chosen[a]
\end{array}
$$

$$
\begin{array}{l}
OnlyOneChosen \triangleq \\
\quad \forall\, a,\, aa \in Acceptors : \\
\qquad (chosen[a] \neq None \land chosen[aa] \neq None) \implies (chosen[a] = chosen[aa])
\end{array}
$$

$$
\begin{array}{l}
VotedForIn(a,\, v,\, b) \triangleq \exists\, m \in msgs : \land\, m.type = \text{“accept”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.val \;\; = v \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.bal \;\; = b \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc \;\; = a
\end{array}
$$

$$
\begin{array}{l}
ProposedValue(v,\, b) \triangleq \exists\, m \in msgs : \land\, m.type = \text{“propose”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.val \;\; = v \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.bal \;\; = b \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.op \;\; = \text{“W”}
\end{array}
$$

$$
\begin{array}{l}
NoOtherFutureProposal(v,\, b) \triangleq \\
\quad \forall\, vv \in Values : \\
\qquad \forall\, bb \;\; \in Ballots : \\
\qquad\quad (bb > b \land ProposedValue(vv,\, bb)) \implies v = vv
\end{array}
$$

$$
\begin{array}{l}
ChosenIn(v,\, b) \triangleq \exists\, Q \in Quorum2 : \forall\, a \in Q : VotedForIn(a,\, v,\, b) \\
ChosenBy(v,\, b) \triangleq \exists\, b2 \in Ballots : (b2 \;\; \leq b \land ChosenIn(v,\, b2)) \\
Chosen(v) \triangleq \exists\, b \in Ballots : ChosenIn(v,\, b)
\end{array}
$$

Invariants.

$$
\begin{array}{l}
LearnInv \triangleq AllChosenWereAcceptedByPhase2 \land OnlyOneChosen \\
ReadInv \triangleq \forall\, b \in Ballots : readLog[b] = None \lor ChosenBy(readLog[b],\, b) \\
ConsistencyInv \triangleq \forall\, v1,\, v2 \in Values : Chosen(v1) \land Chosen(v2) \implies (v1 = v2)
\end{array}
$$

$$
\begin{array}{l}
StabilityInv \triangleq \\
\quad \forall\, v \in Values : \forall\, b \in Ballots : ChosenIn(v,\, b) \implies NoOtherFutureProposal(v,\, b)
\end{array}
$$

$$
\begin{array}{l}
AcceptorInv \triangleq \\
\quad \forall\, a \in Acceptors : \\
\qquad \land\, (maxVal[a] \;\; = None) \equiv (maxABal[a] = \, -1)
\end{array}
$$

$\land\ maxABal[a] \leq maxPBal[a]$
$\land\ (maxABal[a] \geq 0) \implies VotedForIn(a,\ maxVal[a],\ maxABal[a])$
$\land\ \forall\ c \in Ballots :$
$\quad c > maxABal[a] \implies \neg\exists\ v \in Values : VotedForIn(a,\ v,\ c)$

BIBLIOGRAPHY

[1] Amazon.com. https://www.amazon.com/.

[2] Blob storage — Microsoft Azure. https://azure.microsoft.com/en-us/services/storage/blobs/.

[3] Google classroom. https://classroom.google.com/.

[4] Open source cloud computing infrastructure - openstack. https://www.openstack.org/.

[5] Quizlet.com audience insights - Quantcast. https://www.quantcast.com/quizlet.com#trafficCard.

[6] ShareLaTeX. https://sharelatex.com.

[7] Target. https://www.target.com/.

[8] gRPC load balancing, 2017. https://grpc.io/blog/grpc-load-balancing/.

[9] NetBench. https://github.com/ndal-eth/netbench, 2017.

[10] NDP simulator. https://github.com/nets-cs-pub-ro/NDP/wiki/NDP-Simulator, 2018.

[11] Google cloud networking incident #19009, 2019. https://status.cloud.google.com/incident/cloud-networking/19009.

[12] OMNeT++ discrete event simulator. https://omnetpp.org/, 2020.

[13] Admission control - envoy documentation, 2021. https://www.envoyproxy.io/docs/envoy/v1.20.1/configuration/http/http_filters/admission_control_filter.

[14] Envoy proxy, 2021. https://www.envoyproxy.io/.

[15] Istio / traffic management, 2021. https://istio.io/latest/docs/concepts/traffic-management/.

[16] Kubernetes, 2021. https://kubernetes.io/.

[17] Load balancing | linkerd, 2021. https://linkerd.io/2.11/features/load-balancing/.

[18] Loadbalancing reference - v2.5.x — Kong docs, 2021. https://docs.konghq.com/gateway-oss/2.5.x/loadbalancing/.

[19] nginx news, 2021. https://nginx.org/.

[20] Fortio, 2022. https://fortio.org/.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[22] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *NSDI*, 2021.

[23] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, and Alec Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[24] Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *International Symposium on Distributed Computing*, pages 108–122. Springer, 2001.

[25] Marcos Kawazoe Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[26] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Multileader WAN Paxos: Ruling the archipelago with fast consensus. *CoRR*, 2017.

[27] Jyrki Alakuijala, Evgenii Kliuchnikov, Zoltan Szabadka, and Lode Vandevenne. Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms. *Google LLC*, 2015.

[28] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[29] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, 2012.

[30] Peter A Alsberg and John D Day. A principle for resilient sharing of distributed resources. In *ICSE*, 1976.

[31] Amazon Web Services, Inc. Multiple region multi-VPC connectivity. https://aws.amazon.com/answers/networking/aws-multiple-region-multi-vpc-connectivity/, 2019.

[32] Alexey Andreyev, Xu Wang, and Alex Eckert. Reinventing Facebook's data center network. Engineering at Meta, 2019. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/.

[33] Apache Software Foundation. Apache Hadoop distributed copy - DistCp guide, 2021. https://hadoop.apache.org/docs/stable/hadoop-distcp/DistCp.html.

[34] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI*, 2014.

[35] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[36] Daniel Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP tunnels. RFC 3209, 2001.

[37] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[38] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[39] Michael Z. Bell. 100% uptime anybody? http://www.riskythinking.com/articles/article8.php, 2004.

[40] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[41] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: Striking the right utilization-availability balance in WAN traffic engineering. In *SIGCOMM*, 2019.

[42] Alan Edward Branch. *Elements of shipping*. Routledge, 2007.

[43] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[44] Berit D. Brouer, Jakob Dirksen, David Pisinger, Christian E.M. Plum, and Bo Vaaben. The vessel schedule recovery problem (VSRP) – a MIP model for handling disruptions in liner shipping. *European Journal of Operational Research*, 224, 2013.

[45] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *NSDI*, 2020.

[46] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[47] Brandon Butler. Which cloud providers had the best uptime last year? http://www.networkworld.com/article/2866950/cloud-computing/which-cloud-providers-had-the-best-uptime-last-year.html, 2015.

[48] Brandon Butler. And the cloud provider with the best uptime in 2015 is ... http://www.networkworld.com/article/3020235/cloud-computing/and-the-cloud-provider-with-the-best-uptime-in-2015-is.html, 2016.

[49] Viveck R Cadambe, Nancy Lynch, Muriel Médard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.

[50] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 2016.

[51] Anupam Chander and Uyên P Lê. Data nationalism. *Emory LJ*, 64, 2014.

[52] Chia-Hsun Chang, Jingjing Xu, and Dong-Ping Song. Risk analysis for container shipping: from a logistics perspective. *The International Journal of Logistics Management*, 2015.

[53] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[54] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):25, 2017.

[55] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In *USENIX ATC*, 2017.

[56] David Clark and Wenjia Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on networking*, 1998.

[57] David Clark and John Wroclawski. An approach to service allocation in the internet. Internet draft, draft-clark-diff-svc-alloc-00, 1997. Work in Progress.

[58] Zoom Video Communications. Video conferencing, cloud phone, webinars, chat, virtual events. https://zoom.us/.

[59] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[60] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik,

David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[61] Nigel Cory. Cross-border data flows: Where are the barriers, and what do they cost? Technical report, Information Technology and Innovation Foundation, 2017.

[62] Rogério Leão Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using Mininet for emulation and prototyping software-defined networks. In *IEEE Colombian Conference on Communications and Computing (COLCOM)*, 2014.

[63] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[64] Nick Duffield, Carsten Lund, and Mikkel Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Transactions on Information Theory*, 2005.

[65] Nick G Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merive. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[66] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *USENIX ATC*, 2019.

[67] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *EuroSys*, 2020.

[68] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's software-defined networking control plane. In *NSDI*, 2021.

[69] Roy Fielding and Julian Reschke. RFC 7230: Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. *Internet Engineering Task Force (IETF)*, 2014.

[70] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *CACM*, 1986.

[71] Jens Geersbro and Thomas Ritter. CASE: Reinventing the container shipping industry. *Journal of Business Market Management*, 7, 2014.

[72] Amitabha Ghosh, Sangtae Ha, Edward Crabbe, and Jennifer Rexford. Scalable multi-class traffic management in data center backbone networks. *IEEE Journal on Selected Areas in Communications*, 2013.

[73] GitLab. The only single product for the complete devops lifecycle - GitLab. https://about.gitlab.com/.

[74] Google LLC. Google Docs. https://docs.google.com.

[75] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *SIGCOMM*, 2016.

[76] John Graham-Cumming. Cloudflare outage on july 17, 2020. The Cloudflare Blog, 2020. https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/.

[77] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *CoNext*, 2010.

[78] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *SIGCOMM*, 2018.

[79] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[80] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[81] Osama Haq, Mamoon Raja, and Fahad R Dogar. Measuring and improving the reliability of wide-area cloud paths. In *WWW*, 2017.

[82] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *SIGCOMM*, 2015.

[83] Todd Haselton. Salesforce's slack is fixed after 5-hour outage tuesday. *CNBC*, 2022. https://www.cnbc.com/2022/02/22/slack-is-down.html.

[84] Stephen Hemminger et al. Network emulation with NetEm. In *Linux conf au*, 2005.

[85] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 2008.

[86] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

[87] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[88] Todd Hoff. Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[89] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[90] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu Bollineni, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *SIGCOMM*, 2018.

[91] Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Lass, and Keith Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. In *NSDI*, 2017.

[92] Heidi Howard, Aleksey Charapko, and Richard Mortier. Fast Flexible Paxos: Relaxing quorum intersection for Fast Paxos. In *ICDCN*, 2021.

[93] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum intersection revisited. In *OPODIS*, 2016.

[94] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*, 2017.

[95] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure storage. In *USENIX ATC*, 2012.

[96] Intel. Storage - Intel ISA-L — Intel software. https://software.intel.com/en-us/storage/ISA-L.

[97] Mike Isaac and Sheera Frenkel. Gone in minutes, out for hours: Outage shakes facebook. *The New York Times*, 2021. https://www.nytimes.com/2021/10/04/technology/facebook-down.html.

[98] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hö lzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[99] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *NSDI*, 2013.

[100] Chuan Jiang, Sanjay Rao, and Mohit Tawarmalani. PCF: provably resilient flexible routing. In *SIGCOMM*, 2020.

[101] Mikel Jimenez and Henry Kwok. Building express backbone: Facebook's new long-haul network. https://engineering.fb.com/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/, 2017.

[102] Alex Johnson. Google confirms four-hour outage of youtube, gmail, other major services. *NBC News*, 2019. https://www.nbcnews.com/tech/internet/google-confirms-big-outage-youtube-gmail-other-major-services-n1012976.

[103] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *SIGCOMM*, 2014.

[104] Christian Kaufmann. Making the internet fast, reliable and secure. LINX Meeting, 2018. https://web.archive.org/web/20210308171115/https://www.linx.net/wp-content/uploads/LINX101-Akamai-ICN-ChristianKaufmann.pdf.

[105] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *SIGCOMM*, 2015.

[106] Praveen Kumar, Chris Yu, Yang Yuan, Nate Foster, Robert Kleinberg, and Robert Soulé. Yates: Rapid prototyping for traffic engineering systems. In *SOSR*, 2018.

[107] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *NSDI*, 2018.

[108] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Sol: Fast distributed computation over slow networks. In *NSDI*, 2020.

[109] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

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

[111] Leslie Lamport. Generalized consensus and Paxos. 2005.

[112] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[113] Leslie Lamport and Mike Massa. Cheap Paxos. In *DSN*, 2004.

[114] Butler Lampson. The ABCD's of Paxos. In *PODC*, 2001.

[115] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *SIGCOMM*, 2014.

[116] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[117] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPCC: High precision congestion control. In *SIGCOMM*, 2019.

[118] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *SIGCOMM*, 2014.

[119] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *NSDI*, 2013.

[120] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[121] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[122] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *SOSP*, 2015.

[123] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, 2008.

[124] Francisco Maturana and KV Rashmi. Irregular array codes with arbitrary access sets for geo-distributed storage. In *ISIT*, 2021.

[125] Microsoft. Azure Cosmos DB - globally distributed database service — Microsoft Azure. https://azure.microsoft.com/en-us/services/cosmos-db/.

[126] Microsoft. Microsoft global network. https://docs.microsoft.com/en-us/azure/networking/microsoft-global-network.

[127] Microsoft. Microsoft global network, 2017. https://azure.microsoft.com/en-us/blog/how-microsoft-builds-its-fast-and-reliable-global-network/.

[128] Microsoft. SLA for Azure Cosmos DB. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_3/, 2019.

[129] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[130] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review*, 2012.

[131] Iulian Moraru. Epaxos. https://github.com/efficient/epaxos, 2014. commit 791b115.

[132] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.

[133] Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *SoCC*, 2014.

[134] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When Paxos meets erasure code: Reduce network and storage cost in state machine replication. In *HPDC*, 2014.

[135] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *OSDI*, 2014.

[136] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. OceanStore: An architecture for global-scale persistent storage. In *OSDI*, 2014.

[137] Gaya Nagarajan. Evolution of facebook backbone. In *NANOG*, 2014.

[138] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. DPaxos: Managing data closer to users for low-latency and mobile applications. In *SIGMOD*, 2018.

[139] Joseph Noor, Mani Srivastava, and Ravi Netravali. Portkey: Adaptive key-value placement over dynamic edge networks. In *SoCC*, 2021.

[140] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[141] Ping Pan, George Swallow, and Ping Pan. Fast reroute extensions to RSVP-TE for LSP tunnels. RFC 4090, 2005.

[142] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[143] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[144] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *SIGCOMM*, 2013.

[145] Andreea Popescu, Pinar Keskinocak, Ellis Johnson, Mariana LaDue, and Raja Kasilingam. Estimating air-cargo overbooking based on a discrete show-up-rate distribution. *Interfaces*, 36, 2006.

[146] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. 2015.

[147] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.

[148] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI*, 2016.

[149] Red Hat. etcd - a distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/.

[150] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[151] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[152] Tanner Ryan. Cloudflare backbone: A fast lane on the busy internet highway. The Cloudflare Blog, 2021. https://blog.cloudflare.com/cloudflare-backbone-internet-fast-lane/.

[153] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.

[154] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: Novel erasure codes for big data. In *VLDB*, 2013.

[155] Amin Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking (TON)*, 14(SI):2551–2567, 2006.

[156] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM*, 2015.

[157] Slack Technologies. Slack is your digital HQ. https://slack.com/.

[158] Yee Jiun Song, Marcos Kawazoe Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC chains: Efficient client-server communication in geodistributed systems. In *NSDI*, 2009.

[159] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[160] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented DBMS. In *VLDB*, 2005.

[161] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. Network architecture for joint failure recovery and traffic engineering. In *SIGMETRICS*, 2011.

[162] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The resilient geo-distributed SQL database. In *SIGMOD*, 2020.

[163] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[164] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *NSDI*, 2020.

[165] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In *FAST*, 2018.

[166] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[167] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[168] Antonio Visioli. *Practical PID Control*. Springer London, 2006.

[169] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[170] Haoyu Wang, Haiying Shen, Zijian Li, and Shuhao Tian. GeoCol: A geo-distributed cloud storage system with low cost and latency using reinforcement learning. In *ICDCS*, 2021.

[171] Shuaian Wang. A novel hybrid-link-based container routing model. *Transportation Research Part E: Logistics and Transportation Review*, 61, 2014.

[172] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. R3: resilient routing reconfiguration. In *SIGCOMM*, 2010.

[173] Jason Warner. October 21 post-incident analysis, 2018. https://github.blog/2018-10-30-oct21-post-incident-analysis/.

[174] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[175] John Wilkes. Yet more Google compute cluster trace data. Google research blog, 2020. https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.

[176] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[177] Zhe Wu, Edward Wijaya, Muhammed Uluyol, and Harsha V. Madhyastha. Bolt-on global consistency for the cloud. In *SoCC*, 2018.

[178] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[179] Peter Zaitsev. What is the largest amount of data do you store in MySQL? - P ercona database performance blog. https://www.percona.com/blog/2012/11/09/what-is-the-largest-amount-of-data-do-you-store-in-mysql/.

[180] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. AWStream: Adaptive wide-area streaming analytics. In *SIGCOMM*, 2018.

[181] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 2017.

[182] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.

[183] Jiaqi Zheng, Hong Xu, Xiaojun Zhu, Guihai Chen, and Yanhui Geng. We've got you covered: Failure recovery with backup tunnels in traffic engineering. In *ICNP*, 2016.

[184] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. ARROW: restoration-aware traffic engineering. In *SIGCOMM*, 2021.