Cost-Effective Support for Low Latency Cloud Storage

by

Zhe Wu

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
2017

Doctoral Committee:

Assistant Professor Harsha V. Madhyastha, Chair
Associate Professor Vijay G. Subramanian
Professor Jason N. Flinn
Assistant Professor Manos Kapritsos

Zhe Wu

wuzhe@umich.edu

ORCID iD: 0000-0001-9828-5267

To my family.

ACKNOWLEDGEMENTS

There are many people that I want to thank during this long journey of Ph.D. study. I would have not been able to complete this dissertation without the help from them.

First and foremost, I'd like to thank my advisor Professor Harsha Madhyastha, who I am extremely fortunate to have had the opportunity to work with during the past six years. Harsha opened my eyes in this wonderful area of distributed systems, patiently guided me on my research, provided me with many valuable life and career suggestions, and has always been available when I needed help from him. Moreover, his passion for great research and commitment in pursuing cutting-edge ideas never cease to impress and educate me. I cannot ask for a better advisor.

I appreciate the collaboration opportunity with Professor Ethan Katz-Bassett from USC, who has also been very helpful in referring me to internship and career opportunities. I'm very grateful for his help throughout my Ph.D study.

I'm thankful to Professor Vijay Subramanian, Jason Flinn, and Manos Kapritsos for serving on my thesis committee. Their valuable feedbacks and thoughtful comments have greatly helped improve this dissertation.

I want to thank my colleagues and friends at UC Riverside, whom I spent my first three years of Ph.D life with. My groupmates Curtis Yu, Michael Butkiewicz, Masoud Akhoondi, and Dorian Perkines offered tremendous help in my early Ph.D years. They brought great insights into my projects and sometimes assisted me in conducting complicated experiments. We have been great friends since then. Amy

Ricks and Victor Hill were amazing CS department administrative staff who made our graduate student life much easier.

I also want to thank my friends here at University of Michigan who brought me an incredible and enjoyable life in Ann Arbor. There has never been a lack of fun and interesting discussion in our research group with Vaspol Ruamviboonsuk and Muhammed Uluyol. Yuru Shao inspired me for running my first ever marathon. Shichang Xu, Ashkan Nikravesh and I shared many late night office adventures during my early years at Michigan. I also had great fun with Juncheng Gu and his family during after work time. Many thanks to Stephen Reger, who took care of our cloud bills month after month during the past three years.

Finally and most importantly, I owe the greatest thanks to my family. My parents have always given me their trust, encouragement, and unconditional support. My dog Charlie brought countless joy and happiness to my life. My son Leo, who hasn't been around very long, has already encouraged me to be more responsible and gave me a better perspective on life. Last but not least, my wife Melody Sun, the pillar of our small home and the love of my life who has always taken great care of our family, has been an amazing companion and inspiration throughout my years in graduate school. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

ABSTRACT

By offering data centers in several locations across the globe, cloud services enable web services to serve users from nearby data centers, thereby reducing user-perceived latencies. However, web services face many challenges in doing so due to the poor abstractions and lack of performance guarantees offered by cloud services. First, almost every cloud storage service offers an isolated pool of storage in each of its data centers, leaving replication across data centers to applications. Second, the limited low level interface offered by cloud storage makes it challenging for applications to implement any replication protocols. Third, popular public cloud storage services today experience high latency variance which can greatly degrade median application performance.

To address these problems, my thesis enables web services to combine the use of multiple cloud services leveraging the greater diversity in data center locations, different storage performance characteristics, and discrepancies in pricing. Spreading a web services deployment across the data centers of multiple cloud providers however puts the onus of improving performance, reducing cost, and managing data replication on web services. Therefore, I have designed and implemented three systems that tackle these challenges. First, CosTLO judiciously combines several instantiations of the approach of issuing redundant requests and reduces the high latency variance of cloud storage without relying on cloud providers to make any changes. Second, SPANStore greatly simplifies the task of geo-replicating data in a manner that cost-effectively

satisfies any application's high-level latency, consistency, and fault-tolerance requirements. Lastly, CRIC enables strongly consistent access to geo-replicated data with latency and cost comparable to those achievable only if cloud storage offered a richer interface and had lower latency variance.

CHAPTER I

Introduction

With the ubiquitous availability of fast Internet connectivity, modern computer software has shifted from local computer applications to remote Internet-based services. Instead of having to install a bulky software binary which runs all or most of an application's logic locally on a user's device, this new type of software only requires users to have a web browser or install a thin client to access high performance software services with rich functionality. Figure 1.1 illustrates a common architecture of a geo-distributed web service. To use a web service, users first connect to the service's front-end servers. Users interact with the service by communicating their activities, service requests, and user data with the service front-ends. The web service's back-end servers deployed in one or several centralized data centers then process these data and requests in a timely manner, and send back request responses and application data back to the users.

There are many benefits for this shift to service-based applications, including:

- Offloading heavyweight application functionalities to the server side, so that an application's performance is no longer constrained by users' hardware.
- Enabling seamless user experience across user devices since application logic and data are on the server side.

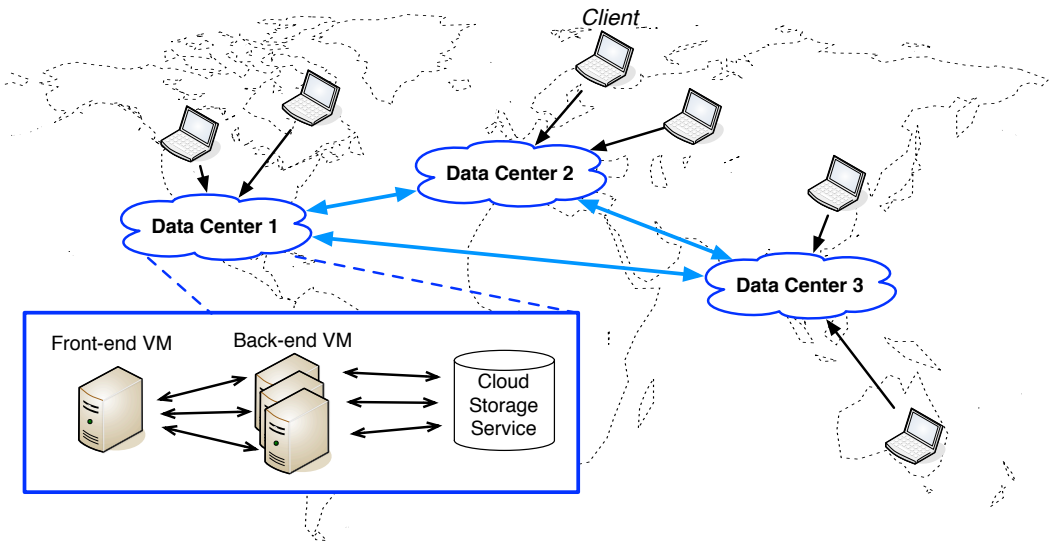


Figure 1.1: Generic architecture of a geo-distributed web service.

- Simplifying collaboration and information sharing among users by deploying web service logic and storing data in few centralized data centers.
- Providing users an on-demand software experience without any up-front commitment.
- Faster rollout of new service functionalities.

Because of these benefits, Internet-based web services have become a popular model for many IT companies to provide their software applications, such as messaging, document managing, social application, database, development software, accounting, etc.. Moreover, it also attracts many traditional businesses such as banking and shopping to provide web services to enrich their customers' experience. Web services have been incorporated into nearly all leading technology, business, and enterprise companies. Popular examples of such applications include Facebook [23], an online social network service that people can conveniently connect to and share information with their friends, Dropbox [22], a file and data storage service that enables users to access their data anywhere anytime, and Google Docs [27], which offers

collaborative document editing services.

However, in order for web services to attract users and make profit, one important criteria is that they must have good performance. Therefore, minimizing user-perceived latency is critical for web services. In fact, several studies have shown that even 100 milliseconds of additional delay can significantly reduce a web service's revenue [29, 1, 60]. Latency incurred by serving user requests usually contains two parts: communication latency caused by sending and receiving messages to and from web services' front-ends, and service latency incurred by processing user requests at web services' back-ends.

In contrast to applications that run as local software binaries, interactions with remote web services can incur significantly higher latency since web servers can be hundreds of milliseconds away from users. As a result, to improve user-perceived latency, web services have to be deployed across many geographically distributed data centers, so that users can be serviced from servers that are nearby. However, as not all web service providers can afford to build their own data centers in many locations across the world, this task can be simplified by leveraging public cloud infrastructure services [3, 13, 24].

1.1 Entering cloud era of web service deployments

Deploying applications in the cloud is an attractive proposition due to the opportunities available for eliminating the needs of building IT infrastructure and reducing cost: usage-based cloud pricing, the ability to scale up and down computation and storage resources on demand, economies of scale, and multiplexing of resource usage. In addition, the use of the cloud is also desirable because cloud providers offer many services that greatly simplify application development. For examples, to setup web servers and provide computation at scale, the cloud offers various types of virtual machines with a configurable number of CPUs, memory size, disk size, and operating

systems for applications to choose from, as well as supporting services such as virtual machine auto-scaling, user request load balancing, and user redirection services. This can greatly reduce the complexity of application providers' job of developing high performance web services.

The task of distributing data in different locations for deploying a low latency web service has also been simplified by the emergence of numerous cloud infrastructure services provided in many locations across the world. Cloud services such as Amazon Web Service [3], Windows Azure [13], and Google Cloud Platform [24] have tens of data centers across the globe offering storage-as-a-service, and every application provider can simply rent resources on-demand. To store data at scale, the cloud offers various types of storage services such as blob storage service for large data, table storage service for relational data, and virtual hard disks for virtual machines. For example, to store blob data, web services can store and retrieve data via PUTs and GETs without dealing with the complexities associated with setting up and managing the underlying storage infrastructure. Cloud storage services are also well designed and developed to provide properties such as scalability, availability and durability, which web service providers can take advantage of to simplify their application development.

The convenience and flexibility of cloud computing have attracted companies, including popular web services like Netflix, Dropbox and Yelp, to abandon self-built infrastructures and use the cloud to service their users. Studies also show that 57% of web services are running on Amazon Web Services (the biggest public cloud service) today, and the cloud computing industry is predicted to grow to a 110 billion dollar market and account for 90 percent of mobile data traffic by 2019 [33].

1.2 Challenges

Despite the convenience of using cloud storage services to build web services, the inability to modify the interface to cloud infrastructure and having to share this infrastructure with other applications make it challenging for application providers to develop and deploy low latency web services. More specifically, three significant issues associated with cloud services complicate their use.

1.2.1 High latency variance

Cloud storage services offer no performance guarantees. Although the median latency of serving user requests is low [128], sharing of resources across cloud tenants leads to very high latency variance: a burst in one web service's workload may increase latencies for other web services accessing the same storage servers or using the same network paths. For example, my measurements of Azure's and Amazon's storage services from hundreds of nodes across the world show an absolute inflation greater than 200ms in the 99th percentile write latency as compared to the median latency; the relative inflation is greater than 2x in both write and read operations. Such high latency in the tail of the latency distribution greatly impacts the average web service performance. My measurements also show that downloading an average webpage containing 100 objects from the closest Amazon data center incurs a 3x latency inflation comparing to the case where there is no latency variance, as the page load time is constrained by the object that is fetched last.

1.2.2 Poor abstractions

Today's cloud services only provide storage services within each of their data centers. Moreover, applications may even choose to use multiple cloud providers to improve their service quality, and no cloud provider has the incentive to offer data synchronization with other providers. Therefore, web services that desire globally

distributed deployments need to implement data replication across data centers by themselves. Managing data replication is complex since there are numerous ways to assign locations to data replicas, and a web service's choice directly impacts the performance that it can offer to its users and the cost that it incurs.

Moreover, ensuring consistency of data across replicas further complicates the task of replication. Web services deployed in the cloud must themselves keep data consistent across data centers. However, cloud storage services only provide simple PUT/GET interface. Therefore, for web services to manage data replication themselves, they must deploy additional virtual machines (VMs) in every data center to handle conflicting concurrent operations to underlying storage. Although cloud storage already provides high performance services, this additional layer of replication can incur high additional cost, become the bottleneck of the entire web service data tier, and cause poor performance on web services.

1.2.3 Performance versus cost tradeoff

Cloud services adopt usage based pricing policy [8, 14]. In addition to the cost for storing data, all reads, writes, and data transfers also add to an application's costs, even when storage or the network is under-utilized. Implicitly, the more cloud resources that a web service buys from the cloud, the better performance its application can have. For example, to ensure low data access latency, web services can replicate all data in all data centers and incur high cost of storing and accessing data comparing to storing data in a single data center. Also, to ensure that replication layer VMs do not cause performance bottleneck, most web services will have to incur high cost in deploying a sufficiently large number of VMs.

Some applications may value lower cost over the best achievable performance, different applications may demand different requirements such as the degrees of data consistency, some objects may only be popular in some regions, and some clients may

be near to multiple data centers, any of which can serve them quickly. All these parameters mean that no single deployment provides the best fit for all applications and all objects. Since cloud providers do not provide a centralized view of storage with rich semantics, every application needs to reason on its own about where and how to store data to satisfy its latency goals and consistency requirements at low cost.

1.3 Thesis and Contributions

In this dissertation, my goal is to address these challenges for web services deployed in the cloud. More specifically, I support the following thesis: ***it is practical to cost-effectively offer better abstractions and support for latency SLOs on legacy cloud storage services.***

My thesis research focuses on developing solutions for web services to better utilize cloud storage services to build their applications. Especially, I focus on devising solutions for web services that do not require cloud providers to make any changes to their infrastructure or their services. To improve user-perceived latency, I develop solutions that enable web services to combine the use of multiple cloud services to increase the diversity of cloud data centers. The price discrepancies of different clouds further reduces the operational cost of using cloud storage services. To offer a better abstraction of cloud storage resources, I design systems that automate the process of geo-replicating data and managing data accesses in order to satisfy a web service's high-level performance requirements with highly efficient cloud resource utilization. Moreover, I place strong emphasis on storage tail latency performance, a critical latency metric in today's cloud applications.

The work presented in this dissertation makes four contributions.

Multi-cloud key-value storage. A key design decision in my thesis research is to design key-value storage systems spanning the data centers of multiple cloud

service providers. There are multiple benefits of doing this.

First, using multiple cloud providers can enable storage services to offer lower GET/PUT latencies. This is because for nearly all data centers, I find that there are nearby data centers from other cloud providers available within 50ms. We can use this greater choice of nearby storage options to meet tighter latency requirements, or to meet a fixed latency requirement using fewer storage replicas (by picking locations nearby to multiple frontends). Intuitively, this benefit occurs because different providers have data centers in different locations, resulting in a variation in latencies to other data centers and to clients.

Second, building systems across multiple cloud providers also enables it to meet application requirements at potentially lower cost due to the discrepancies in pricing across providers. For example, nearby Azure data centers have similar pricing, and so, no cheaper options than local storage exist within 150ms for Azure-based services. However, for the majority of Azure-based frontends, deploying across all Amazon Web Service, Azure, and Google Cloud yields more than 5 storage options that are cheaper for at least some operations. Thus, by judiciously combining resources from multiple providers, we can build systems that use these cheaper options to reduce costs.

Moreover, storing data across multiple cloud services does not impose high development overhead to application developers despite the difference across cloud providers due to two reasons: (1) only data needs to be spread across multiple cloud providers and (2) key-value cloud storage services across different cloud providers have a largely identical interface.

Low latency variance on legacy cloud storage. I developed the CosTLO system [128], which uses the well-known approach of issuing redundant requests so that the earliest response can be considered to reduce the latency variability of cloud storage services. This approach is motivated by my measurement study that high latency variance is caused predominantly by isolated latency spikes.

The primary challenge in CosTLO is to minimize the cost overhead imposed by redundant requests. If only few redundant requests are issued, it may not be effective to reduce latency variability. On the other hand, the cost overhead is high if CosTLO issues too many redundant requests. I have developed algorithms that combine the use of different forms of redundancy (e.g., to download an object, an end-host can issue redundant read requests either to the same object or to different copies of the object) to identify the most cost-effective configuration that can meet a web service's goals for latency variability. Leveraging observations from my measurements, CosTLO captures a cloud service's replication and load balancing policies in order to estimate the latency variance associated with any configuration. Using a trace of Wikipedia's workload, my results show that CosTLO can reduce the spread between 99th percentile and median read latencies by 50% with only a 25% increase in cost.

Cost-effective replica placement for geo-replicated data. My second system, SPANStore [126], recognizes the opportunity to lower cost and to lower user-perceived latencies by combining the use of multiple cloud providers. SPANStore offers a unified view of geographically distributed storage services and judiciously replicates data objects across data centers, which is transparent to any web service that uses it. It automates the process of trading off cost and latency, while satisfying consistency and fault-tolerance requirements.

There are two main challenges in replicating data in a manner that minimizes cost. The first challenge is to identify the granularity of replication. One replication policy per data object will not be cost-optimal due to high volatility in the workloads of individual objects, whereas a single replication policy for all objects will require over-replication in order to satisfy latency constraints. In SPANStore, I leverage application-level hints to group objects based on their access patterns; the aggregate workload across a large set of objects is more stable than the workload of individual objects, and different replication policies for objects with different access patterns

result in a lower cost overhead than a single replication policy for all objects.

The second challenge is to choose from a wide gamut of replication policies, while accounting for the inter-dependencies between web service requirements, workload, and cost. I formulate the problem of determining the cost-optimal replication policy as a mixed integer program to address the trade-off between storage, networking and request costs. Results from my prototype of SPANStore show that, in comparison to alternative designs for geo-replicated storage, SPANStore can lower costs by over 10x with the same or better latency performance.

Strongly consistent access of geo-replicated data. My third system, CRIC, is an efficient strongly consistent data replication solution for web services deployed in the cloud. Though several systems have recently been developed to explore latency versus consistency trade-offs in geo-replicated storage [66, 86, 96, 97, 93, 117], little attention has been paid to the fact whether web services are deployed across data centers they own or on third-party cloud infrastructure.

I identify two challenges that complicate the task of providing low latency data replication across cloud storage services. First, replicating data across cloud data centers and enabling low latency data access are made challenging by the fact that almost every cloud provider offers an isolated pool of storage in each of its data centers, leaving replication and managing data consistency across data centers to individual web services. Moreover, cloud storage services only expose a limited interface to storage, which applications have no ability to modify. Therefore, for web services to manage data replication themselves, they must deploy additional virtual machines (VMs) to run replication protocol such as Paxos [89] in every data center to guarantee the ordering of data updates and handle conflicting concurrent operations to underlying storage. This is clearly inefficient. The second challenge is that the high latency variance that is typical within each data center of popular cloud storage services [128] is further compounded when data is geo-replicated. Due to the long distance between

data centers, even issuing requests to all replicas and waiting for a subset to finish (the traditional wisdom from quorum systems to reduce latency variance) has only limited ability to reduce tail latency.

The design goal in CRIC is to reduce the overhead of deploying a web service managed replication layer, as well as tackle the cloud storage latency variance on shared data. I design CRIC as a library that interposes on any geographically distributed web service's interactions with cloud storage and enables strongly consistent low latency access to geo-replicated data. The core in CRIC is a novel Paxos variant, CPaxos, which does not require web services to setup any additional virtual machines for replica coordination. To reduce user perceived latency variance, instead of using the standard majority quorum as in Paxos, CRIC leverages a two layer structured quorum, hierarchical quorum [87], which increases data and request redundancy inside each replica data center to further reduce client perceived latency variance.

1.4 Organization

The remaining chapters of this thesis are organized as follows. In Chapter II, I provide an overview of the usage of the cloud to deploy web services, and present prior work in the area of geo-distributed storage, storage performance, and cloud system design. Chapter III presents CosTLO, which reduces the latency variance of accessing data in the cloud storage services. Chapter IV focuses on SPANStore, which tackles the challenges of replicating data cost effectively in the cloud. Chapter V describes CRIC, which enables efficient strongly consistent data replication in the cloud. Chapter VI concludes this dissertation by summarizing the contributions and discussing future work.

CHAPTER II

Related Work

In this chapter, I present the prior work in three main areas that this dissertation builds upon: performance oriented web service cloud deployments, distributed storage system performance, and geo-distributed storage system design.

2.1 Web service deployments in the cloud

Evaluating benefits of cloud deployments. With the explosive growth of cloud computing, there has been some work recently to answer the question—when to use cloud services? Answers to this question have largely been restricted to examining how and when to migrate applications from the application provider's data center to a cloud service [75, 116, 125]. While these efforts consider issues such as cost and wide-area latency like we do, none of them seek to provide a storage service with a unified view to geo-replicated storage. Some others compare the performance offered by various cloud services [92, 39]. However, these efforts do not consider issues such as cost and consistency. In this dissertation, I examine the benefits that application providers can obtain by taking the use of cloud services to the next level—spreading web services across multiple cloud infrastructure services.

Choosing among cloud services. Recently, there have been some efforts at measuring and characterizing the performance offered by various cloud services [92,

39]. The focus in these efforts has been either to enable application providers to choose between cloud services (with the application being deployed on one of them) or to monitor an application deployed on a cloud service. The focus of my study instead is to understand the latency and benefits of deploying web services *across multiple cloud services*.

Using multiple cloud services. Several previously developed systems have also considered the use of multiple cloud service providers for storing data. RACS [44] proposes replicating data across multiple cloud providers to obtain similar benefits as RAID disk arrays. SafeStore [85] tries to improve data durability leveraging multi-cloud deployment. DEPSKY [57] uses multi-cloud to increase reliability and security of critical data. MetaStorage [56] addresses cloud vendor lock-in issue leveraging multi-cloud deployment. However, all of these systems focus on issues pertaining to availability, durability, vendor lock-in, performance, and consistency. None of these systems seek to minimize cost by exploiting pricing discrepancies across providers, which is one of the main contribution of this dissertation.

Other complementary efforts have focused on utilizing compute resources from multiple cloud providers. AppScale [64] enables portability of applications across cloud services, but without any attempt to minimize cost. Conductor [121] orchestrates the execution of MapReduce jobs across multiple cloud providers in a manner that minimizes cost. In contrast to these systems, I focus on unifying the use of storage resources across multiple providers. Moreover, I envision application deployments spanning multiple cloud services and explore the latency benefits from multi-cloud deployments.

Trading off cost and performance. Zhang et al. [133] showed how a service provider can trade off cost versus performance by jointly optimizing network routing and client redirection. In contrast, web services deployed on cloud services have little control over Internet routing. Moreover, since costs of running a web service are cru-

cially dependent on its workload and implementation architecture, the computation of costs are best left to web service administrators. Our goal instead is to evaluate the potential latency gains of multi-cloud deployments, narrow down on the regions in which users would benefit the most, and highlight the challenges in harnessing these latency benefits.

Minerva [47], Hippodrome [50], scc [101] and Rome [122] automate the provisioning of cost-effective storage configurations while accounting for workload characterizations. Though these systems share a similar goal of minimizing service cost as ours, their setting is restricted to storage clusters deployed within a data center. This dissertation focuses on geo-replicated storage, and so its deployment strategies must account for inter-data center latencies and multiple administrative domains. Farsite [45] provides scalable storage in an untrusted environment and shares some techniques with our SPANStore system (Chapter IV), e.g., lazily propagating updates. However, Farsite does not consider any optimizations in reducing cost.

2.2 Improving performance of distributed storage

Cloud performance studies. Prior studies have compared the performance offered by different cloud providers [92], reverse-engineered cloud service internals [110], and studied application deployments on the cloud [79]. Our measurement study of Azure and S3 (Chapter III) is the first to quantify the latency variance on these storage services and to characterize the impact of different forms of redundancy. Moreover, unlike Bodik et al. [58], who focused on characterizing and modeling spikes in application workloads, our measurements show that an application using cloud storage can suffer latency spikes even when there is no spike in that application's workload.

Reducing storage tail latencies. Reducing storage tail latencies has been a major focus in storage system design, as high tail latency of single storage requests can greatly impact median application performance. One of the most popular techniques

used to reduce tail latencies is to increase request redundancy, so that the earliest response can be considered. The approach of issuing redundant requests to reduce tail latencies has been considered previously [67, 120], but the focus has primarily been on understanding the implications of redundancy on system load. Before this, redundant task execution was also used to reduce runtimes of analytics jobs by eliminating stragglers [129, 48]. In contrast, our CosTLO system (Chapter III) demonstrates how the approach of using redundant requests should be applied in the context of cloud storage services, in order to meet latency SLOs while minimizing cost overhead. Also, in Chapter V, we show how to apply this technique in a new setting—geo-replicated storage with shared data on third-party infrastructure—to satisfy a new goal: reducing latency variance while providing strong consistency.

Some application providers such as Facebook use in-memory caching of data to reduce tail latencies [107]. However, caching cannot reduce tail latencies associated with PUT requests. Moreover, caching at a single data center cannot tackle latency spikes on Internet paths, and not all application providers will be able to afford caches at multiple data centers that can accommodate enough data to reduce 99^{th} percentile GET latencies.

Combining cloud providers to improve performance. Others have combined the use of multiple cloud providers to improve availability [44, 85], to offer more secure storage [57], and to reduce cost [121]. This dissertation takes advantage of multi-cloud web service deployment to improve web service performance as well. It uses cloud storage services offered by multiple providers because 1) the combination offers more data center pairs that are close to each other (Chapter IV), and 2) latency spikes on the Internet paths to data centers in different cloud services are uncorrelated (Chapter III).

Redesigning cloud services. Several recent proposals redesign storage systems and data centers to improve storage latency performance [115], to offer bandwidth

guarantees to tenants [54, 108, 55, 118], or to ensure predictable completion times for TCP flows [130, 81, 124]. However, all of these proposals require modifications to a cloud service's infrastructure. It is unclear when, and if, cloud services will revamp their infrastructure to these more complex architectures. Therefore, this dissertation instead considers the scenarios that web services cannot change underlying cloud systems and develop solutions that can satisfy latency requirements for applications deployed on the cloud without having to wait for any modifications to cloud services. Moreover, our CRIC system (Chapter V) enables functionality—synchronization of replicas across the data centers of multiple cloud providers—that no cloud provider has the incentive to offer.

2.3 Low-latency geo-replicated storage

Replication protocols. Paxos [90] is considered to be the standard solution for achieving strongly consistent data replication. It makes minimum assumptions about system hardware and its correctness is well studied and proved. However, it is known to incur high latency when being used in the wide-area setting, since every storage request requires two round of wide area communication, which is intolerable for many applications. Therefore, there are many Paxos variants proposed recently to improve its latency with some workload assumptions. For example, Fast Paxos [91] can commit writes in one round when request conflicts are rare. Multi-Paxos [62] can commit writes in one round with a stable leader. And EPaxos [105] can commit writes in one round with a pre-defined request conflict resolution mechanism. All of these replication protocols are well studied for transactional storage. However, none of these protocols have paid attention to the fact that whether web services are deployed across data centers they own or on third-party cloud infrastructure, which is the main problem we try to address in the CRIC system (Chapter V).

Other than Paxos protocol, there are other replication protocols that can achieve

data replication. For example, Lynx [132] and Azure RTable [17] use chain replication to achieve serializable transactions. In contrast to chain replication, which sacrifices write latency for read latency, Paxos replication permits both read and write operations to complete while waiting for responses from only the nearest quorum of nodes. Therefore, in this dissertation, we mainly focus on leveraging the Paxos protocol to achieve strongly consistent data replication.

Many advancements in reducing commit latency also stem from the development of new protocols that reduce network communication in the common case [91, 86, 105, 109, 66, 131, 106, 94, 70]. Building upon these protocols, my contribution lies in improving data replication efficiency and optimizing read and write latency without introducing intermediate virtual machines to enrich the interface to cloud storage.

Latency vs. consistency trade-off. Numerous systems strive to provide fast performance and stronger-than-eventual consistency across the wide area. Recent examples include Walter [114], Spanner [66], Gemini [93], COPS [96], and Eiger [97]. Given that wide-area storage systems cannot simultaneously guarantee both the strongest forms of consistency and low latency [102], these systems strive to push the envelope along one or both of those dimensions. However, none of these systems focus on minimizing cost while meeting performance goals, which is our primary goal. In fact, most of these systems replicate all data to all data centers, and all data centers are assumed to be under one administrative domain. We believe that the work presented in this dissertation can be adapted to achieve the consistency models and performance that these systems offer at lower cost.

Many storage systems allow (or require) applications to choose weak consistency guarantees in exchange for higher availability and lower latency [96, 97, 93, 117, 88, 68]. Incremental consistency [74] allows an application to speculatively process data based on what it currently knows about the consistency of the data, requiring the application to implement rollback behavior as appropriate. The SNOW theorem [98]

explores this tradeoff space in detail, showing that no transaction protocol can provide non-blocking consistent reads in the presence of conflicting writes. Taking these guarantees one step further, Olive takes advantage of conditional writes in cloud storage to provide exactly-once semantics for application logic in the presence of failures [111]. In contrast, my CRIC system (Chapter V) guarantees strong consistency to provide the simplest storage semantics and, like Olive, uses conditional writes to achieve this.

CHAPTER III

Lower Latency Variance on Cloud Storage Services

Minimizing user-perceived latencies is critical for many applications as even hundreds of milliseconds of additional delay can significantly lower revenue [60, 43, 113]. Large-scale cloud services aid application providers in this regard by enabling them to serve every user from the closest among several geographically distributed data centers. For example, our measurements from over 120 PlanetLab nodes across the globe show that, when every node downloads 1 KB-sized objects from the closest Microsoft Azure data center, the median download latency is less than 100ms for over 90% of the nodes.

However, on today's cloud services, both fetching and storing content are associated with high latency variance. For example, for over 70% of the same 120 nodes considered above, the 99^{th} percentile and median download latencies from the closest Azure data center differ by 100ms or more. These high tail latencies are problematic both for popular applications where even 1% of traffic corresponds to a significant volume of requests [68], and for applications where a single request issued by an end-host requires the application to fetch several objects (e.g., web page loads) and user-perceived latency is constrained by the object fetched last. For example, our measurements show that latency variance in S3 more than doubles the *median* page load time for 50% of PlanetLab nodes when fetching a webpage containing 50 objects.

To enable application providers to avail of the cost benefits enabled by cloud services, without having latency variance degrade user experience, we develop CosTLO (Cost-effective Tail Latency Optimizer). Since we observe that the high latency variance is caused predominantly by isolated latency spikes, CosTLO uses the well-known approach [120, 67] for reducing variance by augmenting every GET/PUT request with a set of redundant requests, so that the earliest response can be considered. We tackle three key challenges in using this redundancy-based approach in CosTLO.

First, the end-to-end latency when any end-host uploads to or downloads from a cloud storage service has several components: latency over the Internet, latency over the cloud service's data center network, and latency within the storage service. To tackle the variance in all of these components, CosTLO exploits the fact that redundant requests to cloud storage services can be issued in a variety of ways, each of which impacts a different component of end-to-end latency. For example, while issuing redundant requests to *the same object* may elicit an earlier response due to differences in load across servers hosting replicas of the object, one can further reduce the impact of server load by issuing redundant requests to *a set of objects* which are all copies of the object being accessed. Alternatively, to reduce the impact of spikes in data center network latency, redundant requests can be issued to *different front-ends* of the storage service or relayed to the *same front-end via different virtual machines (VMs)*. Furthermore, when a client is accessing an object stored in a particular data center, redundant requests can be issued to *copies of the object in other data centers* in order to tackle the variance in Internet latencies.

However, not all forms of redundancy have utility in practice due to the complex architectures of cloud services. Therefore, we also empirically evaluate the ways in which redundant requests should be issued for CosTLO's approach to be viable on Amazon S3 and Microsoft Azure, the two largest cloud storage services today. For example, when issuing concurrent requests to multiple data centers, we find that it

is essential to leverage storage services offered by multiple cloud providers; utilizing a single cloud provider's data centers is insufficient to tame the variance in Internet latencies. Our study also shows that, due to load balancing within the data center networks of cloud services, concurrent requests to the same front-end of a storage service are sufficient to tackle spikes in data center network latencies, and more complex approaches are unnecessary. To the best of our knowledge, this is the first work that identifies the key causes for latency variance in cloud storage services and studies the impact of different forms of redundancy.

Third, the number of configurations in which CosTLO can implement redundancy is unbounded—not only can CosTLO *combine* the use of various forms of redundancy, but it can also *vary* the number of redundant requests, the probability with which it issues redundant requests, etc.—and the impact on cost and latencies varies significantly across configurations. Therefore, for CosTLO to add redundancy in a manner that satisfies an application's goals for latency variance cost-effectively, it becomes essential that CosTLO be able to 1) *estimate*, rather than measure, the cost and latencies associated with any particular configuration, and 2) *search* for a cost-effective configuration, instead of enumerating through all possible configurations. To address these challenges, 1) we model the load balancing and replication within cloud storage services in order to accurately capture the dependencies between concurrent requests, and 2) we develop an efficient algorithm to identify a cost-effective CosTLO configuration that can keep latency variance below a target. Note that no prior work that uses redundant requests seeks to minimize cost.

We have implemented and deployed CosTLO across all data centers in S3 and Azure. To evaluate CosTLO, we use PlanetLab nodes at 120 sites as clients and replay a trace of Wikipedia's workload. Our results show that CosTLO can reduce the spread between 99^{th} percentile and median GET latencies by 50% for the median PlanetLab node, with only a 25% increase in cost.

3.1 Characterizing Latency Variance

We begin with a measurement study of Amazon S3 and Microsoft Azure. We 1) quantify the latency variance when using these services, 2) analyze the impact of latency variance on applications, and 3) identify the dominant causes of this variance.

Overview of measurements. To analyze client-perceived latencies when downloading from and uploading to cloud storage services, we gather two types of measurements for a week. First, we use 120 PlanetLab nodes across the world as representative end-hosts. Once every 3 seconds, every node uploaded a new object to and downloaded a previously stored object from the S3 and Azure data centers to which the node has the lowest median RTT. Second, from "small instance" VMs in every S3 and every Azure data center, we issued one GET and one PUT per second to the local storage service. In all cases, every GET from a data center was for a 1 KB object selected at random from 1M objects of that size stored at that data center, and every PUT was for a new 1 KB object. To minimize the impact of client-side overheads, we measure GET and PUT latencies on PlanetLab nodes as well as on VMs using timings from tcpdump.

In addition, we leverage logs exported by S3 [34] and Azure [42] to break down end-to-end latency minus DNS resolution time into its two components: 1) latency within the storage service (i.e., duration between when a request was received at one of the storage service's front-ends and when the response left the storage service), and 2) latency over the network (i.e., for the request to travel from the end-host/VM to a front-end of the storage service and for the response to travel back). We extract storage service latency directly from the storage service logs, and we can infer network latency by subtracting storage service latency from end-to-end request latency.

Quantifying latency variance. Figure 3.1 shows the distribution across nodes of the spread in latencies; for every node, we plot the absolute and relative difference between the 99^{th} percentile and median latencies. In both Azure and S3, the median

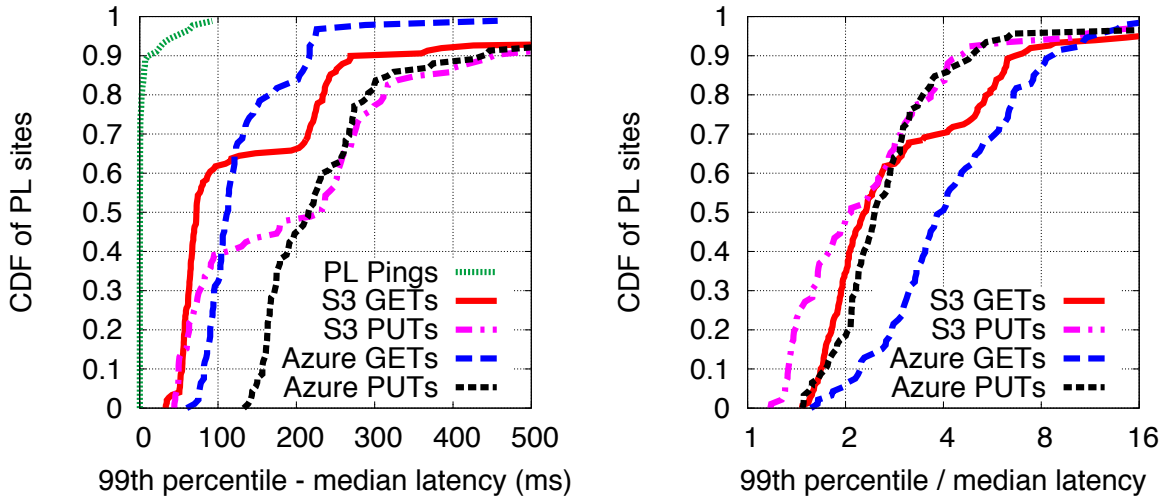
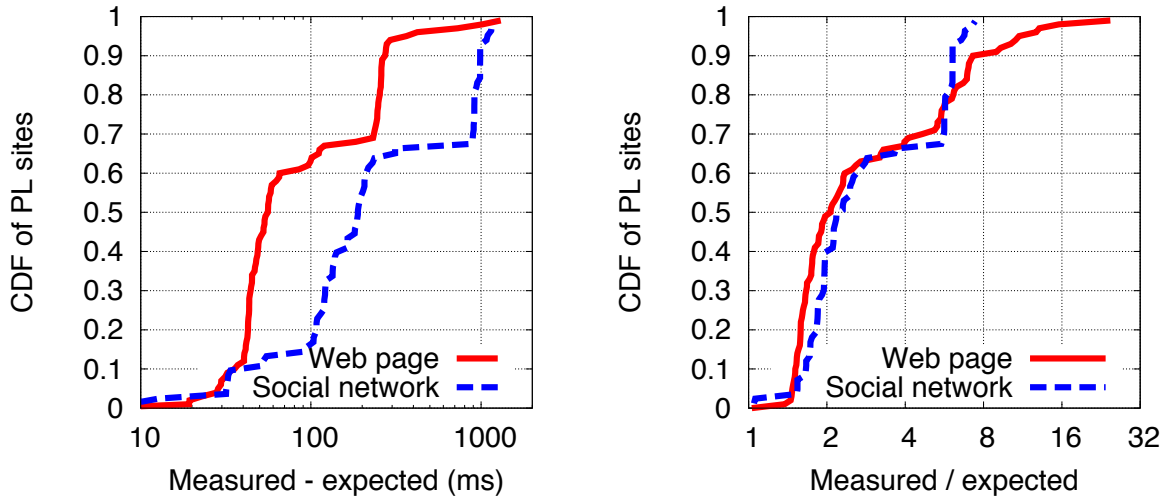


Figure 3.1: (a) Absolute and (b) relative inflation in 99^{th} percentile latency with respect to median. Logscale x-axis in (b).

PlanetLab node sees an absolute inflation greater than 200ms (70ms) in the 99^{th} percentile PUT (GET) latency as compared to the median latency; the median relative inflation is greater than 2x in both PUTs and GETs. To show that this high latency variance is not due to high load or slow access links of PlanetLab nodes, Figure 3.1 also plots for every node the difference between 99^{th} percentile and median latency to the node closest to it among all PlanetLab nodes.

Impact on applications. To show that high latency variance can significantly degrade application performance, we conduct measurement studies in two application scenarios. The first one is a web service that serves static webpages containing 50 objects. The second one is a social network application, where an update from a user triggers a synchronization mechanism to make all of the user's followers fetch that update. In both applications, one user-level request requires the application to issue several requests to cloud storage, and user-perceived latency is constrained by the request that finishes last. We consider a setting in which (1) users only fetch objects from their closest data centers, (2) every user in the social network application has 200 followers [2], and (3) users and their followers have the same closest data centers.



(a) (b)

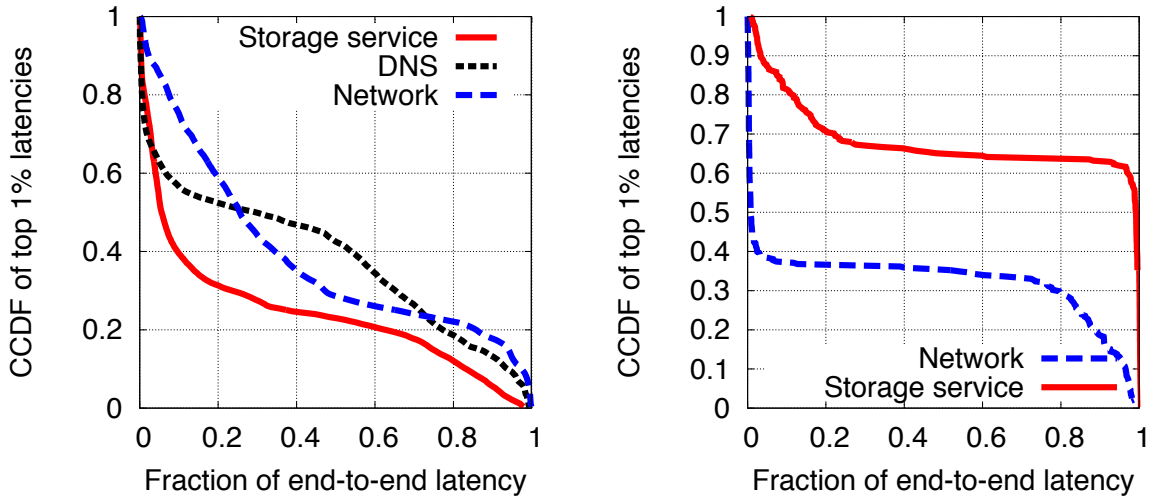
Figure 3.2: (a) Absolute and (b) relative inflation in median user-level request latency with respect to ideal latency. Note logscale on x-axis in both graphs.

We setup clients on PlanetLab nodes and applications on S3, emulate interactions between users and applications using real world traces [30, 95], and measure the page load time/sync completion time.

Ideally, with no latency variance, in the webpage application, page load time should be the same as the latency of fetching a single object if clients fetch all objects on the page in parallel, and in the social network application, the sync completion time should be the same as the latency incurred by the farthest follower to fetch a single object. However, Figure 3.2 shows that, for over 80% of PlanetLab nodes, latency variance causes at least 50ms latency inflation in the *median* page load time and at least 100ms latency inflation in the *median* sync completion time. This corresponds to a 2x relative inflation for more than 50% of users.

Causes for tail latencies. We observe two characteristics that dictate which solutions can potentially reduce the tail of these latency distributions.

First, we find that neither are the top 1% of latency samples clustered together in time nor are they correlated with time of day. Thus, the tail of the latency distribution is dominated by isolated spikes, rather than sustained periods of high latencies.



(a) End-host perspective (b) VM perspective

Figure 3.3: Breakdown of components of tail latencies.

Therefore, a solution that monitors load and reacts to latency spikes will be ineffective.

Second, Figure 3.3(a) shows that all three components of end-to-end latency significantly influence tail latency values. DNS latency, network latency, and latency within the storage service account for over half the end-to-end latency on more than 40%, 25%, and 20% of tail latency samples. Since network latencies as measured from PlanetLab nodes conflate latencies over the Internet and within the cloud service's data center network, we also study the composition of tail latencies as seen in our measurements from VMs to the local storage service. In this case too, Figure 3.3(b) shows that both components of end-to-end latency—latency within the storage service, and latency over the data center network—contribute significantly to a large fraction of tail latency samples. *Thus, any solution that reduces latency variance will have to address all of these sources of latency spikes.*

3.2 Overview of CosTLO

Goal. We design CosTLO to meet any application's service-level objectives (SLOs) for the extent to which it seeks to reduce latency variance for its users. To ensure

that CosTLO is broadly applicable across several classes of applications, we consider the most fundamental SLO that applications can build upon—SLOs that bound the variance of the latencies of individual PUT/GET operations; we discuss CosTLO's ability to handle more complex application-specific SLOs in Section 3.5.

Though there are several ways in which such SLOs can be specified, we do not consider SLOs that bound the absolute value of, say, 99^{th} percentile GET/PUT latency; due to the non-uniform geographic distribution of data centers, a single bound on tail latencies for all end-hosts will not help reduce latency variance for end-hosts with proximate data centers. Instead, we focus on SLOs that limit the tail latencies for any end-host *relative* to the latency distribution experienced by *that* end-host. Specifically, we consider SLOs which bound the difference, for any end-host, between 99^{th} percentile latency and its baseline median latency (i.e., the median latency that it experiences without CosTLO). Every application specifies such a bound separately for GETs and PUTs.

Approach. Since tail latency samples are dominated by isolated spikes, our high-level approach is to augment any GET/PUT request with a set of redundant requests, so that the first response can be considered. Though this is a well-known approach for reducing tail latencies [120, 67, 49], CosTLO is unique in exploiting several ways of issuing redundant requests in combination.

For example, consider downloads from the closest S3 data center at the PlanetLab node in University of Kansas. When this client fetches objects by issuing single GET requests, the difference between the 99^{th} percentile and median latencies is 214ms. The simplest way to reduce variance is to have the client issue two concurrent GET requests to download an object (Figure 3.4(a)). This decreases the gap between 99^{th} percentile and baseline median latency to 110ms, but doubles the cost for GET operations and network bandwidth. Alternatively, the client can issue a single GET request to a VM in the cloud, which can in turn issue two concurrent requests for the

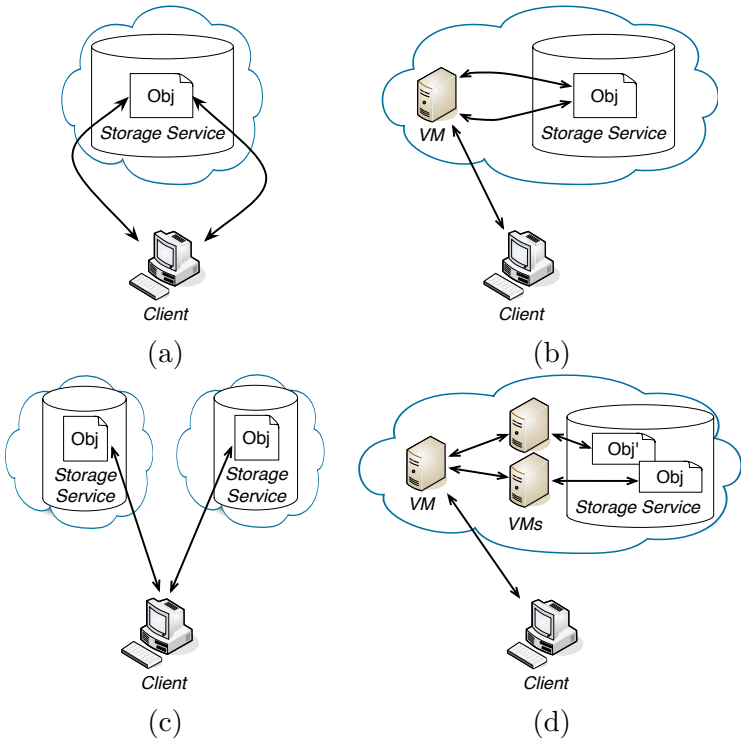


Figure 3.4: Illustration of various ways in which CosTLO can concurrently issue requests: (a) to a single object in a storage service, (b) to a single object via a relay VM, (c) to storage services in multiple data centers, or (d) via multiple relay VMs.

requested object to the local storage service (Figure 3.4(b)). While this adds VM costs and the 99^{th} percentile latency is now 135ms higher than the baseline median latency, relaying redundant requests via VMs reduces bandwidth costs (since a single copy of the object leaves the data center). A third option is to have the client concurrently fetch copies of the object from multiple data centers (Figure 3.4(c)), e.g., the two closest S3 data centers. This strategy—the best of the three options in terms of reducing variance (inflation in 99^{th} percentile compared to baseline median drops to 34ms)—eliminates the overhead of VM costs but increases storage costs.

Challenges. This example illustrates how various forms of redundancy differ in the tradeoff between reducing variance and increasing cost. Choosing from these various options, so as to satisfy an application's SLO cost-effectively, is challenging for several reasons.

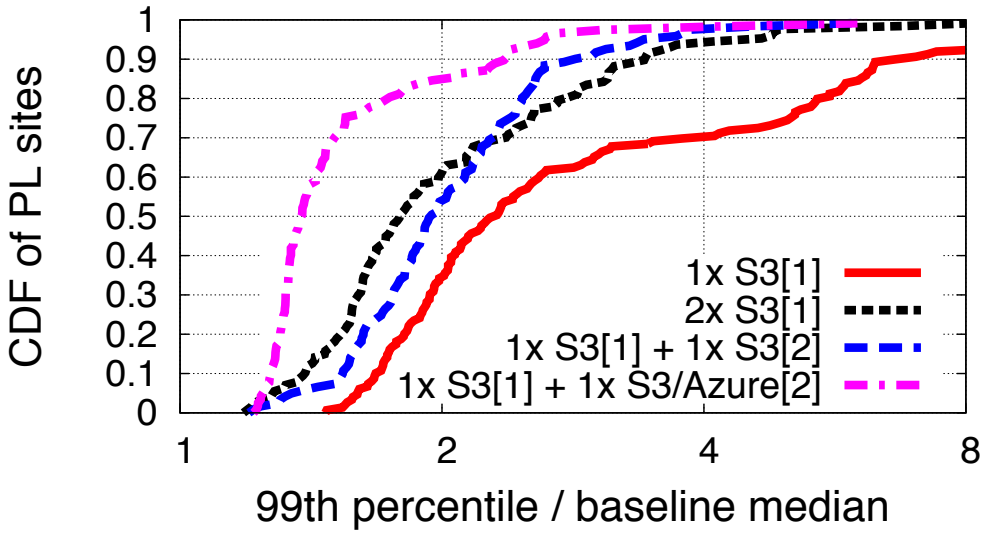


Figure 3.5: Impact on Internet tail latencies of different ways to send two concurrent requests. Logscale on x-axis.

- ***Large configuration space.*** There exist an unbounded number of configurations in which CosTLO can issue redundant requests. This is not only because the degree of parallelism is unbounded, but also because different types of redundancy can be combined with each other. For example, Figure 3.4(d) shows a configuration that both 1) uses multiple relay VMs to route around latency spikes in the data center network, and 2) issues requests to different objects that are copies of each other. This unbounded configuration space makes it impossible to simply *measure* the latency distribution offered by every candidate configuration of CosTLO.
- ***Complex service architectures.*** However, *predicting* the impact on latencies of any particular approach for issuing redundant requests is complicated by the fact that we have little visibility into the architecture of any cloud storage service. As we describe later, due to correlations between concurrent requests, we cannot estimate the latencies obtained with k concurrent requests simply by considering the minimum of k independent samples of a single request's latency distribution.
- ***Multi-dimensional pricing policies.*** Finally, minimizing CosTLO's cost overhead is made complex by the fact that cloud services charge customers based on a

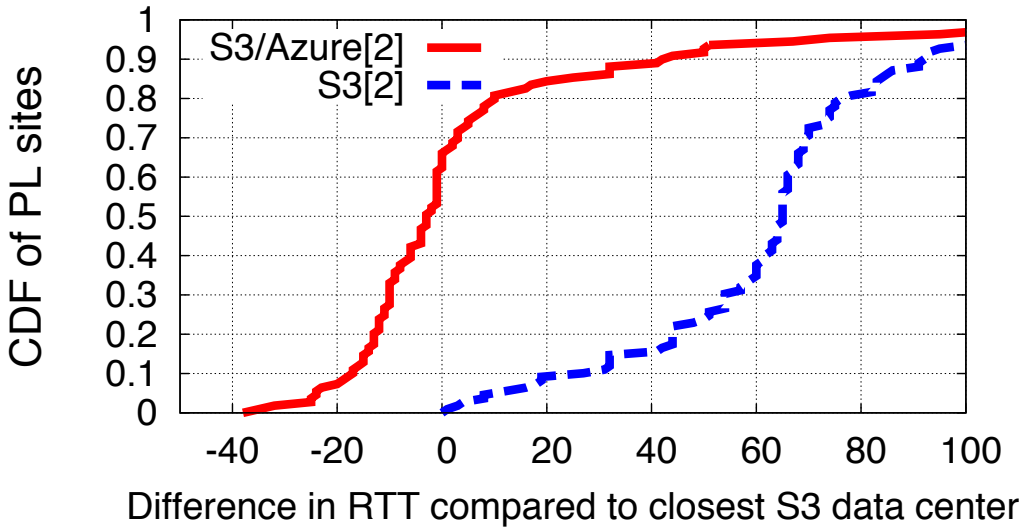


Figure 3.6: Comparison of second closest data center within a cloud service and across cloud services.

Goal	Technique	Section
Characterizing configuration space	Measurement study highlighting viable options for CosTLO to reduce latency variance via redundancy	§ 3.3
Selecting cost-effective configuration	Design a representation of configurations and a cost-effective configuration selection algorithm	§ 3.4.2
Estimating cloud storage latency distribution	Explicitly modeling the sources of concurrent requests' correlations in Internet latency, data center network latency, and cloud storage latency	§ 3.4.3

Table 3.1: Overview of techniques developed to build CosTLO.

combination of storage, request, VM, and bandwidth costs. Each of the potential ways in which redundant requests can be issued impacts a subset of these pricing dimensions, and the extent to which it does so depends on the application's workload.

Table 3.1 summarizes the various techniques used in CosTLO to address these challenges.

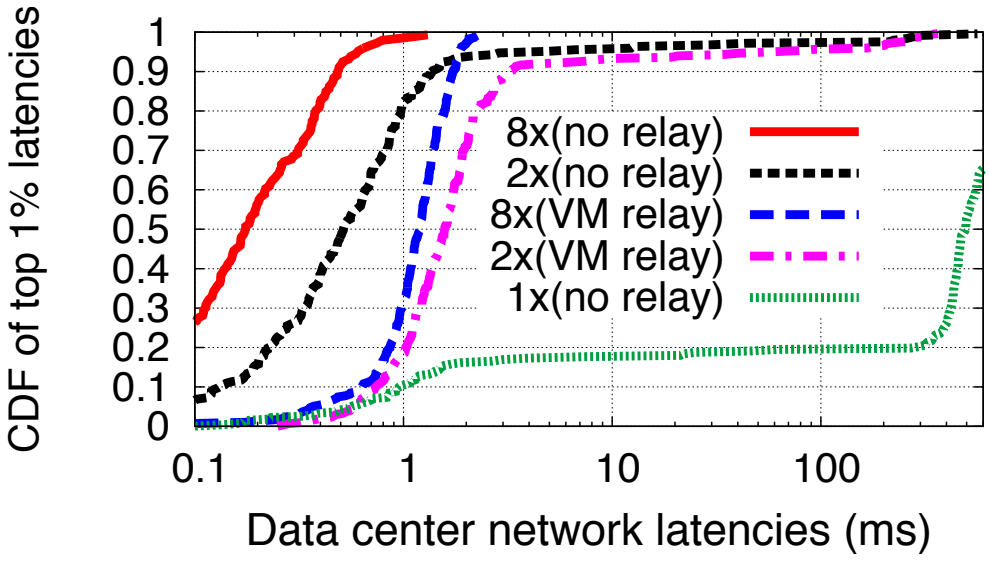


Figure 3.7: Different ways of exploiting path diversity in the data center network. Note logscale on x-axis.

3.3 Characterizing Configuration Space

CosTLO's approach of issuing redundant requests to reduce tail latencies can broadly be applied in two ways. One way is to concurrently issue the same request multiple times in order to *implicitly* exploit load balancing in the Internet or inside cloud services. For example, issuing multiple GET requests concurrently to the same object may lower latencies either because different requests take different paths through the Internet to the same data center, or because different requests may be served by different storage servers that host replicas of the same object. An alternate way is to *explicitly* enforce diversity by concurrently issuing a set of requests that differ from each other, yet have the same effect, e.g., by storing multiple copies of an object and issuing concurrent requests to different copies, or by issuing concurrent requests to different front-ends of a storage service.

Here, we empirically evaluate on both S3 and Azure the efficacy of several approaches for reducing tail latencies in three components of end-to-end latency: Internet latency, data center network latency, and latency in the storage service. We ignore DNS latency since applications often do not control how clients perform DNS

lookups and concurrently querying multiple nameservers to reduce DNS latencies has no impact on cost.

3.3.1 Internet latencies

To examine the utility of different approaches on reducing Internet tail latencies, we issue pairs of concurrent GET requests from each PlanetLab node in three different ways and then compare the measured tail latencies with those seen with single requests. We use the notation "nx $C[m]$" to denote a setting in which every PlanetLab node issues n concurrent requests to its m^{th} closest data center in cloud C, where C is either S3, Azure, or the union of data centers in the two ("S3/Azure").

Multiple requests to same data center. To account for spikes in Internet latency, we first consider every end-host concurrently issuing multiple requests to the storage service in the data center closest to it. Load balancing in the Internet [52] may result in concurrent requests taking different paths to the same data center.[1] However, as shown by the *"2x S3[1]"* line in Figure 3.5, though issuing two concurrent requests to the same data center does reduce the inflation in tail latencies, relative inflation seen at the median PlanetLab node remains close to 2x; the *"1x S3[1]"* line represents the baseline where end-hosts issue single GET requests to their closest data center.

Requests to multiple data centers. Since path diversity to the same data center is insufficient to tame Internet latency spikes, we next consider issuing concurrent requests to multiple data centers; in addition to a GET request to its closest S3 data center, we have every node issue a GET request in parallel to its second closest S3 data center. The *"1x S3[1] + 1x S3[2]"* line in Figure 3.5 shows that this strategy offers little benefit in reducing latency variance. This is because, for most PlanetLab nodes, the second closest data center is too far to help tame latency spikes to the node's closest data center.

[1]Multiple requests may also help in surviving packet losses. However, loss rates in our measurements are below 0.1%, thus making them an insignificant factor in causing latency spikes.

The root cause for this is that any particular cloud service provider provisions its data centers in a manner that maximizes geographical coverage. Hence, any pair of data centers in the same cloud service are distant from each other. For example, the *"S3[2]"* line in Figure 3.6 shows that RTT to the second closest data center in S3 is 40ms greater than the RTT to the closest S3 data center for over 80% of PlanetLab nodes.

Leveraging multiple cloud providers. Though a single cloud provider's data centers are distant from each other, we observe that different cloud providers often have nearby data centers. For example, Figure 3.6 shows that, for over 80% of PlanetLab nodes, RTT to the second closest data center across S3 and Azure is within 25ms of the RTT to the closest S3 data center.

Therefore, leveraging the fact that storage services offered by all cloud providers largely offer the same PUT/GET interface, every client of an application can download copies of an object in parallel from 1) the closest data center among the ones on which the application is deployed, and 2) the second closest data center across all storage services that offer a PUT/GET interface. Figure 3.5 shows that doing so reduces the inflation in 99^{th} percentile GET latency to be less than 1.5x the baseline median at 70% of PlanetLab nodes. Note that the application itself can be deployed across a single cloud provider's data centers. As we describe later (Section 3.4.1), CosTLO can maintain copies of objects without the application's knowledge.

3.3.2 Data center network latencies

Next, we consider strategies for tackling latency spikes within a cloud service's data center network.

In this case, we first attempt to implicitly exploit path diversity by issuing the same PUT/GET request multiple times in parallel from a VM to the local storage service. Load balancing within the data center network [73] may cause concurrent

requests to take different routes to the same front-end of the storage service, thus enabling us to avoid latency spikes that occur on any one path.

Alternatively, we can explicitly exploit path diversity in two ways. When a VM issues a GET/PUT to the local storage service, we can either relay each request through a different VM (Figure 3.4(d)), or issue each request to a different front-end of the storage service. While the latter approach is applicable in S3, all requests issued by the same tenant are submitted to the same front-end [61] in Azure. Therefore, we only consider here the former way of explicitly exploiting path diversity.

In one of Azure's data centers, Figure 3.7 compares the distribution of tail latencies over the network in three scenarios for how a VM downloads objects from the local storage service: 1) a single request is issued, 2) concurrent requests are issued directly to the same front-end, and 3) concurrent requests are relayed via different VMs. In the latter two cases, we experiment with different levels of parallelism. We see that both implicit and explicit exploitation of path diversity significantly reduce tail latencies, with higher levels of parallelism offering greater reduction. However, using VMs as relays adds some overhead, likely due to requests traversing longer routes.

3.3.3 Storage service latencies

Finally, we evaluate two approaches for reducing latency spikes within the storage service, i.e., latency between when a request is received at a front-end and when it sends back the response. When issuing n concurrent requests to a storage service, we either issue all n requests for the same object or to n different objects. The former attempts to implicitly leverage the replication of objects within the storage service, whereas the latter explicitly creates and utilizes copies of objects. In either case, if concurrent requests are served by different storage servers, latency spikes at any one server can be overridden by other servers that are lightly loaded.

At one data center each in Azure and S3, Figure 3.8 shows that both approaches

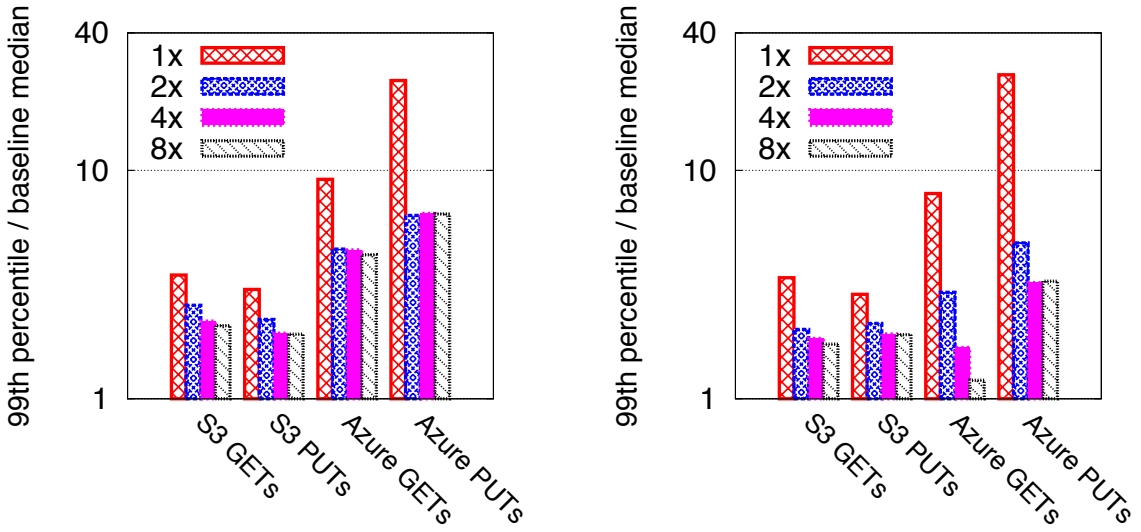


Figure 3.8: Impact on storage service tail latency inflation when issuing concurrent requests to (a) the same object, and (b) to different objects. Note logscale on y-axis.

for issuing concurrent requests significantly reduce tail GET and PUT latencies. However, the takeaways differ between Azure and S3. On S3, irrespective of whether we issue multiple requests to the same object or to different objects, the reduction in 99^{th} percentile latency tails off with increasing parallelism. As seen later in Section 3.4, this is because, in S3, concurrent requests from a VM incur the same latency over the network, which becomes the bottleneck in the tail. In contrast, on Azure, 99^{th} percentile GET latencies do not reduce further when more than two concurrent requests are issued to the same object, but tail GET latencies continue to drop significantly with increasing parallelism when concurrent requests are issued to different objects. In the case of PUTs, the benefits of redundancy tail off at parallelism levels greater than two due to Azure's serialization of PUTs issued by the same tenant [61].

Besides using multiple copies of objects, erasure coding [69] is another approach to explicitly create data redundancy. It is more cost effective than creating full copies of objects since the additional storage for redundancy is only a fraction of the full data. However, we do not consider this approach in CosTLO for two reasons. First,

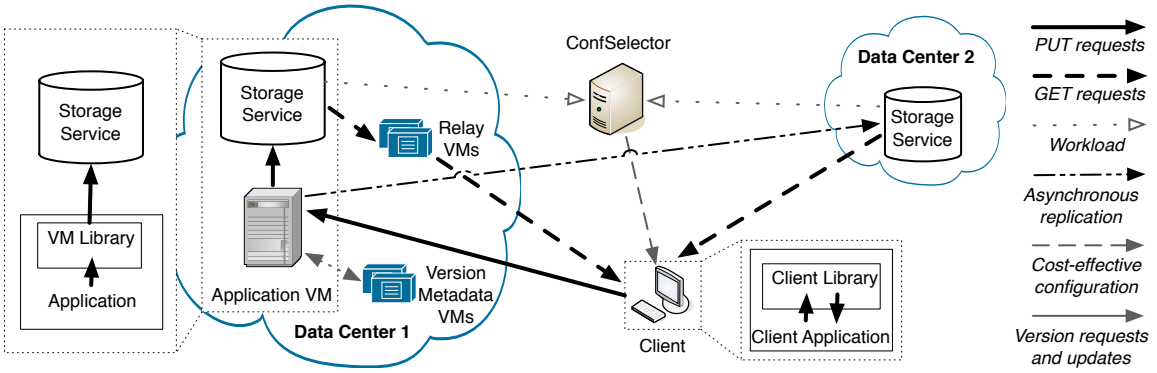


Figure 3.9: CosTLO architecture; VMs that run measurement agents are not shown.

erasure coding cannot reduce the latency variance in writes since all splits of data need to be updated. Second, because our target objects are small, reading a split of an object from disk will take similar latency as reading a full copy of the object.

3.3.4 Takeaways

In summary, our measurement study highlights the following viable options for CosTLO to reduce latency variance via redundancy. First, CosTLO can tackle spikes in Internet latencies by issuing multiple requests to a client's closest data center. If greater reduction in Internet tail latencies is desired, CosTLO must concurrently issue requests to the two closest data centers to the client from the union of data centers in multiple cloud services. Second, for latency spikes in a data center's network, it suffices to issue multiple requests to the storage service in that data center. While explicitly relaying requests via VMs may help reduce bandwidth costs (as seen in our example earlier in Section 3.2), they do not offer additional benefits in reducing latencies. Finally, for latency spikes within the storage service, CosTLO can issue multiple requests either to the same object or to different objects that are all copies of the object being accessed.

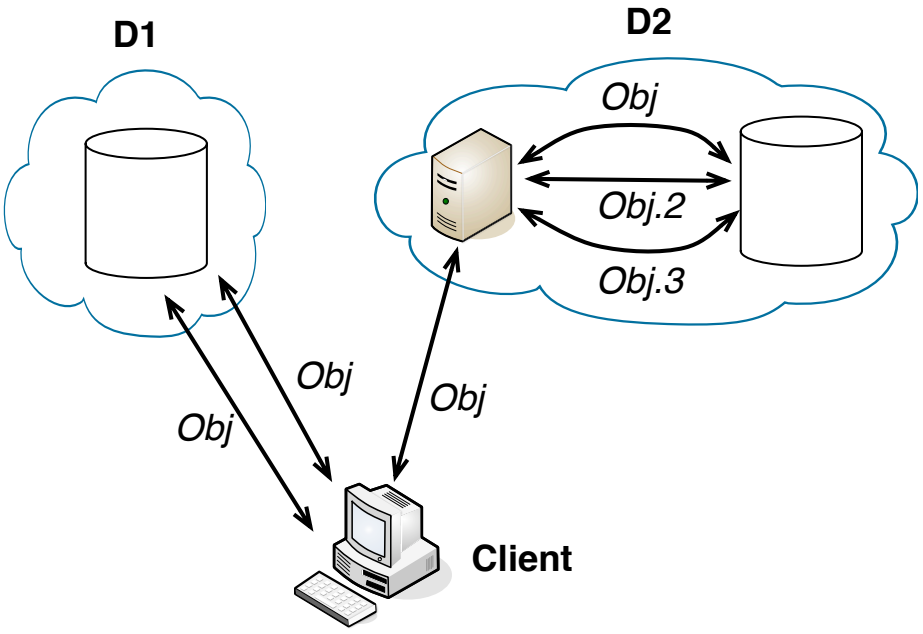


Figure 3.10: Illustration of a configuration in which the tuples for data centers D1 and D2 are *(Copies=1, ReqPerCopy=2, VM=False)* and *(Copies=3, ReqPerCopy=1, VM=True)*. All edges are annotated with the name of the object for which GET requests are issued when the client requests object *Obj*.

3.4 Cost-effective Support for SLOs

Next, we describe how CosTLO combines the use of the above-mentioned viable redundancy options in order to satisfy an application's SLO cost-effectively.

3.4.1 System architecture

Application interface. As shown in Figure 3.9, application code on end-hosts links to CosTLO's client library and uses the GET operation[2] in this library to fetch data from cloud storage. The client library issues a set of GET requests to download an object and returns the object's data to the application as soon as any one GET completes. Unlike downloads, we let client-side application code upload data to its own VMs, because the application may need to update application-specific metadata

[2]When ambiguous, we refer to applications invoking CosTLO's GET/PUT *operations*, and CosTLO issuing GET/PUT *requests* to storage services.

before writing user-uploaded data to cloud storage. The application code in these VMs links to CosTLO's VM library and invokes the PUT operation in this library to write data to the local storage service. The VM library in turn issues a set of PUT requests to the local storage service, and informs the application that the PUT operation is complete once any one of the PUT requests finishes. CosTLO offers the same consistency semantics as S3 [7]: read-after-write consistency for PUTs of new objects and eventual consistency for overwrite PUTs; we discuss how CosTLO can support strong consistency later in Section 3.6.

Configuration selection. CosTLO's central ConfSelector selects the configuration in which its client library and VM library should serve PUTs and GETs. ConfSelector divides time into epochs, and at the start of every epoch, it selects a new configuration separately for every IP prefix, since Internet latencies to any particular data center are similar from all end-hosts in a prefix [100]. To exploit weekly stability in workloads [46], we set epoch durations to one week; we do not consider exploiting diurnal workload patterns because we observe good cost-efficiency even when only leveraging weekly workloads stability. At the start of every epoch, the CosTLO library on every end-host and instances of CosTLO's VM library in every data center fetch the configurations that are relevant to them. Since all objects accessed by a client are replicated as per the configuration associated with the client's prefix, no per-object metadata is necessary. If a client loses its state, it simply re-fetches the configuration in the current epoch for its prefix from ConfSelector.

In the rest of this section, we address three questions: 1) how does ConfSelector identify a cost-effective configuration of CosTLO that can satisfy the application's SLO?, 2) while searching for this cost-effective configuration, how does ConfSelector *estimate* the tail latencies for any configuration, given that is impractical to *measure* the latencies offered by every configuration?, and 3) how does CosTLO preserve data consistency?

3.4.2 Selecting cost-effective configuration

Characterization of workload and cloud services. To estimate the cost overhead and latency variance associated with any CosTLO configuration, ConfSelector 1) takes as input the pricing policies at every data center, 2) uses logs exported by cloud providers to characterize the workload imposed by clients in every prefix, and 3) employs a measurement agent at every data center. Every agent gathers three types of measurements: 1) pings to a representative end-host in every prefix, 2) pairs of concurrent GETs and pairs of concurrent PUTs to the local storage service, and 3) the rates at which VMs can relay PUTs and GETs between end-hosts and the local storage service without any queueing. We ignore the impact of VM failures on tail latency since cloud providers guarantee over 99.95% of uptime for VMs [5, 41].

Representation of configurations. To search through the configuration space, ConfSelector represents every candidate configuration for a prefix as follows. First, a configuration's representation includes two three-tuples, which specify the manner in which end-hosts in the prefix should execute GETs. One three-tuple is for the data center from which the application serves the prefix and another for the data center closest to the prefix among all other data centers on which CosTLO is deployed. Either tuple specifies 1) the number of copies of the object stored in that data center, 2) the number of requests issued to each copy, and 3) whether all of these requests are relayed via a VM.[3] Figure 3.10 depicts an example.

Second, the configuration includes one two-tuple for the manner in which CosTLO's VM library should serve PUTs from the prefix. We use only one tuple in this case, since PUTs from an end-host are served solely at the data center closest to it, and we use a two-tuple, since the VM library does not relay PUTs through other VMs.

Third, due to the inability to cancel redundant requests after getting a response,

[3]If necessary, these three-tuples can be extended to include other dimensions, e.g., whether each request is issued to a different front-end. The dimensions we use here are based on the techniques that we found to be viable in reducing tail latencies on Azure and S3 (Section 3.3).

to reduce the cost overhead associated with redundant requests, the client/VM library initially issues a single request when serving a GET/PUT. If no response is received for a certain period, the client/VM library times out and probabilistically issues redundant requests concurrently as specified by the tuples described above. The timeout period ensures that CosTLO's redundancy is focused on requests that incur a high latency, whereas probabilistically issuing redundant requests offers finer-grained control over latency variance. For both PUTs and GETs, the configuration representation specifies the values of the timeout period and probability parameters. Considering the same example from Figure 3.10 but with 70% probability and 50ms timeout period to issue redundant requests, the configuration would be [(1, 2, False), (3, 1, True), 50ms, 70%] (the PUT tuple is ignored here).

Configuration search. Given this representation of the configuration space, ConfSelector identifies a cost-effective configuration of CosTLO for any particular prefix as follows. It initializes the configuration for a prefix to reflect the manner in which an application serves its clients when not using CosTLO—by always issuing only a single request to the data center closest to a client. CosTLO imposes no cost overhead in this configuration.

Thereafter, our structured representation of the configuration space enables ConfSelector to step through configurations in the increasing order of cost. For this, ConfSelector maintains a pool of candidate configurations, from which it considers the minimum cost configuration in every step. ConfSelector computes the cost associated with a configuration as the sum of expected costs for storage, VMs, requests, and bandwidth based on the workload for the prefix and the manner in which the configuration mandates that GET/PUT operations be served. If the lowest cost configuration in the current pool does not satisfy the SLO, ConfSelector discards this configuration and inserts all neighbors of this configuration into the pool of candidates. Two configurations are neighbors if they differ in the value of exactly one

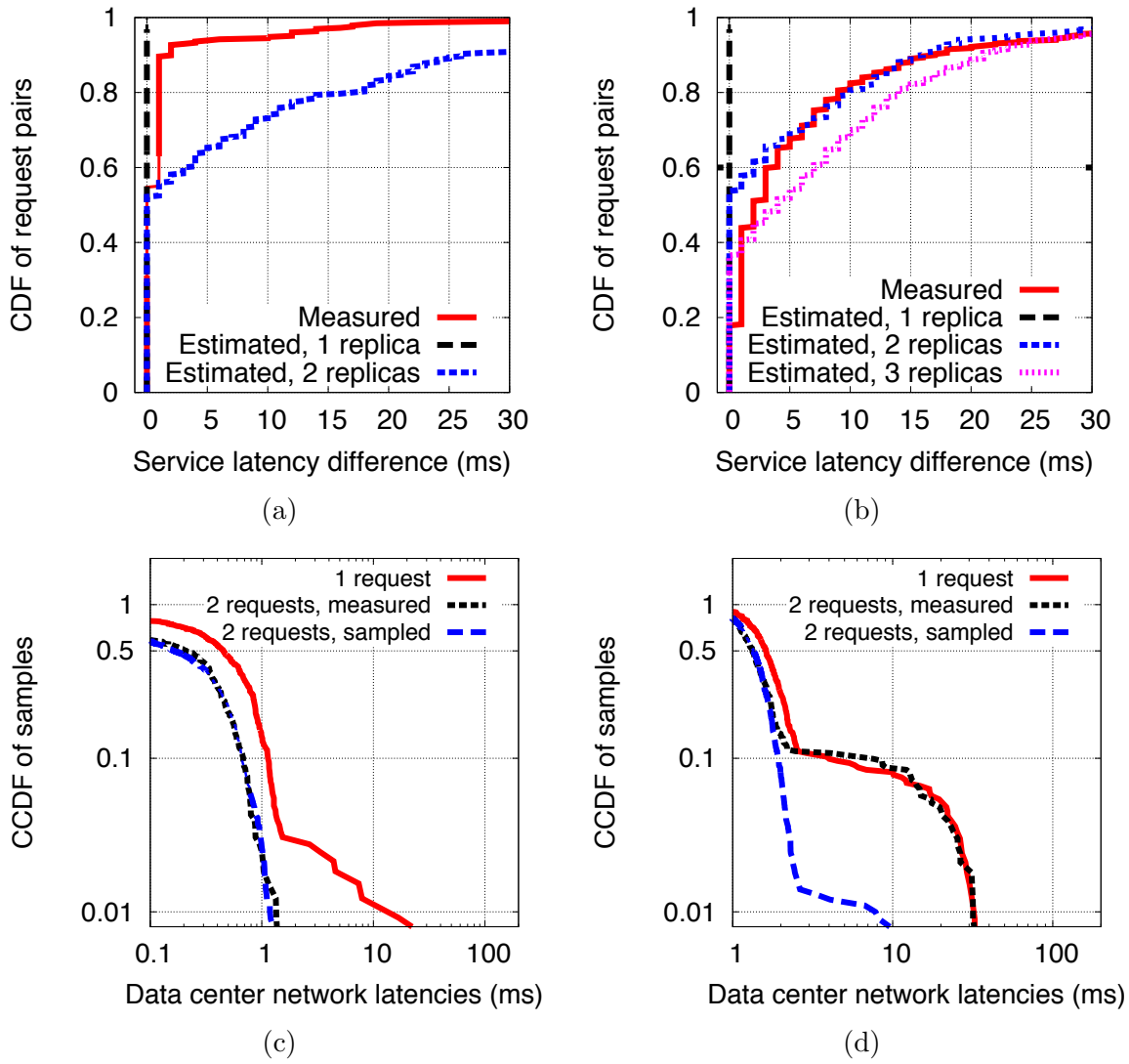
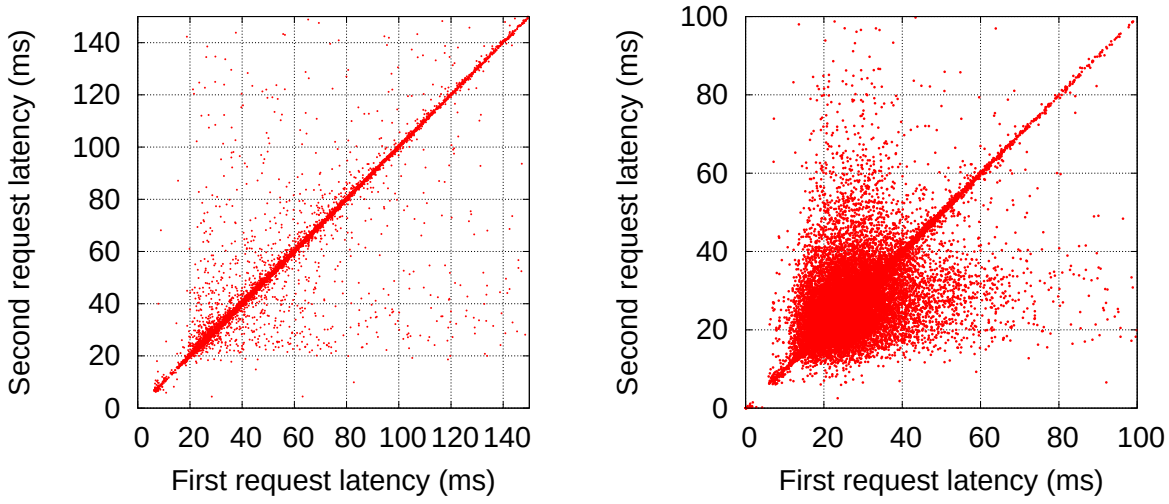


Figure 3.11: Distribution of service latency difference between concurrent GET requests offers evidence for GETs to an object (a) being served by one replica in Azure, and (b) being spread across two replicas in S3. Data center network latencies for concurrent requests are (c) uncorrelated on Azure, and (d) correlated on S3. Note logscale on both axes of (c) and (d).

parameter in the configuration representation. For example, configurations [(1, 2, False), 50ms, 70%] and [(2, 2, False), 50ms, 70%] are neighbors (we only show one GET tuple here for simplicity). This process terminates once ConfSelector finds a configuration that satisfies the SLO.



(a) Azure GETs (b) S3 GETs

Figure 3.12: Scatter plot of first vs. second request GET latency when issuing two concurrent requests to a storage service.

3.4.3 Estimating latency distribution

To identify when it has found a configuration that will satisfy the application's SLO, for any particular configuration for a prefix, ConfSelector must be able to estimate the latency distribution that clients in that prefix will experience when served in that configuration. For brevity, we present here ConfSelector's estimation of latencies only for GETs, which it computes in four steps. First, for either data center used in the configuration, we estimate the latency distribution when a VM in that data center concurrently issues requests to the local storage service, where the number of requests is specified by the data center's tuple in the configuration representation. Second, we estimate the latency distribution for either data center's tuple by adding the distribution computed above with the latency distribution measured to the prefix from a VM in that data center. Simply adding these distributions works when objects are smaller than 1 KB, and in Section 3.6, we discuss how to extrapolate this distribution for larger objects. Third, we estimate the client-perceived latency distribution by independently sampling the latency distributions associated with either tuple in

the configuration and considering the minimum. Finally, we adjust this distribution to account for the timeout and probability parameters.

The primary challenge here is the first step: estimating the latency distribution when a VM issues concurrent requests to the local storage service. This turns out to be hard due to the dependencies between concurrent requests. While Figure 3.12 shows the correlation in latencies between two concurrent GET requests to an object at one of Azure's and one of S3's data centers, we also see similar correlations for PUTs and even when the concurrent requests are for different objects. Attempting to model these correlations between concurrent requests by treating the cloud service as a black box did not work well. Therefore, we explicitly model the sources of correlations: concurrent requests may incur the same latency within the storage service if they are served by the same storage server, or incur the same data center network latency if they traverse the same network path.

Modeling replication in storage service. First, at every data center, we use CosTLO's measurements to infer the number of replicas across which the storage service spreads requests to an object. For every pair of concurrent requests issued during CosTLO's measurements, we compute the difference in service latency (i.e., latency within the storage service) between the two requests. We then consider the distribution of this difference across all pairs of concurrent requests to infer the number of replicas in use per object. For example, if the storage service load balances GET requests to an object across 2 replicas, there should be a 50% chance that two concurrent GETs fetch from the same replica, therefore the service latency difference is expected to be 0 half the time. We compare this measured distribution with the expected distribution when the storage service spreads requests across n replicas, where we vary the value of n. We infer the number of replicas used by the service as the value of n for which the estimated and measured distributions most closely match. For example, though both Azure [20] and S3 [10] are known to store 3 replicas

of every object, Figures 3.11(a) and 3.11(b) show that the measured service latency difference distributions closely match GETs being served from 1 replica on Azure and from 2 replicas on S3.

On the other hand, for concurrent GETs or PUTs issued to different objects, on both Azure and S3, we see that the latency within the storage service is uncorrelated across requests. This is likely because cloud storage services store every object on a randomly chosen server (e.g., by hashing the object's name for load balancing [68]), and hence, requests to different objects are likely to be served by different storage servers.

Modeling load balancing in network. Next, we identify whether concurrent requests issued to the storage service incur the same latency over the data center network, or are their network latencies independent of each other. At any data center, we compute the distribution obtained from the minimum of two independent samples of the measured data center network latency distribution for a single request. We then compare this distribution to the measured value of the minimum data center network latency seen across two concurrent requests.

Figure 3.11(c) shows that, on Azure, the distribution obtained by independent sampling closely matches the measured distribution, thus showing that network latencies for concurrent requests are uncorrelated. Whereas, on S3, Figure 3.11(d) shows that the measured distribution for the minimum across two requests is almost identical to the data center network latency component of any single request; this shows that concurrent requests on S3 incur the same network latency.

Estimating VM-to-service latency. Given these models for replication and load balancing, we estimate the end-to-end latency distribution as follows when a VM issues k concurrent requests to the local storage service. If concurrent requests are known to have the same latency over the service's data center network, we sample the measured data center network latency distribution once and use this value for all

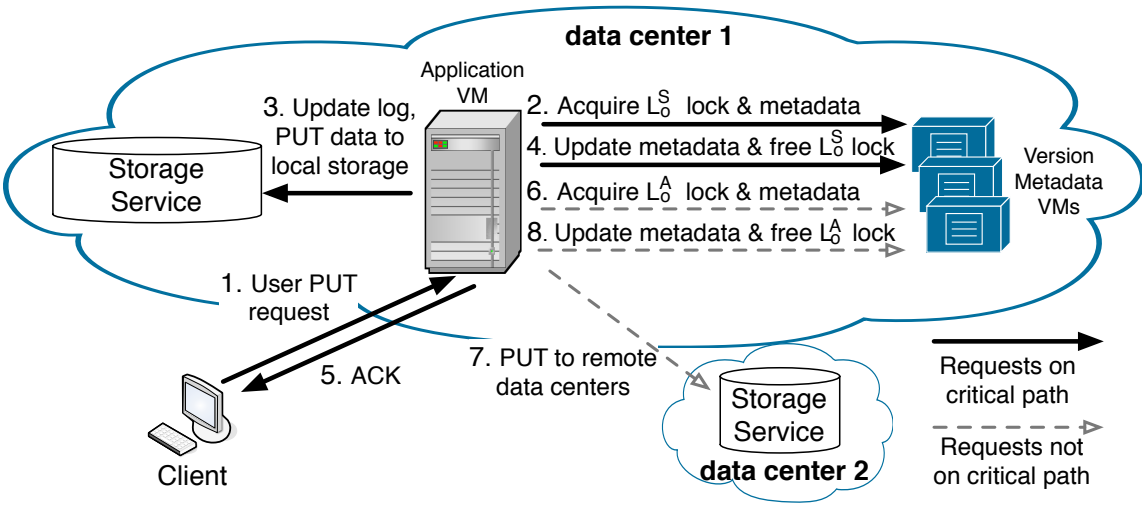


Figure 3.13: Illustration of CosTLO's execution of PUTs.

requests; if not, we independently sample once for each request. If all k requests are to the same object, then we randomly assign every request to one of the replicas of the object, where the number of replicas is identified as described above. If the k requests are for k different objects, then we assume that no two requests are served from the same storage server. In either case, for each storage server, we independently choose a sample from the service latency distribution for a single request and assign that to be the service latency for all requests assigned to that server. Finally, for each of the k requests, we sum up their assigned data center network latency and service latency values, and estimate the end-to-end latency at the VM as the minimum of this sum across the k requests.

Note that our latency estimation models may potentially break down at high storage service load. But, we have not seen any evidence of this so far, since we see the same latency distribution irrespective of whether we issue requests once every three seconds or once every 200ms.

3.4.4 Ensuring data consistency

CosTLO can afford to inform the application that a PUT operation is complete as soon as any of the PUT requests that it issues to serve the operation finish, because the underlying cloud services guarantee that the data written by a completed PUT request will be durable [20, 10]. However, this design decision makes it challenging for CosTLO to ensure that, eventually, all GETs for an object will consistently return the same data. First, if the application issues back-to-back or concurrent PUT operations on the same object, redundant PUT requests that are still pending from a completed PUT operation may potentially overwrite updates written by subsequent PUT operations. Second, if an application VM restarts after only a subset of the PUT requests issued to serve a PUT operation complete, the VM library will not realize if some of the remaining PUT requests fail, thus causing some of the copies of the object to potentially not reflect the latest update to the object.

Figure 3.13 illustrates the execution of PUTs in CosTLO accounting for these concerns. In every data center, CosTLO maintains a set of VMs that store in memory (with a persistent backup) the latest version number and the status of two locks L_o^S and L_o^A for every object o stored in that data center. We use L_o^S for synchronous PUTs to local storage service and L_o^A for asynchronous PUTs to remote storage services. When serving a PUT operation on object o, the VM library first queries the local cluster of CosTLO's VMs to obtain lock L_o^S and learn o's current version. Once it acquires the lock, the library appends to a persistent log (maintained locally on the VM) the update that needs to be written to o and all the PUT requests that the library needs to issue as per the configuration for the client issuing this PUT operation. By appending the status of every response to the log, the library ensures that it knows which PUTs to re-issue, even across VM restarts. Once all PUT requests complete, the library releases lock L_o^S, updating o's version in the process. At some point later, the library attempts to acquire lock L_o^A, and if o's version has not changed by then,

it updates the remaining copies of o and subsequently releases the lock. If o's version has changed, the library just needs to release the lock, since there exists a newer PUT operation on this key and that PUT's asynchronous propagation will suffice to update the remaining copies of o.

Note that, since the application is unaware of the replication of objects across data centers, all PUT operations on an object will be issued by the application's VMs in the same data center. Hence, the VM library needs to acquire locks only from CosTLO's VMs within the local data center, thus ensuring that locking operations add negligible latency. Also note that, when an application issues back-to-back PUT operations, execution of the latter PUT has to wait for the lock L_o^S (for the object o being updated) to be released. This can potentially increase[4] tail latencies if multiple PUT requests need to complete before L_o^S is released. Therefore, in the rare case when an application often issues back-to-back or concurrent PUTs for the same object, the application should choose an SLO that offers no improvement in PUT latency variance; this will ensure that CosTLO executes any PUT operation by issuing a single PUT request.

3.5 Evaluation

We evaluate CosTLO from three perspectives: 1) its ability to satisfy latency SLOs, 2) its cost-effectiveness in doing so, and 3) its efficiency in various respects. We perform our evaluation from the perspective of an application deployed across all of Amazon's data centers. We deploy CosTLO across Azure's and S3's data centers, and use PlanetLab nodes at 120 sites as clients.

[4]Note that we can reduce the extent of this increase in inflation by having CosTLO maintain a lock $L_{o,c}^S$ for every copy c of object o, but we do not present such a design here to keep the discussion simple.

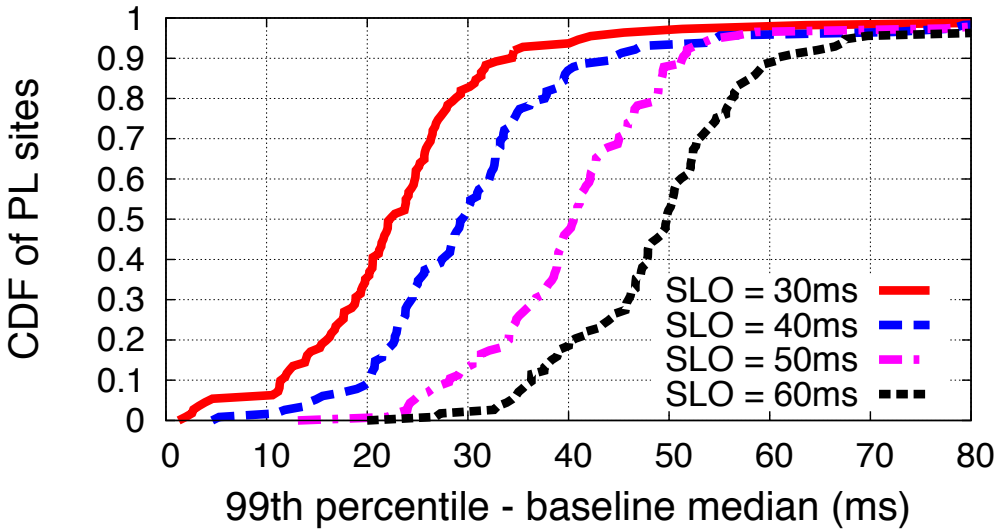


Figure 3.14: Verification of CosTLO's ability to satisfy SLOs.

3.5.1 Ability to satisfy SLOs

SLOs on individual operations. To verify CosTLO's ability to satisfy latency SLOs, we mimic a deployment of Wikipedia using server-side logs of objects requested from the English version of Wikipedia [30]. We randomly select a 1% sample from the datasets for two consecutive weeks. We provide the workload from the first week to ConfSelector as input, and have it select cost-effective configurations for 120 PlanetLab nodes. We then run CosTLO with every node configured in the manner selected by ConfSelector. We replay the workload from the second week, with every GET request assigned to a random PlanetLab node. We repeat this experiment for four SLO values—30ms, 40ms, 50ms, and 60ms. In all cases, since we issue GETs/PUTs to S3 and Azure, our measurements are affected by Internet congestion and by contention with S3's and Azure's customers.

Figure 3.14 shows the distribution of the measured difference between the 99^{th} percentile and baseline median latencies at every PlanetLab node. For all SLOs, the latency variance delivered by CosTLO is within the input SLO on most nodes; without CosTLO, the difference between 99^{th} percentile and baseline median GET latencies is

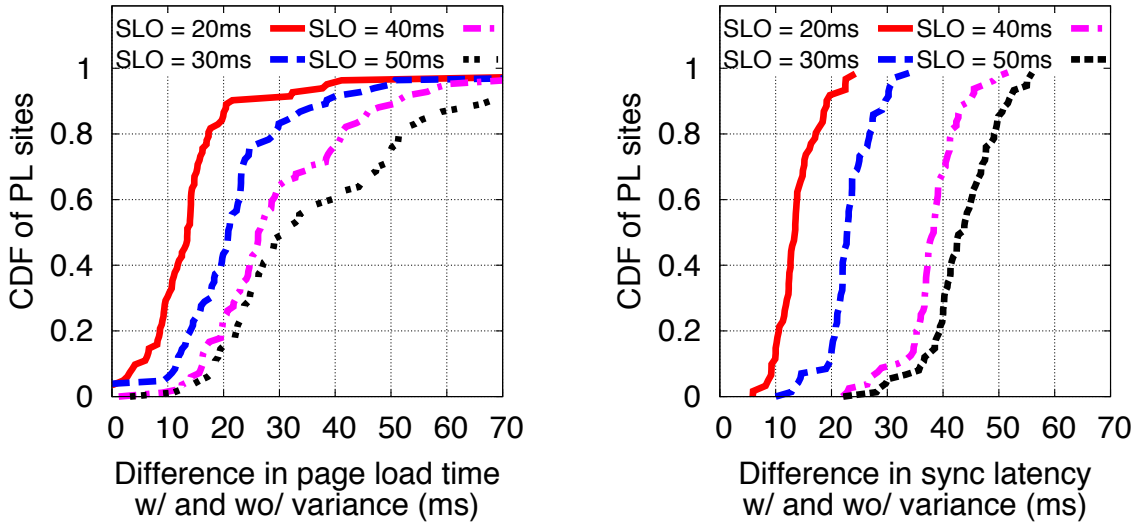
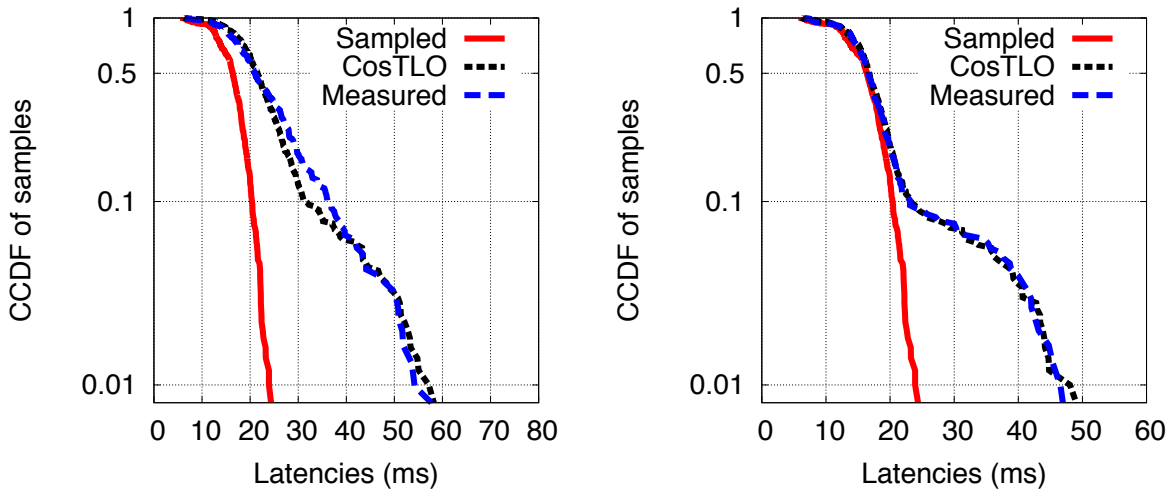


Figure 3.15: CosTLO's ability to satisfy application-specific SLOs for (a) webpage and (b) social network applications.

greater than 60ms for 75% of PlanetLab nodes (Figure 3.1(a)). Latency variance with CosTLO is, in fact, well below the SLO in many cases; due to discontinuous drops in the latency distribution across neighboring configurations, as ConfSelector steps through the configuration space, it often directly transitions from a configuration that violates the SLO to one that exceeds it.

Note that, though we only demonstrate CosTLO's ability to satisfy GET latency SLOs here (because the trace from Wikipedia only contains GETs), CosTLO can also reduce the latency variance for PUTs as described earlier. In contrast, in-memory caching of data can only reduce tail latencies for GETs, but not for PUTs.

Application-specific SLOs. CosTLO's design is easily extensible to handle application-specific SLOs, rather than the SLOs for the latencies of individual PUT/GET operations. Here, we show the results of using CosTLO to reduce user-perceived latencies in the two applications from Section 3.1. In the webpage application, we modify ConfSelector so that it uses the models in Section 3.4.3 to estimate the distribution for the latency incurred when the client library fetches 50 objects in parallel and



(a) Requests to same object

(b) Requests to different objects

Figure 3.16: Accuracy of estimating GET latency distribution for 8 concurrent GET requests from VM to local storage service. (a and b) Comparison of latency distributions in one S3 region.

waits for at least one GET to each of these objects to complete. In the social network application, since we need to estimate latencies from multiple users, we extend the configuration representation in ConfSelector such that it contains the configuration tuples of all of a user's followers. The sync completion time is determined when all followers have at least one GET completed. We use this modified version of ConfSelector to select configurations for all PlanetLab nodes and run CosTLO's client library on every node as per these configurations. Figure 3.15 shows that CosTLO is able to satisfy application-specific SLOs in both applications.

3.5.2 Accuracy of estimating latency distributions

CosTLO is able to meet latency SLOs due to its accurate estimation of the end-to-end latency distributions in any configuration. Our simple approaches of considering the minimum of the latency distributions across data centers and of adding VM-to-prefix and VM-to-service latency distributions work reasonably well; in either case, CosTLO's estimates show less than 15% error for 90% of PlanetLab nodes. Therefore,

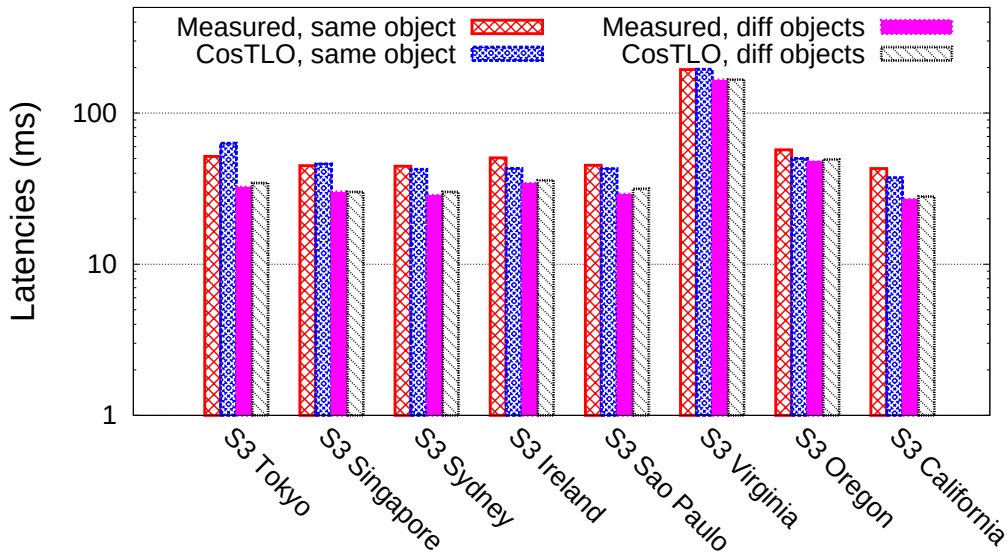


Figure 3.17: Comparison across all S3 regions of 99^{th} percentile latencies.

here we focus on demonstrating the accuracy of our estimation of the latency distribution when a VM concurrently issues a set of requests to the local storage service. Recall that CosTLO only gathers measurements when issuing pairs of concurrent requests. We evaluate its ability to estimate the latency distribution for higher levels of parallelism.

Figures 3.16(a) and 3.16(b) compare the measured and estimated latency distributions when issuing eight concurrent GETs from a VM to the local storage service; all concurrent requests are for the same object in the former and to different objects in the latter. In both cases, our estimated latency distribution closely matches the measured distribution, even in the tail. In contrast, if we estimate the latency distribution for eight concurrent GETs by independently sampling the latency distribution for a single request eight times and considering the minimum, we significantly under-estimate the tail of the distribution. Additionally, Figure 3.17(c) shows that the relative error between the measured and estimated values of the 99^{th} percentile GET latency is less than 5% in the median S3 region; latencies are higher for S3's Virginia data center because it is the most widely used data center in S3.

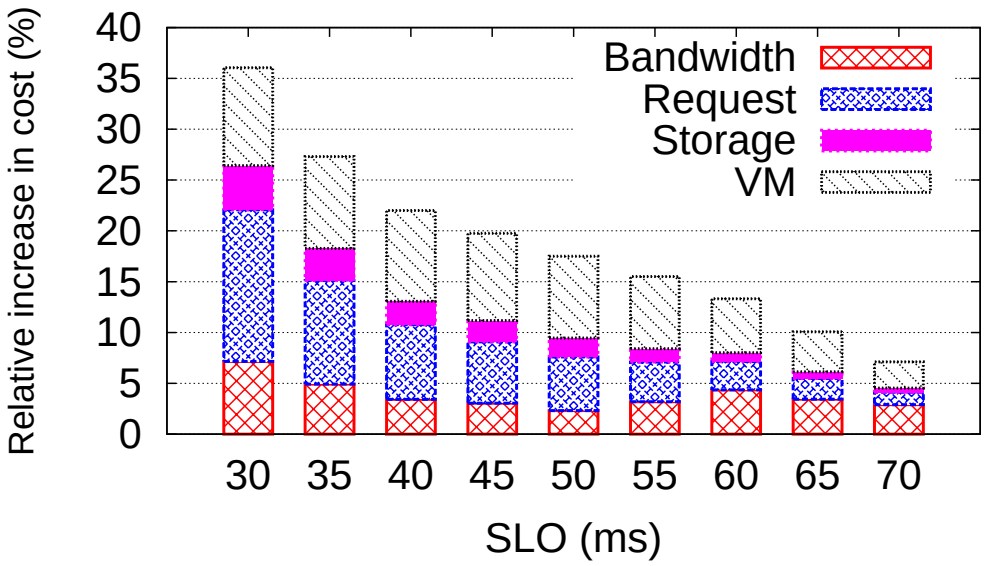
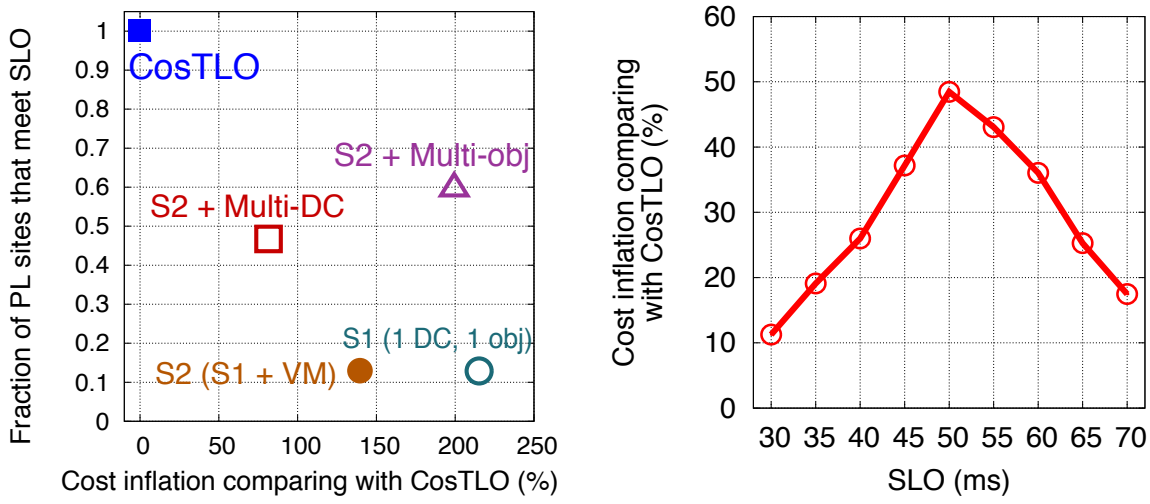


Figure 3.18: CosTLO's cost-effectiveness in satisfying SLOs.

3.5.3 Cost-effectiveness

An application that uses CosTLO incurs additional costs for storing copies of objects, for operations and bandwidth due to redundant requests, and for VMs used either as relays or to manage locks and version numbers. We again use Wikipedia's workload to quantify this overhead on an application provider's costs.

Figure 3.18 shows the relative cost overhead as a function of the latency SLO, with the cost split into its four components. At the higher end of the examined range of SLO values, CosTLO caps tail latency inflation at 70ms—which is less than the inflation observed at the median node when not using CosTLO—with less than 8% increase in cost. As the SLO decreases, i.e., as lower variance is desired, cost increases initially due to an increase in the number of redundant requests. Thereafter, as the SLO further decreases, CosTLO begins to use more relay VMs so that only one copy of any requested object leaves the data center, thus decreasing bandwidth costs at the expense of VM costs. As the SLO decreases further, CosTLO begins concurrently issuing requests to multiple data centers, thus again increasing band-



(a) (b)

Figure 3.19: (a) Utility of CosTLO's components in reducing cost and meeting SLO = 30ms. (b) Cost inflation when not using timeout and probability parameters.

width costs. Storage costs and cost for VMs that manage locks and version numbers remain low for all SLO values, because 1) on both Amazon's and Microsoft's cloud services, storage is significantly cheaper than GET/PUT requests, VMs, and network transfers, and 2) lock status and version numbers for all 70M objects in the English version of Wikipedia fit into the memory of a small instance VM, which costs less than $20 per month on EC2.

3.5.4 Utility of CosTLO's components

CosTLO's ability to satisfy SLOs cost-effectively crucially depends on its *combined use* of various forms of issuing redundancy. We illustrate this in Figure 3.19(a) by comparing CosTLO with several strategies that each use a subset of the dimensions in CosTLO's configuration space. For each strategy, we compute the fraction of PlanetLab nodes for which it is able to satisfy an SLO of 30ms, and across the nodes on which the strategy does meet the SLO, we compare its cost with CosTLO's.

First, the simplest strategy $S1$, which only issues redundant requests to a single copy of any object in the data center closest to any client, can meet the SLO on only

a little over 10% of nodes. Adding the use of relay VMs ($S2$) reduces cost inflation compared to CosTLO from over 200% to less than 150%, but the ability to meet the SLO remains unchanged. We can improve the ability to satisfy the SLO by adding the option of issuing redundant requests either to multiple copies of every object in the closest data center or to multiple data centers. However, the fraction of nodes on which the SLO can be met remains below 60% if we use one of these two options. Only by combining the use of relay VMs, multiple copies of objects, and multiple data centers is CosTLO able to meet the SLO at all nodes, at significantly lower cost.

In addition, we illustrate the utility of CosTLO waiting for a timeout period before issuing redundant requests and issuing redundant requests probabilistically. For every SLO in the range 30ms to 70ms, Figure 3.19(b) compares CosTLO's cost overhead when it uses the timeout and probability parameters versus when it does not. The cost overhead of not using the timeout and probability parameters is low when the SLO is extremely low or extremely high. In the former case, most PlanetLab nodes need to issue redundant requests at all times without any timeout in order to meet the SLO, whereas in the latter case, the SLO is satisfied for most PlanetLab nodes even without redundant requests. However, for many intermediate SLO values—that are neither too loose nor too stringent—not using the timeout and probability parameters increases cost significantly, by as much as 48%.

3.5.5 Efficiency

Measurement cost. The cost associated with CosTLO's measurements depends on the number of latency samples necessary to accurately sample latency distributions. To quantify the stationarity in latencies, we consider a dataset of 200K latency measurements gathered over a week from VMs in every S3 and Azure data center. We then consider subsets of these datasets, varying the number of samples considered. In all datasets, we find that 10K samples are sufficient to obtain a reasonably

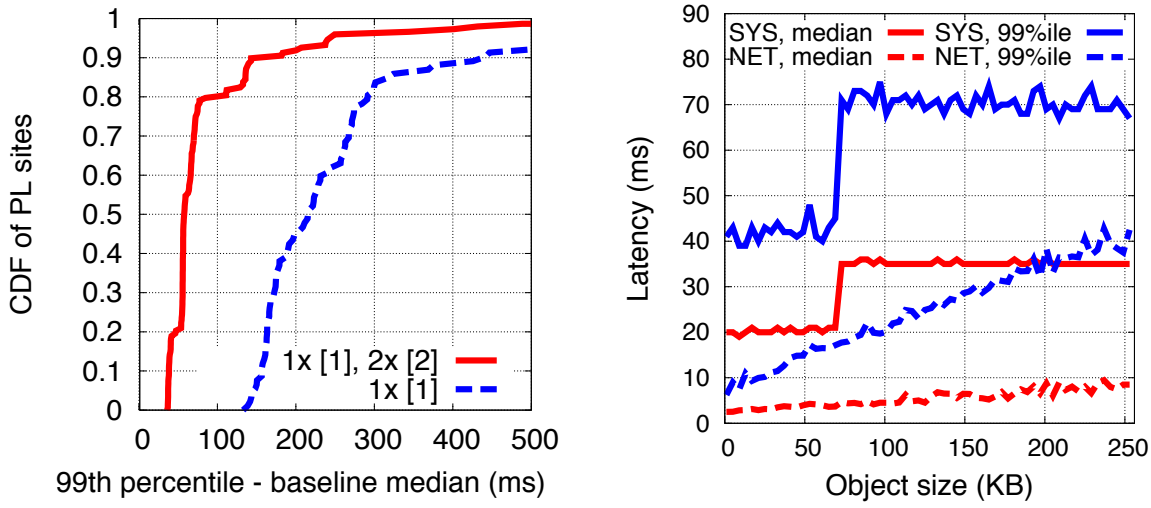
accurate value of the 99^{th} percentile latency. In the ping, GET, and PUT latency measurements, the 99^{th} percentile from a subset of 10K samples was off from the 99^{th} percentile in the entire dataset by only 2.9%, 3.8%, and 2.2% on average.

Thus, at every data center, CosTLO's weekly measurement costs include: 1) 20K GETs and PUTs (since CosTLO gathers data with pairs of concurrent requests), 2) 10K pings to every end-host prefix, and 3) one "small instance" VM (which is sufficient to support this scale of measurements). Accounting for the roughly 120K IP prefixes at the Internet's edge [84], eight S3 data centers, and 13 Azure data centers, these measurements translate to a total cost of $392 per week. These minimal measurement costs are shared across all applications that use CosTLO.

Configuration selection runtime. We run ConfSelector to select the configurations for 120 PlanetLab nodes, and we compute the average runtime per node. We repeat this for SLO values ranging from 20ms to 100ms. Extrapolating the average runtime per node, we estimate that, for all SLO values, ConfSelector needs less than a day to select the configuration for all 120K edge prefixes on a server with 16 cores. Hence, ConfSelector can identify the configurations for a particular week during the last day of the previous week. Moreover, since ConfSelector independently selects configurations for different prefixes, this runtime is easily reduced by parallelizing ConfSelector's execution across a cluster of servers.

3.6 Discussion

Strong consistency. Many applications (e.g., Google Docs) require their underlying storage to offer strong consistency. For such applications, CosTLO uses only strongly consistent storage services, e.g., it can use Azure but not S3. In addition, two modifications are necessary in the execution of a PUT operation on any object o. First, to ensure linearizability of PUTs, the VM library synchronously updates all copies of o before releasing lock L_o^S. Second, instead of the library informing the



(a) (b)

Figure 3.20: (a) CosTLO's utility in reducing latency variance when offering strong consistency; 1 PUT request per copy. (b) latency breakdown for objects of different sizes.

application when any one PUT request completes, the application registers for two callbacks—1) *quorumPUTsDone*, for when at least one PUT request each completes on a quorum of o's copies, and 2) *allPUTsDone*, when all PUTs finish. The *quorumPUTsDone* callback indicates to the application that subsequent GET operations on o will fetch the latest version, if the client library waits for responses from a quorum of copies when serving GETs.

After these changes, Figure 3.20(a) shows the PUT latency variance offered by CosTLO when every object accessed by a PlanetLab node has one copy and two copies, respectively, in the closest Azure data center and the second closest data center across S3 and Azure; for this analysis, we ignore that S3 does not offer strong consistency. Despite having to wait for PUT requests on a quorum of copies to complete, and though a quorum of every object's copies are stored in a different data center than the application VMs that issue PUT operations on the object, CosTLO more than halves the PUT latency inflation for the median node. This again highlights the utility of redundant copies and requests, and of CosTLO's use of multiple cloud

services.

Latency estimation for larger objects. One can potentially extend CosTLO's approach for estimating latency variance to larger objects as follows. We conduct measurements on objects from 1 KB to 256 KB when issuing one GET request at a time to each object from a local VM, and Figure 3.20(b) shows the results from one data center. We see that network latency is proportional to object size, and storage service latency is a step function of object size. Although different data centers may have different step functions (we observe that some data centers have the same storage service latency distribution for all sizes in the 256 KB range), the smallest range that has a fixed storage service latency distribution is until 64 KB, which is a typical block size in distributed storage systems [63]. Therefore, to estimate latencies for objects of different sizes, we can leverage the fact that objects with the same number of blocks have the same storage service latency distribution.

Scale of adoption. CosTLO's approach of issuing redundant requests makes it unviable if all applications adopt it. However, we believe that increasing adoption of CosTLO will emphasize the demand for latency SLOs and spur cloud providers to suitably modify their services. In the interim, CosTLO minimizes the cost overhead incurred by application providers who seek to improve predictability in user-perceived latencies without having to wait for any changes to cloud services. Moreover, cloud service providers have little control over reducing variance in the latency on the Internet path between end-hosts and their data centers.

3.7 Summary

Our measurements of the Azure and S3 storage highlight the high variance in latencies offered by these services. To enable applications to improve predictability, without having to wait for these services to modify their infrastructure, we have designed and implemented CosTLO, a framework that requires minimal changes to

applications. Based on several insights about the causes for latency variance on cloud storage that we glean from our measurements, our design of CosTLO judiciously combines several instantiations of the approach of issuing redundant requests. Our results show that, despite the unbounded configuration space and opaque cloud service architectures, CosTLO cost-effectively enables applications to meet latency SLOs.

CHAPTER IV

Cost-effective Data Placement in the Cloud

Today, several cloud providers offer storage as a service. Amazon S3 [6], Google Cloud Storage (GCS) [25], and Microsoft Azure [13] are notable examples. All of these services provide storage in several data centers distributed around the world. Customers can store and retrieve data via PUTs and GETs without dealing with the complexities associated with setting up and managing the underlying storage infrastructure.

Ideally, web applications should be able to provide low-latency service to their clients by leveraging the distributed locations for storage offered by these services. For example, a photo sharing web service deployed across Amazon's data centers may serve every user from the data center closest to the user.

However, a number of realities complicate this goal. First, almost every storage service offers an isolated pool of storage in each of its data centers, leaving replication across data centers to applications. For example, even though Amazon's cloud platform has eight data centers, customers of its S3 storage service need to read/write data at each data center separately. If a user in Seattle uploads a photo to a photo sharing web service deployed on all of Amazon's data centers, the application will have to replicate this photo to each data center to ensure low latency access to the photo for users in other locations.

Second, while replicating all objects to all data centers can ensure low latency access [96], that approach is costly and may be inefficient. Some applications may value lower costs over the most stringent latency bounds, different applications may demand different degrees of data consistency, some objects may only be popular in some regions, and some clients may be near to multiple data centers, any of which can serve them quickly. All these factors mean that no single deployment provides the best fit for all applications and all objects. Since cloud providers do not provide a centralized view of storage with rich semantics, every application needs to reason on its own about where and how to replicate data to satisfy its latency goals and consistency requirements at low cost.

To address this problem, we design and implement SPANStore ("Storage Provider Aggregating Networked Store"), a key-value store that presents a unified view of storage services present in several geographically distributed data centers. Unlike existing geo-replicated storage systems [96, 97, 114, 66], our primary focus in developing SPANStore is to minimize the cost incurred by latency-sensitive application providers. Three key principles guide our design of SPANStore to minimize cost.

First, SPANStore spans data centers across multiple cloud providers due to the associated performance and cost benefits. On one hand, SPANStore can offer lower latencies because the union of data centers across multiple cloud providers results in a geographically denser set of data centers than any single provider's data centers. On the other hand, the cost of storage and networking resources can significantly differ across cloud providers. For example, when an application hosted in the US serves a user in China, storing the user's data in S3's California data center is more expensive ($0.026 per GB) than doing so in GCS ($0.023 per GB), whereas the price for serving data to the user has the opposite trend ($0.09 per GB in S3 vs. $0.12 per GB in GCS). SPANStore exploits these pricing discrepancies to drive down the cost incurred in satisfying application providers' latency, consistency, and fault tolerance

goals.

Second, to minimize cost, SPANStore judiciously determines where to replicate every object and how to perform this replication. Replicating objects to a larger diversity of locations reduces GET latencies by moving copies closer to clients, but this additional replication increases both storage costs and the expenses necessary to pay for the bandwidth required to propagate updates. For every object that it stores, SPANStore addresses this trade-off by taking into consideration several factors: the anticipated workload for the object (i.e., how often different clients access it), the latency guarantees specified by the application that stored the object in SPANStore, the number of failures that the application wishes to tolerate, the level of data consistency desired by the application (e.g., strong versus eventual), and the pricing models of storage services that SPANStore builds upon.

Lastly, SPANStore further reduces cost by minimizing the compute resources necessary to offer a global view of storage. These compute resources are used to implement tasks such as two-phase locking while offering strong consistency and propagation of updates when offering eventual consistency. To keep costs low, we ensure that all data is largely exchanged directly between application virtual machines (VMs) and the storage services that SPANStore builds upon; VMs provisioned by SPANStore itself—rather than by application provider—are predominantly involved only in metadata operations.

We have developed and deployed a prototype of SPANStore that spans all data centers in the S3, Azure, and GCS storage services. In comparison to alternative designs for geo-replicated storage (such as using the data centers in a single cloud service or replicating every object in every data center from which it is accessed), we see that SPANStore can lower costs by over 10x in a range of scenarios. We have also ported two applications with disparate consistency requirements (a social networking web service and a collaborative document editing application), and we

find that SPANStore is able to meet latency goals for both applications.

4.1 Problem formulation

Our overarching goal in developing SPANStore is to enable applications to interact with a single storage service, which underneath the covers uses several geographically distributed storage services. Here, we outline our vision for how SPANStore simplifies application development and the challenges associated with minimizing cost.

4.1.1 Setting and utility

We assume an application employing SPANStore for data storage uses only the data centers of a single cloud service to host its computing instances, even though (via SPANStore) it will use multiple cloud providers for data storage. This is because different cloud computing platforms significantly vary in the abstractions that applications can build upon; an application's implementation will require significant customization in order for it be deployable across multiple cloud computing platforms. For example, applications deployed on Amazon EC2 can utilize a range of services such as Simple Queueing Service, Elastic Beanstalk, and Elastic Load Balancing. Other cloud computing platforms such as Azure and GCE do not offer direct equivalents of these services.

To appreciate the utility of developing SPANStore, consider a collaborative document editing web service (similar to Google Docs) deployed across all of EC2's data centers. Say this application hosts a document that is shared among three users who are in Seattle, China, and Germany. The application has a range of choices as to where this document could be stored. One option is to store copies of the document at EC2's Oregon, Tokyo, and Ireland data centers. While this ensures that GET operations have low latencies, PUTs will incur latencies as high as 560ms since updates need to be applied to all copies of the document in order to preserve the document's

consistency. Another option is to maintain only one copy of the document at EC2's Oregon data center. This makes it easier to preserve consistency and also reduces PUT latencies to 170ms, but increases GET latencies to the same value. A third alternative is to store a single copy of the document at Azure's data center on the US west coast. This deployment reduces PUT and GET latencies to below 140ms and may significantly reduce cost, since GET and PUT operations on EC2 cost 4x and 50x, respectively, what they do on Azure.

Thus, every application has a range of replication strategies to choose from, each of which presents a different trade-off between latency and cost. Today, the onus of choosing from these various options on a object-by-object basis is left to individual application developers. By developing SPANStore, we seek to simplify the development of distributed applications by presenting a single view to geo-replicated storage and automating the process of navigating this space of replication strategies.

4.1.2 Goals

Four objectives guide our synthesis of geographically distributed storage services into a single key-value store.

- **Minimize cost.** Our primary goal is to minimize costs for applications that use SPANStore. For this, we need to minimize the total cost across SPANStore's use of 1) the storage services that it unifies, and 2) the compute resources offered by the corresponding providers.
- **Respect latency SLOs.** We design SPANStore to serve latency-sensitive applications that have geographically distributed deployments, and therefore stand to benefit from the geo-replicated storage offered by SPANStore. However, the service level objectives (SLOs)[1] for GET/PUT latencies may vary across applications. While minimizing cost for any particular application, SPANStore must strive to

[1]We use the term SLO instead of SLA because violations of the latency bounds are not fatal, but need to be minimized.

meet applications' latency goals.

- **Flexible consistency.** Different applications can also vary in their requirements for the consistency of the data they store. For example, a collaborative document editing service requires strong consistency whereas eventual consistency may suffice for a social networking service. SPANStore should respect requirements for strong consistency and exploit cases where eventual consistency suffices to offer lower latencies.
- **Tolerate failures.** Applications seek to tolerate failures of data centers and Internet paths. However, applications may differ in the cost that they are willing to bear for increased fault tolerance. SPANStore should account for an application's fault tolerance requirements while storing objects written by the application.

4.1.3 Challenges

Satisfying these goals is challenging for several reasons.

Inter-dependencies between goals. To minimize cost, it is critical that SPANStore *jointly* considers an application's latency, consistency, and fault tolerance requirements. For example, if an application desires strongly consistent data, the most cost-effective strategy is for SPANStore to store all of the application's data in the cheapest storage service. However, serving all PUTs and GETs from a single replica may violate the application's latency requirements, since this replica may be distant from some of the data centers on which the application is deployed. On the other hand, replicating the application's data at all data centers in order to reduce PUT/GET latencies will increase the cost for SPANStore to ensure strong consistency of the data.

Dependence on workload. Even if two applications have the same latency, fault tolerance, and consistency requirements, the most cost-effective solution for storing their data may differ. The lowest cost configuration for replicating any object

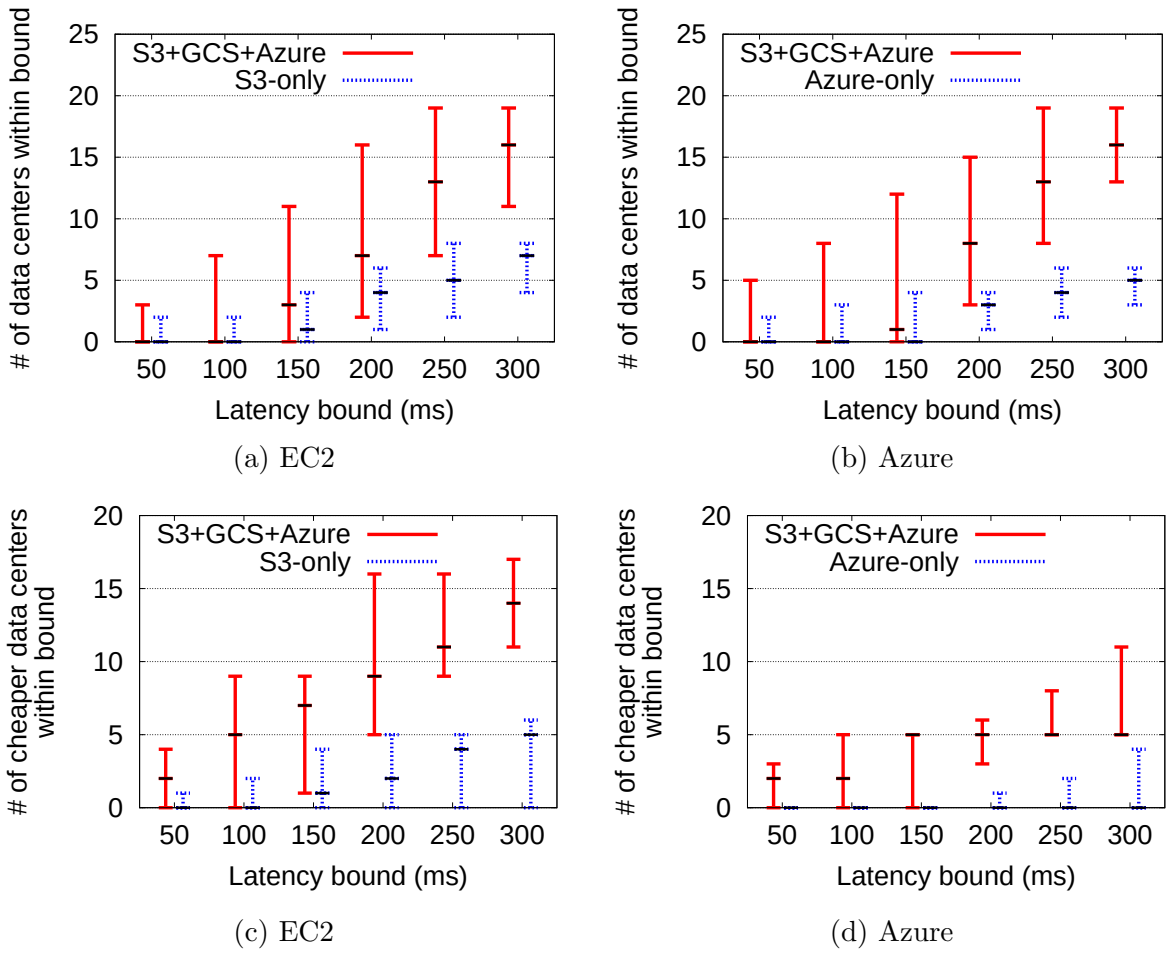
depends on several properties of an application's workload for that object:

- The set of data centers from which the application accesses the object, e.g., an object accessed only by users within the US can be stored only on data centers in the US, whereas another object accessed by users worldwide needs wider replication.
- The number of PUTs and GETs issued for that object at each of these data centers, e.g., to reduce network transfer costs, it is more cost-effective to replicate the object more (less) if the workload is dominated by GETs (PUTs).
- The temporal variation of the workload for the object, e.g., the object may initially receive a high fraction of PUTs and later be dominated by GETs, thus requiring a change in the replication strategy for the object.

Multi-dimensional pricing. Cost minimization is further complicated by the fact that any storage service prices its use based on several metrics: the amount of data stored, the number of PUTs and GETs issued, and the amount of data transferred out of the data center in which the service is hosted. No single storage service is the cheapest along all dimensions. For example, one storage service may offer cheap storage but charge high prices for network transfers, whereas another may offer cheap network bandwidth but be expensive per PUT and GET to the service. Moreover, some storage services (e.g., GCS) charge for network bandwidth based on the location of the client issuing PUTs and GETs.

4.2 Why multi-cloud?

A key design decision in SPANStore is to have it span the data centers of *multiple* cloud service providers. In the previous chapter, we show that how to use multi-cloud to reduce latency variance of accessing single objects stored in cloud storage. In this section, we motivate our design decision of leveraging multi-cloud deployments by presenting measurements which demonstrate that deploying across multiple cloud



(a) EC2 (b) Azure

(c) EC2 (d) Azure

Figure 4.1: For applications deployed on a single cloud service (EC2 or Azure), a storage service that spans multiple cloud services offers a larger number of data centers (a and b) and more cheaper data centers (c and d) within a latency bound.

providers can potentially lead to reduced wide area latencies for clients and reduced cost for applications.

4.2.0.1 Lower latencies

We first show that using multiple cloud providers can enable SPANStore to offer lower GET/PUT latencies. For this, we instantiate VMs in each of the data centers in EC2, Azure, and GCE. From the VM in every data center, we measure GET latencies to the storage service in every other data center once every 5 minutes for a week. We consider the latency between a pair of data centers as the median of the

measurements for that pair.

Figure 4.1 shows how many other data centers are within a given latency bound of each EC2 [4.1(a)] and Azure [4.1(b)] data center. These graphs compare the number of nearby data centers if we only consider the single provider to the number if we consider all three providers—Amazon, Google, and Microsoft. For a number of latency bounds, either graph depicts the minimum, median, and maximum (across data centers) of the number of options within the latency bound.

For nearly all latency bounds and data centers, we find that deploying across multiple cloud providers increases the number of nearby options. SPANStore can use this greater choice of nearby storage options to meet tighter latency SLOs, or to meet a fixed latency SLO using fewer storage replicas (by picking locations nearby to multiple frontends). Intuitively, this benefit occurs because different providers have data centers in different locations, resulting in a variation in latencies to other data centers and to clients.

4.2.0.2 Lower cost

Deploying SPANStore across multiple cloud providers also enables it to meet latency SLOs at potentially lower cost due to the discrepancies in pricing across providers. Figures 4.1(c) and 4.1(d) show, for each EC2 and Azure data center, the number of other data centers within a given latency bound that are cheaper than the local data center along some dimension (storage, PUT/GET requests, or network bandwidth). For example, nearby Azure data centers have similar pricing, and so, no cheaper options than local storage exist within 150ms for Azure-based services. However, for the majority of Azure-based frontends, deploying across all three providers yields multiple storage options that are cheaper for at least some operations. Thus, by judiciously combining resources from multiple providers, SPANStore can use these cheaper options to reduce costs.

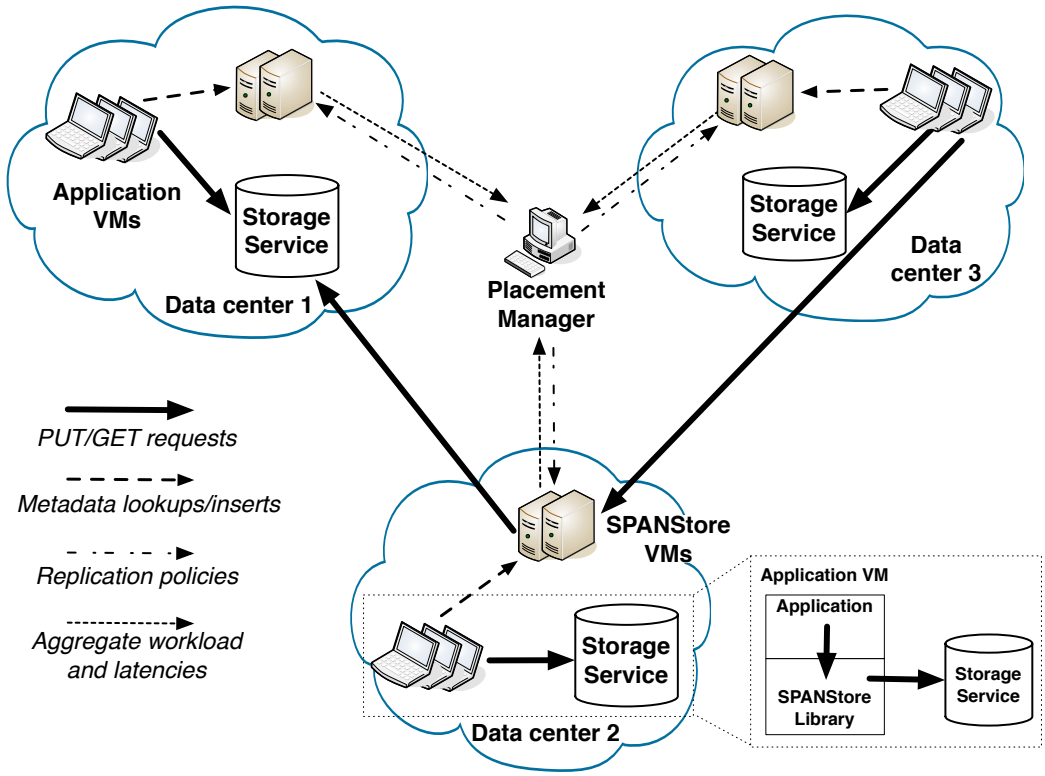


Figure 4.2: Overview of SPANStore's architecture.

Note that leveraging these price discrepancies across clouds to reduce web service cost is unlikely to result in all load being concentrated at the cheapest data center, which could cause pricing to equalize [78] across cloud providers. This is because workload and performance requirements vary across web services, which can result in completely distinct replication policies and storage request execution configurations. Therefore, different web services are likely to exploit pricing discrepancies in differing ways. Moreover, certain distinctions in factors such as cloud system implementations and cloud-ISP relationships will also cause the price discrepancies to remain in the near future.

4.3 Overview of SPANStore

We design SPANStore such that every application uses a separate deployment of SPANStore. Figure 4.2 summarizes SPANStore's deployment for any particular

application. At every data center in which the application is deployed, the application issues PUT and GET requests for objects to a SPANStore library that the application links to. The SPANStore library serves these requests by 1) looking up in-memory metadata stored in the SPANStore-instantiated VMs in the local data center, and thereafter 2) issuing PUTs and GETs to underlying storage services. To issue PUTs to remote storage services, the SPANStore library may choose to relay PUT operations via SPANStore VMs in other data centers.

The manner in which SPANStore VMs should serve PUT/GET requests for any particular object is dictated by a central PlacementManager (*PMan*). We divide time into fixed-duration epochs; an epoch lasts one hour in our current implementation. At the start of every epoch, all SPANStore VMs transmit to *PMan* a summary of the application's workload and latencies to remote data centers measured in the previous epoch. *PMan* then computes the optimal replication policies to be used for the application's objects based on its estimate of the application's workload in the next epoch and the application's latency, consistency, and fault tolerance requirements. In our current implementation, *PMan* estimates the application's workload in a particular epoch to be the same as that observed during the same period in the previous week. *PMan* then communicates the new replication policies to SPANStore VMs at all data centers. These replication policies dictate how SPANStore should serve the application's PUTs (where to write copies of an object and how to propagate updates) and GETs (where to fetch an object from) in the next epoch.

Table 4.1 summarizes the various insights that SPANStore leverages to reduce cost of using cloud storage services.

4.4 Determining replication policies

In this section, we discuss *PMan*'s determination of the replication policies used in SPANStore's operation. We first describe the inputs required by *PMan* and the

Insight	Overview	Section
Multi-cloud deployment	Lowering cost due to the discrepancies in pricing across providers	§ 4.2
Using aggregate workload per access set	Leveraging application-level hints to group objects based on their access patterns for better workload predictability	§ 4.4.1
Relay propagation	Reducing cost by relaying propagation at data centers with cheaper data transfer cost	§ 4.5
Cost-optimal configuration formulation	Formulating the problem of determining the cost-optimal replication policy as a mixed integer program to address the trade-off between storage, networking and request costs	§ 4.4.2 & § 4.4.3

Table 4.1: Summary of insights to reduce cost in SPANStore.

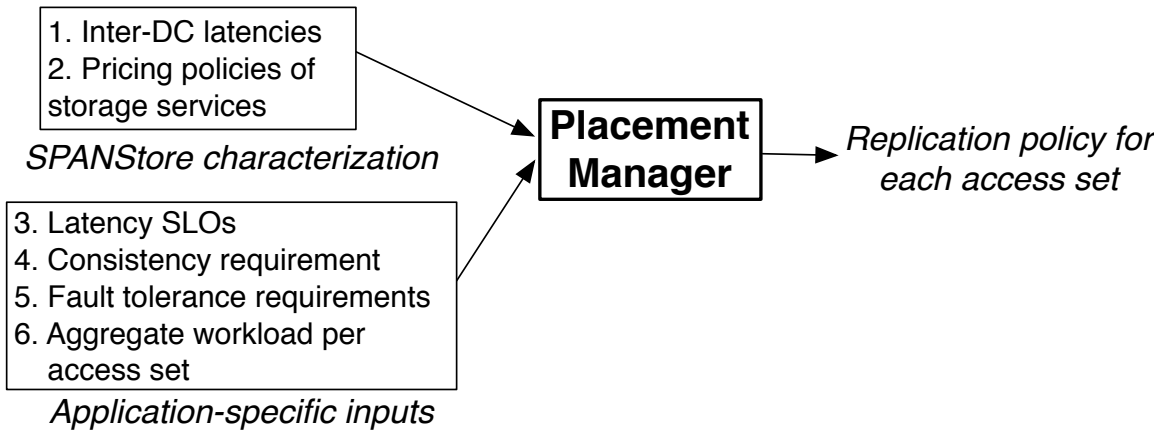


Figure 4.3: Overview of *PMan*'s inputs and output.

format in which it outputs replication policies. We then present the algorithms used by *PMan* in two different data consistency scenarios.

4.4.1 Inputs and output

As shown in Figure 4.3, *PMan* requires three types of inputs: 1) a characterization of SPANStore's deployment, 2) the application's latency, fault tolerance, and consistency requirements, and 3) a specification of the application's workload.

Characterization of SPANStore deployment. *PMan* requires two pieces of information about SPANStore's deployment. First, it takes as input the distribution of latencies between every pair of data centers on which SPANStore is deployed.

These latencies include measurements of PUTs, GETs, and pings issued from a VM in one data center to the storage service or a VM in another data center. Second, *PMan* needs the pricing policy for the resources used by SPANStore. For each data center, we specify the price per byte of storage, per PUT request, per GET request, and per hour of usage for the type of virtual machine used by SPANStore in that data center. We also specify, for each pair of data centers, the price per byte of network transfer from one to the other, which is determined by the upload bandwidth pricing at the source data center.

Application requirements. *PMan* also needs as input the application's latency, data consistency, and fault tolerance requirements. For the latency goals, we let the application separately specify SLOs for latencies incurred by PUT and GET operations. Either SLO is specified by a latency bound and the fraction of requests that should incur a latency less than the specified bound.

To capture consistency needs, we ask the application developer to choose between strong and eventual consistency. In the strong consistency case, we provide linearizability [80], i.e., all PUTs for a particular object are ordered and any GET returns the data written by the last committed PUT for the object. In contrast, if an application can make do with eventual consistency, SPANStore can satisfy lower latency SLOs. Our algorithms for the eventual consistency scenario are extensible to other consistency models such as causal consistency [96] by augmenting data transfers with additional metadata.

In both the eventual consistency and strong consistency scenarios, the application developer can specify the number of failures—either of data centers or of Internet paths between data centers—that SPANStore should tolerate. As long as the number of failures is less than the specified number, SPANStore should ensure the availability of all GET and PUT operations while also satisfying the application's consistency and latency requirements. When the number of failures exceeds the specified number,

SPANStore may make certain operations unavailable or violate latency goals in order to ensure that consistency requirements are preserved.

Workload characterization. Lastly, *PMan* accounts for the application's workload in two ways. First, for every object stored by an application, we ask the application to specify the set of data centers from which it will issue PUTs and GETs for the object. We refer to this as the *access set* for the object. An application can determine the access set for an object based on the sharing pattern of that object across users. For example, a collaborative online document editing web service knows the set of users with whom a particular document has been shared. The access set for the document is then the set of data centers from which the web service serves these users. In cases where the application itself is unsure which users will access a particular object (e.g., in a file hosting service like RapidShare), it can specify the access set of an object as comprising all data centers on which the application is deployed; this uncertainty will translate to higher costs. In this work, we consider every object as having a fixed access set over its lifetime. SPANStore could account for changes in an object's access set over time, but at the expense of a larger number of latency SLO violations; we defer the consideration of this scenario to future work.

Second, SPANStore's VMs track the GET and PUT requests received from an application to characterize its workload. Since the GET/PUT rates for individual objects can exhibit bursty patterns (e.g., due to flash crowds), it is hard to predict the workload of a particular object in the next epoch based on the GETs and PUTs issued for that object in previous epochs. Therefore, SPANStore instead leverages the stationarity that typically exists in an application's aggregate workload, e.g., many applications exhibit diurnal and weekly patterns in their workload [46, 58]. Specifically, at every data center, SPANStore VMs group an application's objects based on their access sets. In every epoch, for every access set, the VMs at a data center report to *PMan* 1) the number of objects associated with that access set and

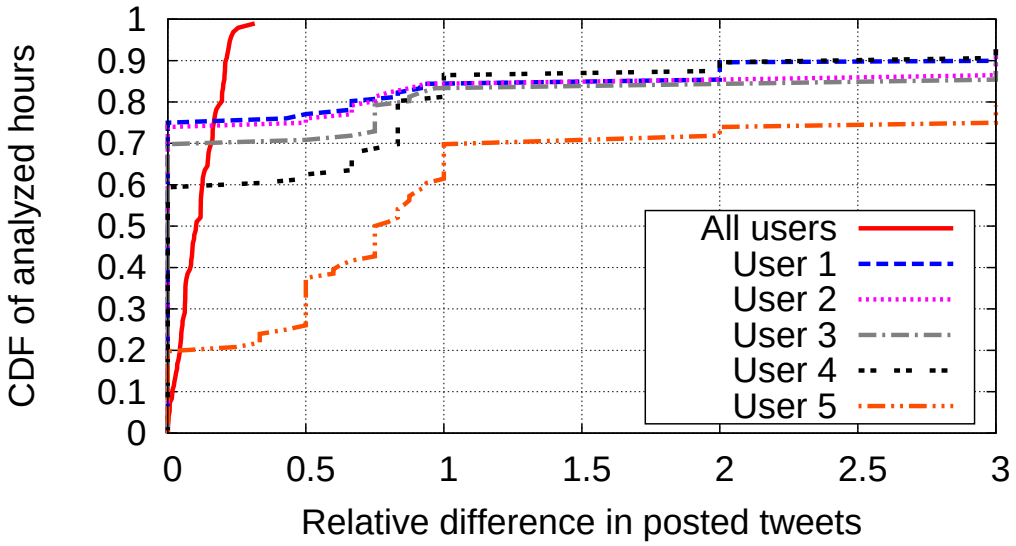


Figure 4.4: Comparison at different granularities of the stationarity in the number of posted tweets.

the sum of the sizes of these objects, and 2) the aggregate number of PUTs and GETs issued by the application at that data center for all objects with that access set.

To demonstrate the utility of considering aggregate workloads in this manner, we analyze a Twitter dataset that lists the times at which 120K users in the US posted on Twitter over a month [95]. We consider a scenario in which every user is served from the EC2 data center closest to the user, and consider every user's Twitter timeline to represent an object. When a user posts a tweet, this translates to one PUT operation on the user's timeline and one PUT each on the timelines of each of the user's followers. Thus, the access set for a particular user's timeline includes the data centers from which the user's followers are served.

Here, we consider those users whose timelines have their access set as all EC2 data centers in the US. Figure 4.4 presents the stationarity in the number of PUTs when considering the timelines of all of these users in aggregate and when considering five popular individual users. In either case, we compare across two weeks the number of PUTs issued in the same hour on the same day of the week, i.e., for every hour, we compute the difference between the number of tweets in that hour and the number of

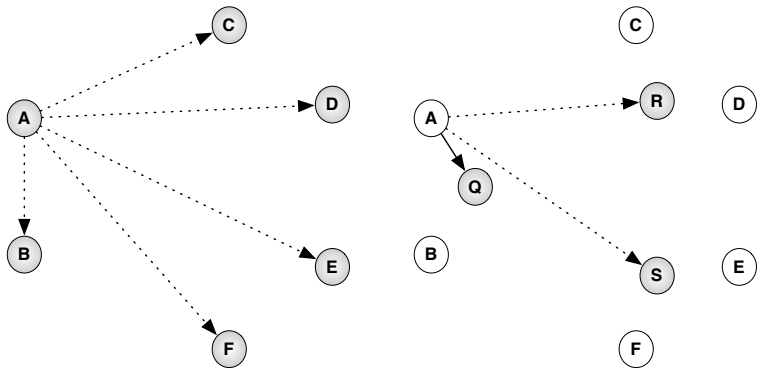
tweets in the same hour the previous week, normalized by the latter value. Aggregate across all users, the count for every hour is within 50% of the count for that hour the previous week, whereas individual users often exhibit 2x and greater variability. The greater stationarity in the aggregate workload thus enables more accurate prediction based on historical workload measurements.

Replication policy. Given these inputs, at the beginning of every epoch, *PMan* determines the replication policy to be used in the next epoch. Since we capture workload in aggregate across all objects with the same access set, *PMan* determines the replication policy separately for every access set, and SPANStore employs the same replication strategy for all objects with the same access set. For any particular access set, the replication policy output by *PMan* specifies 1) the set of data centers that maintain copies of all objects with that access set, and 2) at each data center in the access set, which of these copies SPANStore should read from and write to when an application VM at that data center issues a GET or PUT on an object with that access set.

Thus, the crux of SPANStore's design boils down to: 1) in each epoch, how does *PMan* determine the replication policy for each access set, and 2) how does SPANStore enforce *PMan*-mandated replication policies during its operation, accounting for failures and changes in replication policies across epochs? We next describe separately how SPANStore addresses the first question in the eventual consistency and strong consistency cases, and then tackle the second question in the next section.

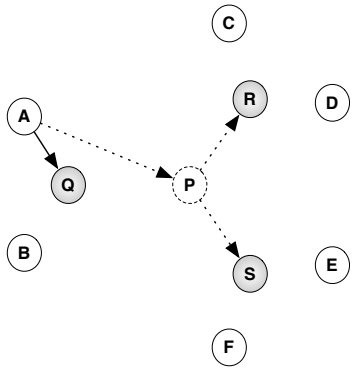
4.4.2 Eventual consistency

When the application can make do with eventual consistency, SPANStore can trade-off costs for storage, PUT/GET requests, and network transfers. To see why this is the case, let us first consider the simple replication policy where SPANStore maintains a copy of every object at each data center in that object's access set (as

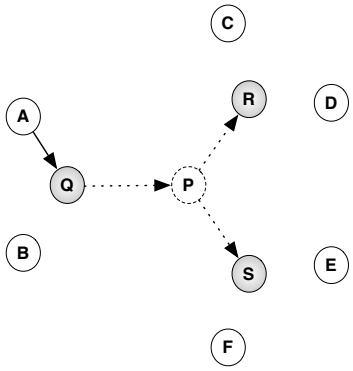


(a) Full replication minimizes latency

(b) Partial replication reduces costs



(c) Relayed update propagation reduces bandwidth costs



(d) Two-hop relaying may reduce costs further

Figure 4.5: When eventual consistency suffices, illustration of different replication policies for access set {A, B, C, D, E, F}. In all cases, we show how PUTs from data center A are propagated. Shaded circles are data centers that host replicas, and dotted circles represent data centers that propagate updates. Solid arrows correspond to transfers that impact PUT latencies, and dotted arrows represent asynchronous propagation.

shown in Figure 4.5(a)). In this case, a GET for any object can be served from the local storage service. Similarly, PUTs can be committed to the local storage service and updates to an object can be propagated to other data centers in the background; SPANStore considers a PUT as complete after writing the object to the local storage service because of the durability guarantees offered by storage services. By serving PUTs and GETs from the storage service in the same data center, this replication policy minimizes GET/PUT latencies, the primary benefit of settling for eventual consistency. In addition, serving GETs from local storage ensures that GETs do not

incur any network transfer costs.

However, as the size of the access set increases, replicating every object at every data center in the access set can result in high storage costs. Furthermore, as the fraction of PUTs in the workload increase, the costs associated with PUT requests and network transfers increase as more copies need to be kept up-to-date.

To reduce storage costs and PUT request costs, SPANStore can store replicas of an object at fewer data centers, such that every data center in the object's access set has a nearby replica that can serve GETs/PUTs from this data center within the application-specified latency SLOs. For example, as shown in Figure 4.5(b), instead of storing a local copy, data center A can issue PUTs and GETs to the nearby replica at Q.

However, SPANStore may incur unnecessary networking costs if it propagates a PUT at data center A by having A directly issue a PUT to every replica. Instead, we can capitalize on the discrepancies in pricing across different cloud services and relay updates to the replicas via another data center that has cheaper pricing for upload bandwidth. For example, in Figure 4.5(c), SPANStore reduces networking costs by having A send each of its updates to P, which in turn issues a PUT for this update to all the replicas that A has not written to directly. In some cases, it may be even more cost-effective to have the replica to which A commits its PUTs relay updates to a data center that has cheap network pricing, which in turn PUTs the update to all other replicas, e.g., as shown in Figure 4.5(d).

PMan addresses this trade-off between storage, networking, and PUT/GET request costs by formulating the problem of determining the replication policy for a given access set AS as a mixed integer program (shown in Algorithm 1; for simplicity, we present the formulation without including VM costs). For every data center $i \in AS$, *PMan* chooses $f + 1$ data centers (out of all those on which SPANStore is deployed) which will serve as the replicas to which i issues PUTs and GETs (*line 27*).

Algorithm 1 Replication policy selection for eventual consistency.

Inputs:
T = Duration of epoch
AS = Set of data centers that issue PUTs and GETs
f = Number of failures that SPANStore should tolerate
SLO, p = SLO on p^{th} percentile of PUT/GET latencies
$L^C_{ij} = p^{th}$ percentile latency between VMs in data centers i and j
$L^S_{ij} = p^{th}$ percentile latency between a VM in data center i and the storage service in data center j
$PUTs_i, GETs_i$ = Total no. of PUTs and GETs issued at data center i across all objects with access set AS
$Size^{avg}, Size^{total}$ = Avg. and total size of objects with access set AS
$Price^{GET}_i, Price^{PUT}_i, Price^{Storage}_i$ = Prices at data center i per GET, per PUT, and per byte per hour of storage
$Price^{Net}_{ij}$ = Price per byte of network transfer from data center i to j

Variables:
$\forall i \in AS, j$ s.t. $L^S_{ij} \leq SLO : R_{ij}$ *// whether j is a replica to which i issues PUTs and GETs; only permitted if a VM at i can complete a GET/PUT on the storage service at j within the SLO*
$\forall i \in AS, j, k$ s.t. p^{th} percentile of $L^C_{ij} + L^S_{jk} \leq SLO : P^S_{ijk}$ *// whether i synchronously forwards its PUTs to k via j; only permitted if a VM at i can forward data to the storage service at j via a VM at k within the SLO*
$\forall i \in AS, j, k, m : P^A_{ijkm}$ *// whether i's PUTs are asynchronously forwarded to m via j and k*
$\forall i \in AS, j, k$ s.t. $j \neq k : F_{ijk}$ *// whether PUTs from i are relayed to k via j*
$\forall j : C_j$ *//whether j is a replica*

Objective: Minimize (Cost for GETs + Cost for PUTs + Storage cost)
// GETs issued at i fetch data only from i's replicas
Cost for GETs = $\sum_i GETs_i \cdot (\sum_j (R_{ij} \cdot (Price^{GET}_j + Price^{Net}_{ji} \cdot Size^{avg})))$
// Every PUT is propagated to all replicas
Cost for PUTs = $\sum_i PUTs_i \cdot (\sum_j (C_j \cdot Price^{PUT}_j) + \sum_{j,k} (F_{ijk} \cdot Price^{Net}_{jk} \cdot Size^{Avg}))$
// every replica stores one copy of every object
Storage cost = $\sum_j (C_j \cdot Price^{Storage}_j \cdot Size^{Total} \cdot T)$

Constraints:
// Every data center in the access set has $f+1$ GET/PUT replicas
$\forall i \in AS : \sum_j R_{ij} = f + 1$
// j is a replica if it is a GET/PUT replica for any i in the access set
$\forall j : (C_j = 1)$ iff $(\sum_{i \in AS} R_{ij} > 0)$ [2]
// i's PUTs must be synchronously forwarded to k iff k is one of i's replicas
$\forall i \in AS, k : (R_{ik} = 1)$ iff $(\sum_j P^S_{ijk} > 0)$
// For every data center in access set, its PUTs must reach every replica
$\forall i \in AS, m : C_m = \sum_j (P^S_{ijm} + \sum_k P^A_{ijkm})$
// PUTs from i can be forwarded over the path from j to k as part of either synchronous or asynchronous forwarding
$\forall i, j, k$ s.t. $i \neq j : (F_{ijk} = 1)$ iff $(P^S_{ijk} + \sum_m (P^A_{ijkm} + P^A_{imjk}) > 0)$
$\forall i, k : (F_{iik} = 1)$ iff $(P^S_{iik} + \sum_m P^S_{ikm} + \sum_{m,n} P^A_{ikmn} > 0)$

SPANStore then stores copies of all objects with access set AS at all data centers in the union of PUT/GET replica sets (*line 29*).

The integer program used by *PMan* imposes several constraints on the selection

of replicas and how updates made by PUT operations propagate. First, whenever an application VM in data center i issues a PUT, SPANStore synchronously propagates the update to all the data centers in the replica set for i (*line 31*) and asynchronously propagates the PUT to all other replicas of the object (*line 33*). Second, to minimize networking costs, the integer program used by *PMan* allows for both synchronous and asynchronous propagation of updates to be relayed via other data centers. Synchronous relaying of updates must satisfy the latency SLOs (*lines 13 and 14*), whereas in the case of asynchronous propagation of updates, relaying can optionally be over two hops (*line 15*), as in the example in Figure 4.5(d). Finally, for every data center i in the access set, *PMan* identifies the paths from data centers j to k along which PUTs from i are transmitted during either synchronous or asynchronous propagation (*lines 35 and 36*).

PMan solves this integer program with the objective of minimizing total cost, which is the sum of storage cost and the cost incurred for serving GETs and PUTs. The storage cost is simply the cost of storing one copy of every object with access set AS at each of the replicas chosen for that access set (*line 24*). For every GET operation at data center i, SPANStore incurs the price of one GET request at each of i's replicas and the cost of transferring the object over the network from those replicas (*line 20*). In contrast, every PUT operation at any data center i incurs the price of one PUT request each at all the replicas chosen for access set AS, and network transfer costs are incurred on every path along which i's PUTs are propagated (*line 22*).

4.4.3 Strong consistency

When the application using SPANStore for geo-replicated storage requires strong consistency of data, we rely on quorum consistency [72]. Quorum consistency imposes two requirements to ensure linearizability. For every data center i in an access set,

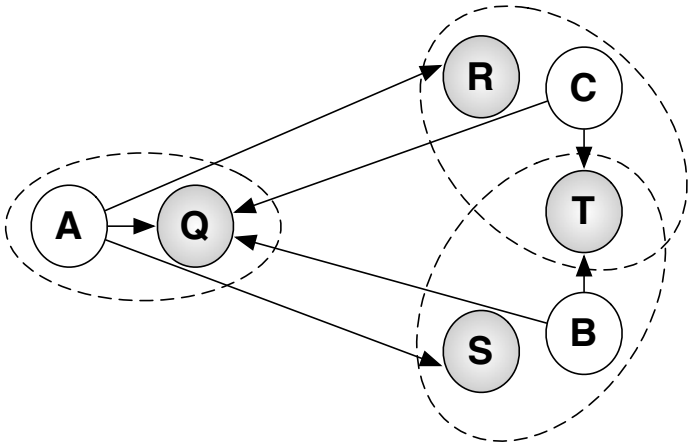


Figure 4.6: Example use of asymmetric quorum sets. Solid unshaded circles represent data centers in the access set, and shaded circles are data centers that host replicas. Directed edges represent transfers to PUT replica sets, and dashed ovals represent GET replica sets.

1) the subset of data centers to which i commits each of its PUTs—the PUT replica set for i—should intersect with the PUT replica set for every other data center in the access set, and 2) the GET replica set for i should intersect with the PUT replica set for every data center in the access set. The cardinality of these intersections should be greater than the number of failures that the application wants SPANStore to tolerate.

In our design, we use asymmetric quorum sets [104] to instantiate quorum consistency as above. With asymmetric quorum sets, the PUT and GET replica sets for any particular data center can differ. We choose to use asymmetric quorum sets due to the non-uniform geographic distribution of data centers. For example, as seen in Figure 4.1(a), EC2 data centers have between 2 and 16 other data centers within 200ms of them. Figure 4.6 shows an example where, due to this non-uniform geographic distribution of data centers, asymmetric quorum sets reduce cost and help meet lower latency SLOs.

The integer program that *PMan* uses for choosing replication policies in the strong consistency setting (shown in Algorithm 2) mirrors the program for the eventual consistency case in several ways: 1) PUTs can be relayed via other data centers to

Algorithm 2 Replication policy selection for strong consistency.

1: **Inputs:**
2: *Same as the inputs in the eventual consistency scenario, except*
3: SLO_{GET}, SLO_{PUT} = SLOs on GET and PUT latencies

4: **Variables:**
5: $\forall i \in AS, j : PR_{ij}$ *// whether j is in i's PUT replica set*
6: $\forall i \in AS, j$ s.t. $L^S_{ij} \leq SLO_{GET} : GR_{ij}$ *// whether j is in i's GET replica set; only permitted if a VM at i can complete a GET on the storage service j within the SLO*
7: $\forall j : C_j$ *// whether j is a replica*
8: $\forall i, j \in AS, k : U^P_{ijk}$ *// whether k is in the union of i's and j's PUT replica sets*
9: $\forall i, j \in AS, k : U^G_{ijk}$ *// whether k is in the union of i's GET replica set and j's PUT replica set*
10: $\forall i \in AS, j, k$ s.t. p^{th} percentile of $L^C_{ij} + L^C_{ik} + L^S_{kj} \leq SLO_{PUT} : F_{ikj}$ *// whether i forwards its PUTs to j via k; only permitted if a VM at i can acquire the lock on a VM at j and then forward data to the storage service at j via a VM at k within the SLO*
11: $\forall i \in AS, k : R_{ik}$ *// whether k serves as a relay from i to any of i's PUT replicas*

12: **Objective:** Minimize (Cost for GETs + Cost for PUTs + Storage cost)
13: *// at each of i's GET replicas, every GET from i incurs one GET request's cost and the network cost of transferring the object from the replica to i*
14: Cost for GETs $= \sum_i GETs_i \cdot (\sum_j (GR_{ij} \cdot (Price^{GET}_j + Price^{Net}_{ji} \cdot Size^{avg})))$
15: *// every PUT from i incurs one PUT request's cost at each of i's PUT replicas and the network cost of transferring the object to these replicas*
16: Cost for PUTs $= \sum_i PUTs_i \cdot \sum_k (R_{ik} \cdot Price^{Net}_{ik} \cdot Size^{avg} + \sum_j (F_{ikj} \cdot (Price^{PUT}_j + Price^{Net}_{kj} \cdot Size^{avg})))$
17: *// every replica stores one copy of every object*
18: Storage cost $= \sum_j (C_j \cdot Price^{Storage}_j \cdot Size^{total} \cdot T)$

19: **Constraints:**
20: *// k is in the union of i's and j's PUT replica sets if it is in either set*
21: $\forall i, j \in AS, k : (U^P_{ijk} = 1)$ iff $(PR_{ik} + PR_{jk} > 0)$
22: *// for the PUT replica sets of any pair of data centers in access set, the sum of their cardinalities should exceed the cardinality of their union by $2f$*
23: $\forall i, j \in AS : \sum_k (PR_{ik} + PR_{jk}) > \sum_k U^P_{ijk} + 2f$
24: *// for any pair of data centers in access set, GET replica set of one must have intersection larger than $2f$ with PUT replica set of the other*
25: $\forall i, j \in AS, k : (U^G_{ijk} = 1)$ iff $(GR_{ik} + PR_{jk} > 0)$
26: $\forall i, j \in AS : \sum_k (GR_{ik} + PR_{jk}) > \sum_k U^G_{ijk} + 2f$
27: *// a PUT from i is relayed to k iff k is used to propagate i's PUT to any of i's PUT replicas*
28: $\forall i \in AS, k : (R_{ik} = 1)$ iff $(\sum_j F_{ikj} > 0)$
29: *// some k must forward i's PUTs to j iff j is in i's PUT replica set*
30: $\forall i \in AS, j : PR_{ij} = \sum_k F_{ikj}$
31: *// a data center is a replica if it is either a PUT replica or a GET replica for any data center in access set*
32: $\forall j : (C_j = 1)$ iff $(\sum_{i \in AS} (GR_{ij} + PR_{ij}) > 0)$

reduce networking costs (*lines 10 and 11, and 27–30*), 2) storage costs are incurred for maintaining a copy of every object at every data center that is in the union of the GET and PUT replica sets of all data centers in the access set (*lines 7, 18, and 32*), and 3) for every GET operation at data center i, one GET request's price and the

price for transferring a copy of the object over the network is incurred at every data center in i's GET replica set (*line 14*).

However, the integer program for the strong consistency setting does differ from the program used in the eventual consistency case in three significant ways. First, for every data center in the access set, the PUT and GET replica sets for that data center may differ (*lines 5 and 6*). Second, *PMan* constrains these replica sets so that every data center's PUT and GET replica sets have an intersection of at least $2f + 1$ data centers with the PUT replica set of every other data center in the access set (*lines 20–26*). Finally, PUT operations at any data center i are propagated only to the data centers in i's PUT replica set, and these updates are propagated via at most one hop (*line 16*).

4.5 SPANStore dynamics

Next, we describe SPANStore's operation in terms of the mechanisms it uses to execute PUTs and GETs, to tolerate failures, and to handle changes in the application's workload across epochs. First, we discuss the metadata stored by SPANStore to implement these mechanisms.

4.5.1 Metadata

At every data center, SPANStore stores in-memory metadata across all the VMs that it deploys in that data center. At the beginning of an epoch, *PMan* computes the new replication policy to use in that epoch, and it transmits to every data center the replication policy information needed at that data center. A data center A needs, for every access set AS that contains A, the PUT and GET replica sets to be used by A for objects with access set AS. Whenever the application issues a PUT for a new object at data center A, it needs to specify the access set for that object. SPANStore then inserts an (object name $\rightarrow$ access set) mapping into the in-memory metadata at

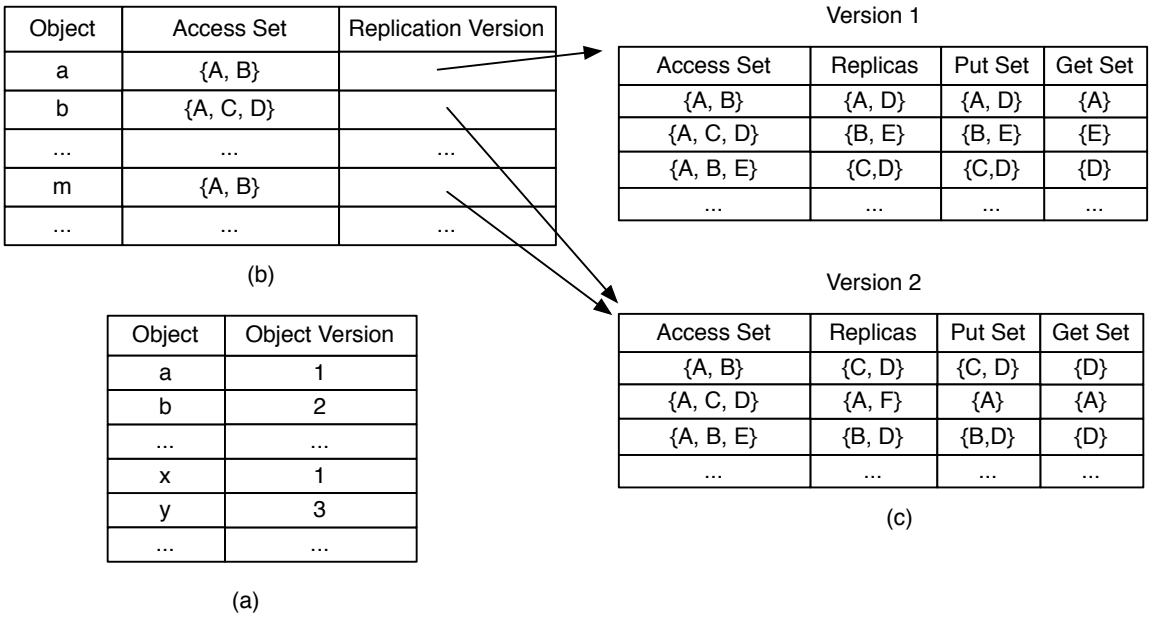


Figure 4.7: At any data center A, SPANStore stores an (a) in-memory version mapping for objects stored at A. If the application is deployed at A, SPANStore also stores (b) the access set mapping for objects whose access set includes A, and (c) replication policy versions for different epochs.

every data center in the access set.

As we describe later in Section 4.5.4, when serving the first operation for an object in a particular epoch, SPANStore needs to account for both the replication policy currently in use for that object and the new replication policy computed by *PMan* for the current epoch. Therefore, we store both current and historical versions of the (access set $\rightarrow$ replica sets) mapping. As shown in Figure 4.7, the access set mapping for an object includes the replication policy version that currently applies for that object. SPANStore eventually destroys an old replication policy when no object is using it.

In addition, at any data center, SPANStore also stores an in-memory version mapping for all objects stored in the storage service at that data center. Note that, at any data center A, the set of objects stored at A can differ from the set of objects whose access set includes A.

4.5.2 Serving PUTs and GETs

Any application uses SPANStore by linking to a library that implements SPANStore's protocol for performing PUTs and GETs. If the application configures SPANStore to provide eventual consistency, the library looks up the local metadata for the current PUT/GET replica set for the queried object. Upon learning which replicas to use, the SPANStore library issues PUT/GET requests to the storage services at those replicas and returns an ACK/the object's data to the application as soon as it receives a response from any one of those replicas.

When using strong consistency, SPANStore uses two phase locking (2PL) to execute PUT operations. First, the SPANStore library looks up the object's metadata to discover the set of replicas that this data center must replicate the object's PUTs to. The library then acquires locks for the object at all data centers in this replica set. If it fails to acquire any of the locks, it releases the locks that it did acquire, backs off for a random period of time, and then retries. Once the library acquires locks at all data centers in the PUT replica set, it writes the new version of the object to the storage services at those data centers and releases the locks.

The straightforward implementation of this protocol for executing PUTs can be expensive. Consider a VM at data center A performing a PUT operation on a replica set that includes data center B. The VM at A can first send a request to a VM at B to acquire the lock and to obtain the version of the object stored at data center B. The VM at A can then send the data for the object being updated to the VM at B (possibly via another data center C that relays the data). The VM at B can write the data to the local data center, release the lock, and update the in-memory version mapping for the object. However, this requires SPANStore's VMs to receive object data from remote data centers. To meet the bandwidth demands of doing so, we will need to provision a large number of VMs, thus inflating cost.

Instead, we employ a modified 2PL protocol as shown in Figure 4.8. As before,

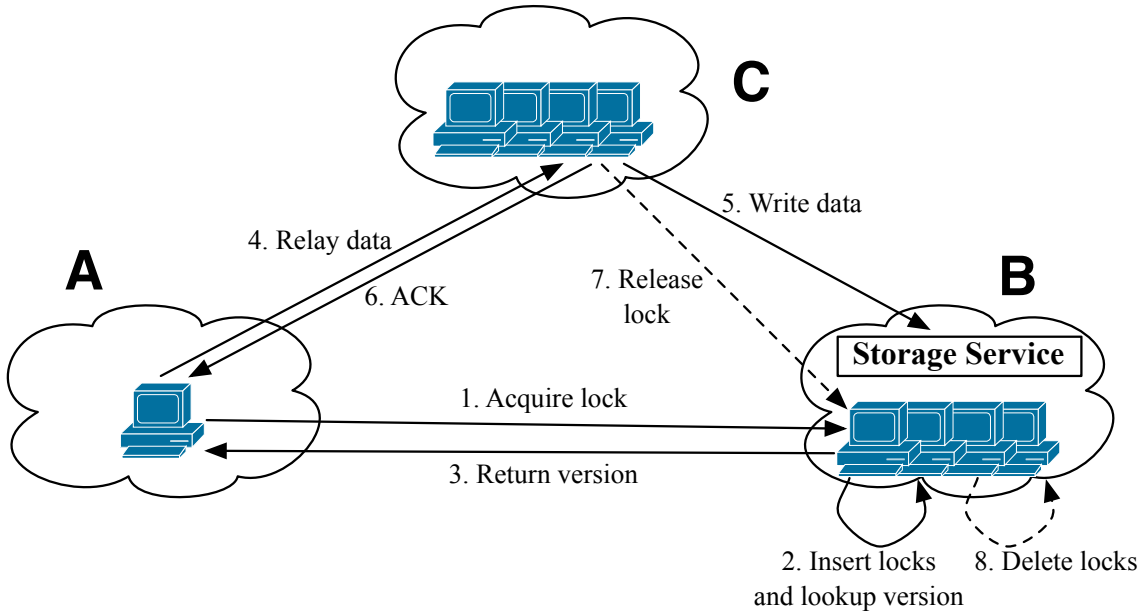


Figure 4.8: Illustration of SPANStore's two-phase locking protocol. Solid lines impact PUT latency, whereas operations along dashed lines are performed asynchronously.

the VM at A communicates with a VM at B to acquire the lock and obtain the version number of B's copy of the object. To acquire the lock for object o, the VM at B inserts two objects into the local in-memory metadata cluster—L_o^T that times out after 5 seconds, and L_o^U that does not have any timeout. Once it acquires the lock, the VM at A directly issues a PUT to the storage service at data center B, rather than asking the VM at B to perform this PUT. While writing the object, we prepend the version number to the object's data. Once the PUT to the storage service is complete, SPANStore lazily requests a VM at B to release the lock by deleting both L_o^T and L_o^U, and to also update the version number stored in memory for the updated object.

In the case where the Internet path from A to B fails after the new version of the object has been written to B or if the VM at A that is performing the PUT fails before it releases the locks, the VM at A cannot explicitly delete L_o^T and L_o^U at B, yet L_o^T will timeout. When a VM at B receives a request to lock object o in the future and finds that L_o^T is absent but L_o^U is present, it issues a GET for the object to the

local storage service and updates the in-memory version mapping for o to the version prepended to the object's data.

This modified 2PL protocol eliminates the need for SPANStore's VMs to send or receive object data, other than when PUTs are relayed via another data center. As a result, our 2PL protocol is significantly more cost-effective than the strawman version, e.g., a small VM on EC2 can handle 105 locking operations per second, but can only receive and write to the local storage service 30 100KB objects per second.

In the strong consistency setting, serving GETs is simpler than serving PUTs. When an application VM at a particular data center issues a GET, the SPANStore library on that VM looks up the GET replica set for that object in the local in-memory metadata, and it then fetches the copy of the requested object from every data center in that set. From the retrieved copies of the object, the library then returns the latest version of the object to the application. We could reduce networking costs by first querying the in-memory metadata at every replica for the current version of the object at that data center, and then fetching a copy of the object only from the nearest data center which has the latest version. However, for small objects whose data can fit into one IP packet, querying the version first and then fetching the object will double the wide-area RTT overhead.

4.5.3 Fault tolerance

SPANStore needs to respect the application's fault-tolerance needs, which *PMan* accounts for when it determines replication policies. In the eventual consistency case, every data center in the access set is associated with $f + 1$ replicas, and SPANStore considers a GET/PUT as complete once the operation successfully completes on any one of the replicas chosen by *PMan*. It suffices for SPANStore to consider a PUT as complete even after writing the update to a single replica because of the durability guarantees offered by the storage services that SPANStore builds upon. Every storage

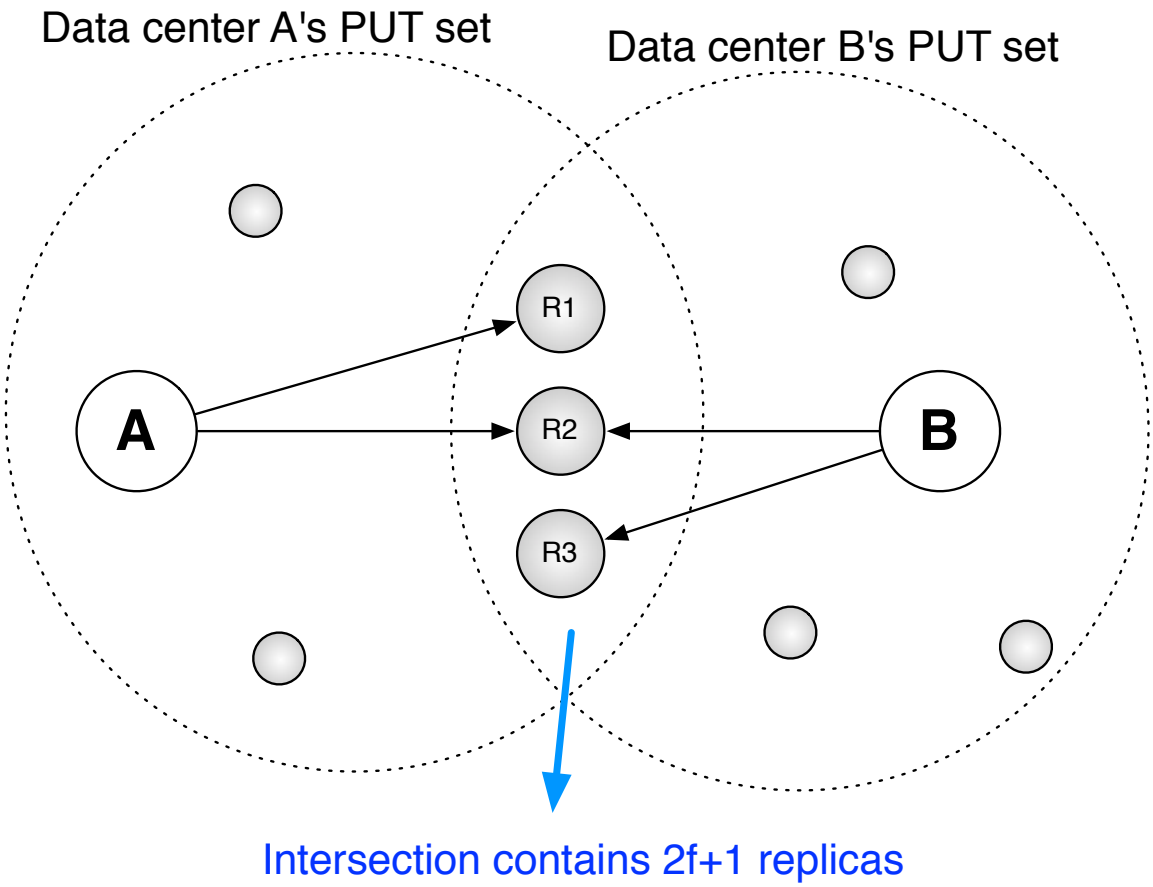


Figure 4.9: Illustration of SPANStore's fault tolerance. The intersection of any two data centers' PUT sets contains $2f + 1$ replicas, and PUT operations require getting responses from at least $f + 1$ replicas from all intersections. This guarantees that any two data centers have at least one common replica to write to, and therefore guarantees strong consistency.

service replicates objects across servers within a data center to ensure that it is very unlikely to lose an update committed by a PUT.

When configured for strong consistency, SPANStore relies on the fault tolerance offered by our use of quorum sets. As shown in Figure 4.9, an intersection of at least $2f + 1$ data centers between the PUT replica set of every data center and the PUT and GET replica sets of every other data center in the access set enables SPANStore to be resilient to up to f failures [103]. This is because, even if every data center is unable to reach a different set of f replicas, the intersection larger than $2f+1$ between PUT-PUT replica set pairs and GET-PUT replica set pairs ensures that every pair

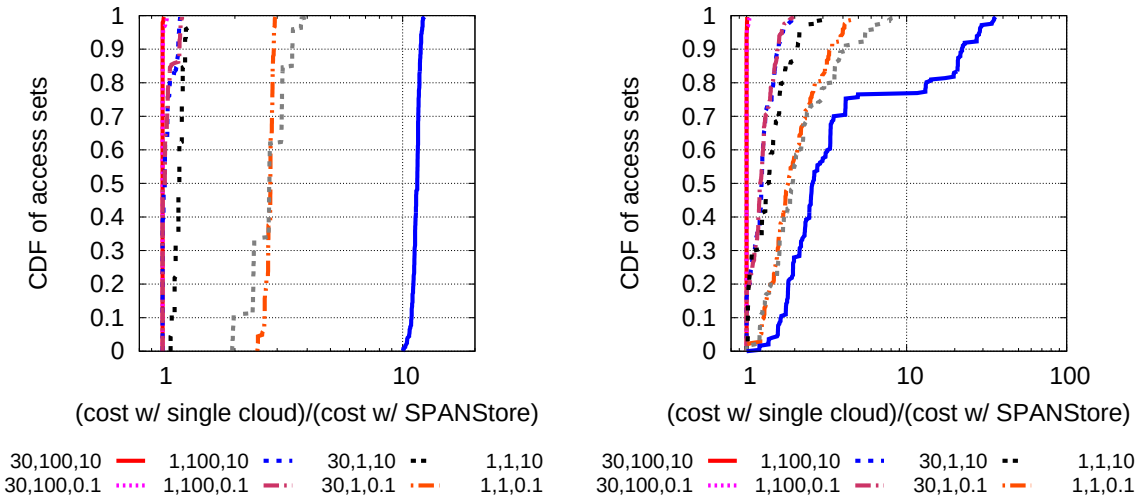
of data centers in the access set has at least one common replica that they can both reach.

Thus, in the strong consistency setting, the SPANStore library can tolerate failures as follows when executing PUT and GET operations. At any data center A, the library initiates a PUT operation by attempting to acquire the lock for the specified object at all the data centers in A's PUT replica set for this object. If the library fails to acquire the lock at some of these data centers, for every other data center B in the object's access set, the library checks whether it failed to acquire the lock at at most f replicas in B's PUT and GET replica sets for this object. If this condition is true, the library considers the object to be successfully locked and writes the new data for the object. If not, the PUT operation cannot be performed and the library releases all acquired locks.

The SPANStore library executes GETs in the strong consistency setting similarly. To serve a GET issued at data center A, the library attempts to fetch a copy of the object from every data center in A's GET replica set for the specified object. If the library is unsuccessful in fetching copies of the object from a subset S of A's replica set, it checks to see whether S has an intersection of size greater than f with the PUT replica set of any other data center in the object's access set. If yes, the library determines that the GET operation cannot be performed and returns an error to the application.

4.5.4 Handling workload changes

The replication policy for an access set can change when there is a significant change in the aggregate workload estimated for objects with that access set. When *PMan* mandates a new replication policy for a particular access set at the start of a new epoch, SPANStore switches the configuration for an object with that access set at the time of serving the first GET or PUT request for that object in the new epoch.



(a) Strong, 90%ile G-SLO=250ms (b) Strong, 90%ile G-SLO=100ms

Figure 4.10: Cost with SPANStore compared to that possible using the data centers of a single cloud service. Legend indicates GET:PUT ratio, average object size (in KB), and overall data size (in TB).

SPANStore can identify the first operation on an object in the new epoch based on a version mismatch between the replication policy associated with the object and the latest replication policy.

In a new epoch, irrespective of whether the first operation on an object is a GET or a PUT, the SPANStore library on the VM issuing the request attempts to acquire locks for the object at all data centers in the object's access set. In the case of a PUT, SPANStore commits the PUT to the new PUT replica set associated with the object. In the case of a GET, SPANStore reads copies of the object from the current GET set and writes the latest version among these copies to the new PUT set. SPANStore then switches the replication policy for the object to the current version at all data centers in its access set. Thereafter, all the PUTs and GETs issued for the object can be served based on the new replication policy.

This procedure for switching between replication policies leads to latency SLO violations and cost overhead (due to the additional PUTs incurred when the first request for an object in the new epoch is a GET). However, we expect the SLO

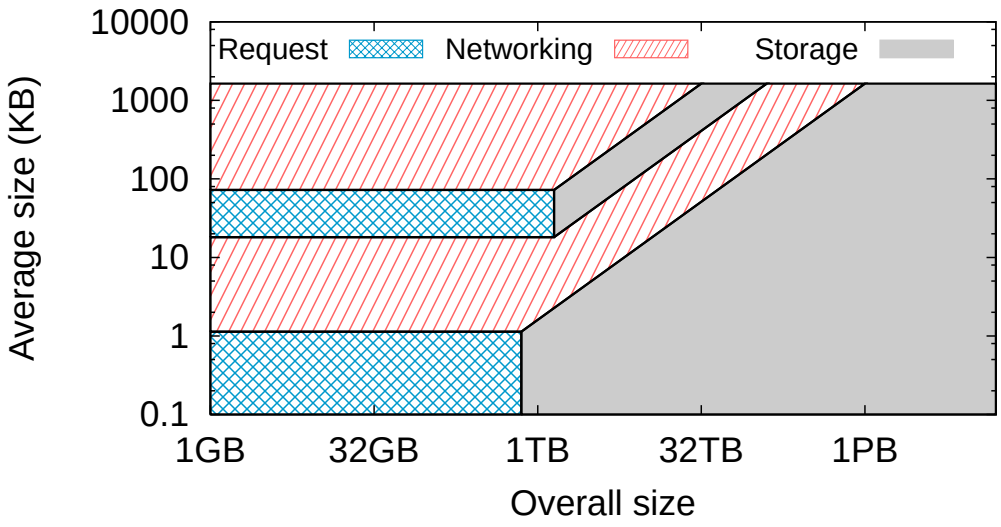


Figure 4.11: Variation in the dominant component of SPANStore's cost as a function of the application's workload.

violations to be rare and the cost overhead to be low since only the first operation on an object in a new epoch is affected.

4.6 Implementation

We have implemented and deployed a prototype of SPANStore that spans all the data centers in Amazon S3, Microsoft Azure, and Google Cloud Storage. Our implementation has three components—1) *PMan*, 2) a client library that applications can link to, and 3) an XMLRPC server that is run in every VM run by SPANStore. In addition, in every data center, we run a memcached cluster across all SPANStore instances in that data center to store SPANStore's in-memory metadata.

PMan initially bootstraps its state by reading in a configuration file that specifies the application's latency, consistency, and fault tolerance requirements as well as the parameters (latency distribution between data centers, and prices for resources at these data centers) that characterize SPANStore's deployment. To determine optimal replication policies, it then periodically invokes the CPLEX solver to solve the formulation (Algorithm 1 or Algorithm 2) appropriate for the application's consistency

needs. *PMan* also exports an XMLRPC interface to receive workload and latency information from every data center at the end of every epoch.

The client library exports two methods: *GET(key)* and *PUT(key, value, [access_set])*. The library implements these methods as per the protocols described in Section 4.5. To lookup the metadata necessary to serve GET and PUT requests, the library uses DNS to discover the local memcached cluster.

The XMLRPC server exports three interfaces. First, it exports a *LOCK(key)* RPC for the client library to acquire object-specific locks. Second, its *RELAY(key, data, dst)* enables the library or a SPANStore VM to indirectly relay a PUT in order to reduce network bandwidth costs. Lastly, the XMLRPC server receives replication policy updates from *PMan*.

In addition, the XMLRPC server 1) gathers statistics about the application's workload and reports this information to *PMan* at the end of every epoch, and 2) exchanges heartbeats and failure information with SPANStore's VMs in other data centers. Both the client library and the XMLRPC server leverage open-source libraries for issuing PUT and GET requests to the S3, Azure, and GCS storage services.

4.7 Evaluation

We evaluate SPANStore from four perspectives: the cost savings that it enables, the cost-optimality of its replication policies, the cost necessary for increased fault tolerance, and the scalability of *PMan*. Here, we show results for the case where the application is deployed across EC2's data centers, and SPANStore is deployed across the storage services offered by S3, Azure, and GCS. Our results are qualitatively similar when we consider application deployments on Azure or GCE.

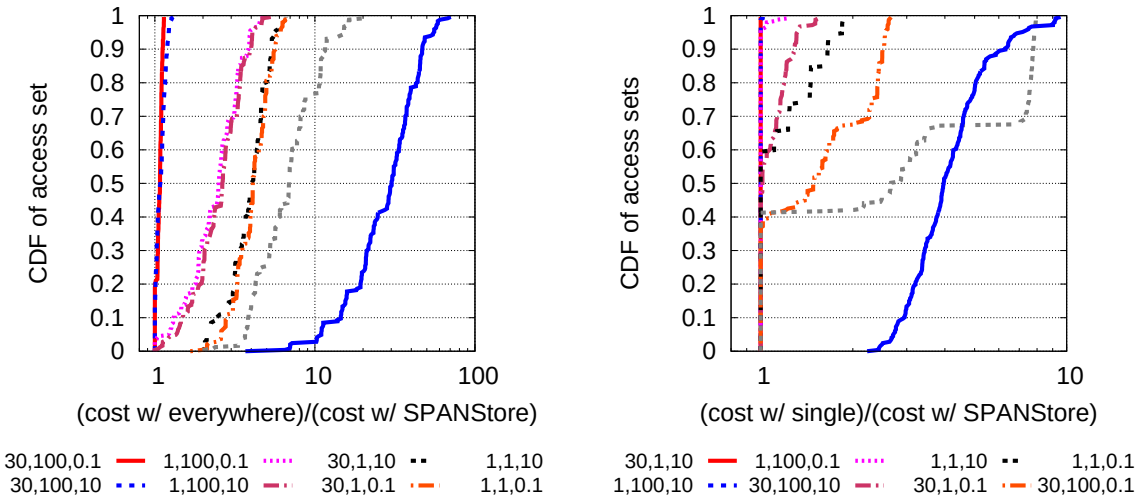


Figure 4.12: Cost savings enabled by SPANStore compared to *Everywhere* (left) and *Single* (right) replication policies. Legend indicates GET:PUT ratio, average object size (in KB), and overall data size (in TB).

4.7.1 Cost savings

Workload and SLOs. To evaluate the cost savings enabled by SPANStore, we consider all 18 combinations of a) GET:PUT ratios of 1, 10, and 30, b) average object sizes of 1 KB and 100 KB, and c) aggregate data size of 0.1 TB, 1 TB and 10 TB. Our choice of these workload parameters is informed by the GET:PUT ratio of 30:1 and objects typically smaller than 1 KB seen in Facebook's workload [51]. For brevity, we omit here the results for the intermediate cases where GET:PUT ratio is 10 and overall data size is 1 TB. In all workload settings, we fix the number of GETs at 100M and compute cost over a 30 day period.

When analyzing the eventual consistency setting, we consider two SLOs for the 90^{th} percentile values of GET and PUT latencies—100 ms and 250 ms; 250 ms is the minimum SLO possible with a single replica for every object and 100 ms is less than half of that. In the strong consistency case, we consider two SLO combinations for the 90^{th} percentile GET and PUT latencies—1) 100ms and 830ms, and 2) 250 ms and 830 ms; 830 ms is the minimum PUT SLO if every object was replicated at all

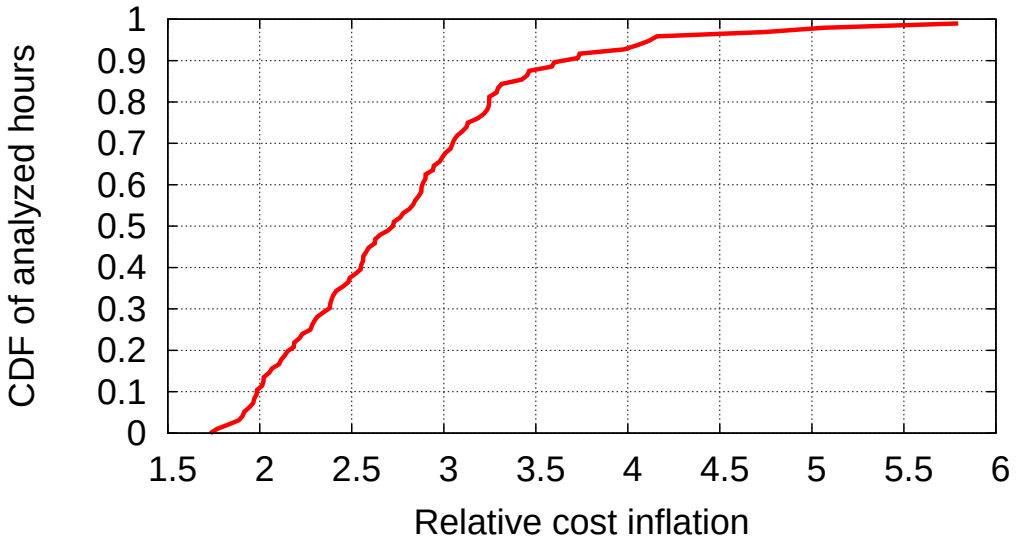


Figure 4.13: Cost inflation when predicting workload using individual objects compared with aggregate workload prediction.

data centers in its access set. Since the cost savings enabled by SPANStore follow similar trends in the eventual consistency and strong consistency settings, we show results only for the latter scenario for brevity.

Comparison with single-cloud deployment. First, we compare the cost with SPANStore with the minimum cost required if we used only Amazon S3's data centers for storage. Figure 4.10 shows that SPANStore's use of multiple cloud services consistently offers significant cost savings when the workload includes 1 KB objects. The small object size makes networking cost negligible in comparison to PUT/GET requests costs, and hence, SPANStore's multi-cloud deployment helps because PUT and GET requests are priced 50x and 4x cheaper on Azure as compared to on S3. When the average object size is 100KB, SPANStore still offers cost benefits for a sizeable fraction of access sets when the PUT/GET ratio is 1 and overall size size is small. In this case, since half of the workload (i.e., all PUT operations) require propagation of updates to all replicas, SPANStore enables cost savings by exploiting discrepancies in network bandwidth pricing across cloud services. Furthermore, when the total data size is 10 TB, SPANStore reduces storage costs by storing fewer copies

of every object.

Comparison with fixed replication policies. We also compare the cost incurred when using SPANStore with that imposed by two fixed replication policies: *Everywhere* and *Single*. With the *Everywhere* policy, every object is replicated at every data center in the object's access set. With the *Single* replication policy, any object is stored at one data center that minimizes cost among all single replica alternatives that satisfy the PUT and GET latency SLOs. We consider the same workloads as before, but ignore the cases that set the SLO for the 90^{th} percentile GET latency to 100ms since that SLO cannot be satisfied when using a single replica.

In Figure 4.12(left), we see that SPANStore significantly outdoes *Everywhere* in all cases except when GET:PUT ratio is 30 and average object size is 100KB. On the other hand, in Figure 4.12(right), we observe a bi-modal distribution in the cost savings as compared to *Single* when the object size is small. We find that this is because, for all access sets that do not include EC2's Sydney data center, using a single replica (at some data center on Azure) proves to be cost-optimal; this is again because the lower PUT/GET costs on Azure compensate for the increased network bandwidth costs. When the GET:PUT ratio is 1 and the average object size is 100KB, SPANStore saves cost compared to *Single* by judiciously combining the use of multiple replicas.

Dominant cost analysis. Finally, we analyze how the dominant component of SPANStore's cost varies based on the input workload. For one particular access set, Figure 4.11 shows which among network, storage, and request cost dominates the total cost when varying average object size from 0.1 KB to 1 MB and total data size from 1 GB to 1 PB. Here, we use a GET:PUT ratio of 30 and set GET and PUT SLOs as 250ms and 830ms with the need for strong consistency, but the takeaways are similar in other scenarios.

When both the average object size and the total data size are small, costs for

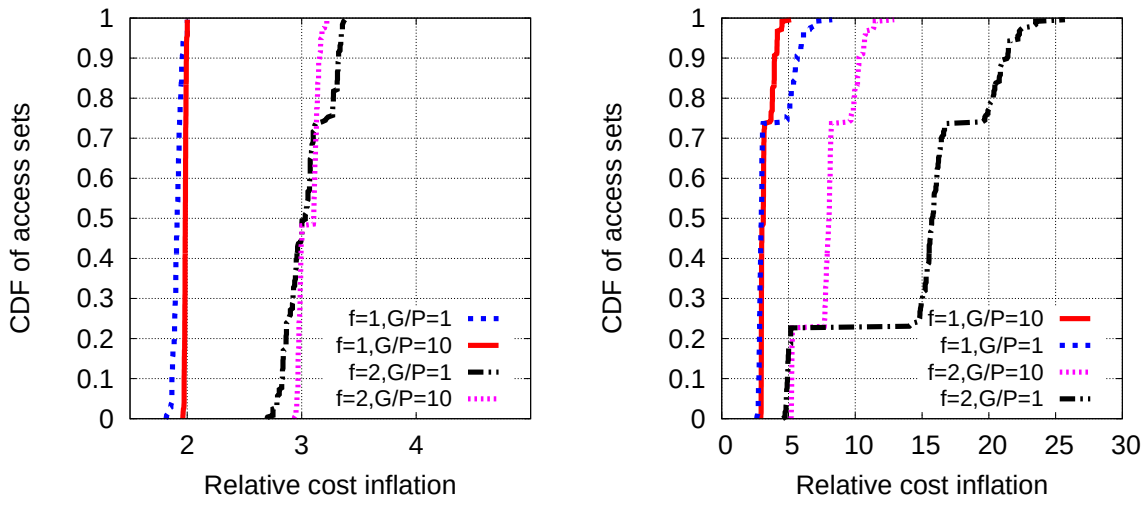
PUT and GET requests initially dominate, but network transfer costs increase as the average object size increases. However, as the average object size increases further, SPANStore transitions to storing data locally at every data center in the access set. This eliminates the need to use the network when serving GETs, thus making request costs the dominant component of cost again. However, network transfers cannot be completely eliminated due to the need to propagate updates to all replicas. Therefore, network transfer costs again begin to dominate for large object sizes.

When the total data size is large, SPANStore stores a single copy of all data when the average object size is small and storage cost is dominant. As the average object size increases, network transfer costs initially exceed the storage cost. However, as network costs continue to increase, SPANStore begins to store multiple replicas of every object so that many GETs can be served from the local data center. This reduces network transfer costs and makes storage cost dominant again. Eventually, as the average object size increases further, even if SPANStore stores a replica of every object at every data center in its access set, the need to synchronize replicas results in networking costs exceeding storage costs.

4.7.2 Impact of aggregation of objects

SPANStore's cost-effectiveness critically depends on its ability to estimate the application's workload. As discussed previously in Section 4.4, we choose to characterize workload in aggregate across all objects with the same access set due to the significantly greater stationarity that exists in aggregate workloads as compared to the workloads of individual objects. Here, we quantify the cost benefits enabled by this design decision.

From the Twitter dataset previously described in Section 4.4, we randomly choose 10K users. Since the dataset only specifies the times at which these users post tweets, we generate the times at which they check their Twitter timelines based on Twitter's



(a) Eventual,P-SLO=250ms (b) Strong,P-SLO=830ms

Figure 4.14: Cost inflation when tolerating f failures compared to the cost with $f = 0$. SLOs are on 90^{th} percentile latencies and G-SLO=250ms.

usage statistics [28]. Considering hour-long epochs, we estimate the workload in a particular hour as being the same as that in the same hour in the previous week. We consider workload estimation at two granularities: 1) per-object, and 2) in aggregate across all objects with the same access set.

Figure 4.13 shows the cost inflation arising from estimating workloads on a per-object granularity as compared to the cost when estimating aggregate workloads. The high inflation is due to the significant variation seen in any individual user's usage of Twitter. Since most users rarely post or access Twitter, the use of the per-object workload estimator causes SPANStore to typically choose EC2's data centers as replicas since they have lower storage cost. However, this often turns out to be a mis-prediction of the workload, and when a user does post or access her timeline, SPANStore has to incur greater request costs that necessary by serving these requests from EC2's data centers. The greater accuracy of estimating workloads in aggregate enables SPANStore to replicate data in a more cost-effective manner.

4.7.3 Cost for fault tolerance

To tolerate failures, SPANStore provisions more data centers to serve as replicas. As expected, this results in higher cost. In Figure 4.14, we show the cost inflation for various levels of fault-tolerance as compared to the cost when SPANStore is provisioned to not tolerate any failures. For most access sets, the cost inflation is roughly proportional to $f + 1$ in the eventual consistency scenario and proportional to $2f + 1$ in the strong consistency case. However, the need for fault tolerance increases cost by a factor greater than $f + 1/2f + 1$ for many other access sets. This is because, as f increases, the need to pick a greater number of replicas within the latency SLO for every data center in the access set requires SPANStore to use as replicas data centers that have greater prices for GET/PUT requests.

In addition, in both the eventual consistency and strong consistency scenarios, the need to tolerate failures results in higher cost inflation when the GET:PUT ratio is low as compared to when the GET:PUT ratio is high. This is because, when the GET:PUT ratio is low, SPANStore can more aggressively reduce costs when $f = 0$ by indirectly propagating updates to exploit discrepancies in network bandwidth pricing.

4.7.4 Scalability of PlacementManager

Finally, we evaluate the scalability of *PMan*, the one central component in SPANStore. At the start of every epoch, *PMan* needs to compute the replication policy for all access sets; there are 2^N access sets for an application deployed across N data centers. Though the maximum number of data centers in any one cloud service is currently 8 (in EC2), we test the scalability of *PMan* in the extreme case where we consider all the data centers in EC2, Azure, and GCE as being in the same cloud service on which the application is deployed. On a cluster of 16 servers, each with two quad-core hyperthreaded CPUs, we find that we can compute the replication policy within an hour for roughly 33K access sets. Therefore, as long as the application can estimate

its workload for the next epoch an hour before the start of the epoch (which is the case with our current way of estimation based on the workload in the same hour in the previous week), our *PMan* implementation can tackle cases where the application is deployed on 15 or fewer data centers. For more geographically distributed application deployments, further analysis is necessary to determine when the new aggregate workload for a access set will not cause a significant change in the optimal replication policy for that set. This will enable *PMan* to only recompute the replication policy for a subset of access sets.

4.8 Case studies

We have used our SPANStore prototype as the back-end storage for two applications that can benefit from geo-replicated storage: 1) Retwis [32] is a clone of the Twitter social networking service, which can make do with eventual consistency, and 2) ShareJS [36] is a collaborative document editing web service, which requires strongly consistent data. To support both applications, we add a few operations that are wrappers over PUTs and GETs, such as "append to a set", "get i^{th} element from a set", and "increment a global counter". We deploy both applications across all of EC2's data centers.[3]

Retwis. At each EC2 data center, we run emulated Retwis clients which repeatedly make two types of operations: *Post* operations represent a user posting a tweet, and *GetRange* operations fetch the last 100 tweets in a user's timeline (the set of tweets posted by those that the user follows). We set the ratio of number of *Post* operations to number of *GetRange* operations as 0.1, i.e., on average, every user makes a post every 10 times that she checks her timeline. A GetRange operation issues 1) a GET to fetch the user's timeline object, and then 2) GETs for the post IDs in the

[3]Though ShareJS is not amenable to distributed deployment, we modify it suitably so as to enable every user to be served from her closest EC2 data center.

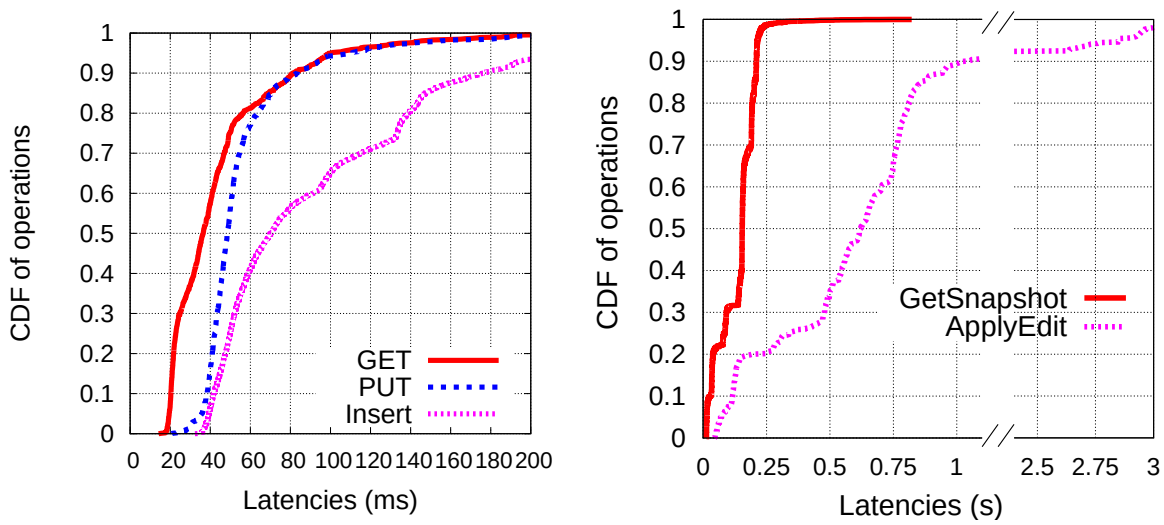


Figure 4.15: CDF of operation latencies in (a) Retwis and (b) ShareJS applications.

specified range in the timeline. A Post operation executes the following sequence: 1) a PUT to store the post, 2) a GET to fetch the list of the user's followers, and 3) an Insert operation to append the post's ID to the timeline object of each of the user's followers. The Insert operation on an object fetches the object by issuing GET, modifies it locally, and then writes back the updated object with a PUT.

We run Retwis with a randomly generated social network graph comprising 10K users, where every user follows a randomly chosen subset of 200 users [2]. For every user, we assign the data center from which the user is served at random. We run this workload on SPANStore configured with the latency SLOs specifying that the 90^{th} percentile PUT and GET latencies should be less than 100ms. Figure 4.15(a) shows that SPANStore satisfies these latency goals as over 90% of all operations are within the specified SLOs.

ShareJS. We run the ShareJS application with a similar setup; we consider 1K documents, each of which is associated with a randomly chosen access set. At each EC2 data center, we then run a ShareJS client which iteratively issues *GetSnapshot* and *ApplyEdit* operations on randomly chosen documents whose access set includes

the local data center. These operations correspond to the client fetching a document and applying an edit to a document, respectively. Since we need strong consistency in this case, we use SLOs on the 90^{th} percentile PUT and GET latencies as 830ms and 250ms. Note that the GetSnapshot operation directly maps to a GET, but the ApplyEdit operation requires the application to issue a GET for the latest version of the document and then issue a PUT to incorporate the edit.

Figure 4.15(b) shows the distribution of latencies incurred for the GetSnapshot and ApplyEdit operations. We can see that more than 95% of GetSnapshot operations satisfy the latency SLO, and a vast majority of ApplyEdit operations are within the SLO, given that an ApplyEdit includes a GET followed by a PUT. The small minority of operations that exceed the latency bounds are due to contention between concurrent updates to the same document; when a writer fails on the two-phase locking operation, it retries after 2 seconds.

4.9 Summary

Though the number of cloud storage services available across the world continues to increase, the onus is on application developers to replicate data across these services. We develop SPANStore to export a unified view of geographically distributed storage services to applications and to automate the process of trading off cost and latency, while satisfying consistency and fault-tolerance requirements. Our design of SPANStore achieves this goal by spanning data centers of multiple cloud providers, by judiciously determining replication policies based on workload properties, and by minimizing the use of compute resources. We have deployed SPANStore across Amazon's, Microsoft's, and Google's cloud services and find that it can offer significant cost benefits compared to simple replication policies.

CHAPTER V

Efficient Geo-Replication of Data in the Cloud

Minimizing user-perceived latency is a critical goal for web services, because even a few 100 milliseconds of additional delay can significantly reduce revenue [29]. Key to achieving this goal is to deploy web servers in multiple locations so that every user can be served from a nearby server. Therefore, cloud services such as Azure and Amazon Web Services are an attractive option for web service deployments as these platforms offer data centers in tens of locations spread across the globe [12, 16].

However, the potential for cloud services to democratize low latency web services remains largely unfulfilled today, as web services in the cloud seldom opt for a geographically distributed deployment [9, 79]. We posit that a key reason is that, while cloud providers make it easy to setup *servers* in multiple locations, they lack support for managing *data* replicated across data centers. Geo-replication of data is crucial to achieve low latency since web services increasingly enable users to share content with each other (e.g., collaborative document editing, online gaming, online social networks). In the absence of geo-replication of data, web servers would have to use a centrally located copy of shared data to serve users' requests, nullifying the latency benefits of deploying web servers in multiple locations.

In this chapter, we seek to fill this gap between what web services need (a globally consistent view to data replicated *across* data centers) and what cloud providers offer

(strongly consistent storage services *within* each data center). Even if cloud providers fill this gap in the future, the onus for consistently geo-replicating data will remain on web service providers who choose to spread their data across the data centers of *multiple* cloud providers for the associated cost, performance, and fault-tolerance benefits [126, 123, 83, 82].

Unlike prior systems [66, 86, 53, 96] designed to replicate data across a web service provider's own data centers, the primary difference in our setting is that we build upon a multi-tenant storage service in each data center and not upon storage servers under the provider's control. This difference leads to two unique constraints, which make it challenging to minimize latency and cost.

- **Limited interface:** There is a mismatch between the interface offered by cloud storage (e.g., PUT/GET) and the interface necessary to reuse existing low latency replication protocols (e.g., Accept/Learn to use Fast Paxos [91]). Application providers could augment the storage interface by deploying virtual machines (VMs) at every data center to proxy requests to the local storage service. However, coping with the operational complexities of managing these proxy VMs would nullify a primary benefit of using cloud storage: not having to worry about issues such as scalability and fault-tolerance. Moreover, for the proxy VMs to not become a bottleneck, these VMs can significantly increase a web service provider's costs.

  Alternatively, web services could use replication protocols [71, 76] that can make do with the limited interface offered by cloud storage. Doing so would however significantly degrade latencies.

- **High latency variance:** Sharing of cloud storage by multiple tenants leads to high tail latencies [128]. The oft-used solution to address latency variance—redundantly access all replicas and wait for responses from a quorum—is insufficient when data is geo-replicated. This is because, for a client to access *more* replicas than the quorum of replicas closest to it, the client must access replicas that are *farther* and

hence are less effective at mitigating latency spikes at nearby replicas.

To address these challenges, we present CRIC (*Consistent Replication in the Cloud*). CRIC enables a geo-distributed web service's VMs to directly read from and write to cloud storage, yet offers read/write latencies that would be achievable with existing replication techniques only if cloud storage offered a richer interface and had lower latency variance. We achieve these properties by making contributions primarily on two fronts.

First, we present CPaxos, a variant of Paxos optimized for use in the cloud. To cope with the limited interface to cloud storage, CPaxos stores the state maintained by Paxos acceptors in cloud storage but moves the logic executed by Paxos acceptors into the client. Despite doing so, CPaxos can execute both reads and writes with one round-trip of communication in the common case when conflicts are rare. CPaxos achieves this property by leveraging the ability offered by cloud storage services to conditionally update data only if it has not been updated since it was last read.

Second, to reduce latency variance in the context of geo-replicated data, we modify Paxos's use of redundancy. In a traditional implementation of Paxos, a proposal is committed once it is accepted by a majority of acceptors. In contrast, CRIC leverages the notion of *hierarchical quorums* [87], explicitly accounting for the data centers in which replicas are stored and storing multiple replicas within each data center. A client's attempt to update any object is successful once it updates a quorum of replicas in each of a quorum of data centers. This design enables CRIC to mitigate the impact of latency variance associated with cloud storage by requiring redundant requests to be issued only to the quorum of data centers closest to the client.

We have implemented and deployed a prototype of CRIC across AWS and Azure. It offers performance comparable to approaches requiring additional VMs to support replication, but at 20–50% lower cost. Whereas, compared to existing replication protocols compatible with a limited storage interface, CPaxos can halve write latency

and also lower cost by 10–60%. Lastly, we simulate CRIC's use in a strongly consistent equivalent of Twitter and show that hierarchical quorum significantly reduce median user-perceived latencies.

5.1 Motivation

We begin by describing our target setting and goals, and discuss the challenges associated with enabling strongly consistent access to data replicated across cloud data centers.

5.1.1 Setting and Goals

In a geo-distributed web service deployment in the cloud, user-facing web servers are deployed in multiple data centers, so that any user's request can be served from a nearby data center. The latency incurred by any user includes the back-end latency for web servers to read or write data as necessary to serve the user's request; we focus on web services that rely on key-value storage to store objects that are typically smaller than 1KB [59, 51, 107]. To minimize back-end latency and maximize availability, web services can geo-replicate data—potentially across the data centers of multiple cloud providers[1]—and have web servers read or write to any object by accessing a subset of nearby replicas, rather than a centrally located copy of the object.

While cloud providers currently offer a separate storage service within each data center, our goal is to enable web service developers to outsource to CRIC the task of managing the consistency of data replicated across data centers (as shown in Figure 5.1). CRIC offers the same PUT-GET interface as strongly consistent key-value stores available within a cloud data center, but executes operations on geo-replicated

[1]Note that a web service can deploy all of its servers across the data centers of a single cloud provider, but spread its data across multiple cloud providers, leveraging the common interface offered by key-value stores.

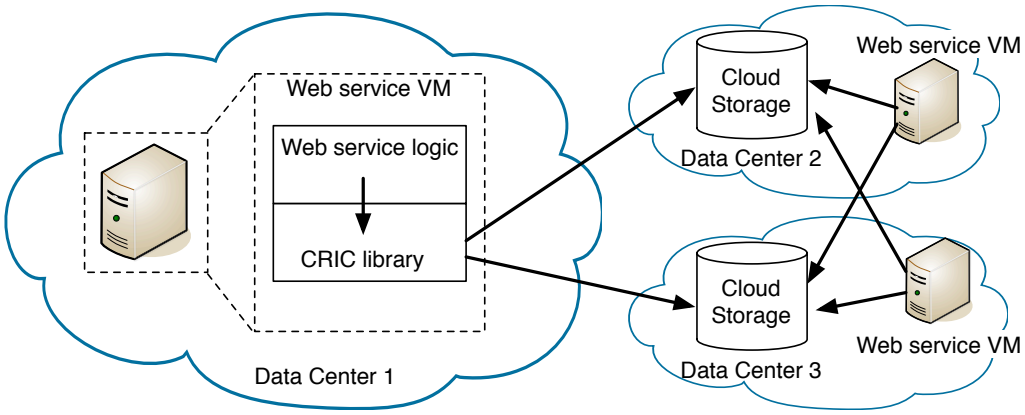


Figure 5.1: Illustration of CRIC's use in a geo-distributed web service deployed in the cloud.

data. In developing CRIC to offer this functionality, we are guided by four objectives:

- **Strong consistency:** All writes must be linearizable and any read should return the latest version of any object. Strongly consistent replicated storage simplifies application development, enables seamless porting of web services written to use centralized storage, and is essential in many web services, including collaborative document editing (e.g., Google Docs), online auctions (e.g., eBay, stock trading), and multi-player online gaming.
- **Fault-tolerance:** CRIC must ensure that any object continues to be readable/writable as long as at most f of the object's replicas are unavailable.
- **Low latency:** When reading from or writing to an object, CRIC should 1) access nearby copies of the object, 2) minimize the number of round-trips of wide-area communication, and 3) account for the latency variance in cloud storage services in order to minimize tail latencies.
- **Cost-effectiveness:** CRIC should minimize the cost overhead that it imposes on web services. We account for the cost of data transfers, GET/PUT requests, and any VMs necessary to support geo-replication.

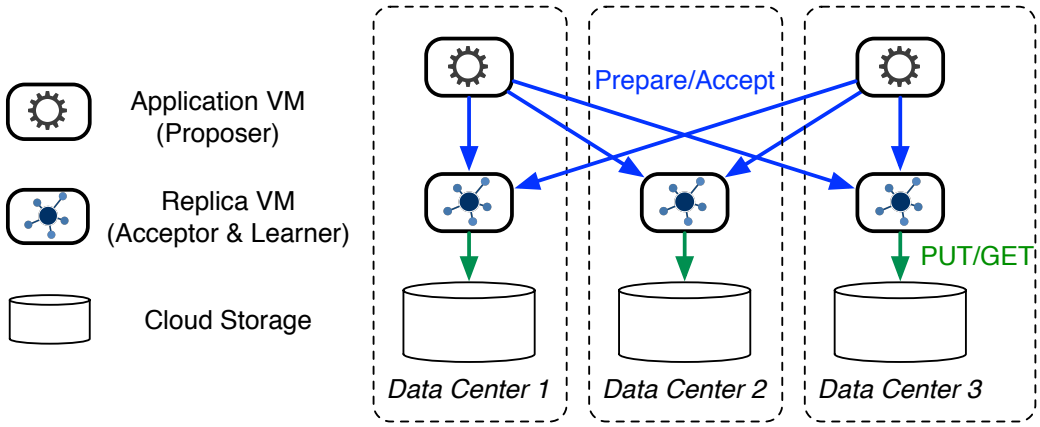


Figure 5.2: Illustration of how Fast Paxos can be used to consistently replicate data across cloud data centers.

5.1.2 Overheads of preserving consistency

Augmenting storage interface imposes operational and cost overheads. The problem of enabling low latency geo-replication has been studied extensively in recent years. Replication protocols such as Fast Paxos [91] and EPaxos [105] as well as new systems such as MDCC [86] and TAPIR [131] can achieve consensus among replicas within one round of communication when conflicts are rare. All of these methods require replicas to participate in a specific replication protocol in order to determine whether to accept or reject updates, to detect conflicts, or to report to a coordinator the order in which they receive requests.

In the cloud, the challenge in using one of these prior solutions is that cloud storage services provide no APIs to support these protocols. Web services that want to use these approaches would have to rent additional *replica VMs* in each data center to augment the interface to the local storage service, so as to implement the desired replication protocol. For example, if an application wants to use Fast Paxos [91] (MDCC [86] and Tapir [131] complete writes in one RTT in a similar manner), it has to rent VMs to serve as "Acceptors" and "Learners", which in turn will have to use cloud storage's PUT/GET interface to store and read both protocol state and

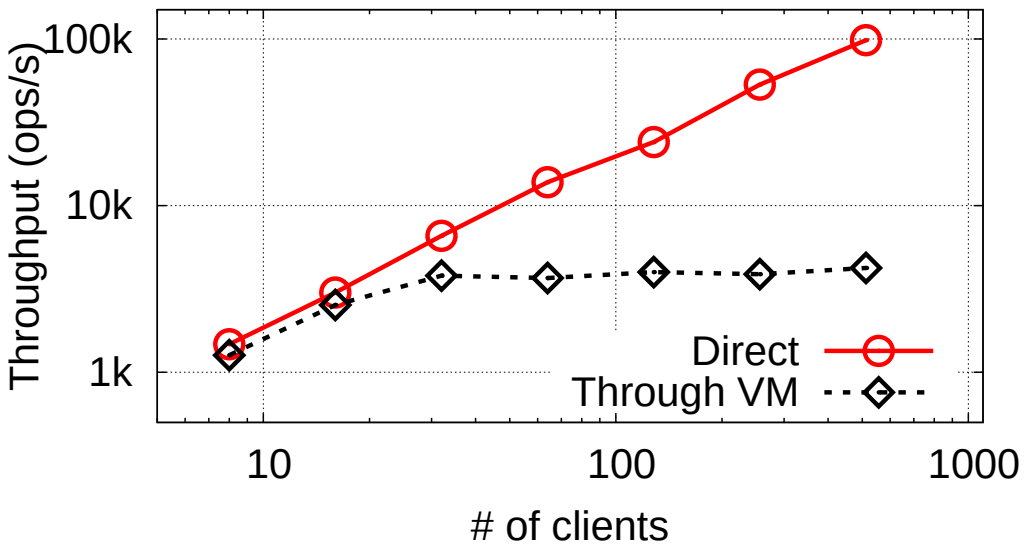


Figure 5.3: GET throughput as a function of # of clients; throughput for "Direct" continues to increase with more clients.

application data; see Figure 5.2.

Relaying all interactions with storage through a layer of VMs is undesirable on multiple fronts. On the one hand, web service providers would have to manage the replica VMs to ensure scalability, fault-tolerance, and load balancing, nullifying the benefits of using cloud storage services which already handle these concerns. On the other hand, having all reads and writes go through replica VMs can significantly increase a web service provider's costs.

To empirically quantify the expected cost overhead due to replica VMs, in one of Azure's data centers, we continually issued GETs, relayed via an A2 instance [15] to the local storage service, for objects of size 1 KB from an increasing number of client threads (running on separate VMs). Figure 5.3 shows that throughput caps out at roughly 3000 ops/s,[2] compared to a linear increase when clients directly issue GETs to storage. We believe that a more efficient implementation of replica VMs is unlikely to offer greater throughput per VM. Our observation is that the key bottleneck in

[2]With other VM types, throughput shows either a linear or sub-linear growth with the price of the VM, thus making A2 instances the most cost-effective option in terms of throughput per $.

Figure 5.4: When reads/writes are relayed through VMs, fraction of cost accounted for by these VMs relative to total cost to execute reads and writes.

the throughput per replica VM is that every request to the storage service is required to include an authentication signature which is a function of the request's metadata and the private encryption key assigned to the tenants account. For small objects, our implementation of replica VMs, which uses libraries provided by cloud providers to issue requests, saturates the CPU in computing these signatures.

This limit on achievable throughput per replica VM implies that many web services deployed in the cloud will incur a significant increase in cost in order to serve the thousands of requests per second they receive on average from users [11], where each user request potentially translates to several requests to storage. Figure 5.4 shows that the VMs that a web service will need to deploy to interpose on its interactions with cloud storage will account for 45% or more of the cost (i.e., $ for GET/PUT requests, network transfers, and VMs) of accessing geo-replicated data when the average object is smaller than 1 KB.

Existing client-based approaches inflate latency and cost. These undesirable effects of using replica VMs are largely avoidable because write conflicts are rare in typical web service workloads [99, 66]; in the absence of conflicts, the replica VMs

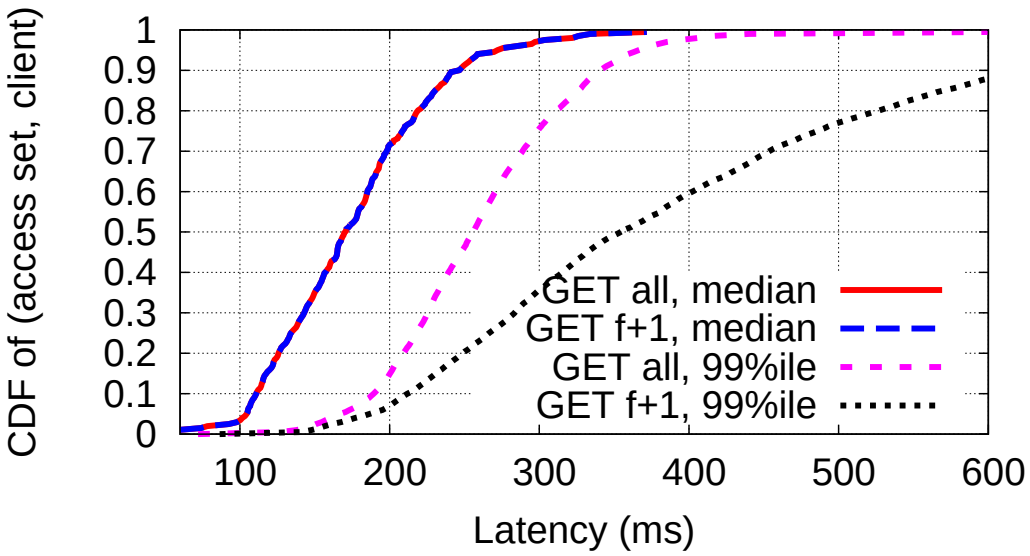


Figure 5.5: Median and tail latency when reading from either $f + 1$ or all replicas and waiting for responses from a quorum; $f = 1$.

used to implement existing replication protocols will simply have to relay data between application VMs and cloud storage. Therefore, it would be desirable to enable application VMs to directly communicate with cloud storage without sacrificing consistency. Indeed, replication protocols such as Disk Paxos [71] and pPaxos [76] are well-suited for this purpose, as they are designed assuming a limited interface at each replica, which is the case in cloud storage. For example, Disk Paxos can make do with replicas offering only a PUT/GET interface, whereas pPaxos requires an APPEND operation in addition.

However, while replication protocols such as Disk Paxos and pPaxos eliminate the cost overhead associated with replica VMs, they double the number of rounds of communication necessary to commit a write to storage, even in the absence of conflicts. Moreover, when conflicts arise, both protocols increase the amount of data transferred over the network, thereby inflating cost. As we show later in Section 5.4, Disk Paxos and pPaxos can even prove to be more expensive than the approach of using replica VMs.

	Write latency (RTTs) in common case		**Read latency (RTTs)**
	Fast path	Slow path	
Disk Paxos [71]	2	4	1
pPaxos [76]	2	4	1
Classic Paxos [90]	1	2	1
CPaxos	1	2	1

Table 5.1: Latency comparison of CPaxos to prior protocols. We assume no request conflict.

	When executing writes, # of copies of object's data transferred per write attempt	
	No conflict	n concurrent proposers
Disk Paxos [71]	$O(m)$	$O(m)$
pPaxos [76]	3	$O(n)$
Classic Paxos [90]	1	1
CPaxos	1	$O(1)$

Table 5.2: Data transfer cost comparison of CPaxos to prior protocols. m is the number of clients. For pPaxos, we assume older versions are garbage collected as soon as a new version is globally accepted.

5.1.3 Impact of latency variance

In addition to the limited interface they offer, cloud storage services also make low latency geo-replication challenging due to the high latency variance when reading/writing data from them [128]. In fact, latency variance gets exacerbated in the context of geo-replicated storage for two reasons.

- On the one hand, accessing replicated data without sacrificing consistency typically requires reading from or writing to *multiple* replicas. For example, to execute reads and writes using Paxos on an object with $2f + 1$ replicas, one must access at least $f+1$ replicas. This will lead to higher tail latencies than seen at any individual data center since read/write latency, which is constrained by the last of $f + 1$ responses,

	Average # of copies of object's data per replica
Disk Paxos [71]	m
pPaxos [76]	$\approx$1
Classic Paxos [90]	1
CPaxos	1

Table 5.3: Data storage cost comparison of CPaxos to prior protocols. m is the number of clients. For pPaxos, we assume older versions are garbage collected as soon as a new version is globally accepted.

will be high if *any one* response is slow.

- On the other hand, redundantly accessing more replicas than the minimum necessary to preserve consistency does not completely eliminate latency variance. For example, with Paxos, one can read from or write to all $2f + 1$ replicas and wait for the first $f + 1$ responses. However, when data is geo-replicated, the redundant requests are directed to replicas that are farther away and hence are limited in their ability to mitigate the impact of latency spikes at any of the closer replicas.

We empirically quantify the latencies when accessing geo-replicated data in the cloud as follows. We consider a web service that has web servers at all AWS data centers and replicates data across all Azure and AWS data centers. For any object, we refer to the subset of data centers from which the object is accessed as its *access set*. Using a 6 month long dataset of measurements, for every possible access set, we place replicas of the object being accessed at 3 data centers chosen at random but within the proximity of data centers in the access set (dataset and methodology for replica placement described later in Section 5.4). Figure 5.5 shows that, when we access only the closest $f + 1$ replicas for reading objects, compared to the median latency, the 99^{th} percentile latency increases by 250ms in the median case.[3] Moreover, even if we read from all $2f + 1$ replicas, there remains a 100ms gap between the tail and median latencies. This gap is significant as the inflation in tail latencies for individual

[3]Since the availability of both cloud services [40] and the Internet paths between their data centers is over 99.9% [77], failures have no impact on 99^{th} percentile read and write latency.

Metric to optimize	Technique	Section
Write latency	Treat cloud storage as a passive acceptor and use conditional-PUTs to update Paxos state	§ 5.2.1
Read latency	Clients act as learners, asynchronously updating Paxos state to mark a version as globally accepted	§ 5.2.2
Network transfer costs	Garbage collect old versions in Accept phase Store Paxos state as object's metadata so that clients need not transfer object data in Prepare phase	§ 5.2.3
Throughput under conflicts	Stateless Propose Assistant VMs to eliminate false propose failures	§ 5.2.4

Table 5.4: Techniques used by CRIC to minimize the cost and communication necessary for strongly consistent reads and writes.

reads and writes can degrade *median* user-perceived latency when a web service must execute many reads and writes to serve a user's request.

5.2 Global consistency atop limited interface

In this section, we describe how CRIC enables clients to directly read from and write to cloud storage, yet closely matches the cost and performance possible if cloud storage offered a richer interface. Specifically, despite having only a PUT-GET interface to storage, CRIC enables the use of Paxos to execute reads and writers in a manner that mimics the corresponding execution if cloud services were to expose the interface of Paxos "acceptors" and "learners" [89]. Table 5.4 summarizes the various techniques used in CRIC.

5.2.1 Low latency writes

To write to an object, CRIC relies on CPaxos (Cloud Paxos), a new variant of Paxos which we design to work with passive Paxos acceptors (i.e., acceptors that only store state but cannot run any computation). In CRIC, every object transitions through a sequence of monotonically increasing versions, where each version corre-

sponds to one CPaxos instance. If a client writes to a specific version of an object (e.g., a version that is one greater than the version it previously read), the CRIC library uses CPaxos to try to get a majority of acceptors to accept the specified data for the specified version and returns an error if a value has already been globally accepted for this version. If the client chooses last-writer-wins semantics, the CRIC library repeatedly retries the write with higher version numbers until it succeeds.

Like prior variants of Paxos [71, 76] designed for passive acceptors, CPaxos moves the acceptor and learner logic into the proposer. However prior Paxos variants for passive acceptors inflate latency and cost (see Table 5.1, 5.2, 5.3) because they require two rounds of communication for a proposer to complete each phase of Paxos: one round to perform conflict-free writes at a quorum of acceptors (e.g., write a new object [71] or perform an atomic append [76]) and another round to read the state of the acceptors to check for success.

Leveraging conditional updates. In contrast, CPaxos can complete each phase of Paxos in one round of communication in the common case, when there are no conflicts [99, 66]. To achieve this property, CPaxos leverages the *conditional-PUT* operation supported by all popular cloud storage services [19, 4, 26] that offer strong consistency. A conditional-PUT is a PUT augmented with a boolean condition, where the storage service executes the PUT only if the specified condition is true. For example, in Azure, a GET operation on an object returns the object's data and the time at which the object was last updated; a subsequent conditional-PUT can specify that the object should not have been updated since that time.

Leveraging the ability to perform conditional updates, proposers in CPaxos work as shown in Figure 5.6. A proposer first gathers the state from a majority of acceptors, a step which can often be omitted because 1) if an object is being created, it will have no state at any acceptor, whereas 2) when a client is updating an object, it has likely read that object in the past, enabling the CRIC library to cache acceptors' states.

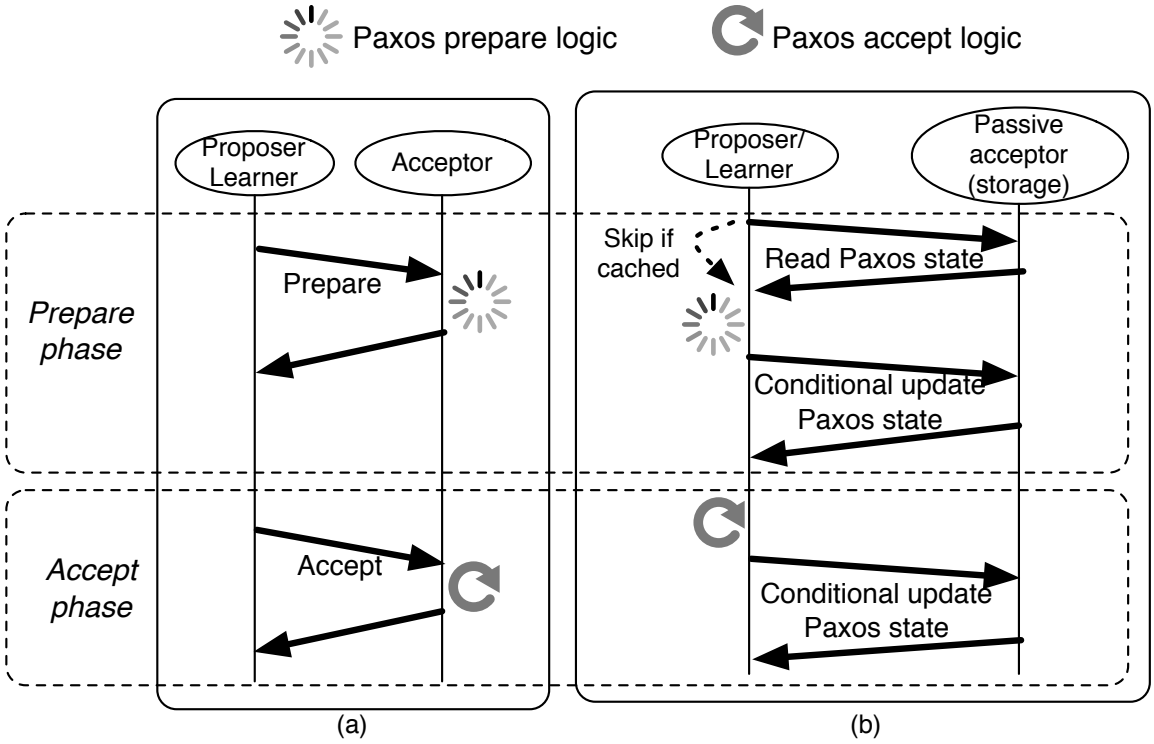


Figure 5.6: Comparison of (a) classic Paxos and (b) CPaxos.

For each acceptor, the proposer then runs the same logic that the acceptor would have executed upon receiving the proposer's Prepare message. If the proposer finds that a majority of acceptors would have promised to accept its proposal, it then attempts to update the state of these acceptors accordingly. Here, the proposer uses conditional-PUTs to ensure that it updates any acceptor's state only if the state has not already been updated since when the proposer read it. The Prepare phase of Paxos is complete once a majority of passive acceptors have been updated.

If some of the conditional updates fail and a majority of acceptors are not successfully updated, the proposer re-collects acceptor states, re-runs the logic for the Prepare phase (possibly with a higher proposal number), and tries to update the acceptors' states again. The proposer repeatedly does so until it either gets a majority of acceptors to promise to accept its proposal, or discovers that a conflicting value has already been globally accepted.

The Accept phase is similar to the Prepare phase: the proposer computes the new

acceptor states based on the Accept messages it wants to send to the acceptors and uses conditional-PUTs to update acceptors' states. The primary difference is that the proposer does not need to collect the acceptors' states at the start of the Accept phase, since it already knows the state of at least a majority of acceptors after its conditional updates succeed in the Prepare phase.

Enabling one round write. Taking inspiration from Fast Paxos [91], we can further reduce write latency to one round of communication. Fast Paxos has two types of execution flows: fast round and slow round. Different proposal numbers indicate different types of rounds. Whether a proposal number is for a fast round or a slow round is a predetermined information that all proposers know. In CRIC, we designate a special *proposal #0* as a fast round so that any write to an object can start with an attempt to fast accept the update by skipping the Prepare phase and running the Accept phase for *proposal #0*. The value is considered as committed once a super quorum $\lceil 3/4N + 1 \rceil$ of the object's N replicas accept the value. The fast round will succeed in the common case because conflicts are rare in typical web service workloads [99, 66].

The reason for requiring a super quorum in fast round is the same as in Fast Paxos [91]. Since any writer can propose a value in the fast round, when a reader collects the Paxos logs from a majority of replicas, there can be more than one value accepted by different acceptors in the fast round. To distinguish the globally accepted value among all the values, fast round requires acceptance from a super quorum of replicas so that the intersection between a super quorum and a regular quorum contains a majority of the regular quorum. This resolves the ambiguity as to which value is the globally accepted value since there can be only one value appearing at a majority of replicas in any regular quorum. $\lceil 3/4N + 1 \rceil$ of replicas is the minimum size of a super quorum that satisfies this requirement. During the Prepare phase, if a proposer observes multiple values accepted in *proposal #0* and none of them is

globally accepted (none of them appear at a majority of responses), like in Fast Paxos, it will pick the value with the highest frequency and try to get it globally accepted.

Failure of the fast round indicates that there may be conflicting writes, so in CPaxos, we only use *proposal #0* as the fast round. When the fast round fails, the proposer must fall back to the regular two-round (slow round) CPaxos protocol using a higher proposal number. This increases the likelihood and reduces the latency that one of the conflicting writers will be successful. To prevent starvation, writers should also use random backoff in slow rounds.

However, when objects are geo-replicated, using a fast round may not always lower write latency even when conflicts are rare. For example, consider an object with replicas in Azure's US-East, US-West, and Southeast Asia data centers. A client in Azure's Central-US data center has a latency of 40ms, 50ms, and 230ms, respectively to these 3 replicas. Using the fast round requires the client to contact all replicas, which takes 230ms. In contrast, a write using regular CPaxos will complete in 100ms, despite taking two rounds, because the client need only wait for responses from the closest majority of replicas.

Our current implementation of CRIC requires the application to specify whether to begin writes with a fast round. But, given the placement of an object's replicas, it is possible to identify whether a particular client would benefit from using a fast round to update the object.

5.2.2 Low latency reads

Since typical web service workloads are dominated by reads [51, 66], it is particularly vital to minimize read latency. Hence, in the common case, CRIC enables any client to read the latest version of an object in one round of communication with the closest majority of the object's replicas. We achieve this property by having clients act as learners.

We augment the CPaxos state for any version of an object with a commit bit to indicate whether this version has been globally accepted. After a client has its update to an object's data accepted by a quorum of acceptors, the client asynchronously issues conditional-PUTs to update the commit bit for this new version in the object's replicas. The utility of this commit bit is that, when a client wants to read the latest version of an object, it can issue GET requests to fetch the CPaxos logs for this object from all of the object's replicas but need not wait to hear from all replicas in order to identify the highest globally accepted version. Instead, once a reader is done fetching CPaxos logs from any majority of acceptors (typically the quorum of acceptors closest to it), if it finds the commit bit for the highest version enabled at any replica, the reader can safely use the corresponding data.

A client may find that the commit bit for the highest version it sees is not set at any of the majority of replicas that it first hears from, e.g., because this read occurs after a new version has been globally accepted but before the writer updates the commit bit for that version. In this case, the client needs to simply wait to receive CPaxos logs for the object from all replicas to recognize the committed version.

When storage services in some cloud data centers are unavailable, a reader may be unable to confirm that a particular version has been globally accepted even if it reads the CPaxos logs from a majority of replicas that are available; that version may have been accepted by a different majority of acceptors, some of whom are currently unavailable. In this case, a client identifies the highest version accepted at any of the acceptors from which it has read the CPaxos log. The client then re-proposes the data corresponding to the highest accepted proposal number for this version in order to either get this value globally accepted or discover a different value that is already globally accepted. The purpose of a reader performing such a write back is to ensure that, in case an object is an inconsistent state (e.g., because a client failed in the midst of performing a write), the object is left in a consistent state which reflects

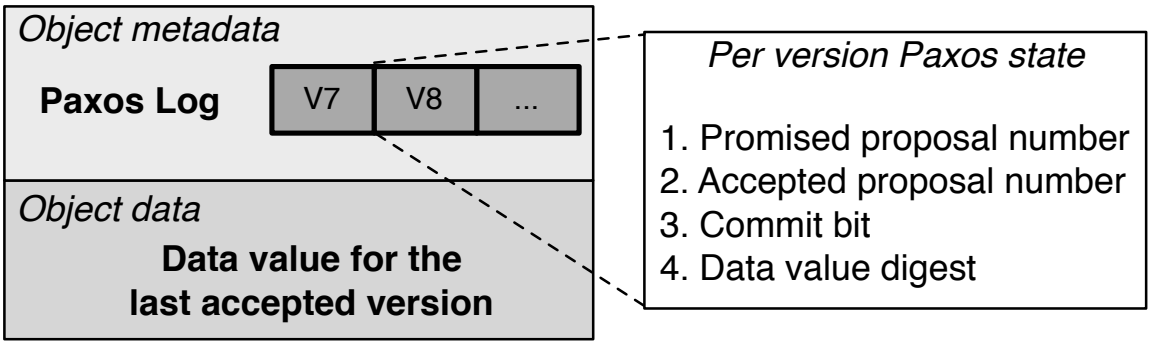


Figure 5.7: CRIC's format for storing objects in cloud storage.

the data returned to the application. Since failures in the cloud are rare [40, 37, 21], readers will seldom have to incur the latency overhead of performing write backs.

5.2.3 Minimizing cost

In comparison with a traditional usage of Paxos, CRIC's use of CPaxos to account for cloud storage's limited PUT-GET interface inflates the cost associated with network transfers in two ways: 1) from every acceptor, a reader has to fetch the entire CPaxos log; it cannot fetch only the last accepted version, and 2) since a client updates an acceptor's state by performing a conditional-PUT to the CPaxos log stored at that acceptor, the client has to redundantly transfer all data previously stored in the log. Next, we address both these sources of cost overhead.

Garbage collection of CPaxos logs. For executing reads as described earlier in Section 5.2.2, we observe that readers need to know only the highest version accepted by each acceptor. Therefore, we can afford to garbage collect CPaxos state for older versions during the Accept phase of a write. As a result, any replica of an object will contain data only for the last version of the object accepted at that replica.

A consequence of CRIC's garbage collection strategy is that the state for older versions at an acceptor is discarded without confirming whether any proposal for the new version has been globally accepted. Doing so does not violate safety because,

even if a reader finds that the highest version seen across a majority of acceptors is not globally accepted, as described earlier in Section 5.2.2, the reader will perform a write back to get some value accepted for this version. In settings where conflicts are known to be not so rare, one can minimize the chances of readers having to perform write backs by configuring CRIC so that writers garbage collect *some*, but not *all*, older versions in the Accept phase.

The key point to note here is that CRIC can safely afford to always store a single copy of an object's data at any replica. In contrast, to not violate safety, prior variants of Paxos designed for passive acceptors [71, 76] require every writer to create an *additional* copy of the object's data at a quorum of replicas. This in turn leads to overheads in network transfers as subsequent readers must necessarily fetch *multiple* copies of the object's data from each replica.

Separation of object data and Paxos state. During the Prepare phase of a write to an object, only the CPaxos state for that object needs to be updated but not the object's data. Therefore, to preempt writers in the Prepare phase from having to redundantly write back object data that they haven't modified, we leverage the fact that cloud storage services permit storing for any key a limited amount of metadata which can be modified independently of the key's value [18]. As shown in Figure 5.7, CRIC stores the CPaxos log at any of an object's replicas within that replica's metadata. Moreover, we also store within the metadata a *data digest*: a hash of the data last accepted at that replica. In low conflict workloads where replicas are typically in sync with each other, CRIC reduces cost by having a client read an object's data only from the closest replica and read only the metadata from the other replicas to confirm that they have the same data for the object. Note that the limits on object metadata (e.g., 8 KB on Azure [35]) suffice for storing the data digest and CPaxos state for the last accepted version as well as the CPaxos state for any subsequent versions not yet accepted.

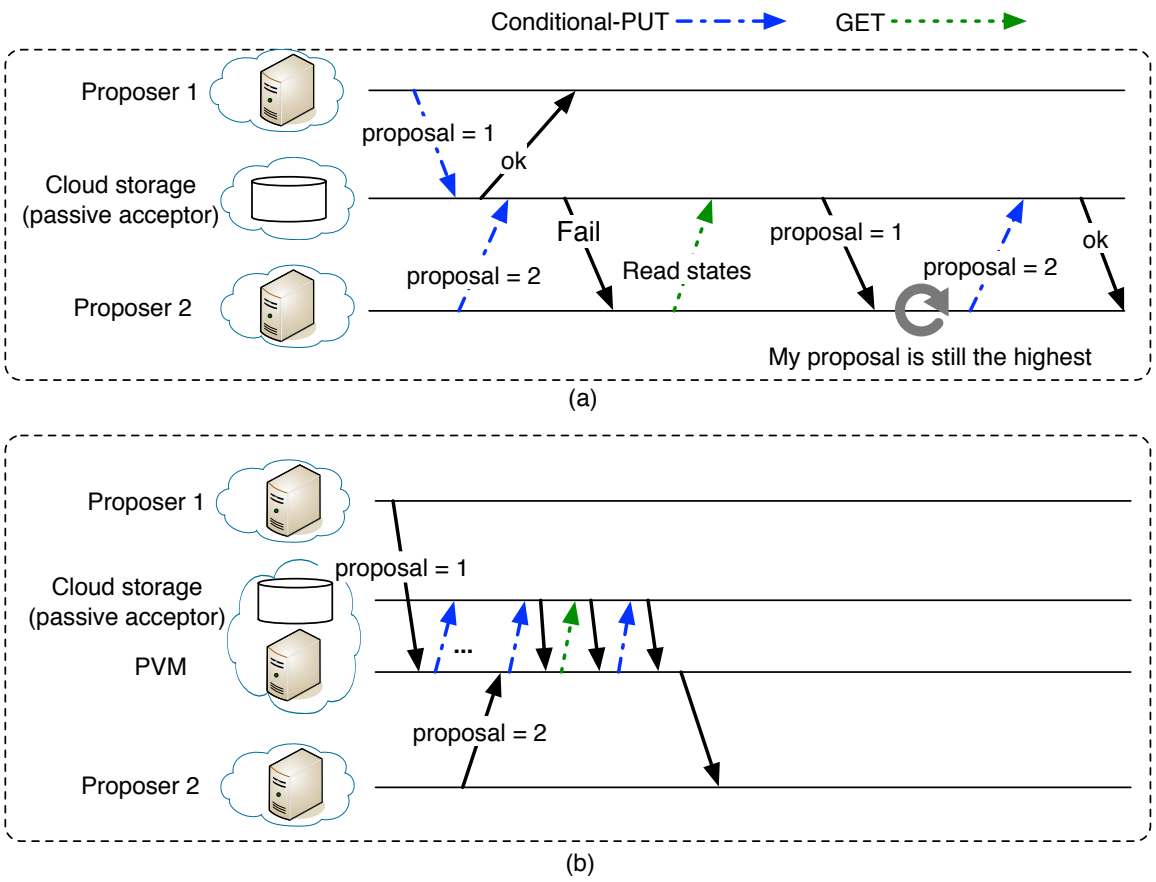


Figure 5.8: Illustration of (a) false propose failure, and (b) the use of Propose Assistant VMs.

5.2.4 Improving throughput under conflicts

Despite CRIC making do with passive Paxos acceptors, it is able to serve reads and writes with one round of communication when conflicts are rare. However, in comparison with classic Paxos, write throughput with CPaxos degrades more rapidly if conflicts become common.

To appreciate the cause for the throughput degradation, consider two clients trying to execute the Prepare phase on the same Paxos instance, as shown in Figure 5.8. *Client 1* proposes *proposal #1* and successfully updates the Paxos state at a majority of replicas. Before *Client 1* executes the Accept phase, *Client 2* proposes *proposal #2*. In classic Paxos, all acceptors will promise to accept *Client 2*'s proposal since it has a higher proposal number. However, in CPaxos, *Client 2*'s conditional-PUT

will fail since the acceptors' states after *Client 1*'s update are unknown to *Client 2*. *Client 2* needs to read the Paxos state again from storage, re-run the Paxos logic, and try again to obtain promises for its proposal. As we can see in Figure 5.8, this at least triples the write latency and the number of requests issued to cloud storage. Moreover, these extra rounds increase the window for contention between requests.

To reduce such false propose failures, we move the execution of the Prepare logic close to the storage by augmenting CPaxos with Propose Assistant VMs (PVMs) at every data center. A PVM only receives Prepare messages from proposers, reads and updates the metadata of the relevant object to run the Prepare phase's logic for the local acceptor, and returns the Prepare phase's result as well as metadata necessary for the proposer to issue subsequent conditional-PUTs (Figure 5.8(b)). For the Accept phase, proposers directly interact with cloud storage; a conditional-PUT that fails in this phase would fail even if relayed through PVMs because there must be another proposer who has updated the highest proposal number in the Paxos state.

Note that PVMs are stateless, and unlike replica VMs that proxy all client interactions with cloud storage (Section 5.1.2), PVMs never deal with any object's data. As a result, each PVM can sustain a much higher throughput than replica VMs. Furthermore, when conflicts are rare and writes can be successfully committed using the fast round, the execution of writes sidesteps PVMs.

5.3 Tackling latency variance

Our description of CRIC in the previous section outlined how CRIC is able to execute reads and writes in one round of wide-area communication in the common case. Now, we turn our attention to minimizing the tail of the read/write latency distribution.

Inadequacy of quorum-based replication. Paxos, the basis for CPaxos, implicitly accounts for stragglers as, in either phase of the protocol, a client needs to

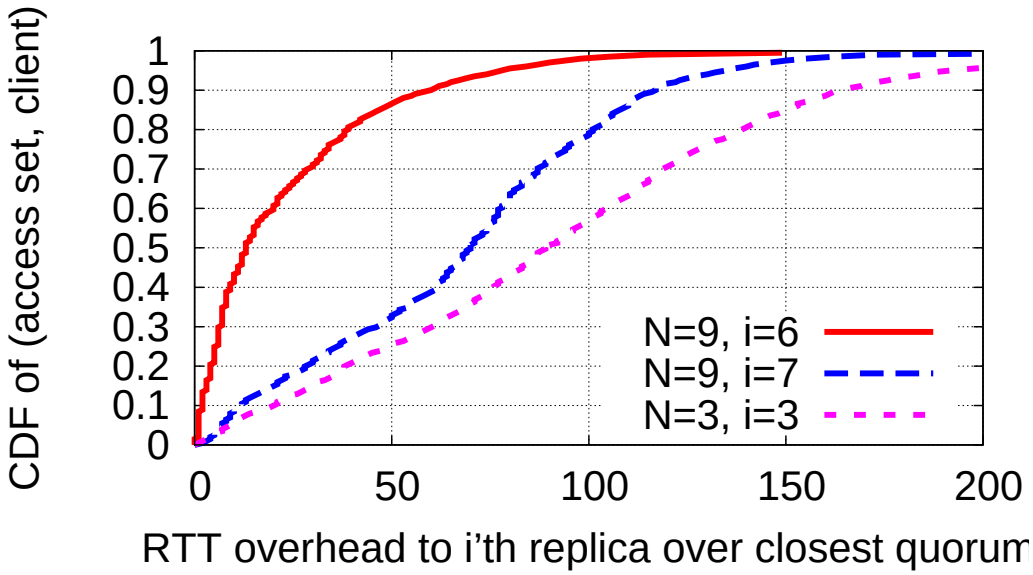


Figure 5.9: Latency overhead a client incurs to communicate with a farther replica compared to the quorum closest to it.

wait for responses from only a majority of acceptors. However, when replicas are in multiple locations, this approach has limited ability to reduce client-perceived latency variance. As we showed in Section 5.1.3, even when clients issue requests to all replicas and only wait for responses from a quorum, there remains a significant gap between the tail and median latencies. Using the same data used earlier in Figure 5.4, Figure 5.9 highlights the cause for this gap: since the 3^{rd} closest replica is 90ms farther from the client than the 2^{nd} closest replica in the median case, the 3^{rd} closest replica's response has to necessarily incur this additional delay before reaching the client.

To help mitigate this problem, one may consider storing replicas at more data centers than necessary to ensure fault-tolerance, e.g., storing replicas at 9 data centers despite needing to tolerate only $f = 1$ failure. By having replicas in more data centers, the RTT overhead of a client communicating with a replica farther than the closest quorum is significantly diminished; from Figure 5.9, with replicas at 9 data centers, the 6^{th} closest replica is less than 15ms farther than the 5^{th} closest replica in the

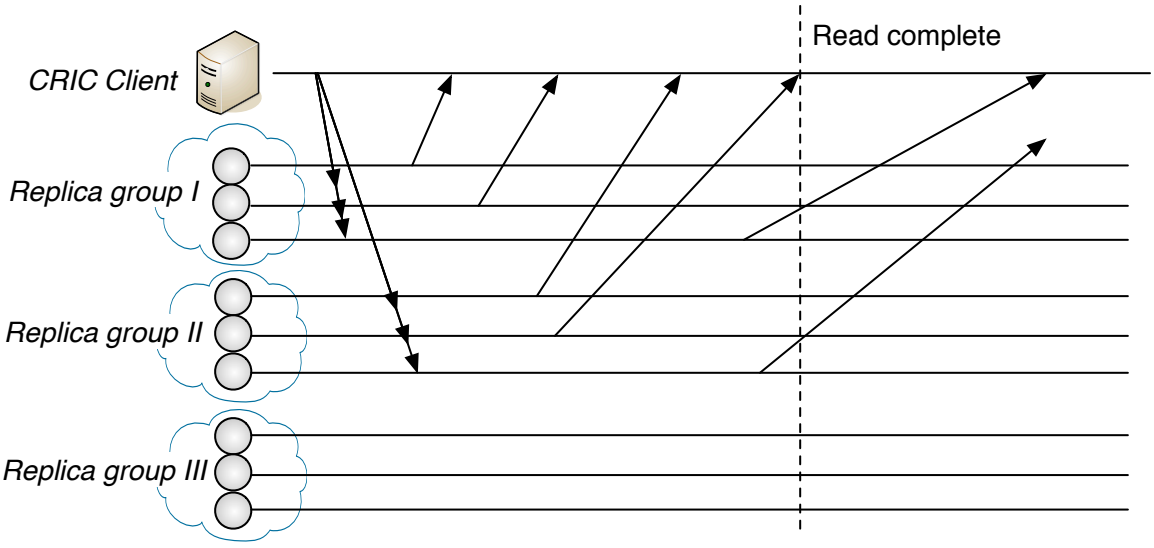


Figure 5.10: Illustration of hierarchical quorum.

median case.

However, as we show later in Section 5.4, inflation in tail latencies remains significant even with replicas at more data centers. This is because more replicas implies a larger quorum, and having to wait for a greater number of responses ends up inflating tail latencies such that more redundant responses become necessary to overcome the latency variance, nullifying the benefits of having more replicas. For example, when replicas are stored in 9 data centers, we find that clients need to wait for responses from at least 7 replicas to reduce tail latency, with additional redundant requests to farther replicas offering little benefit. As we see in Figure 5.9, when storing replicas in 9 data centers, RTT from a client to its 7^{th} nearest replica is higher than the RTT to the 5^{th} closest replica by an amount commensurate with the difference between the client's RTTs to the 2^{nd} and 3^{rd} closest replicas when using 3 replicas.

More replicas per data center. In order for tail latency to match median latency, our primary insight is that a client must rely on redundant responses only from the nearest quorum of data centers which host an object's replicas, since these are the data centers from which the client receives the first quorum of responses in the median case.

Based on this insight, our approach is to create additional replicas of an object at each one of $2f + 1$ data centers, rather than creating replicas at more data centers. The number of replicas of an object to store at any data center can be chosen based on the extent of latency variance at that data center.

Given these additional replicas, a natural way to execute reads and writes would be to have every client wait for responses from a quorum of replicas. To see why this approach can prove to be problematic, consider an object whose replicas are stored at three data centers D_1, D_2, and D_3, with 1 replica each at D_1 and D_2 and 3 replicas at D_3. A client that is proximate to D_1 and D_2 will have to wait for responses from all 3 data centers for *every* read/write in order to receive responses from a quorum of replicas. Thus, while the redundancy at D_3 helps tame the latency variance at that data center, doing so can end up inflating the *median* latency for many clients.

Hierarchical quorums. To prevent such undesirable outcomes, we leverage a two-tiered quorum-based approach, hierarchical quorum [87], which groups replicas stored in the same data center as a replica group. In both the Prepare and Accept phases of a CPaxos write and when executing a read, a client must receive positive responses from a quorum of replica groups, where the response from a group is complete once the client receives positive responses from a quorum of replicas within that group. It is easy to see that a quorum of quorum intersects with any other quorum of quorum. Here, we give a formal definition of a hierarchical quorum system and a proof of the intersection property of using hierarchical quorums.

Definition V.1. Given a set $S = \{s_1, s_2, ..., s_n\}(n > 1)$, a hierarchical quorum system **HQ** is a quorum system [119] over a set of quorum systems $Q = \{q_1, q_2, ..., q_m\}(m > 1)$ that $\forall q_i : q_i$ is a quorum system over a non-empty subset of S.

Theorem V.2. *Two hierarchical quorums have non-empty intersection over S.*

Proof. $\because \forall nq_1, nq_2 \subset \mathbf{HQ}$, nq_1 and nq_2 have non-empty intersection over Q, and $\forall nq \subset nq_1 \cap nq_2$ and $\forall q_1, q_2 \subset nq$, q_1 and q_2 have non-empty intersection over S.

$\therefore nq_1$ and nq_2 have non-empty intersection over S. □

Figure 5.10 shows an example of three replica groups and three replicas in each group. To execute reads and writes, a client must receive positive responses from at least four replicas, two replicas each from two replica groups. Thus, in comparison with the approach of treating all replicas as equal, hierarchical quorum not only has no impact on median latency but also reduces the number of replicas that a client must wait for responses from, resulting in lower tail latencies.

When using hierarchical quorum in CPaxos's fast round for writes, at each of a super quorum $\lceil 3/4n + 1 \rceil$ of data centers, a client's conditional-PUTs must succeed at a super quorum $\lceil 3/4m + 1 \rceil$ of replicas. Here, n is the number of replica groups, and m is the number of replicas in a replica group. Here, we provide a proof of correctness of using super hierarchical quorum in CPaxos's fast round.

Lemma V.3. *In Fast Paxos, any two super quorums and any regular quorum have non-empty intersection of replicas.*

Proof (see [91]). □

Theorem V.4. *Using super hierarchical quorum in CPaxos fast round guarantees safety (only one value can be committed for a given version of an object and only committed values can be read).*

Proof. Based on [91], the requirement for Fast Paxos to guarantee safety is that any two fast round quorums and any regular round quorum have non-empty intersection. Therefore, to prove that using hierarchical quorums can guarantee safety, we only need to prove that any two super hierarchical quorums and any regular hierarchical quorum have non-empty intersection of replicas.

Based on Lemma V.3, any two super hierarchical quorums and any regular hierarchical quorum have non-empty intersection of replica groups. Similarly, within each

replica group, any two super quorums and any regular quorum have non-empty intersection of replicas. Therefore, any two super hierarchical quorums and any regular hierarchical quorum have non-empty intersection of replicas. □

Like any approach that uses redundancy to reduce latency variance, the use of hierarchical quorum increases cost (to store more replicas, to issue requests to storage, and to transfer more data). CRIC reduces this cost overhead by leveraging the data digests that CPaxos stores in an object's metadata. For example, when reading the 3 copies of an object in a replica group, a client need read the data from only 2 of them and read only the metadata from the third, thus reducing network transfer costs by a third.

5.4 Evaluation

We evaluate CRIC in three parts. First, in a prototype deployment of CRIC, we evaluate the latency and throughput performance under different conflict rates, in comparison with alternative designs not optimized for the cloud. Second, to demonstrate CRIC's cost-effectiveness and its ability to reduce tail latency in a broader setting, we conduct a simulation-based evaluation on a large-scale dataset of measurements from 39 cloud data centers. Finally, we showcase the utility of CRIC on a real-world web service's workload. The primary take-aways from our evaluation are:

- CRIC's performance is comparable to approaches that require replica VMs, but at 20–50% lower cost.
- Compared to existing replication protocols compatible with passive Paxos acceptors, CPaxos can halve write latency and also lower cost by 10–60%.
- Hierarchical quorums can reduce tail read latency by 50ms in the median case, a more than 50% reduction over what traditional quorums can achieve.

5.4.1 Prototype Evaluation

Implementation. Our prototype implementation of CRIC is roughly 5400 lines of Java code. We use Thrift [112] for RPCs between VMs, and interact with every cloud storage service using the official client libraries. Our prototype is compatible with Microsoft Azure and Amazon AWS.

To compare the performance of CRIC with existing solutions, we also implemented two comparison systems:

- **Fast Paxos:** We implement Fast Paxos to represent a range of existing solutions, such as MDCC [86] and TAPIR [131], that would require replica VMs to replicate data in the cloud. Although these systems provide richer functionality such as transactions, using them to offer a globally consistent view of key-value storage would offer performance similar to the Fast Paxos protocol. Our implementation of Fast Paxos contains two components: a client library that implements a Fast Paxos proposer, and code that runs on VMs which proxy requests to cloud storage and mimic Fast Paxos acceptors and learners.
- **pPaxos*:** Among prior Paxos-based replication protocols that work with passive acceptors, pPaxos [76] is strictly better than Disk Paxos [71] in terms of performance and cost. Therefore, we compare CRIC only with pPaxos. Our implementation, pPaxos*, improves pPaxos's efficiency by using fast round writes and optimal garbage collection.

  Not all cloud storage services support Append operations as needed by pPaxos. Even storage services that do support Append may impose restrictions, e.g., though Azure Storage supports Append Blobs, updating and deleting existing blocks in a Blob is not supported, thus preempting garbage collection. We therefore implement Appends via replica VMs that interpose on requests to storage but ignore the cost of these VMs in our comparisons.

Deployment setting. We deploy clients in A2 instances at 5 Azure data centers: East US, West US, Japan, West Europe, and Southeast Asia. In the blob storage service at each of these data centers, we store a copy of one million objects. When replica VMs are necessary, we deploy a sufficient number of them for these VMs to not be a bottleneck.

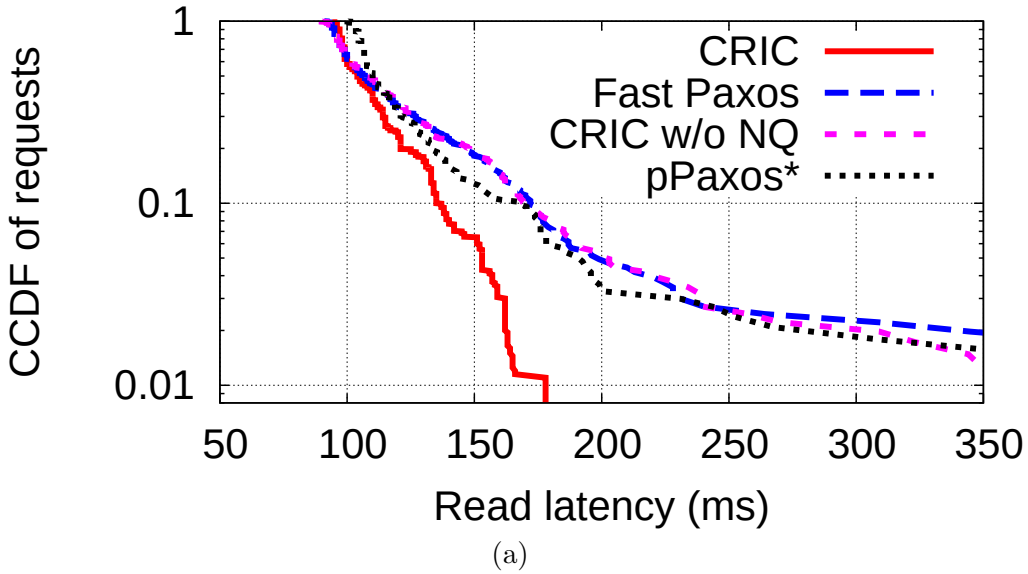
Clients issue reads and writes using YCSB [65], a key value store benchmark which emulates cloud workloads. We vary the read-to-write ratio, average object size, and Zipfian distribution that reflects how popularity varies across objects. We vary the Zipfian coefficient in the range $[0.5, 1)$. The higher the value, the higher the rate of conflicts.

Performance under low conflict rates. First, when conflicts are rare, we show that CRIC 1) offers performance comparable to approaches that would require replica VMs for deployment in the cloud, and 2) outperforms prior replication protocols compatible with passive acceptors. In this experiment, we use 1KB objects, set the read-to-write ratio to 30, and set the Zipfian coefficient to 0.5.

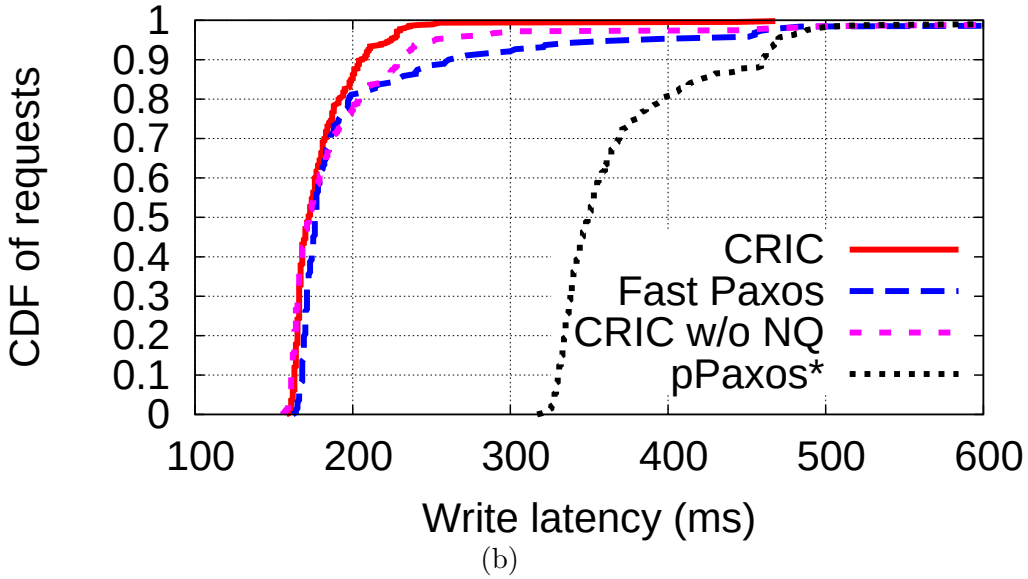
Figure 5.11 shows the distribution of read and write latencies observed by clients in the Azure East US region. In terms of read latency, Fast Paxos, pPaxos*, and CRIC all offer similar latencies since all of them require one round trip to a majority of replicas to execute reads in the common case. Whereas, for writes, latencies with pPaxos* are double that with CRIC and Fast Paxos. This is because, after appending to acceptors' logs, a pPaxos* proposer has to read back those logs to check whether its request was accepted. For both reads and writes, CRIC's use of hierarchical quorum enables it to reduce the 99^{th} percentile latency by over 100ms.

Note that, since we deploy a sufficient number of replica VMs when they are necessary, when conflicts are rare, throughput is constrained only by cloud storage.

Performance under high conflict rate. Next, we showcase CRIC's ability to minimize performance degradation even if conflicts are common. Here, we set the



(a)



(b)

Figure 5.11: (a) Read and (b) write latencies under low conflicts.

read-to-write ratio to 1 and use 1KB objects.

Under high conflict rates, a client may require multiple rounds of communication to complete a read or write, worsening both latency and throughput. Write-write conflicts cause the fast round to fail, and multiple proposers compete in one CPaxos instance. For read-write conflicts, readers may see an inconsistent state due to ongoing writes and perform write backs, which may then interfere with ongoing writes.

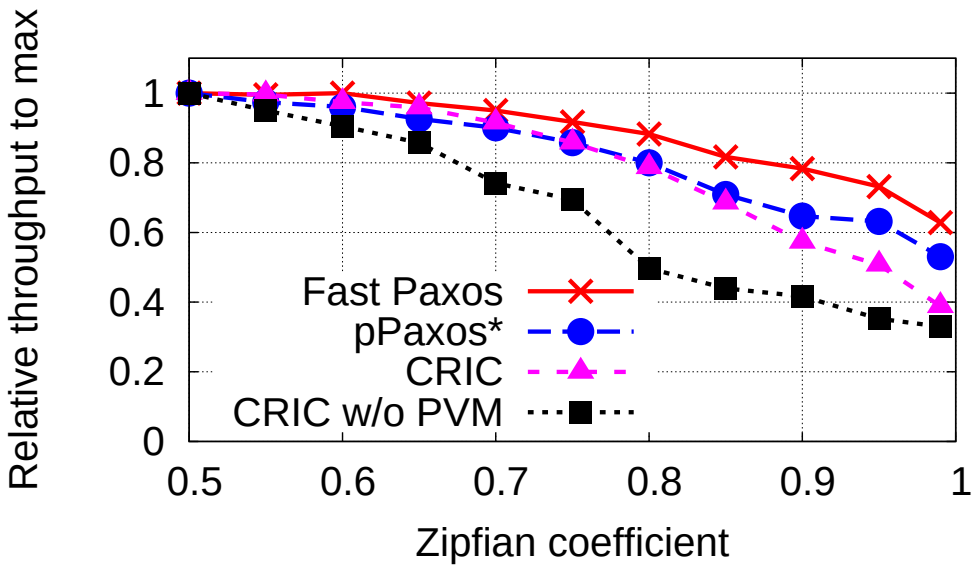


Figure 5.12: Degradation in write throughput under different conflict rates.

Performance degradation can be further amplified due to false propose failures (Section 5.2.4).

Write-write conflicts. Figure 5.12 shows how write throughput degrades with different replication approaches under different conflict rates; we normalize values compared to the throughput achieved when conflicts are rare. As conflicts increase, CRIC's throughput without PVMs degrades at a significantly faster rate than when using PVMs. For example, when the Zipfian coefficient is 0.8, throughput when using CRIC without PVMs is less than half that achieved in the absence of conflicts. With PVMs, CRIC comes close to the throughput achieved with Fast Paxos, achieving a normalized throughput of 0.8.

As conflicts increase (Zipfian coefficient of 0.9 and higher), CRIC starts to degrade faster than Fast Paxos due to the failure of conditional-PUTs in the Accept phase. In this case, CRIC requires an extra round to read passive acceptors' states whereas Fast Paxos's replica VMs can include an acceptor's state when rejecting a client's Accept message. However, under such high conflict rates, it would be more prudent to use pessimistic concurrency control.

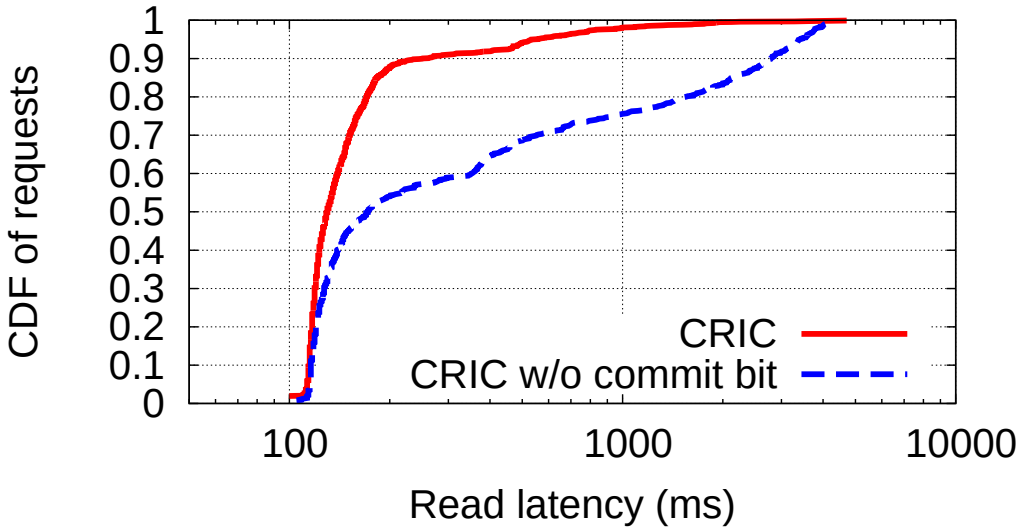


Figure 5.13: Client perceived read latency distribution when conflict rate is high.

Note that false propose failures do not happen in pPaxos* since a proposer knows the exact status of its request after reading the passive acceptors' logs. However, we still see that throughput with pPaxos* degrades relative to Fast Paxos at a rate similar to CRIC. This is because a long log of requests accumulate in acceptors' logs under high contention, increasing processing time at proposers.

Read-write conflicts. In CRIC, when a write to an object is globally accepted, but has not yet been applied to all replicas, clients who read that object may have to perform a write back. CRIC uses the commit bit to reduce the incidence of such write backs.

In our experimental setup, Figure 5.13 shows the read latency distribution observed at Azure's Japan data center when the Zipfian coefficient is set to 0.8. We see that having writers asynchronously mark their completed write as committed greatly improves read latencies; as long as the commit bit is set at any replica with highest version, a reader need not perform write backs.

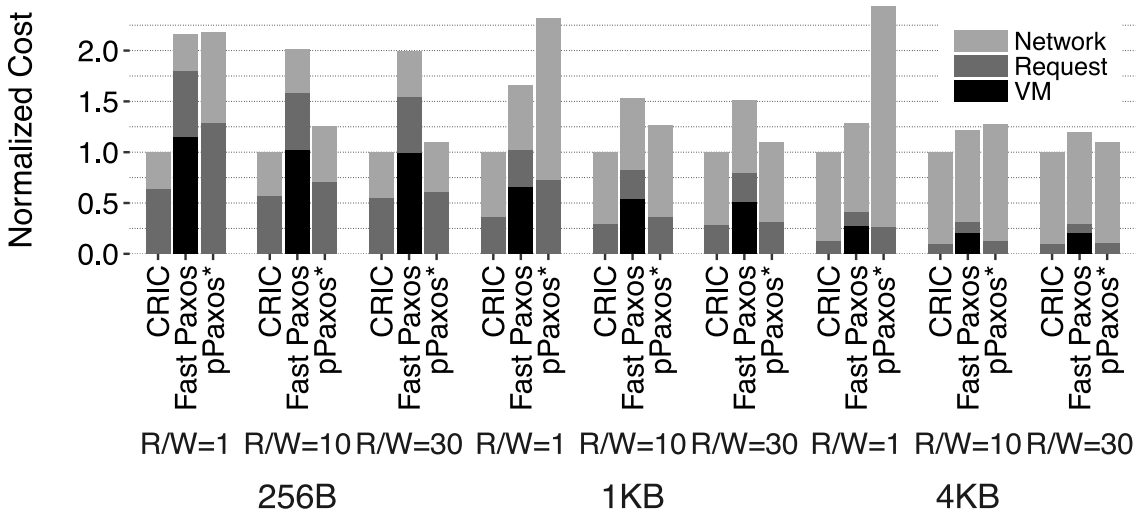


Figure 5.14: Comparison of cost necessary to execute reads and writes on geo-replicated data; R/W is the read-to-write ratio.

5.4.2 Cost

While we have shown thus far that latencies and throughput with CRIC are comparable to those with Fast Paxos when conflicts are rare, using Fast Paxos incurs significant cost overhead due to the need for replica VMs to augment the storage interface. Figure 5.14 compares CRIC against Fast Paxos and pPaxos* in terms of the cost necessary to sustain any specific throughput; we consider three read-to-write ratios (30, 10, and 1), three object sizes (256 bytes, 1KB, and 100KB), and a Zipfian coefficient of 0.5. These parameter choices are informed by prior studies of web service workloads [51, 66, 59].

We see that CRIC reduces cost by 20–50% compared to Fast Paxos and pPaxos* in our target setting: conflicts are rare and objects are small [59, 51]. In such workloads, CRIC's cost savings over Fast Paxos stem from more efficient use of VMs, and the savings compared to pPaxos* are due to efficient use of storage and network resources. Only if both objects are large and the read-to-write ratio is low, do CRIC's cost benefits reduce. Note that, even though, like CRIC, pPaxos* too does not require

replica VMs, its use may result in higher cost than Fast Paxos when objects are large. This is because a pPaxos* proposer has to always *append* a new copy of an object's data to any acceptor's log and has to therefore transfer multiple versions when reading the log to check the status of its proposal.

5.4.3 Tackling latency variance

Simulation setting. To assess CRIC's tail latency performance and associated cost overhead in a broader range of settings, we use a dataset containing measurements from 39 cloud data centers: 14 in Amazon AWS and 25 in Microsoft Azure. We collect this dataset by continually measuring latencies between all pairs of the 39 data centers, as well as response times for 1KB GETs and PUTs at the storage service in each of those data centers. Our dataset contains more than 100K measurement samples per pair of data centers and 100K measurement samples per GETs and PUTs at every data center. By simulating CRIC's operation using this data, we are able to conduct a more comprehensive evaluation of CRIC than possible with our prototype.

Like in Section 5.1.3, we consider a web service that has web servers in each Azure data center and replicates data across all Azure and AWS data centers; we refer to the subset of data centers from which an object is accessed as its *access set* and consider 2000 randomly chosen access sets. For every access set, we choose the set of data centers in which to store replicas using a strategy that combines random selection (for fault-tolerance) with affinity for data centers in the access set (for performance). We group data centers by continents, and evenly distributes replicas across continents in which there is a data center from the access set, randomly choosing data centers within any continent. For each access set, we choose 10 random replica placements based on this strategy and report the median performance.

We compare the tail latency improvements enabled by CRIC's use of hierarchical quorum when storing replicas at 3 data centers against those enabled when storing

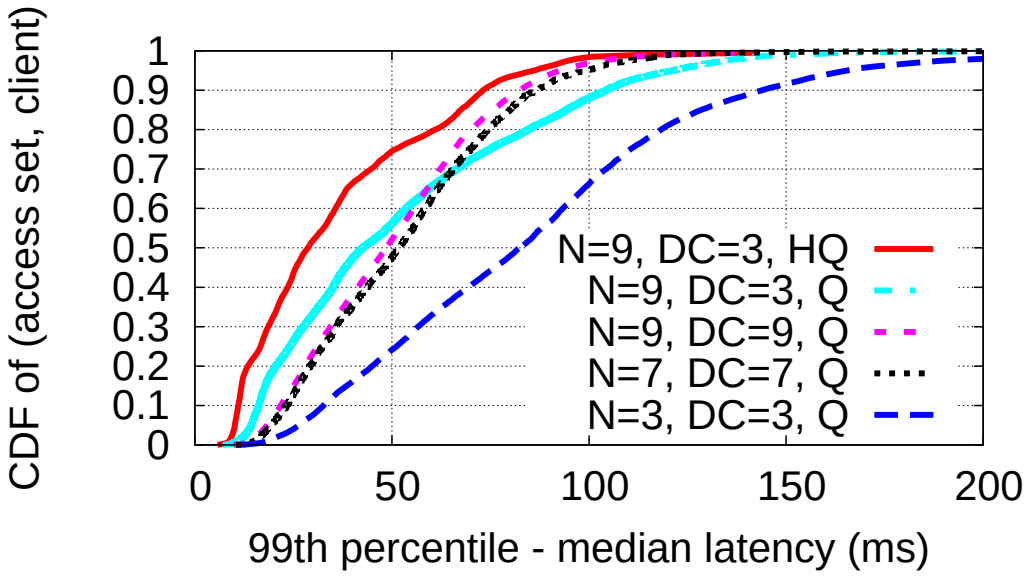


Figure 5.15: Utility of hierarchical quorums in reducing tail latency. N is the number of replicas and DC is the number of data centers.

replicas at additional data centers and treating all replicas as equal.

Ability to reduce tail latency. For every client in each of the access sets considered, Figure 5.15 shows the absolute latency difference between the 99^{th} percentile and median latencies for the two approaches. We use hierarchical quorums to store 3 replicas each in 3 data centers, and compare it with replication configurations that (1) place 3 replicas each in 3 data centers but use the traditional quorums to execute requests, (2) place 9 replicas in 9 different data centers, and (3) place 7 replicas in 7 different data centers (to match the minimum number of replica failures hierarchical quorums cannot tolerate). Although when using the traditional majority quorum, issuing requests to all replicas and waiting for the first majority responses can reduce tail latency by 42ms in the median case, there remains a 45ms gap between the tail and median latencies. This is because, when replicas are spread out, redundancy to farther away replicas has little utility in reducing latency variance. However, since hierarchical quorum creates redundancy within the closest quorum of data centers and the size of responses it requires is smaller than the traditional quorum, it can further reduce the gap between tail and median latencies. From Figure 5.15, we can

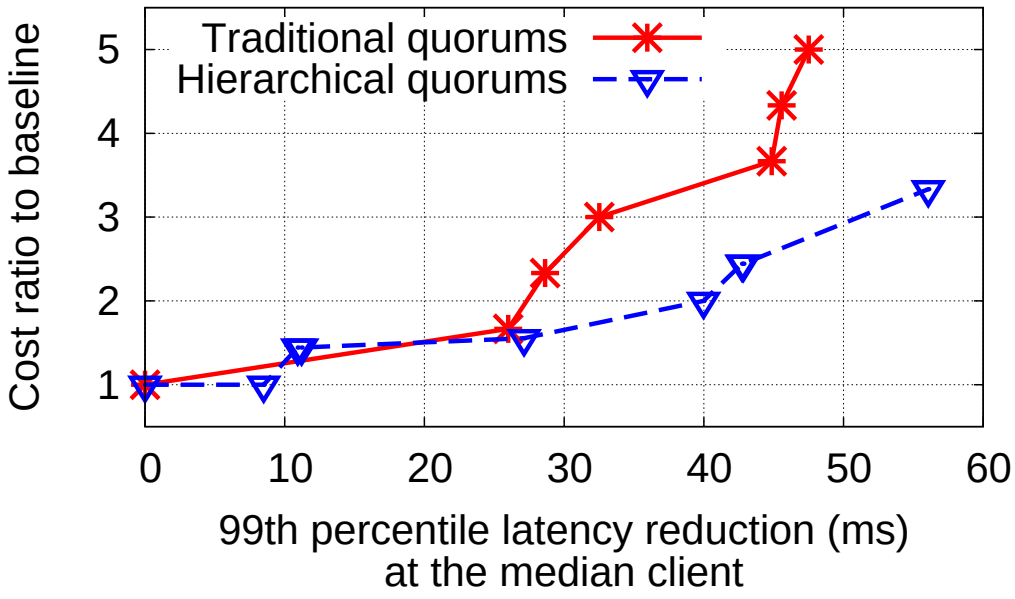


Figure 5.16: Utility of hierarchical quorums in enabling better cost vs. tail latency trade-offs.

see that hierarchical quorum can reduce the gap between tail and median latencies to less than 30ms in the median case.

Tail latency vs. cost tradeoff. Hierarchical quorums not only enable low latency variance, but also incur low cost overhead in achieving those latency benefits. To evaluate the tradeoff enabled by different quorum approaches, we consider different replication configurations by varying the number of replicas, and then determine the tail latency and cost in each case. For hierarchical quorum, we consider replicas at 3 data centers, but vary the number of replicas per data center from 1 to 5. With traditional quorums, we consider 1 replica per data center and vary the total number of replicas from 1 to 15.

For both hierarchical quorum and traditional quorums, Figure 5.16 shows one point for each replication configuration. For each configuration, we show the tail latency reduction seen at the median client against the cost overhead over the baseline configuration with 1 replica each at 3 data centers. Compared to traditional quorums,

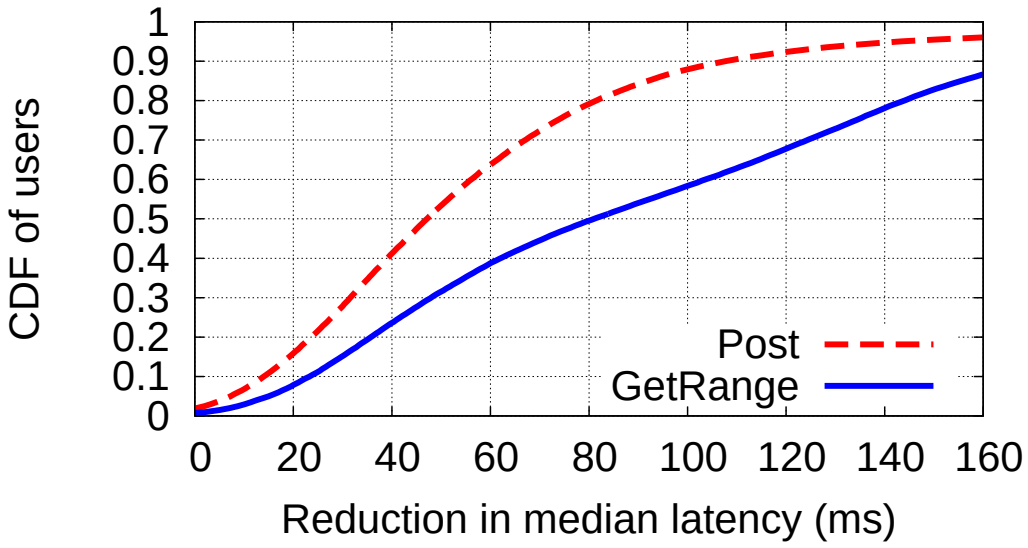


Figure 5.17: In a Twitter clone, CRIC improves performance compared to Fast Paxos for users with at least 100 followers.

hierarchical quorums can provide the same tail latency performance at much lower cost, or better tail latencies at the same cost. Hierarchical quorums thus provide web service developers with a better tail latency versus cost tradeoff than when using traditional quorums.

5.4.4 Application case study

In the final part of our evaluation, we estimate CRIC's benefits by considering the characteristics of a real-world web service's workload. We use data from Twitter as an example of a latency-sensitive geo-distributed web service, even though Twitter only offers eventual/causal consistency.

We simulate a strongly consistent deployment of Retwis [32], an open source Twitter clone; once a user's post is accepted by the service, the user's followers are guaranteed to see the post. Our simulation considers 1 million users, uses location information in their profiles [95] to map each user to be served by a web server in the closest AWS data center, and picks the number of followers and followees for every user based on publicly available information [38]. We compare CRIC's performance

to Fast Paxos.

We focus on the latency for two types of operations: *GetRange* operations fetch the last 100 tweets in a user's timeline (the set of tweets posted by those that the user follows), and *Post* operations represent a user posting a tweet. With *GetRange* operations, Retwis first reads the user's timeline object to get the IDs for all posts on her timeline, and then reads 100 posts (in parallel). For *Post* operations, there are three steps: 1) write the post, 2) read the list of the user's followers (done in parallel with writing the post), and 3) insert the post ID into their timelines. All posts and timeline objects are replicated as per the replica placement strategy described in § 5.4.3, whereas a user's list of followers/followees is stored only at the data center which serves that user.

For the 40% of users who have over 100 followers, Figure 5.17 shows the reduction in *median* latency for *GetRange* and *Post* operations.

5.5 Discussion

Improving read performance. Read latency with CRIC could be lowered by using read leases, like in Megastore [53]. Whether a read lease has been granted on a particular replica of the object can be stored in the metadata of that copy of the object. Once a client completes a read, it can perform a conditional-PUT to update the metadata at the replica closest to it, thereby enabling clients close to this replica to complete reads by reading only from this replica. This improvement in read latency will come at the expense of higher write latency as writers will now have to invalidate read leases at all replicas.

Alternatively, if an application can make do with reads returning stale data, CRIC is also extensible to reduce read latency without compromising on the linearizability of writes. In this case, to read an object, a client need fetch only the closest replica of the object. The application can then use the highest version that has the commit

bit set in the CPaxos log at this replica. Only if this client attempts to update this object in the future will it have to read from a quorum of the object's replicas.

Support for transactions. CRIC can be used to execute multi-key transactions on geo-replicated data if the cloud storage service within each data center offers support for transactions. For example, on Azure, CRIC can leverage the Table service's support for batched updates [31] to execute a transaction as a batch of conditional updates; on any particular replica, the batch is applied only if all the conditional updates within the batch succeed. In the absence of transaction support from cloud storage, CRIC can still be used to execute transactions with an additional round of latency to acquire locks (stored in the metadata of each replica) at a quorum of replicas.

5.6 Summary

Our work was motivated by the observation that a key challenge today in deploying a web service that spans data centers in the cloud is for the web service to itself manage the consistency of data replicated across data centers. With CRIC, we have demonstrated that it is indeed feasible to address this shortcoming efficiently at the application layer without requiring cloud providers to modify cloud storage either to augment its interface or to reduce its latency variance. We hope our work will spur more web service providers to opt for geo-distributed deployments, thereby realizing the untapped potential for cloud providers to democratize low latency web services.

CHAPTER VI

Conclusions

6.1 Thesis Contributions

This dissertation presents my effort to support my thesis: ***it is practical to cost-effectively offer better abstractions and support for latency SLOs on legacy cloud storage services.***

My thesis research can greatly ease the burden on the developers of geo-distributed cloud web services. It focuses on addressing the problems faced by web services in the cloud due to the poor abstractions and high latency variance offered by cloud storage services. More specifically, the work presented in this dissertation makes the following contributions:

Measurement study quantifying cloud storage latency performance and reducing latency variance of using cloud storage. My measurements of the Azure and S3 storage services highlight the high variance in latencies and point out the main source of the latency variance offered by these services. To address this issue, I designed CosTLO to improve predictability of using cloud storage services without having to wait for these services to modify their infrastructure. It is a framework that requires minimal changes to applications and judiciously combines several redundancy forms to execute storage requests to reduce latency variance in a cost effective manner.

Automating the process of placing geographically distributed replicas

in the cloud. I developed the SPANStore system which provides a unified view of geographically distributed storage services across multiple clouds to web services deployed in the cloud. It can automate the process of trading off cost and latency, while satisfying various application requirements such as data consistency and fault-tolerance.

Efficient strongly consistent geo-replication in the cloud. I designed and implemented CRIC, which enables strongly consistent data replication across cloud storage services with limited interface, and tackles latency variance on shared data replicated in multiple cloud storage services. I demonstrate that it is indeed feasible to address this shortcoming efficiently at the application layer without requiring cloud providers to modify cloud storage either to augment its interface or to reduce its latency variance.

Though this dissertation focuses on developing client side solutions to improve the utility of cloud storage services, many techniques can also be adopted by cloud providers to improve their storage services internally. For example, CosTLO's technique of using multiple forms of redundancy can be used by cloud storage services to provide latency SLOs for their users. With detailed information on storage implementation, cloud providers can devise a more accurate latency model in CosTLO's latency estimation.

Although cloud storage services may evolve in the future and provide functionalities proposed in this dissertation, the fact that no cloud providers would like to provide data synchronization services with other cloud providers shows the fundamental value of this dissertation. For web service providers who choose to adopt a multi-cloud deployment for the associated cost, performance, and fault-tolerance benefits, they still require solutions such as SPANStore to replicate data across multiple clouds and CRIC to keep data consistent. Therefore, this dissertation provides long-term benefits for web services deployed in the cloud.

6.2 Future Work

My thesis research demonstrates that there is a big gap between what cloud applications need from cloud services and what cloud services currently provide. The desire of solving this issue leads me to my future work.

Cloud system design to meet different application requirements. Emerging cloud applications such as self-driving cars, wearable devices, and Internet of Things may have very different performance, data consistency, fault tolerance and resource requirements compared to traditional applications. For example, the data consistency requirement for self-driving cars is likely to be distance based: for cars that are nearby, the data shared among those cars has to be strongly consistent to prevent accidents, but for cars that are hundreds of miles away, it is likely that the data shared between these cars does not need to be strongly consistent. Identifying those additional requirements and designing cloud systems that are tunable to meet different applications' needs are challenging. Moreover, I'd like to build solutions that follow the same underlying principle as SPANStore: automating the cloud system reconfiguration process to suit different application requirements. One path I'd like to explore is to leverage machine learning techniques to enable system reconfiguration automation.

Redesigning distributed system interface. As I have demonstrated in CRIC, there is currently a mismatch between the functionalities that a cloud service provides, and the functionalities that an application desires. This issue is especially enlarged by the trend that more and more emerging web services and applications are relying on some third party services that they do not have control over. Therefore, I argue that system interface design should be an important aspect of future cloud system design. For example, instead of providing a simple GET/PUT interface for key-value storage, incorporating some forms of request dependency will make the storage more usable. It will also make web service developers' life easier in reasoning about storage

behavior, even though they do not have visibility inside the storage system. This line of research is going to require a deeper analysis and study of different types of distributed systems and applications, as well as revisiting the theoretical aspect of distributed systems in order to generalize rules or common patterns for designing interfaces of the future.

Make cloud infrastructure more configurable. Moreover, I'm interested in exploring new possibilities in designing cloud infrastructures in a way that cloud resources can be highly configurable by cloud applications and third party service providers to enrich the functionalities of the cloud. Existing cloud model suffer from a key limitation: they only enable third-parties to offer new functionality that is independent from any service offered by the cloud provider. As I have pointed out the gap between cloud providers' offerings and cloud applications' needs, one way to bridge this gap is to redesigning cloud service marketplaces such that third parties can also augment existing cloud services. As illustrated in my vision paper [127], leveraging emerging hardware technologies such as programmable switches, we can redesign cloud service marketplaces to enable efficient deployment and use of third party add-ons. An add-on's VMs can interpose on an application's interactions with a cloud service, but need to do so only for a portion of the application's traffic. The net effect of my proposed redesign of cloud service marketplaces is a win-win for all parties involved: cloud providers benefit from a greater rate of innovation, applications can avail richer functionality without sacrificing performance, and third-party developers incur low cost.

6.3 Summary

In this dissertation, I present CosTLO, SPANStore, and CRIC, systems that offer better abstractions of cloud storage, low complexity of using cloud storage in geo-distributed deployments, and help tackle the latency variability intrinsic to cloud

storage. These systems are designed as middlewares and libraries that sit in between applications and cloud services, and enable web service providers to make better performance, cost, consistency, and availability trade-offs without requiring any changes to cloud services. Note that cloud providers can also easily incorporate these systems into their future offerings. I hope my work will spur more web services to opt for geo-distributed deployments in the cloud, thereby realizing the untapped potential for cloud providers to democratize low latency web services. I hope the outcome of this research can lead to a greater innovation in cloud environments.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] 100% uptime anybody? http://www.riskythinking.com/articles/article8.php.

[2] 250+ Amazing Twitter Statistics. http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/.

[3] Amazon AWS. http://aws.amazon.com.

[4] Amazon DynamoDB PutItem. http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_PutItem.html.

[5] Amazon EC2 Service Level Agreement. http://aws.amazon.com/ec2/sla/.

[6] Amazon S3. http://aws.amazon.com/s3.

[7] Amazon S3 FAQs. http://aws.amazon.com/s3/faqs/.

[8] Amazon S3 Pricing. http://aws.amazon.com/s3/pricing/.

[9] Amazon Web Services Outage Reveals Critical Lack of Redundancy Across the Internet. http://www.geekwire.com/2017/amazon-web-services-outage-reveals-critical-lack-of-redundancy-across-the-in

[10] Announcing Amazon S3 Reduced Redundancy Storage. http://aws.amazon.com/about-aws/whats-new/2010/05/19/announcing-amazon-s3-reduced-redundancy-storage/.

[11] AWS Case Studies. http://aws.amazon.com/solutions/case-studies/all/.

[12] AWS Global Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/.

[13] Azure. http://azure.microsoft.com/.

[14] Azure Blob Storage Pricing. http://azure.microsoft.com/en-us/pricing/details/storage/blobs/.

[15] Azure Documentation: Virtual Machines. http://azure.microsoft.com/en-us/documentation/articles/virtual-machines-size-specs/.

[16] Azure Regions. https://azure.microsoft.com/en-us/regions/.

[17] Azure Replicated Table Library. http://github.com/Azure/rtable.

[18] Azure Set Blob Metadata. http://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/set-blob-metadata.

[19] Azure Storage Documentation: Specifying Conditional Headers for Blob Service Operations. http://msdn.microsoft.com/en-us/library/dd179371.aspx.

[20] Azure Storage Pricing. http://azure.microsoft.com/en-us/pricing/details/storage/.

[21] CloudSquare Service Status. https://cloudharmony.com/status-1year-group-by-regions-and-provider.

[22] Dropbox. http://www.dropbox.com.

[23] Facebook. http://www.facebook.com.

[24] Google Cloud Platform. http://cloud.google.com/.

[25] Google Cloud Storage. http://cloud.google.com/ storage.

[26] Google Cloud Storage: Update object. http://cloud.google.com/storage/docs/json_api/v1/objects/update.

[27] Google Docs. http://docs.google.com.

[28] Infographic: Who is using Twitter, how often, and why? http://www.theatlantic.com/ technology/archive/2011/07/infographic-who-is-using-twitter-how-often-and-why/241407/.

[29] 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.

[30] Page View Statistics for Wikimedia Projects. http://dumps.wikimedia.org/other/pagecounts-raw/.

[31] Performing Entity Group Transactions. https://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/performing-entity-group-transactions.

[32] Retwis. http://retwis.antirez.com.

[33] Roundup Of Cloud Computing Forecasts And Market Estimates Q3 Update, 2015. http://www.forbes.com/sites/louiscolumbus/2015/09/27/roundup-of-cloud-computing-forecasts-and-market-estimates-q3-update-2015/.

[34] Server Access Log Format - Amazon Simple Storage Service. http://docs.aws.amazon.com/AmazonS3/latest/dev/LogFormat.html.

[35] Setting and Retrieving Properties and Metadata for Blob Resources. http://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/Setting-and-Retrieving-Properties-and-Metadata-for-Blob-Resources?redirectedfrom=MSDN.

[36] ShareJS. https://github.com/josephg/ShareJS/.

[37] The Cloud Provider with the Best Uptime in 2015. http://www.networkworld.com/article/3020235/cloud-computing/and-the-cloud-provider-with-the-best-uptime-in-2015-is.html.

[38] Tweets Loud and Quiet. https://www.oreilly.com/ideas/tweets-loud-and-quiet.

[39] VMware vFabric Hyperic. http://www.vmware. com/products/datacenter-virtualization/vfabric-hyperic/.

[40] 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.

[41] Windows Azure Service Level Agreements. http://azure.microsoft.com/en-us/support/legal/sla/.

[42] Windows Azure Storage Logging: Using Logs to Track Storage Requests. http://blogs.msdn.com/b/windowsazurestorage/archive/2011/08/03/windows-azure-storage-logging-using-logs-to-track-storage-requests.aspx.

[43] Amazon Found Every 100ms of Latency Cost Them 1% in Sales. http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, 2008.

[44] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of the 1st Annual Symposium on Cloud Computing*, 2010.

[45] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[46] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[47] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. A. Golding, A. Merchant, M. Spasojevic, A. C. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 2001.

[48] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, 2010.

[49] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Rao. Improving web availability for clients with MONET. In *Proceedings of the 2nd USENIX Conference on Networked Systems Design and Implementation*, 2005.

[50] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. C. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[51] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[52] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006.

[53] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Conference on Innovative Data Systems Research*, volume 11, pages 223–234, 2011.

[54] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2011.

[55] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[56] D. Bermbach, M. Klems, S. Tai, and M. Menzel. MetaStorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *IEEE International Conference on Cloud Computing*, 2011.

[57] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DEPSKY: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th ACM European Conference on Computer Systems*, 2011.

[58] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st Annual Symposium on Cloud Computing*, 2010.

[59] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[60] J. Brutlag. Speed matters for Google web search. `http://services.google.com/fh/files/blogs/google_delayexp.pdf`, 2009.

[61] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[62] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2007.

[63] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[64] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and open AppEngine application development and deployment. In *CloudComp*, 2009.

[65] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st Annual Symposium on Cloud Computing*, 2010.

[66] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, 2012.

[67] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[68] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[69] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized erasure codes for distributed networked storage. *IEEE/ACM Transactions on Networking (TON)*, 14(SI):2809–2816, 2006.

[70] R. Escriva and R. van Renesse. Consus: Taming the paxi. *Computing Research Repository*, 2016.

[71] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[72] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, 1979.

[73] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference*, 2009.

[74] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental consistency guarantees for replicated objects. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, 2016.

[75] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[76] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. E. Anderson, and D. Wetherall. Metasync: File synchronization across multiple untrusted storage services. In *Proceedings of the USENIX Annual Technical Conference*, pages 83–95, 2015.

[77] O. Haq, M. Raja, and F. R. Dogar. Measuring and improving the reliability of wide-area cloud paths. In *Proceedings of the World Wide Web Conference*, 2017.

[78] A. Haurie and P. Marcotte. On the relationship between Nash-Cournot and Wardrop equilibria. *Networks*, 15(3):295–308, 1985.

[79] K. He, L. Wang, A. Fisher, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure. In *Proceedings of the 13th ACM SIGCOMM Conference on Internet Measurement*, 2013.

[80] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[81] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[82] Z. Hu, L. Zhu, C. Ardi, E. Katz-Bassett, H. V. Madhyastha, J. Heidemann, and M. Yu. The need for end-to-end evaluation of cloud availability. In *Proceedings of the 15th International Conference on Passive and Active Measurement*, 2014.

[83] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. Supercloud: Opportunities and challenges. *ACM SIGOPS Operating Systems Review*, 49(1):137–141, 2015.

[84] E. Katz-Bassett, H. Madhyastha, J. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proceedings of the 5th USENIX Conference on Networked Systems Design and Implementation*, 2008.

[85] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

[86] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[87] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE transactions on Computers*, 40(9):996–1004, 1991.

[88] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[89] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 1998.

[90] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[91] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[92] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010.

[93] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, 2012.

[94] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, 2016.

[95] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: Unified and discriminative influence model for inferring home locations. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012.

[96] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23th Symposium on Operating Systems Principles*, 2011.

[97] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 313–328, 2013.

[98] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, 2016.

[99] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[100] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[101] H. V. Madhyastha, J. C. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat. scc: Cluster storage provisioning informed by application characteristics and SLAs. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[102] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical report, Univ. of Texas, 2011.

[103] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the ACM Symposium on the Theory of Computing*, 1997.

[104] J.-P. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2002.

[105] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th Symposium on Operating Systems Principles*, 2013.

[106] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the AM SIGMOD Conference*, 2015.

[107] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[108] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[109] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[110] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[111] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, 2016.

[112] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[113] S. Souders. Velocity and the bottom line. `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`, 2009.

[114] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23th Symposium on Operating Systems Principles*, 2011.

[115] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[116] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam. To move or not to move: The economics of cloud computing. In *Proceedings of the ACM Workshop on Hot Topics in Cloud Computing*, 2011.

[117] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th Symposium on Operating Systems Principles*, 2013.

[118] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[119] M. Vukolić. Quorum systems: With applications to storage and consensus. *Synthesis Lectures on Distributed Computing Theory*, 3(1):1–146, 2012.

[120] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, 2013.

[121] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[122] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the IEEE/ACM International Symposium on Quality of Service*, 2001.

[123] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[124] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2011.

[125] T. Wood, E. Cecchet, K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *Proceedings of the ACM Workshop on Hot Topics in Cloud Computing*, 2010.

[126] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th Symposium on Operating Systems Principles*, 2013.

[127] Z. Wu and H. V. Madhyastha. Rethinking cloud service marketplaces. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 134–140. ACM, 2016.

[128] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[129] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 2008.

[130] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[131] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[132] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the 24th Symposium on Operating Systems Principles*, 2013.

[133] Z. Zhang, M. Zhang, A. Greenberg, Y. C. Hu, R. Mahajan, and B. Christian. Optimizing cost and performance in online service provider networks. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, 2010.