Cost-Effective Geo-Distributed Storage for Low-Latency Web Services

Zhe Wu and Harsha V. Madhyastha
University of Michigan

Abstract

We present two systems that we have developed to aid geo-distributed web services deployed in the cloud to serve their users with low latency. First, CosTLO [23] helps mitigate the high latency variance observed when accessing data stored in multi-tenant cloud storage services. To cost-effectively satisfy a web service's tail latency goals, CosTLO uses measurement-driven inferences about replication and load balancing within cloud storage to combine the use of multiple forms of redundancy. Second, to support web services that enable users to share data, SPANStore [22] leverages application hints about access patterns and knowledge about cloud latencies and pricing to cost-effectively replicate data across data centers. Key to SPANStore's design is to spread data across the data centers of multiple cloud providers, in order to present better cost vs. latency tradeoffs than possible with the use of any single cloud provider. In this paper, we summarize the key takeaways from our work in developing both systems.

1 Introduction

Minimizing user-perceived latency is a critical goal for web services, because even a few 100 milliseconds of additional delay can significantly reduce revenue [6]. 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 [3, 4].

However, web services deployed in the cloud are faced with two challenges in ensuring that users have low latency access to their data.

- **High latency variance.** Since cloud storage services are used concurrently by multiple tenants, both fetching and storing content on these services is associated with high latency variance. For example, when we had 120 PlanetLab nodes across the globe each download 1 KB file from their respective closest Microsoft Azure data center, for over 70% of these nodes, the 99^{th} percentile and median download latencies 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 [16], 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 Amazon S3 more than doubles the *median* page load time for 50% of PlanetLab nodes when fetching a web page containing 50 objects.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- **Cost vs. latency tradeoff.** Collaborative services that enable users to share data with each other must also determine which data centers must store replicas of each object. While replicating all objects to all data centers can ensure low latency access [19], 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 parameters mean that no single deployment provides the best fit for all applications and all objects.

In this paper, we describe two systems that we have developed to address these challenges. First, CosTLO (Cost-effective Tail Latency Optimizer) enables application providers to avail of the cost benefits enabled by cloud services, without having latency variance degrade user experience. Second, SPANStore (Storage Provider Aggregating Networked Store) is a key-value store that presents a unified view of storage services present in several geographically distributed data centers.

CosTLO's design is based on three principles. First, since our measurements reveal that the high latency variance in cloud storage is caused predominantly by isolated latency spikes, CosTLO augments every GET/PUT request with a set of redundant requests [15,21], so that it suffices to wait for responses to a subset of the requests. Second, to tackle the variance in all components of end-to-end latency—latency over the Internet, latency over the cloud service's data center network, and latency within the storage service—CosTLO combines the use of multiple forms of redundancy, such as issuing redundant requests to the same object, to multiple copies of an object, to multiple front-ends, or even to copies of an object in multiple data centers. Lastly, since the number of configurations in which CosTLO can implement redundancy is unbounded, to add redundancy in a manner that satisfies an application's goals for tail latency cost-effectively, we model the load balancing and replication within cloud storage services in order to accurately capture the dependencies between concurrent requests.

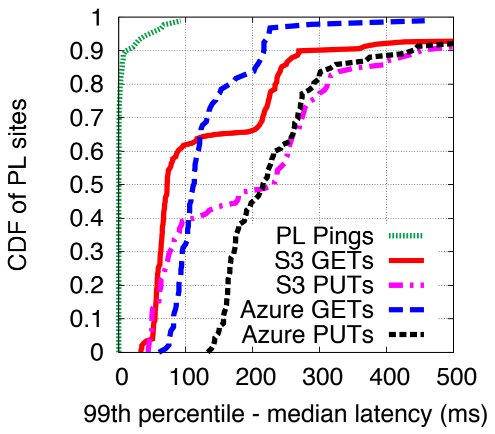
The key premise underlying SPANStore is that, when geo-replicating data, spreading data across the data centers of multiple cloud providers offers significantly better latency versus cost tradeoffs than possible with any single cloud provider. This is not only because the union of data centers across cloud providers results in a geographically denser set of data centers than any single provider's data centers, but also because the cost of storage and networking resources can significantly differ across cloud providers. To exploit these factors to drive down the cost incurred in satisfying application providers' latency, consistency, and fault tolerance goals, SPANStore judiciously determines where to replicate every object and how to perform this replication. For every object that it stores, SPANStore makes this determination 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.

Our goal in this paper is not to provide a detailed description of the design and implementation of either system and our results from evaluating them; we refer the interested reader to [23] and [22] for those details. Here, we instead summarize the main takeaways from both CosTLO and SPANStore in enabling low latency geo-distributed storage in the cloud.

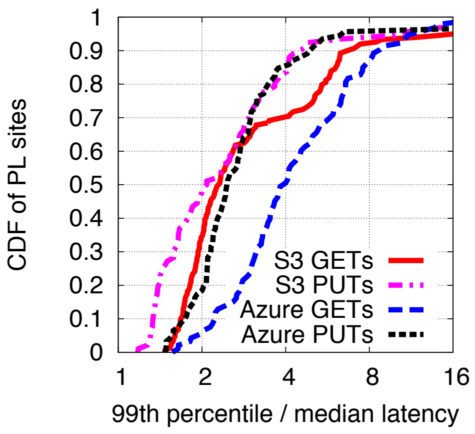
2 Motivation

We begin by presenting measurement data that motivates our design of CosTLO and SPANStore.[1]

[1]Though the data presented here was gathered between 2013 and 2015, similar trends exist today.



(a)



(b)

Figure 1: (a) Absolute and (b) relative inflation in 99^{th} percentile latency with respect to median. Note logscale x-axis in (b).

2.1 Characterizing latency variance

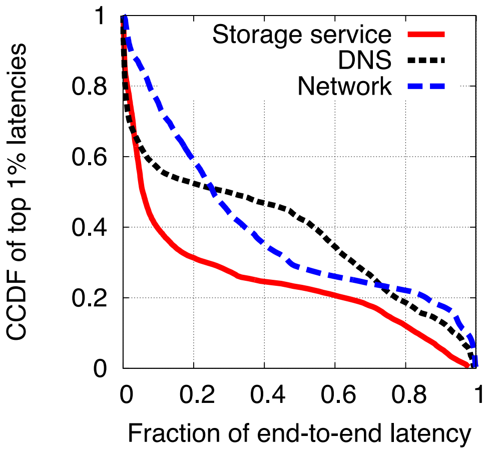
Overview of measurements. To analyze client-perceived latencies when downloading from and uploading to cloud storage services, we gather two types of measurements for a week. First, we use 120 PlanetLab nodes across the world as representative end-hosts. Once every 3 seconds, every node uploaded a new 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 [1] and Azure [7] 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.

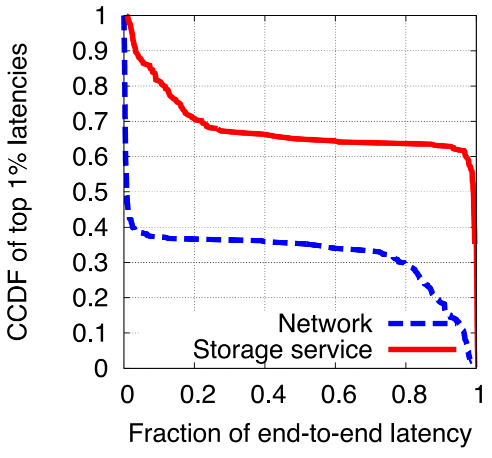
Quantifying latency variance. Figure 1 shows the distribution across nodes of the spread in latencies; for every node, we plot the absolute and relative difference between the 99^{th} percentile and median latencies. In both Azure and S3, the median PlanetLab node sees an absolute inflation greater than 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. Figure 1 shows that this high latency variance is not due to high load or slow access links of PlanetLab nodes; for every node, the figure plots the difference between 99^{th} percentile and median latency to the node closest to it among all PlanetLab nodes.

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

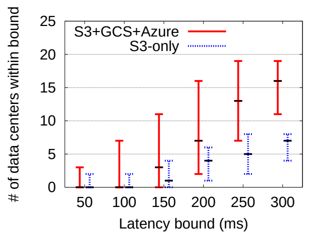


(a) End-host perspective

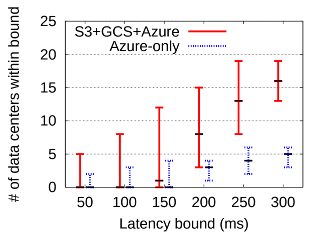


(b) VM perspective

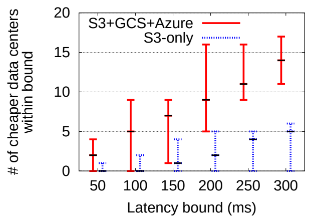
Figure 2: Breakdown of components of tail latencies.



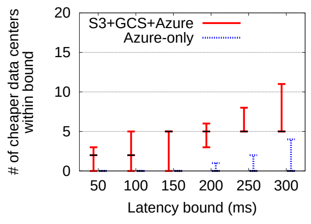
(a) EC2



(b) Azure



(c) EC2



(d) Azure

Figure 3: 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.

Second, Figure 2(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 2(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.*

2.2 Why multi-cloud?

Next, we motivate the utility of spread data across multiple cloud providers by presenting data which shows that doing so can potentially lead to reduced latencies for clients and reduced cost for applications.

Lower latencies. We first show that using multiple cloud providers can enable lower GET/PUT latencies. For this, we instantiated VMs in each of the data centers in EC2, Azure, and Google Cloud. From every VM, we measured 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 3 shows how many other data centers are within a given latency bound of each EC2 [3(a)] and

Azure [3(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. An application 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 front-ends). 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.

Lower cost. Replicating data across multiple cloud providers also enables support for latency SLOs at potentially lower cost due to the discrepancies in pricing across providers. Figures 3(c) and 3(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 front-ends, 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, an application can use these cheaper options to reduce costs.

3 Cost-effective support for lower latency variance with CosTLO

Goal and Approach. 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. 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.

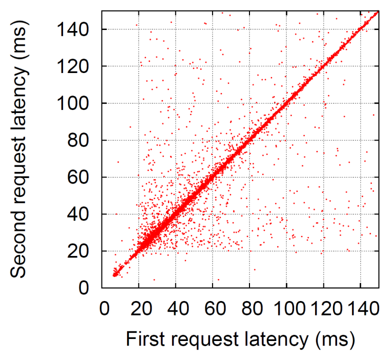
Since tail latency samples are dominated by isolated spikes, our high-level approach is to augment any GET/PUT request with a set of redundant requests, so that either the first response or a subset of early responses can be considered. Though this is a well-known approach for reducing tail latencies [9, 15, 21], CosTLO is unique in exploiting several ways of issuing redundant requests in combination.

3.1 Effective redundancy techniques

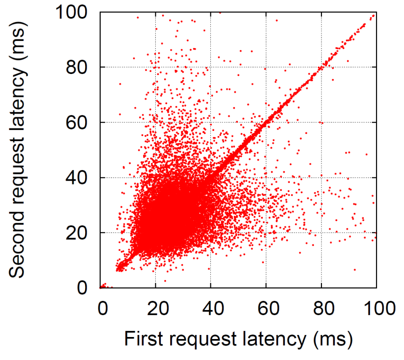
A wide range of redundancy approaches are feasible to tackle variance in each component of end-to-end latency. Here, we summarize what our measurements reveal to be the most effective techniques.

Latency over the Internet. To examine the utility of different approaches on reducing Internet tail latencies, we issued pairs of concurrent GET requests from each PlanetLab node in three different ways and then compared the measured tail latencies with those seen with single requests.

First, relying on load balancing in the Internet [12], when every end-host concurrently issues multiple requests to the storage service in the data center closest to it, relative latency inflation seen at the median PlanetLab node remains close to 2x. Second, in addition to a GET request to its closest S3 region, having every node issue a GET request in parallel to its second closest S3 region offers little benefit in reducing latency variance. This is because, for most PlanetLab nodes, the second closest region within a cloud service is too far to help tame latency spikes to the region closest to the node. Therefore, to reduce variance in latencies over the Internet, it



(a) Azure GETs



(b) S3 GETs

Figure 4: Scatter plot of first vs. second request GET latency when issuing two concurrent requests to the same object stored in a cloud storage service.

is crucial for redundancy to exploit multiple cloud providers. 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.

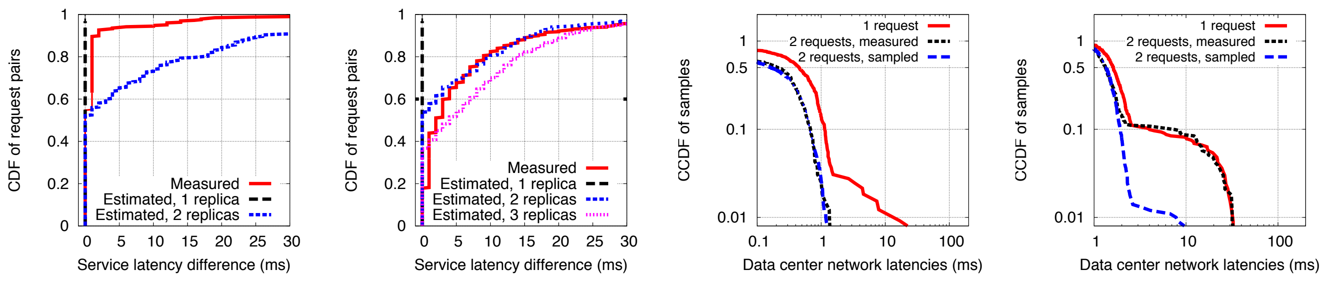
Data center network latencies. Latency spikes within a cloud service's data center network can either be addressed implicitly by issuing the same PUT/GET request multiple times in parallel to exploit path diversity or addressed explicitly by relaying each request through a different VM. In both S3 and Azure, we find 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.

Storage service latencies. Lastly, we considered 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 can 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.

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. 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 2 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 2 due to Azure's serialization of PUTs issued by the same tenant [14].

3.2 Selecting cost-effective configuration

The primary challenge in combining these various forms of redundancy in CosTLO is to select for each edge network a redundancy configuration that helps meet the application's tail latency goals at minimum cost. For this, CosTLO 1) takes as input the pricing policies at every data center, 2) uses logs exported by cloud providers



(a) (b) (c) (d)

Figure 5: 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).

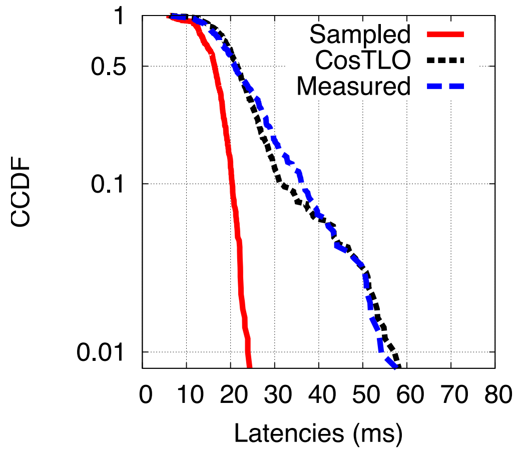
to characterize the workload imposed by clients in every prefix, and 3) employs a measurement agent at every data center. Every agent gathers three types of measurements: 1) pings to a representative end-host in every prefix, 2) pairs of concurrent GETs and pairs of concurrent PUTs to the local storage service, and 3) the rates at which VMs can relay PUTs and GETs between end-hosts and the local storage service without any queueing.

Given these measurements, CosTLO identifies a cost-effective configuration for end-hosts in each IP address prefix. We refer the reader to [23] for a description of the algorithm CosTLO uses to search through the configuration space, and focus here on the key question in doing so: for any particular configuration for an IP prefix, how do we estimate the latency distribution that clients in that prefix will experience when served in that configuration?

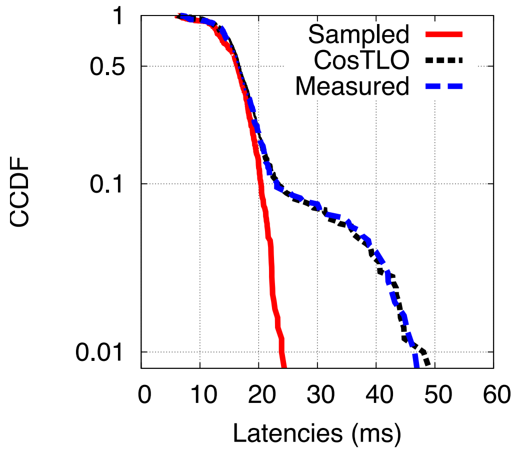
The reason it is hard to estimate the latency distribution for any particular redundancy configuration is due to dependencies between concurrent requests. While Figure 4 shows the correlation in latencies between two concurrent GET requests to an object at one of Azure's and one of S3's data centers, 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 [5] and S3 [2] are known to store 3 replicas of every object, Figures 5(a) and 5(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



(a) Requests to same object



(b) Requests to different objects

Figure 6: Accuracy of estimating GET latency distribution for 8 concurrent GET requests from VM to local storage service in one S3 region. Note logscale on y-axis.

storage services store every object on a randomly chosen server (e.g., by hashing the object's name for load balancing [16]), 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 5(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 5(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.

By leveraging these models of replication and load balancing, CosTLO is able to accurately estimate the latency distribution when sets of requests are concurrently issued to a storage service. Figures 6(a) and 6(b) compare the measured and estimated latency distributions when issuing 8 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 8 concurrent GETs by independently sampling the latency distribution for a single request 8 times and considering the minimum, we significantly under-estimate the tail of the distribution.

4 Cost-effective geo-replication of data with SPANStore

Thus far, we have assumed that every object stored by an application is stored in a single data center, and CosTLO reduces tail latencies for executing GETs and PUTs on such objects. Now, we turn our attention to addressing the storage needs of web services in which sharing of data among users is an intrinsic part of the service (e.g., collaborative applications like Google Docs, and file sharing services like Dropbox), and geo-replication of shared data is a must to ensure low latency. Instead of every user having to access a centrally located copy of an object, replication of the object across data centers permits every user to access a nearby subset of the object's copies.

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.[2] The key benefit here is that the onus of navigating the space of replication strategies on an object-by-object basis is offloaded to SPANStore, rather than individual application developers having to deal with this problem, as is the case today. We are guided by four objectives: 1) minimize cost, 2) respect latency SLOs, 3) tolerate failures, and 4) provide support for both eventual and strong consistency.

While we refer the reader to [22] for a detailed description of SPANStore's design, implementation, and evaluation, here we summarize the key problem at the heart of its design: for any particular object, how to determine the most cost-effective replication policy that satisfies the application's latency, consistency, and fault-tolerance requirements? We first describe the inputs required by SPANStore and the format of the replication policies it identifies. We then separately present the algorithms used by SPANStore in two different data consistency scenarios.

4.1 Inputs and output

SPANStore requires three types of inputs: 1) a characterization of the cloud services on which it is deployed, 2) the application's latency, fault tolerance, and consistency requirements, and 3) a specification of the application's workload.

Characterization of SPANStore deployment. We require two pieces of information about SPANStore's deployment. First, we measure the distribution of latencies between every pair of data centers on which SPANStore is deployed. These measurements includes PUTs, GETs, and pings issued from a VM in one data center to the storage service or a VM in another data center. Second, we need 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. We account for an application's latency goals in terms of separate 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, e.g., 90^{th} percentile latency should be less than 100ms.

To capture consistency needs, we ask the application developer to choose between strong and eventual consistency. In the strong consistency case, we provide linearizability, i.e., all PUTs for a particular object are totally 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 [19] 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 ensures 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, we account for the application's workload in two ways.

[2]We assume an application employing SPANStore for data storage uses only the data centers of a single cloud service to host its computing instances (due to the differences across cloud providers), even though (via SPANStore) it will use multiple cloud providers for data storage.

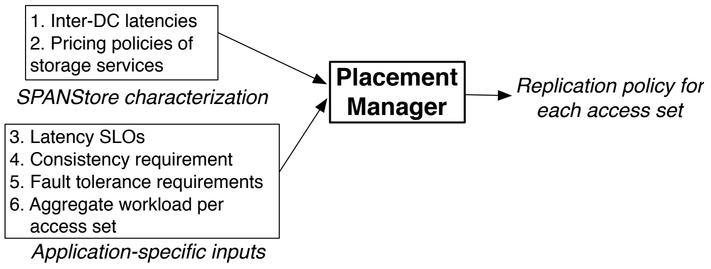

1. Inter-DC latencies
2. Pricing policies of storage services
SPANStore characterization
3. Latency SLOs
4. Consistency requirement
5. Fault tolerance requirements
6. Aggregate workload per access set
Application-specific inputs
Placement Manager
Replication policy for each access set

Figure 7: Overview of inputs and output in identifying cost-effective replication policies.

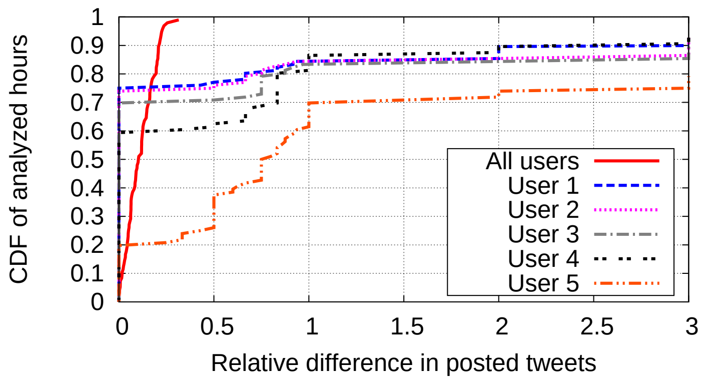
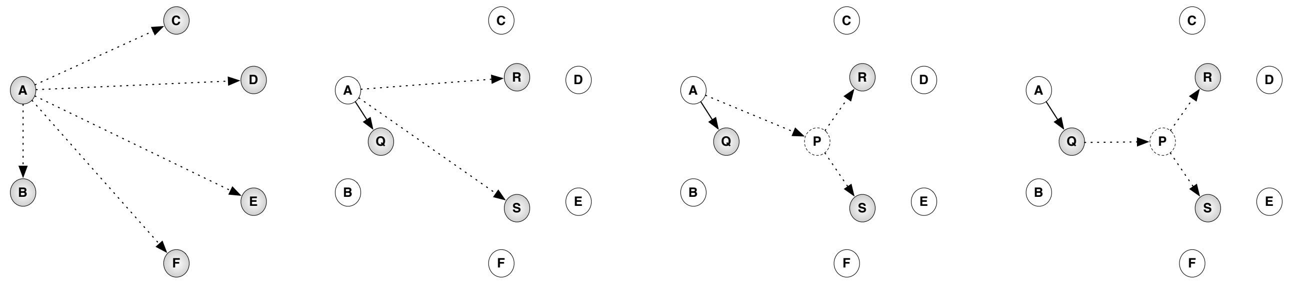


Figure 8: Comparison at different granularities of the stationarity in the number of posted tweets.

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 webservice 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. While SPANStore considers every object as having a fixed access set over its lifetime, subsequent work [10] addresses this limitation.

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 [8, 13]. 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 a central Placement Manager 1) the number of objects associated with that access set and 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 [18]. 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



(a) Full replication minimizes read latency (b) Partial replication reduces costs (c) Relayed update propagation reduces bandwidth costs (d) Two-hop relaying may reduce costs further

Figure 9: 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.

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 8 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. When we 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, the central Placement Manager 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, the Placement Manager 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 the Placement Manager 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 the Placement Manager determine the replication policy for each access set, and 2) how does SPANStore enforce Placement Manager-mandated replication policies during its operation, accounting for failures and changes in replication policies across epochs? We refer the reader to [22] for the answer to the latter question, and we describe here separately how SPANStore addresses the first question in the eventual consistency and strong consistency cases.

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 shown in Figure 9(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 cloud 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 9(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 (see Section 2) and relay updates to the replicas via another data center that has cheaper pricing for upload bandwidth. For example, in Figure 9(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 9(d).

We address 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. For every data center $i \in AS$, the integer program 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. SPANStore then stores copies of all objects with access set AS at all data centers in the union of PUT/GET replica sets.

The integer program we use 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 and asynchronously propagates the PUT to all other replicas of the object. Second, to minimize networking costs, the integer program 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, whereas in the case of asynchronous propagation of updates, relaying can optionally be over two hops, as in the example in Figure 9(d). Finally, for every data center i in the access set, the integer program identifies the paths from data centers j to k along which PUTs from i are transmitted during either synchronous or asynchronous propagation.

SPANStore's Placement Manager 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. 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. 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.

4.3 Strong consistency

When the application using SPANStore for geo-replicated storage requires strong consistency of data, we rely on quorum consistency [17]. Quorum consistency imposes two requirements to ensure linearizability. For every data center i in an access set, 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

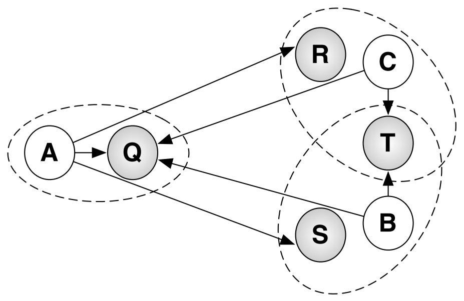


Figure 10: Example use of asymmetric quorum sets. Solid unshaded circles represent data centers in the access set, and shaded circles are data centers that host replicas. Directed edges represent transfers to PUT replica sets, and dashed ovals represent GET replica sets.

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 [20] 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 3(a), EC2 data centers have between 2 and 16 other data centers within 200ms of them. Figure 10 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.

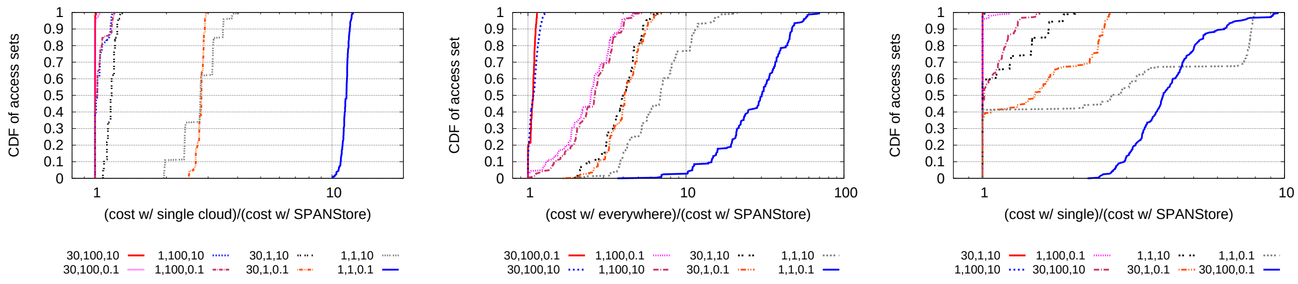
The integer program that the Placement Manager uses for choosing replication policies in the strong consistency setting mirrors the program for the eventual consistency case in several ways: 1) PUTs can be relayed via other data centers to reduce networking costs, 2) storage costs are incurred for maintaining 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, 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.

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. Second, we constrain 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. 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.

4.4 Cost savings

As an example of the cost savings that SPANStore can enable, Figure 11 compares cost with SPANStore against three other replication strategies. We consider a) GET:PUT ratios of 1 and 30, b) average object sizes of 1 KB and 100 KB, and c) aggregate data size of 0.1 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 [11]. In all workload settings, we fix the number of GETs at 100M and compute cost over a 30 day period. The results shown here are for the strong consistency case with SLOs for the 90^{th} percentile GET and PUT latencies set at 250ms and 830ms; 830 ms is the minimum PUT latency SLO if every object was replicated at all data centers in its access set.

Comparison with single-cloud deployment. First, Figure 11(a) compares the cost with SPANStore with the minimum cost required if we used only Amazon S3's data centers for storage. When the objects are small (i.e., average object size of 1KB), SPANStore's cost savings predominantly stem from differences in pricing



(a) (b) (c)

Figure 11: Cost savings enabled by SPANStore compared to (a) when data is replicated across the data centers of a single cloud service, and when using the (b) *Everywhere* and (c) *Single* replication policies. Legend indicates GET:PUT ratio, average object size (in KB), and overall data size (in TB). Note log-scale on x-axis.

per GET and PUT request across cloud providers. 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 data 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 exploiting the greater density of data centers and storing fewer copies of every object.

Comparison with fixed replication policies. Figure 11 also compares 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.

In Figure 11(b), 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 11(c), 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.

5 Conclusion

In summary, high latency variance and the need for judicious data replication are two challenges for any geo-distributed web service deployed in the cloud. To address these challenges, we presented two systems. First, CosTLO combines the use of multiple forms of redundancy by inferring models of replication and load balancing within cloud storage services. Second, SPANStore replicates data across the data centers of multiple cloud providers in order to exploit pricing discrepancies and to benefit from the greater geographical density of data centers. Together, these systems enable cost-effective support for low latency data access in the cloud.

Acknowledgments

Many others contributed to the design, implementation, and evaluation of the systems presented in this paper: Michael Butkiewicz, Dorian Perkins, and Ethan Katz-Bassett for SPANStore, and Curtis Yu for CosTLO. This work was supported in part by a NetApp Faculty Fellowship and by the National Science Foundation under grants CNS-1150219 and CNS-1463126.

References

[1] Amazon simple storage service - server access log format. `http://docs.aws.amazon.com/AmazonS3/latest/dev/LogFormat.html`.

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

[3] AWS global infrastructure. `https://aws.amazon.com/about-aws/global-infrastructure/`.

[4] Azure regions. `https://azure.microsoft.com/en-us/regions/`.

[5] Azure storage pricing. `http://azure.microsoft.com/en-us/pricing/details/storage/`.

[6] 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`.

[7] 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`.

[8] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[9] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Rao. Improving web availability for clients with MONET. In *NSDI*, 2005.

[10] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI*, 2014.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[12] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *IMC*, 2006.

[13] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *SoCC*, 2010.

[14] 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 *SOSP*, 2011.

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

[16] 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 *SOSP*, 2007.

[17] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

[18] 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 *KDD*, 2012.

[19] 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 *SOSP*, 2011.

[20] J.-P. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *DSN*, 2002.

[21] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *CoNEXT*, 2013.

[22] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[23] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *NSDI*, 2015.