A Self-Configurable Geo-Replicated Cloud Storage System

Masoud Saeida Ardekani[1,2] and Douglas B. Terry[3]

[1]Inria
[2]Sorbonne Universités, UPMC Univ Paris 06
[3]Microsoft Research Silicon Valley

Abstract

Reconfiguring a cloud storage system can improve its overall service. Tuba is a geo-replicated key-value store that automatically reconfigures its set of replicas while respecting application-defined constraints so that it adapts to changes in clients' locations or request rates. New replicas may be added, existing replicas moved, replicas upgraded from secondary to primary, and the update propagation between replicas adjusted. Tuba extends a commercial cloud-based service, Microsoft Azure Storage, with broad consistency choices (as in Bayou), consistency-based SLAs (as in Pileus), and a novel replication configuration service. Compared with a system that is statically configured, our evaluation shows that Tuba increases the reads that return strongly consistent data by 63%.

1 Introduction

Cloud storage systems can meet the demanding needs of their applications by dynamically selecting when and where data is replicated. An emerging model is to utilize a mix of strongly consistent primary replicas and eventually consistent secondary replicas. Applications either explicitly choose which replicas to access or let the storage system select replicas at run-time based on an application's consistency and performance requirements [15]. In either case, the configuration of the system significantly impacts the delivered level of service.

Configuration issues that must be addressed by cloud storage systems include: (i) where to put primary and secondary replicas, (ii) how many secondary replicas to deploy, and (iii) how frequently secondary replicas should synchronize with the primary replica. These choices are complicated by the fact that Internet users are located in different geographical locations with different time zones and access patterns. Moreover, systems must consider the growing legal, security, and cost constraints about replicating data in certain countries or avoiding replication in others.

For a stable user community, static configuration choices made by a system administrator may be acceptable. But many modern applications, like shopping, social networking, news, and gaming, not only have evolving world-wide users but also observe time-varying access patterns, either on a daily or seasonal basis. Thus, it is advantageous for the storage system to *automatically* adapt its configuration subject to application-specific and geo-political constraints.

Tuba is a geo-replicated key-value store based on Pileus [15]. It addresses the above challenges by configuring its replicas automatically and periodically. While clients try to maximize the utility of individual read operations, Tuba improves the overall utility of the storage system by automatically adapting to changes in access patterns and constraints. To this end, Tuba includes a configuration service that periodically receives from clients their consistency-based service level agreements (SLAs) along with their hit and miss ratios. This service then changes the locations of primary and secondary replicas to improve the overall delivered utility. A key property of Tuba is that both read and write operations can be executed in parallel with reconfiguration operations.

We have implemented Tuba as middleware on top of Microsoft Azure Storage (MAS) [3]. It extends MAS with broad consistency choices as in Bayou [14], and provides consistency-based SLAs like Pileus. Moreover, it leverages geo-replication for increased locality and availability. Our API is a minor extension to the MAS Blob Store API, thereby allowing existing Azure applications to use Tuba with little effort while experiencing the benefits of dynamic reconfiguration.

An experiment with clients distributed in datacenters (sites) around the world shows that reconfiguration every two hours increases the fraction of reads guaranteeing strong consistency from 33% to 54%. This confirms that automatic reconfiguration can yield substantial benefits which are realizable in practice.

The outline of the paper is as follows. We review Pileus and Tuba in Section 2. We look under the hood of Tuba's configuration service in Section 3. Section 4 describes execution modes of clients in Tuba. In Section 5, we explain implementation details of the system. Our evaluation results are presented in Section 6. We review related work in Section 7 and conclude the paper in Section 8.

2 System Overview

In this section, we first briefly explain features that Tuba inherits from Pileus. Since we do not cover all technical issues of Pileus, we encourage readers to read the original paper [15] for more detail. Then, we overview Tuba and its fundamental components, and how it extends the features of the Pileus system.

2.1 Tuba Features from Pileus

Storage systems cannot always provide rapid access to strongly consistent data because of the high network latency between geographical sites and diverse operational conditions. Clients are forced to select less ideal consistency/latency combinations in many cases. Pileus addresses this problem by allowing clients to declare their consistency and latency priorities via SLAs. Each SLA comprises several subSLAs, and each subSLA contains a desired consistency, latency and utility.

The utility of a subSLA indicates the value of the associated consistency/latency combination to the application and its users. Inside a SLA, higher-ranked subSLAs have higher utility than lower-ranked subSLAs. For example, consider the SLA shown in Figure 1. Read operations with strong consistency are assigned utility 1 as long as they complete in less than 50 ms. Otherwise, the application tolerates eventually consistent data and longer response times though the rewarded utility is very small (0.01). Pileus, when performing a read operation with a given SLA, attempts to maximize the delivered utility by meeting the highest-ranked subSLA possible.

The replication scheme in Pileus resembles that of other cloud storage systems. Like BigTable [4], each key-value store is horizontally partitioned by key-ranges into *tablets*, which serve as the granularity of replication. Tablets are replicated at an arbitrary collection of storage sites. Storage sites are either primary or secondary. All write operations are performed at the primary sites. Secondary sites periodically synchronize with the primary sites in order to receive updates.

Rank	Consistency	Latency(ms)	Utility
1	Strong	50	1
2	Eventual	1000	0.01

Figure 1: SLA Example

Depending on the desired consistency and latency as specified in an SLA, the network delays between clients and various replication sites, and the synchronization period between primary and secondary sites, the Pileus client library decides on the site to which a read operation is issued. Pileus provides six consistency choices that can be included in SLAs: (i) strong (ii) eventual (iii) read-my-writes (RMW) (iv) monotonic reads (v) bounded(t), and (vi) causal.

Consider again the SLA shown in Figure 1. A Pileus client reads the most recent data and *hits* the first subSLA as long as the round trip latency between that client and a primary site is less than 50ms. But, the first subSLA *misses* for clients with a round trip latency of more than 50ms to primary sites. For these clients, Pileus reads data from any replica site and hits the second subSLA.

Pileus helps developers find a suitable consistency/latency combination given a fixed configuration of tablets. Specifically, the locations of primary and secondary replication sites, the number of required secondary sites, and the synchronization period between secondary and primary sites need to be specified by system administrators manually. However, a worldwide distribution of users makes it extremely hard to find an optimal configuration where the overall utility of the system is maximized with a minimum cost. Tuba extends Pileus to specifically address this issue.

2.2 Tuba's New Features

The main goal of Tuba is to periodically improve the overall utility of the system while respecting replication and cost constraints. To this end, it extends Pileus with a configuration service (CS) delivering the following capabilities:

1. performing a reconfiguration periodically for different tablets, and

2. informing clients of the current configuration for different tablets.

We note that the above capabilities do not necessarily need to be collocated at the same service. Yet, we assume they are provided by the same service for the sake of simplicity.

In order for the CS to configure a tablet's replicas such that the overall utility increases, it must be aware of the way the tablet is being accessed globally. Therefore, all clients in the system periodically send their *observed latency* and the *hit and miss ratios* of their SLAs to the CS.

The observed latency is a set comprising the latency between a client (e.g., an application server) and different datacenters. The original Pileus system also requires clients to maintain this set. Since the observed latency between datacenters does not change very often, this set is only sent every couple of hours, or when it changes by more than a certain threshold.

Tuba clients also send their SLAs' hit and miss ratios periodically. It has been previously observed that placement algorithms with client workload information (such as the request rate) perform two to five times better than workload oblivious random algorithms [10]. Thus, every client records aggregate ratios of all hit and missed subSLAs for a sliding window of time, and sends them to the CS periodically. The CS then periodically (or upon receiving an explicit request) computes a new configuration such that the overall utility of the system is improved, all constraints are respected, and the cost of the migrating to and maintaining the new configuration remains below some threshold.

Once a new configuration is decided, one or more of the following operations are performed as the system changes to the new configuration: (i) changing the primary replica, (ii) adding or removing secondary replicas, and (iii) changing the synchronization periods between primary and secondary replicas. In the next section, we explain in more detail how the above operations are performed with minimal disruption to active clients.

3 Configuration Service (CS)

The CS is responsible for periodically improving the overall utility of the system by computing and applying new configurations. The CS selects a new configuration by first generating all reasonable replication scenarios that satisfy a list of defined constraints.

For each configuration possibility, it then computes the expected gained utility and the cost of reconfiguration. The new chosen configuration is the one that offers the highest utility-to-cost ratio. Once a new configuration is chosen, the CS executes the reconfiguration operations required for making a transition from the old configuration to the new one.

In the remaining of this section, we first explain the different types of constraints and the cost model used by the CS. Then, we introduce the algorithm behind the CS to compute a new configuration. Finally, we describe how the CS executes different reconfiguration operations to install the new configuration.

3.1 Constraints

Given the simple goal of maximizing utility, the CS would have a *greedy* nature: it would generally decide to add replicas. Hence, without constraints, the CS could ultimately replicate data in all available datacenters. To address this issue, a system administrator is able to define constraints for the system that the CS respects.

Through an abstract constraint class, Tuba allows constraints to be defined on any attribute of the system. For example, a constraint might disallow creating more than three secondary replicas or disallow a reconfiguration to happen if the total number of online users is greater than 1 million. Tuba abides by all defined constraints during every reconfiguration.

Several important constraints are currently implemented and ready for use including: (i) Geo-replication factor, (ii) Location, (iii) Synchronization period, and (iv) Cost.

With geo-replication constraints, the minimum and maximum number of replicas can be defined. For example, consider an online music store. Developers may set the maximum geo-replication factor of tablets containing less popular songs to one, and set the minimum geo-replication factor of a tablet containing top-ten best selling songs to three. Even if the storage cost is relatively small, limiting the replication factor may still be desirable due to the cost of communication between sites for replica synchronization.

Location constraints are able to explicitly force replication in certain sites or disallow them in others. For example, an online social network application can respond to security concerns of European citizens by allowing replication of their data only in Europe datacenters.

With the synchronization period constraint, application developers can impose bounds on how often a secondary replica synchronizes with a primary replica.

The last and perhaps most important constraint in Tuba is the cost constraint. As mentioned before, the CS picks a configuration with the greatest ratio

of gained utility over cost. With a cost constraint, application developers can indicate how much they are willing to pay (in terms of dollars) to switch to a new configuration. For instance, one possible configuration is to put secondary replicas in all available datacenters. While the gained utility for this configuration likely dominants all other possible configurations, the cost of this configuration may be unacceptably large. In the next section, we explain in more detail how these costs are computed in Tuba.

Should the system administrator neglect to impose any constraint, Tuba has two default constraints in order to avoid aggressive replication and to avoid frequent synchronization between replicas: (1) a lower bound for the synchronization period, and (2) an upper bound on the recurring cost of a configuration.

3.2 Cost Model

The CS considers the following costs for computing a new configuration:

- Storage: the cost of storing a tablet in a particular site.
- Read/Write Operation: the cost of performing read/write operations.
- Synchronization: the cost of synchronizing a secondary replica with a primary one.

The first two costs are computed precisely for a certain period of time, and the third cost is estimated based on read/write ratios.

Given the above categories, the cost of a primary replica is the sum of its storage and read/write operation costs, and the cost of a secondary replica is the sum of storage, synchronization, and read operation costs. Since Tuba uses batching for synchronization to a secondary replica and only sends the last write operation on an object in every synchronization cycle, the cost of a primary replica is usually greater than that of secondary replicas.

In addition to the above costs, the CS also considers the cost of creating a new replica; this cost is computed as one-time synchronization cost.

3.3 Selection

Potential new configurations are computed by the CS in the following three steps:

Ratios aggregation. Clients from the same geographical region usually have similar observed access latencies. Therefore, as long as they use the same SLAs, their hit and miss ratios can be aggregated to reduce the computation. We note that this phase does not necessarily need to be in the critical path, and aggregations can be done once clients send their ratios to the CS.

Rank	Consistency	Latency(ms)	Utility
1	Strong	100	1
2	RMW	100	0.9
3	Eventual	1000	0.01

Figure 2: SLA of a Social Network Application

Configuration computation. In this phase, possible configurations that can improve the overall utility of the system are generated and sorted. For each missed subSLA, and depending on its consistency, the CS may produce several potential configurations along with their corresponding reconfiguration operations. For instance, for a missed subSLA with strong consistency, two scenarios would be: (i) creating a new replica near the client and making it the solo primary replica, or (ii) adding a new primary replica near the client and making the system run in multi-primary mode.

Each new configuration has an associated cost of applying and maintaining it for a certain period of time. The CS also computes the overall gained utility of every new configuration that it considers. Finally, the CS sorts all potential configurations based on their gained utility over their cost.

Constraints satisfaction. Configurations that cannot satisfy all specified constraints are eliminated from consideration. Constraint classes also have the ability to add configurations being considered. For instance, the minimum geo-replication constraint might remove low-replica configurations and create several new ones with additional secondary replicas at different locations.

3.4 Operations

Once a new configuration is selected, the CS executes a set of reconfiguration operations to transform the system from the current configuration. In this section, we explain various reconfiguration operations and how they are executed abstractly by the CS, leaving the implementation specifics to Section 5.

3.4.1 Adjust the Synchronization Period

When a secondary replica is added to the system for a particular tablet, a default synchronization period is set, which defines how often a secondary replica synchronizes with (i.e., receives updates from) the primary replica. Although this value does not affect

the latency of read operations with strong or eventual consistency, the average latency of reads with intermediary consistencies (i.e., RMW, monotonic reads, bounded, and causal) can depend heavily on the frequency of synchronization. Typically, the cost of adjusting the synchronization period is smaller than the cost of adding a secondary replica or of changing the locations of primary/secondary replicas. Hence, it is likely that the CS will decide to decrease this period to increase the hit ratios of subSLAs with intermediary consistencies.

For example, consider a social network application with the majority of users located in Brazil and India accessing a storage system with a primary replica located in Brazil, initially, and a secondary replica placed in South Asia with the synchronization period set to 10 seconds. Assume that the SLA shown in Figure 2 is set for all read operations. Given the fact that the round trip latency between India and Brazil is more than 350 ms, the first subSLA will never hit for Indian users. Yet, depending on the synchronization period and frequency of write operations performed by Indian users, the second subSLA might hit. Thus, if the CS detects low utility for Indian users, a possible new configuration would be similar to the old one but with a reduced synchronization period.

In this case, the chosen operation to apply the new configuration is *adjust_sync_period*. Executing this operation is very simple since the value of the synchronization period need only be changed in the secondary replica. Clients do not directly observe any difference between the new configuration and the old one, but they benefit from a more up-to-date secondary replica.

3.4.2 Add/Remove Secondary Replica

In certain cases, the CS might decide to add a secondary replica to the system. For example, consider an online multiplayer game with the SLA shown in Figure 3 and where the primary replica is located in the East US region. In order to deliver a better user experience to gamers around the globe, the CS may add a secondary replica near users during their peak times. Once the peak time has passed, in order to reduce costs, the CS may decide to remove the added, but now lightly used, secondary replica.

Executing *add_secondary*($site_i$) is straightforward. A dedicated thread is dispatched to copy objects from the primary replica to the secondary one. Once the whole tablet is copied to the secondary replica, the new configuration becomes available to clients. Clients with the old configuration may continue submitting read operations to previously known replicas, and they eventually will become aware of the newly added secondary replica at $site_i$.

Rank	Consistency	Latency(ms)	Utility
1	RMW	50	1
2	Monotonic Read	50	0.5
3	Eventual	500	0

Figure 3: SLA of an online multiplayer game

Executing *remove_secondary*($site_i$) is also simple. The CS removes the secondary replica from the current configuration. In addition, a thread is dispatched to physically delete the secondary replica.

3.4.3 Change Primary Replica

In cases where the system maintains a single primary site, the CS may decide to change the location of the primary replica. For instance, consider the example given in Section 3.4.1. The CS may detect that adjusting the synchronization period between the primary and secondary replicas cannot improve the utility. In this case, the CS may decide to swap the primary and secondary replica roles. During peak times in India, the secondary replica in South Asia becomes the primary replica. Likewise, during peak times in Brazil, the replica in Brazil becomes primary.

The CS calls the *change_primary*($site_i$) operation to make the configuration change. If a secondary replica does not exist in $site_i$, the operation is performed in three steps. First, the CS creates a new empty replica at $site_i$. It also invalidates the configuration cached in clients. As we shall see later, when a cached configuration is invalid, a client needs to contact the CS when executing certain operations. Second, once every cached configuration becomes invalid, the CS makes $site_i$ a WRITE_ONLY primary site. In this mode, all write operations are forwarded to both the primary site and $site_i$, but $site_i$ is not allowed to execute read operations. Finally, once $site_i$ catches up with the primary replica, the CS makes it the solo primary site. If a replica exists in $site_i$, the first step is skipped. We will explain the implementation of this operation in Section 5.3.

3.4.4 Add Primary Replica

For applications that require clients to read up-to-date data as fast as possible, the system may benefit from having multiple primary sites that are strongly consistent. In multi-primary mode, write operations are applied synchronously in several sites before the client is informed that the operation has completed.

Operation	Effect	Cost
Decrease synchronization period of secondary replica at $site_i$	Increase hit ratios of subSLAs with bounded, causal, or RMW consistencies for clients near $site_i$	Increase in communication
Add $site_i$ as a secondary replica	Increase hit ratios of subSLAs with eventual or intermediary consistencies for clients near $site_i$	Additional storage; increased communication
Upgrade $site_i$ from secondary to primary, and downgrade $site_j$ from primary to secondary	Increase hit ratios of subSLAs with strong or intermediary consistency for clients near $site_i$; decrease hit ratios of subSLAs with strong or intermediary consistency for clients near $site_j$	No change
Add $site_i$ as a primary replica (upgraded from secondary)	Increase hit ratios of subSLAs with strong or intermediary consistency for clients near $site_i$	Increased communication; increased write latency

Figure 4: Summary of Common Reconfiguration Operations, Effects on Hit Ratios, and Costs.

The operation that performs the configuration transformation is called *add_primary*($site_i$). Its execution is very similar to *change_primary*($site_i$) with one exception. In the third step, instead of making the WRITE_ONLY $site_i$ the solo primary, $site_i$ is added to the list of primary replicas, thereby making the system multi-primary. In this mode, multiple rounds of operations are needed to execute a write. The protocol that we use is described in Section 5.2.3.

3.4.5 Summary

Figure 4 summarizes the reconfiguration operations that are generally considered by the CS (inverse and other less common operations are not shown). Note that the listed effects are only potential benefits. Adjusting the synchronization period or adding a secondary replica to $site_i$ does not impact the observed consistency or write latency of clients that are not near this site. These operation can possibly increase the hit ratios of subSLAs with intermediary consistencies observed by clients close to $site_i$. Adding a secondary replica can increase the hit ratios of subSLAs with eventual consistency. Making $site_i$ the solo primary increases the hit ratios of subSLAs with both strong and intermediary consistencies for clients close to $site_i$. However, clients close to the previous primary replica now may miss subSLAs with strong or intermediary consistencies. Adding a primary replica can boost strong consistency without having a negative impact on read operations; but, it increases the cost of write operations for all clients.

4 Client Execution Modes

Since the CS may reconfigure the system periodically, clients need to be aware of possible changes in the locations of primary and secondary replicas. Instead of clients asking the CS for the latest configuration before executing each operation, Tuba allows clients to cache the configuration of a tablet (called the *cview*) and use it for performing read and write operations. In this section, we explain how clients avoid two potential safety violations: (i) performing a read operation with strong consistency on a non-primary replica, or (ii) executing a write operation on a non-primary replica.

Based on the freshness of a client's cview, the client is either in fast or slow mode. Roughly speaking, a client is in the fast mode for a given tablet if it knows that it has the latest configuration. That is, it knows exactly the locations of primary and secondary replicas, and it is guaranteed that the configuration will not change in the near future. On the other hand, whenever a client suspects that a configuration may have changed, it enters slow mode until it refreshes its local cache.

Initially, every client is in slow mode. In order to enter fast mode, a client requests the latest configuration of a tablet (Figure 5). If the CS has not scheduled a change to the location of a primary replica, the client obtains the current configuration along with a promise that the CS will not modify the set of primary replicas within the next Δ seconds. Suppose the duration from when the client issues its request to when it receives the latest configuration is measured to be δ seconds. The client then enters the fast mode for $\Delta - \delta$ seconds. During this period, the client is sure that the CS will not perform a reconfiguration that compromises safety.

In order to remain in fast mode, a client needs to periodically refresh its cview. As long as it receives the latest configuration within the fast mode window,

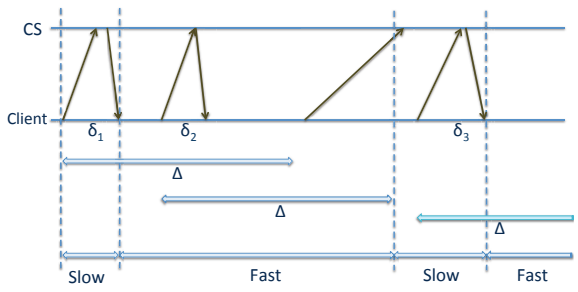


Figure 5: Clients Fast and Slow Execution Modes

it will remain in fast mode, and its fast mode window is extended.

The CS can force all clients to enter slow mode at any time by preventing them from refreshing their configuration views. This feature is used before executing *change_primary*() and *add_primary*() operations (see Section 5.3).

Fast Mode. When a client is in fast mode, read and single-primary write operations involve a single round-trip to one selected replica. No additional overhead is imposed on these operations. Multi-primary write operations use a three-phase protocol in fast or slow mode (see Section 5.2.3).

Slow Mode. Being in slow mode (for a given tablet) means that the client is not totally sure about the latest configuration, and the client needs to take some precautions. Slow mode has no affect on read operations with relaxed consistency, i.e., with any desired consistency except strong consistency. Because read operations with strong consistency must always go to a primary replica, when a client is in slow mode it needs to confirm that such an operation is indeed executed at a current primary replica. Upon completion of a strong consistency read, the client validates that the responding replica selected from its cview is still a primary replica. If not, the client retries the read operation.

Unlike read operations, write operations are more involved when a client is in slow mode. More precisely, any client in slow mode that wishes to execute a write operation on a tablet needs to take a non-exclusive lock on the tablet's configuration before issuing the write operation. On the other hand, the CS needs to take an exclusive lock on the configuration if it decides to change the set of primary replicas. This lock procedure is required to ensure the linearizability [7] of write operations.

5 Implementation

Tuba is built on top of Microsoft Azure Storage (MAS) [3] and provides a similar API for reading and writing blobs. Every MAS storage account is associated with a particular storage site. Although MAS supports Read-Access Geo-Redundant Storage (RA-GRS) in which both strong and eventual consistencies are provided, it lacks intermediary consistencies, and replication is limited to a single primary site and a single secondary site. Our implementation extends MAS with: (i) multi-site geo-replication (ii) consistency-based SLAs, and (iii) automatic reconfiguration.

A user of Tuba supplies a set of storage accounts. This set determines all available sites for replication. The CS then selects primary and secondary replica sites by choosing storage accounts from this set. Thus, a configuration is a set of MAS storage accounts tagged with PRIMARY or SECONDARY.

In the rest of this section, we explain the communication between clients and the CS, and how operations are implemented in Tuba. We ignore the implementation of consistency guarantees and consistency-based SLAs since these aspects of Tuba are taken directly from the Pileus system [15].

5.1 Communication

Clients communicate with the CS through a designated Microsoft Azure Storage container. Clients periodically write their latency and hit/miss ratios to storage blobs in this shared container. The CS reads this information and stores the latest configuration as a blob in this same container. Likewise, clients periodically read the latest configuration blob from the shared container and cache it locally.

As we explained in Section 4, when a client reads the latest configuration, it enters fast mode for $\Delta - \delta$ seconds. Since there is no direct communication between the client and the CS, we also need to ensure that the CS does not modify a primary replica and install a new configuration within the next Δ seconds. Our solution is simple. When the CS wants to perform certain reconfiguration operations (i.e., changing or adding a primary replica), it writes a special reconfiguration-in-progress (*RiP*) flag to the configuration blob's metadata. The CS then waits for at least Δ seconds before installing the new configuration. If a client fails to refresh its cview on time or if it finds that the *RiP* flag is set, then the client enters slow mode. Once the CS completes the operations needed to reconfigure the system, it overwrites the configuration blob with the latest configuration

and clears the *RiP* flag. Clients will re-enter fast mode when they next retrieve the new configuration.

5.2 Client Operations

5.2.1 Read Operation

For each read operation submitted by an application, the client library selects a replica based on the client's latency, cview, and a provided SLA (as in Pileus). The client then sends a read request to the chosen replica. Upon receiving a reply, if the client is in fast mode or if the read operation does not expect strong consistency, the data is returned immediately to the application. Otherwise, the client confirms that the contacted replica had been the primary replica at the time it answered the read request. More precisely, when a client receives a read reply message in slow mode, it reads the latest configuration and confirms that the timestamp of the configuration blob has not changed.

5.2.2 Single-primary Write Operation

To execute a single-primary write operation, a client first checks that it is in fast mode and that the remaining duration of the fast mode interval is longer than the expected time to complete the write operation. If not, it refreshes its cview. Assuming the *RiP* flag is not set, the client then writes to the primary replica. Once the client receives a positive response to this write operation, the client checks that it is still in fast mode. If so, the write operation is finished. If the write operation takes more time than expected such that the client enters slow mode during the execution of the write operation, the client confirms that the primary replica has not changed.

When a client discovers a reconfiguration in progress and remains in slow mode, we considered two approaches for performing writes. The simplest approach is for the client to wait until a new configuration becomes available. In other words, it could wait until the *RiP* flag is removed from the configuration blob's metadata. The main drawback is that no write operation is allowed on the tablet being reconfigured for Δ seconds and, during this period, the CS does nothing while waiting for all clients to enter slow mode.

Instead, Tuba allows a client in slow mode to execute a write operation by taking a lock. A client acquires a non-exclusive lock on the configuration to ensure that the CS does not change the primary replica before it executes the write operation. The CS, on the other hand, grabs an exclusive lock on the configuration before changing it. This locking mechanism is implemented as follows using MAS's existing lease support. To take a non-exclusive lock on the configuration, a client obtains a lease on the configuration blob and stores the lease-id as metadata in the blob. Other clients wishing to take a non-exclusive lock simply read the lease-id from the blob's metadata and renew the lease. To take an exclusive lock, the CS breaks the client's lease and removes the lease-id from the metadata. The CS then acquires a new lease on the configuration blob. Note that no new write is allowed after this point. After some safe threshold equal to the maximum allowed leased time, the CS updates the configuration.

5.2.3 Multi-primary Write Operation

Tuba permits configurations in which multiple servers are designated as primary replicas. A key implementation challenge was designing a protocol that atomically updates any number of replicas on conventional storage servers and that operates correctly in the face of concurrent readers and writers. Our multi-primary write protocol involves three phases: one in which a client marks his intention to write on all primary replicas, one where the client updates all of the primaries, and one where the client indicates that the write is complete. To guard against concurrent writers, we leverage the concept of ETags in Microsoft Azure, which is also part of the HTML 1.1 specification. Each blob has a string property called an ETag that is updated whenever the blob is modified. Azure allows clients to perform a conditional write operation on a blob; the write operation executes only if the provided ETag has not changed.

When an application issues a write operation to a storage blob and there are multiple primary replicas, the Tuba client library performs the following steps.

Step 1: Acquire a non-exclusive lock on the configuration blob. This step is the same as previously described for a single-primary write in slow mode. In this case, the configuration is locked even if the client is in fast mode since the multi-primary write may take longer than Δ seconds to complete. This ensures that the client knows the correct set of primary replicas throughout the protocol.

Step 2: At the main primary site, add a special write-in-progress (*WiP*) flag to the metadata of the blob being updated. The main primary site is the one listed first in the set of primary replicas. This metadata write indicates to readers that the blob is being updated, and it returns an ETag that is used later when the data is actually written. Updates to different blobs can take place in parallel.

Step 3: Write the *WiP* flag to the blob's metadata

on all other primary replicas. Note that these writes can be done in any order or in parallel.

Step 4: Perform the write on the main primary site using the ETag acquired in Step 2. Note that since writes are performed first at the main primary, this replica always holds the *truth*, i.e. the latest data. Other primary replicas hold stale data at this point. This conditional write may fail because the ETag is not current, indicating that another client is writing to the same blob. In the case of concurrent writers, the last writer to set the *WiP* flag will successfully write to the main primary replica; clients whose writes fail abandon the write protocol and possibly retry those writes later.

Step 5: Perform conditional writes on all the other primary replicas using the previously aquired Etags. These writes can be done in parallel. Again, a failed write indicates that a concurrent write is in progress. In this case, this client stops the protocol even though it may have written to some replicas already; such writes will be (or may already have been) overwritten by the latest writer (or by a recovery process as discussed in section 5.4).

Step 6: Clear the *WiP* flags in the metadata at all non-main primary sites. These flags can be cleared in any order or in parallel. This allows clients to now read from these primary replicas and obtain the newly written data. To ensure that one client does not prematurely clear the flag while another client is still writing, these metadata updates are performed as conditional writes using the ETags obtained from the writes in the previous step.

Step 7: Clear the *WiP* flag in the metadata on the main primary using a conditional write with the ETag obtained in Step 4. Because this is done as the final step, clients can check if a write is in progress simply by reading the metadata at the main primary replica.

An indication that the write has been successfully completed can be returned to the caller at any time after Step 4 where the data is written to the main primary. Waiting until the end of the protocol ensures that the write is durable since it is held at multiple primaries.

If a client attempts a strongly consistent read while another client is performing a multi-primary write, the reader may obtain a blob from the selected primary replica whose metadata contains the *WiP* flag. In this case, the client redirects its read to the main primary replica who always holds the latest data. Relaxed consistency reads, to either primary or secondary replicas, are unaffected by writes in progress.

5.3 CS Reconfiguration Operations

In this section, we only explain the implementation of *change_primary*() and *add_primary*() since the implementation details of adjusting a synchronization period and adding/removing secondary replicas are straightforward.

As we explained before, *change_primary*($site_i$) is the operation required for making $site_i$ the solo primary. If a secondary replica does not exist in $site_i$, the operation is performed in three steps. Otherwise, the first step is skipped.

Step 1: The CS starts by creating a replica at $site_i$, and synchronizing it with the primary replica.

Step 2: Before making $site_i$ the new primary replica, the CS synchronizes $site_i$ with the existing primary replica. Because write operations can run concurrently with a *change_primary*($site_i$) operation, $site_i$ might never be able to catch up with the primary replica. To address this issue, the CS first makes $site_i$ a WRITE_ONLY replica by creating a new temporary configuration. As its name suggests, write operations are applied to both WRITE_ONLY replicas and primary replicas (using the multi-primary write protocol described previously).

The CS installs this configuration as follows:

(i) It writes the *RiP* flag to the configuration blob's metadata, and waits Δ seconds to force all clients into slow mode.

(ii) Once all clients have entered the slow mode, the CS breaks the lease on the configuration blob and removes the lease-id from the metadata.

(iii) It then acquires a new lease on the blob and waits for some safe threshold.

(iv) Once the threshold is passed, the CS safely installs the temporary configuration, and removes the *RiP* flag.

Consequently, clients again switch to fast mode execution while the $site_i$ replica catches up with the primary replica.

Step 3: The final step is to make $site_i$ the primary replica, once $site_i$ is completely up-to-date. The CS follows the procedure explained in the previous step to install a new configuration where the old primary replica is downgraded to a secondary replica, and the WRITE_ONLY replica is promoted to be the new primary. Once the new configuration is installed, $site_i$ is the sole primary replica.

Note that write operations are blocked from the time when the CS takes an exclusive lease on the configuration blob until it installs the new configuration in both steps 2 and 3. However, this duration is short: a round trip latency from the CS to the configuration blob plus the safe threshold.

The *add_primary*() operation is implemented exactly like *change_primary*() with one exception. In the third step, instead of making $site_i$ the solo primary, this site is added to the list of primary replicas.

5.4 Fault-Tolerance

Replica Failure. A replica being unavailable should be a very rare occurrence since each of our replication sites is a collection of three Azure servers in independent fault domains. In any case, failed replicas can easily be removed from the system through reconfiguration. Failed secondary replicas can be ignored by clients, while failed primary replicas can be replaced using previously discussed reconfiguration operations.

Client Failure. Most read and write operations from clients are performed at a single replica and maintain no locks or leases. The failure of one client during such operations does not adversely affect others. However, Tuba does need to deal explicitly with client failures that may leave a multi-primary write partially completed. In particular, a client may crash before successfully writing to all primary replicas or before removing the *WiP* flags on one or more primary replicas.

When a client, through normal read and write operations, finds that a write to a blob has been in progress for an inordinate amount of time, it invokes a recovery process to complete the write. The recovery process knows that the main primary replica holds the truth. It reads the blob from the main primary and writes its data to the other primary replicas using the multi-write protocol described earlier. Having multiple recovery processes running simultaneously is acceptable since they all will be attempting to write the same data. The recovery process, after successfully writing to every primary replica, clears all of the *WiP* flags for the recovered blob.

CS Failure. Importantly, the Tuba design does not depend on an active CS in continuous operation. The CS may run only occasionally to check whether a reconfiguration is warranted. Since clients read the latest configuration directly from the configuration blob, and do not rely on responses from the CS, they can stay in fast mode even when the CS is not available as long as the configuration blob is available (and the *RiP* flag is not set). Since the configuration blob is replicated in MAS, it obtains the high-availability guarantees provided by Azure. If higher availability is desired, the configuration blob could be replicated across sites using Tuba's own multi-primary write protocol.

The only troubling scenario is if the CS fails while in the midst of a reconfiguration leaving the *RiP* flag set on the configuration blob. This is not a concern when the CS fails while adjusting a synchronization period or adding/removing a secondary replica. Likewise, a failure before the second step of changing/adding a primary replica does not pose any problem. Even if a CS failure leaves the *RiP* flag set, clients can still perform reads and writes in slow mode.

Recovery is easy if the CS fails during step 2 or during step 3 of changing/adding a primary replica (i.e., after setting the *RiP* flag and before clearing it). When the CS wants to performs a reconfiguration, it obtains an ETag upon setting the *RiP* flag. To install a new configuration, the CS writes the new configuration conditional on the obtained ETag.

A client clears the *RiP* flag upon waiting too long in slow mode. Doing so will prevent the CS from writing a new configuration blob and abort any reconfiguration in progress in the unlikely event that the CS had not crashed but was simply operating slowly. In other words, the CS cannot write the new configuration if some client had impatiently cleared the *RiP* flag and consequently changed the configuration blob's ETag.

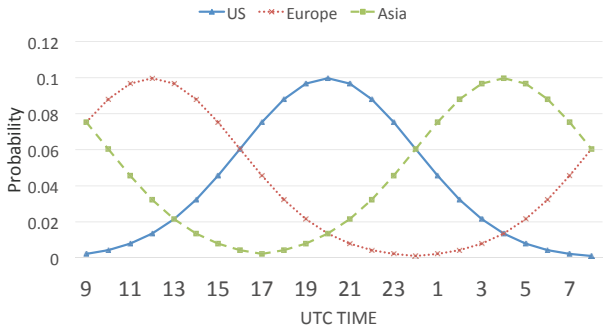
Finally, if the CS fails after step 2 of adding/changing a primary replica, clients can still enter fast mode. In case the CS was executing *change_primary*() before its crash, write operations will execute in multi-primary mode. Thus, they will be slow until the CS recovers and finishes step 3.

6 Evaluation

In this section, we present our evaluation results, and show how Tuba improves the overall utility of the system compared with a system that does not perform automatic reconfiguration.

6.1 Setup and Benchmark

To evaluate Tuba, we used three storage accounts located in the South US (SUS), West Europe (WEU), and South East Asia (SEA). We modeled the number of active clients with a normal distribution, and placed them in the US West Coast, West Europe, and Hong Kong (Figure 6). This is to mimic the workload of clients in different parts of the world during working hours. The mean of the normal distribution is set to 12 o'clock local time, and the variance is set to 8 hours. Considering the above



	SUS	WEU	SEA
US Clients (West US)	53	153	190
Europe Clients (West Europe)	132	<1	277
Asia Clients (Hong Kong)	204	296	36

Figure 6: Client Distribution and Latencies (in ms)

Rank	Consistency	Latency(ms)	Utility
1	Strong	100	1
2	RMW	100	0.7
3	Eventual	250	0.5

Figure 7: SLA for Evaluation

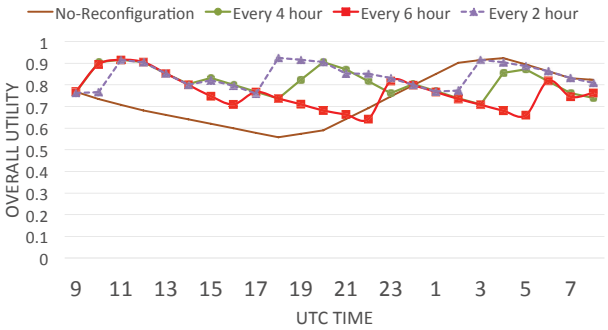
normal distribution, the number of online clients at each hour is computed as a total number of clients times the probability distribution at that hour. The total number of clients at each site is 150 over a 24 hour period. Hence, each tablet is accessed by 450 distinct clients in one day.

We used the YCSB benchmark [6] with workload B (95% Reads and 5% writes) to generate the load. Each tablet contains 10^5 objects, and each object has a 1KB payload. Figure 7 shows the SLA used in our evaluation, which resembles one used by a social networking application [15].

The initial setup places the primary replica in SEA and a secondary replica in WEU. We set the geo-replication factor to two, allowing the CS to replicate a tablet in at most two datacenters. Moreover, we disallowed multi-primary schemes during reconfigurations.

6.2 Macroscopic View

Figure 8 compares the overall utility for read operations when reconfiguration happens every 2, 4, and 6 hours over a 24 hour period, and when no reconfiguration happens. We note that without re-



	Reconf. Every		
	6h	4h	2h
AOU	0.76	0.81	0.85
AOU Impr. over No Reconf.	5%	12%	18%
AOU Impr. over Max. Ach.	20%	45%	65%

AOU: Averaged Overall Utility in 24 hours;
No Reconf. AOU: 0.72; Max. Ach. AOU: 0.92

Figure 8: Utility improvement with different reconfiguration rates

configuration Tuba performs exactly as Pileus. The average overall utility (AOU) is computed as the average utility delivered for all read operations from all clients. The average utility improvement depends on how frequently the CS performs reconfigurations. When no reconfiguration happens in the system, the AOU in the 24 hour period is 0.72. Observe that without constraints, the maximum achievable AOU would have been 1. However, limiting replication to two datacenters and a single primary decreases the maximum achievable AOU to 0.92.

Performing a reconfiguration every 6 hours improves the overall utility for almost 12 hours, and degrades it for 8 hours. This results in a 5 percent AOU improvement. When reconfiguration happens every 4 hours, the overall utility improves for 16 hours. This leads to a 12 percent AOU improvement. Finally, with 2 hour reconfigurations, AOU is improved 18 percent. Note that this improvement is 65 percent of the maximum possible improvement.

Interestingly, when no reconfiguration happens, the overall utility is better than other configurations around UTC midnight. The reason behind this phenomena is that at UTC midnight, the original replica placement is well suited for the client distribution at that time.

Figure 9 shows the hit percentages of different subSLAs. With no reconfiguration, 34% of client reads return eventually consistent data (i.e., hit the third subSLA). With 2 hour reconfigurations, Tuba

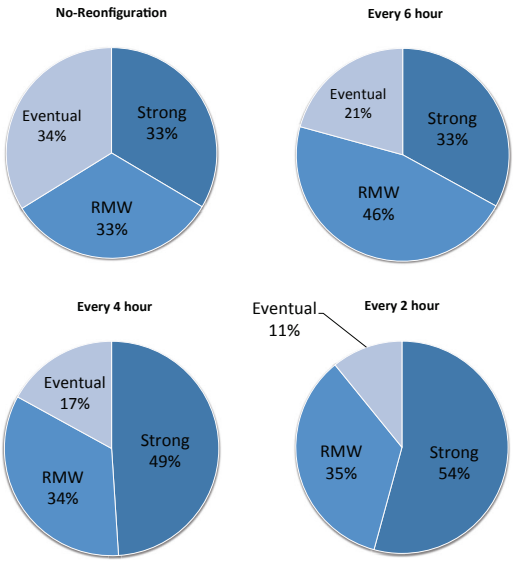
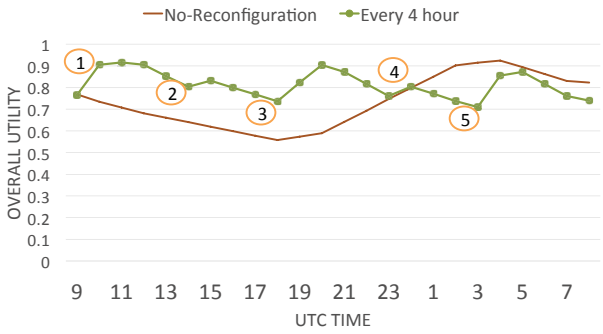


Figure 9: Hit Percentage of subSLAs



Epoch	Configuration Pri.	Configuration Sec.	Reconfiguration Operation
0	SEA	WEU	$change\_primary(WEU)$
1	WEU	SEA	$add\_secondary(SUS)$ $remove\_secondary(SEA)$
2	WEU	SUS	$change\_primary(SUS)$
3	SUS	WEU	$add\_secondary(SEA)$ $remove\_secondary(WEU)$
4	SUS	SEA	$change\_primary(SEA)$
5	SEA	SUS	

Figure 10: Tuba with Reconfigurations Every 4 hour

reduces this to 11% (a 67% improvement). Likewise, the percentage of reads returning strongly consistent data increases by around 63%.

Although the computed AOU depends heavily on the utility values specified in the SLA, we believe that the qualitative comparisons in this study are insensitive to the specific values. Certainly, the hit percentages in Figure 9 would be unaffected by varying utilities as long as the rank order of the subSLAs is unchanged.

In addition to reduced utility, systems without support for automatic reconfiguration have additional drawbacks stemming from the way they are manually reconfigured. A system administrator must stop the system (at least for certain types of configuration changes), install the new configuration, inform clients of the new configuration, and then restart the system. Such systems are unable to perform frequent reconfigurations. Moreover, the effect of a reconfiguration on throughput can be substantial since all client activity ceases while the reconfiguration is in progress.

6.3 Microscopic View

Figure 10 shows how Tuba adapts the system configuration in our experiment where reconfiguration happens every 4 hours. The first five reconfigurations are labeled on the plot. Initially, the primary replica is located in SEA, and the secondary replica is located in WEU. Upon the first reconfiguration, the CS decides to make WEU the primary replica. Though the number of clients in Asia is decreasing at this time, the overall utility stays above 0.90 for two hours before starting to degrade.

The second reconfiguration happens around 2PM (UTC time) when the overall utility is decreased by 10%. At this time, the CS detects poor utility for users located in the US, and decides to move the secondary replica from SEA to SUS. Since the geo-replication factor is set to 2, the CS necessarily removes the secondary replica in SEA to comply with the constraint. At 6PM, the third reconfiguration happens, and SUS becomes the primary replica. This reconfiguration improves the AOU to more than 0.90. In the fourth reconfiguration, the CS decides to create a secondary replica again in the SEA region. Like the second reconfiguration, in order to respect the geo-replication constraint, the secondary replica in WEU is removed. Note that the fourth reconfiguration is suboptimal since the CS does not predict clients' future behavior and solely focuses on their past behavior. A better reconfiguration would have been to make SEA the primary replica rather than the secondary replica. After 4 hours, the CS performs another reconfiguration and again is able to boost the overall utility of the system.

Although the CS performs $adjust\_sync\_period()$ with two hour reconfiguration intervals, this operation is never selected by the CS when reconfigurations happen every 4 hours. This is because changing the primary or secondary replica boosts the util-

	Fast Mode		Slow Mode	
	Read	Write	Read	Write
Client in Europe	54	143	270	785
Client in Asia	297	899	533	1598

Figure 11: Average Latency (in ms) of Read/Write Operations in Fast and Slow Modes

ity enough that reducing the synchronization period would result in little additional benefit.

6.4 Fast Mode vs. Slow Mode

In this experiment, we compare the latency of read and write executions in fast and slow modes. Since the latency of read operations with any consistency other than strong does not change in fast and slow modes, we solely focus on the latency of executing read operations with strong consistency and write operations. We placed the configuration blob in the West US (WUS) datacenter, a data tablet in West Europe (WEU), and clients in Central Europe and East Asia. The latency (in ms) between the two clients and the two storage sites are as follows:

	WEU	WUS
Client in Europe	54	210
Client in Asia	296	230

Figure 11 compares the average latencies of read and write operations in slow and fast modes. Executing strongly consistent read operations in slow mode requires also reading the configuration blob to ensure that the primary replica has not changed. Therefore, the latency of a read operation in slow mode is more than 200 ms longer than in fast mode.

Executing write operations in slow mode requires three additional RPC calls to the US (where the configuration blob is stored) in the case where no client has written a lease-id to the configuration's metadata (as in this experiment). Specifically, slow mode writes involve reading the latest configuration, taking a non-exclusive lease on the configuration blob, and writing the lease-id to the configuration's metadata. If a lease-id is already set in the configuration's metadata, the last phase is not needed, and two RPC calls are enough. We note that, with additional support from the storage servers, the overhead of write operations in slow mode could be trimmed to only one additional RPC call. This is achievable by taking or renewing the lease in one RPC call to the server that stores the configuration.

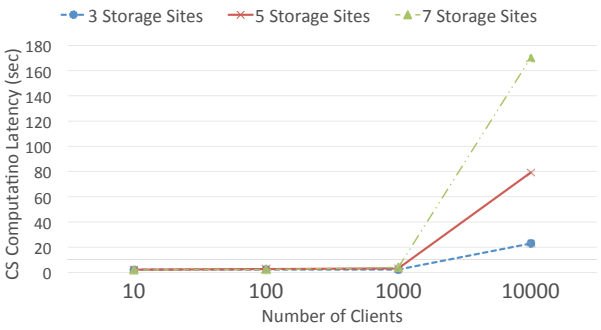


Figure 12: Scalability of the CS

6.5 Scalability of the CS

As we explained in Section 3.3, the CS considers a potentially large number of candidates when selecting a new configuration. To better understand the limitations of the selection algorithm used by our CS, we studied its scalability in practice. We put clients at four sites: East US, West US, West Europe, and Southeast Asia. Each client's SLA has three subSLAs, and all SLAs are distinct; thus, no ratio aggregation is possible. Initially, the East US site is chosen as the primary replica, and no secondary replica is deployed. We also impose the following three constraints: (i) Do not replicate in East US, (ii) Replicate in at least two sites, and (iii) Replicate in a maximum of three sites. We ran the CS on a dual-core 2.20 GHz machine with 3.5GB of memory.

Figure 12 plots the latency of computing a new configuration with 3, 5, and 7 available storage sites when the CS performs an exhaustive search of all possible configurations. With one hundred clients, it takes less than 3 seconds to compute the expected utility gain for every configuration and to select the best one. With one thousand clients, the computation time for 3 available storage sites is still less than 3 seconds, while it reaches 3.8 seconds for 7 sites. When the number of clients reaches ten thousand, the CS computes a new configuration for 3 available storage sites in 20 seconds, and for 7 available storage sites in 170 seconds.

This performance is acceptable for many systems since typically the set of cloud storage sites (i.e., the datacenters in which data can be stored) is small and reconfigurations are infrequent. For systems with very large numbers of clients and a large list of possible storage sites, heuristics for pruning the search space could yield substantial improvements and other techniques like ILP or constraint programming should be explored.

7 Related Work

Lots of previous work has focused on data placement and adaptive replication algorithms in LAN environments (e.g., [2, 8, 11–13, 18]). These techniques are not applicable for WAN environments mainly because: (i) intra-datacenter transfer costs are negligible compared to inter-datacenter costs, (ii) data should be placed in the datacenters that are closest to users, and (iii) the system should react to users' mobility around the globe. Therefore, in the remaining of this section, we only review solutions tailored specifically for WAN environments.

Kadambi et al. [9] introduce a mechanism for selectively replicating large databases globally. Their main goal is to minimize the bandwidth required to send updates and the bandwidth required to forward reads to remote datacenters while respecting policy constraints. They extend Yahoo! PNUTs [5] with a per-record selective replication policy. Their dynamic placement algorithm is based on work by Wolfson et al. [18] and responds to changes in access patterns by creating and removing replicas. They replicate all records in all locations either as a full copy or as a stub. The full replica is a normal copy of the data while the stub contains only the primary-key and some metadata. Instead of recording access patterns as in Tuba, they rely on a simple approach: a stub replica becomes full when a read operation is delivered at its location, and a full replica demotes when a write operation is observed in another location or if there has not been any read at that location for some period. Unlike Tuba, changing the primary replica is not studied in this work. Moreover, once data is inserted into a tablet, policy constraints cannot be changed. In contrast, Tuba allows modifying or adding new constraints, and the current set of constraints will be respected in the next reconfiguration cycle.

Tran et al. [16] introduce a key-value store called Nomad that allows migrating of data between datacenters. They propose and implement an abstraction called overlays. These overlays are responsible for caching and migrating object containers across datacenters. Nomad considers the following three migration policies: (i) count, (ii) time, and (iii) rate. Users can specify the number of times, a certain period, and the rate that data is accessed from the same remote location. In comparison, Tuba focuses on maximizing the overall utility of the storage system and respecting replication constraints.

Volley [1] relies on access logs to determine data locations. Their goal is to improve datacenter capacity skew, inter-datacenter traffic, and client latency. In each round, Volley computes the data placement for *all* data items, while the granularity in Tuba is a tablet. Unlike Tuba, Volley does not take into account the configuration costs or constraints. Moreover, the Volley paper does not suggest any migration mechanisms.

Venkataramani et al. [17] propose a bandwidth-constrained placement algorithm for WAN environments. Their main goal is to place copies of objects at a collection of caches to minimize access time. However, complex coordination between distributed nodes and the assumption of a fixed size for all objects makes this scheme less practical than the techniques presented in this paper.

8 Conclusion

Tuba is a replicated key-value store that, like Pileus, allows applications to specify their desired consistency and dynamically selects replicas in order to maximize the utility delivered to read operations. Additionally, Tuba automatically reconfigures itself while respecting user defined constraints so that it adapts to changes in users locations or request rates. The system is built on Microsoft Azure Storage (MAS), and extends MAS with broad consistency choices, consistency-based SLAs, and explicit geo-replication configurations.

Our experiments with clients distributed in different datacenters around the world show that Tuba with two hour reconfiguration intervals increases the reads that return strongly consistent data by 63% and improves average utility up to 18%. This confirms that automatic reconfiguration can yield substantial benefits which are realizable in practice.

Acknowledgements

We thank Marcos K. Aguilera, Mahesh Balakrishnan, and Ramakrishna Kotla for their insightful discussions as well as for their contributions to the design and implementation of the original Pileus system. We would like to also thank Pierpaolo Cincilla, Tyler Crain, Gilles Muller, Marc Shapiro, Pierre Sutra, Marek Zawirski, the anonymous reviewers, and our shepherd, Emin Gün Sirer, for their thoughtful suggestions and feedback on this work.

References

[1] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: automated data placement for geo-distributed cloud services. In *Networked Sys. Design and Implem. (NSDI)*, page 2. USENIX Association, Apr. 2010.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Usenix Annual Tech. Conf. (Usenix-ATC)*. USENIX Association, June 2000.

[3] B. Calder, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, J. Wang, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, L. Rigas, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, and J. Wu. Windows Azure Storage. In *Symp. on Op. Sys. Principles (SOSP)*, pages 143—-157, New York, New York, USA, Oct. 2011. ACM Press.

[4] 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. *Trans. on Computer Sys.*, 26(2):1–26, June 2008.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symp. on Cloud Computing (SoCC)*, pages 143—-154, New York, NY, USA, 2010. ACM.

[7] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463—-492, 1990.

[8] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, OSDI '99, pages 187–200. USENIX Association, Feb. 1999.

[9] S. Kadambi, J. Chen, B. F. Cooper, D. Lomax, A. Silberstein, E. Tam, and H. Garcia-molina. Where in the World is My Data ? In *Int. Conf. on Very Large Data Bases (VLDB)*, pages 1040–1050, 2011.

[10] G. M. V. Lili Qiu, Venkata N. Padmanabhan. On the placement of web server replicas. In *Int. Conf. on Computer Communications (INFOCOM)*, pages 1587—-1596, 2001.

[11] R. R. Madhukar R. Korupolu, C. Greg Plaxton. Placement Algorithms for Hierarchical Cooperative Caching. In *Symp. on Discrete Algorithms (SODA)*, pages 586–595. Society for Industrial and Applied Mathematics, 1999.

[12] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *Euro. Conf. on Comp. Sys. (EuroSys)*, number 4, page 89, New York, New York, USA, Oct. 2006. ACM.

[13] C. Stewart, S. Dwarkadas, and M. Scott. *Distributed Systems Online*, 05(10):1–1, Oct. 2004.

[14] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149. IEEE Computer Society, Sept. 1994.

[15] 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 *Symp. on Op. Sys. Principles (SOSP)*, pages 309–324, New York, New York, USA, Nov. 2013. ACM Press.

[16] N. Tran, M. K. Aguilera, and M. Balakrishnan. Online migration for geo-distributed storage systems. In *Usenix Annual Tech. Conf. (Usenix-ATC)*, Berkeley, CA, USA, 2011. USENIX Association.

[17] A. Venkataramani, P. Weidmann, and M. Dahlin. Bandwidth constrained placement in a WAN. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 134–143, New York, New York, USA, Aug. 2001. ACM Press.

[18] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *Trans. on Database Sys.*, 22(2):255–314, June 1997.