Consistency-Based Service Level Agreements for Cloud Storage

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla,
Mahesh Balakrishnan, Marcos K. Aguilera, Hussam Abu-Libdeh[†]

Microsoft Research Silicon Valley
[†]Cornell University

Abstract

Choosing a cloud storage system and specific operations for reading and writing data requires developers to make decisions that trade off consistency for availability and performance. Applications may be locked into a choice that is not ideal for all clients and changing conditions. Pileus is a replicated key-value store that allows applications to declare their consistency and latency priorities via consistency-based service level agreements (SLAs). It dynamically selects which servers to access in order to deliver the best service given the current configuration and system conditions. In application-specific SLAs, developers can request both strong and eventual consistency as well as intermediate guarantees such as read-my-writes. Evaluations running on a worldwide test bed with geo-replicated data show that the system adapts to varying client-server latencies to provide service that matches or exceeds the best static consistency choice and server selection scheme.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design--*Distributed systems*; H.2.4 [**Database Management**]: Systems--*Distributed databases*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services--*Data sharing*

General Terms: Design, Performance, Reliability

Keywords: Cloud Computing, Storage, Replication, Consistency, Service Level Agreement

SOSP'13, Nov. 3-6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522731

1 Introduction

Cloud storage systems, such as the currently popular class of "NoSQL" data stores, have been designed to meet the needs of diverse applications from social networking to electronic commerce. Such storage services invariably replicate application data on multiple machines to make it highly available. Many provide a relaxed form of consistency, eventual consistency, in order to achieve elastic scalability and good performance while some strive for strong consistency to maintain the semantics of one-copy serializability. To allow local access and ensure data survivability even during a complete datacenter failure, many storage systems offer "geo-replication," the option of replicating data across different regions of the world.

With data being replicated on a worldwide scale, the inherent trade-offs between performance and consistency are accentuated due to the high communication latencies between datacenters. The performance difference between reads with different consistencies can be substantial. This is not surprising. Strongly consistent reads generally involve multiple replicas or must be serviced by a primary replica, whereas eventually consistent reads can be answered by the closest replica. Even within a datacenter, the latency of strongly consistent reads has been measured as eight times that of reads with weaker session guarantees [26]. With geo-replication, our studies show that the performance can differ by more than two orders of magnitude.

Recognizing these fundamental trade-offs [12], storage systems like Amazon's SimpleDB and DynamoDB [43], Google's AppEngine data store [22], Yahoo!'s PNUTS [17], and Oracle's NoSQL Database [32] now offer read operations with a choice between strong consistency and eventual consistency. Applications can choose between stronger consistency with lower performance and relaxed consistency with higher performance.

A major problem with existing multi-consistency storage services is that application developers need to decide at development time which consistency to embrace. They use different API calls to request

different guarantees. Unfortunately, developers have insufficient information to make the best decision for all situations. The actual performance differences depend heavily on the degree of replication, relative locations of servers and clients, network and server load, and other configuration issues. In some cases, such as when a client is located on the other side of the globe from the master copy but near a potentially out-of-date replica, a strong read may be 100 times slower than an eventually consistent read, while in other cases the performance differences may be minimal or non-existent. Although some systems allow the master replica to move so that it is near active clients [17], this is not possible for data that is shared by a global user community. Different clients will observe different performance or consistency or both. This makes it difficult for developers to pick a consistency-performance pair that satisfies all users in all situations.

The Pileus storage system was designed to relieve application developers from the burden of explicitly choosing a single ideal consistency. The key novelty lies in allowing application developers to provide a service level agreement (SLA) that specifies the application's consistency/latency desires in a declarative manner. For example, an application might request the strongest consistency that can be met within a given response time. Applications issue read operations with an associated SLA. The Pileus system chooses to which server (or set of servers) each read is directed in order to comply with the SLA. Customers of a public cloud may pay extra for better levels of service, thereby incentivizing the storage service provider to meet the application's needs as closely as possible. Pileus adapts to different configurations of replicas and clients and to changing conditions, including variations in network or server load. If an application, such as a multi-player game, favors latency over consistency, then reads are directed to nearby servers regardless of whether they hold stale data, whereas applications with stronger consistency requirements have their reads directed to servers that are fully or mostly up-to-date.

Our target applications are those that tolerate relaxed consistency but, nevertheless, benefit from improved consistency. In the cloud computing world, many applications have been built on data stores that offer only eventual consistency. This includes applications for social networking, shopping, entertainment, news, personal finances, messaging, crowd sourcing, mobility, and gaming. In such applications, the cost of accessing stale data manifests itself in many ways: user inconvenience, lost revenue, compensating actions, duplicate work, and so on. If the system, with knowledge of the application's performance requirements, can return data that is as up-to-date as possible, then the applications and their users profit. Even applications that operate on strongly consistent data may take advantage of SLAs that permit some relaxed consistency. For example, as discussed further in Section 2.3, a program might beneficially perform speculative execution on eventually consistent data. Thus, we believe that a large class of applications could benefit from consistency-based SLAs.

The key challenge addressed in this paper is how to define and implement consistency-based SLAs that offer an extensive set of service levels when accessing a scalable key-value store. We make two main contributions:

First, we present the design of a cloud storage system with a range of consistency choices that lie between strong and eventual consistency. Others have demonstrated that a variety of consistency guarantees are useful including monotonic reads, read-my-writes, and bounded staleness [36][39][42][46]. Pileus allows different applications (or different users of an application) to obtain different consistency guarantees even while sharing the same data.

Second, we introduce the notion of an SLA that incorporates consistency as well as latency. With the consistency choices we offer, an application's SLA can indicate a broad set of acceptable consistency/latency trade-offs. We propose expressing such consistency-based SLAs as a series of alternative choices with decreasing value to the application. We present and evaluate techniques that attempt to maximize the delivered value when reading data.

The next section presents several concrete scenarios in which applications declare and benefit from consistency-based SLAs. Section 3 presents the services offered by Pileus to application developers including the range of consistency choices and the expressiveness of consistency-based SLAs. Section 4 then discusses the implementation of these services and the rationale for our design decisions. Section 5 evaluates the effectiveness of SLAs in improving the value provided to different applications. Section 6 outlines some extensions and areas for future exploration. Related work is reviewed in Section 7, and Section 8 concludes by revisiting the key characteristics of consistency-based SLAs.

2 Application scenarios

In this section, we explore a number of applications and illustrate the consistency-based SLAs that those applications desire. We envision a cloud computing ecosystem in which storage service providers charge clients for increasing levels of service. That is, the price that clients pay for accessing storage depends not only on the amount of data they read or write but also on the latency for accessing that data and the

consistency of the data returned for read operations. Amazon, for instance, already charges twice as much for strongly consistent reads as for eventually consistent reads in DynamoDB [4]. Major web service companies have reported that increased latency directly results in lower revenue per customer; for example, Amazon observed that they lose 1% of sales for each additional 100 ms of latency [24][40]. Some cloud companies already include latency guarantees in their SLAs, and differentiated pricing for low latency cannot be far behind [31]. Given such a pricing model, applications will not simply request strong consistency and zero latency. Applications should request the consistency and latency that provides suitable service to their customers and for which they will pay.

Even if applications are willing to pay for maximum consistency and minimal latency, current operating conditions may prevent the storage system from providing ideal service. Thus, applications also want to express less favorable but acceptable latency-consistency combinations in their SLAs. Each application has a range of acceptable consistency and tolerable latency based on the application semantics and the data it accesses. For example, although an application may prefer to read strongly consistent data, it may perform correctly even when presented stale data, and although the application may prefer 100 millisecond response times, it may be usable with delays of up to one second. An application's consistency-based SLA communicates to the storage provider, in cases where its ideal consistency and latency cannot be met, whether the application favors relaxed consistency or higher latency or both. Sample applications fit into one of three classes.

2.1 Latency-favoring applications

Shopping carts for holding electronic purchases are a well-known example of an application that has been built on an eventually consistent platform in order to maximize availability and performance [20][42]. High availability is critical since shoppers should never be prevented from adding items to their carts, and low latency is important so that customers can check out quickly. If a user at check-out time occasionally experiences a shopping cart that is not quite up-to-date, that is acceptable. For example, an Amazon shopper may occasionally see items that she previously removed from her cart. In this case, the person simply needs to remove the item again before confirming her purchases. Thus, the shopping cart application seems well suited to a data store that provides eventual consistency. However, the customers' shopping experience is clearly improved with stronger consistency. If shopping carts were inaccurate most of the time, then shoppers would get annoyed and shop elsewhere.

Ideally, this shopping cart application wants an SLA that offers the strongest consistency possible for a given latency budget. Read-my-writes consistency is sufficient since the customer is the only client that ever updates her own shopping cart. But weaker consistency is tolerable and may be required to meet strict performance standards. Suppose, as in the Dynamo system [20], the application developer is given a target of retrieving a user's shopping cart in under 300 milliseconds. The application needs the ability to say:

> *"I'd ideally like to be able to see my own updates but I'll accept any consistency as long as data is returned in under 300 ms."*

With such an SLA, clients that are near a primary replica can always read from this replica and observe perfectly accurate data, assuming the server is not overloaded. This is preferred over reading from a randomly chosen server. Clients with a round-trip time to the primary that is consistently over 300 ms, that is, customers located in remote parts of the world, should read from the closest server even if that server is not up-to-date. This is the best that can be achieved under the circumstances. The developer need not choose between strong reads or eventually consistent reads, which can result in suboptimal performance or consistency for many customers. And the storage service provider, with knowledge of the applications' performance and consistency needs, can improve the overall resource utilization.

Other latency-sensitive and inconsistency-tolerant applications include real-time multiplayer games, computer-supported collaborative work, and some data analytics. These applications would benefit from similar consistency-based SLAs.

2.2 Consistency-favoring applications

Other applications may have rigid consistency preferences but tolerate a wide range of response times for read operations. For example, a web search application may want search results with bounded staleness. While this application will accept different latencies, and, indeed, users are accustomed to unpredictable response times, the application loses advertising revenue when fetching data is slower. Essentially, the application wants an SLA that allows it to express the following:

> *"I want data that is at most 5 minutes out-of-date, and I will pay $0.00001 for reads that return in under 200 ms, $0.000008 for reads with latency under 400 ms, $0.000005 for reads in under 600 ms, and nothing for reads over 600 ms."*

Many other applications that are funded with advertising revenue fit into the same class and could use similar SLAs, though perhaps with different

```
data = WeakRead (key);
display (data);
latestdata = StrongRead (key);
if data != latestdata {
   display (latestdata);
}
```

Figure 1. Code segment for displaying inaccurate data quickly and more accurate data later.

consistency choices. These include social networking applications like Facebook, web-based e-mail and calendaring programs, news readers, personal cloud file systems, and photo sharing sites.

2.3 Applications with other trade-offs

As another example, consider applications that want to display information quickly in response to a user's request and then update the display with more accurate data as it arrives. For example, a browsing application might display results based on locally available information and then later add additional results from a more extensive search. Or a news reading application might display a slightly outdated list of news stories and then update the list with the latest stories. The code pattern depicted in Figure 1 arises in a number of applications that retrieve data from a service that offers both strong and eventually consistent reads.

When writing this code, the developer assumes that simply issuing a StrongRead will make the user wait too long and is aiming for a better user experience. He assumes that the WeakRead has a low response time and that the StrongRead will take longer to return. And he must pessimistically assume that the two reads return different results. But what if these assumptions are incorrect? If strongly consistent data can be fetched as quickly (or almost as quickly) as eventually consistent data, then performing two reads is wasteful, and the client unnecessarily pays for both operations. If the application runs on a machine that is near a primary replica, for instance, the client will likely obtain accurate data with the first read operation. Unfortunately, the application code has no way of determining this. On the other hand, if the application is far from any replicas, even the WeakRead operation may have an unsatisfactory response time. In this case, the application may prefer to wait for accurate data since the user already experiences a noticeable delay.

A consistency-based SLA allows the developer to precisely state his desires in a declarative manner. He might decide that accessing data in under 150 ms is fast enough. For this application, the desired SLA says the following:

> *"I want a reply in under 150 ms and prefer strongly consistent data but will accept any data; if no data can be obtained quickly then I am willing to wait up to a second for up-to-date data."*

With this SLA applied to the first read in Figure 1, the operation will return strongly consistent data if it can be fetched quickly or if no data is accessible in a timely manner. The client is informed whether the data was retrieved from a primary replica so that it can skip the second, unnecessary read operation. This SLA allows the client to obtain better performance and reduced execution costs in many, but not all, situations.

Similar code patterns arise in other applications. For example, many systems maintain user credentials in an eventually consistent distributed database, like Microsoft's Active Directory. Some password checking programs first issue a weakly consistent read to obtain a user's credentials. Then, if the password check fails, they issue a strong read to ensure that the password is checked against the user's latest credentials. The intent, as with the code snippet above, is to perform the password check as fast as possible by using data that can be quickly obtained. Since users change their passwords infrequently, reading from any server will almost always return a user's current credentials. Nevertheless, reading data that is guaranteed to be accurate is preferable if it can be done with bounded latency. Thus, these password checking programs desire a consistency-based SLA like the one above.

This password checking program is one example of a broad class of programs that want strongly consistent data but may benefit from speculatively executing on data returned by an eventually consistent read. Such programs have a similar structure to the code in Figure 1 except the calls to display data are replaced by code that executes on the data. In other words, data is fetched quickly and execution is started using this data as input. In the background, strongly consistent data is fetched; only if it differs from the data that was previously returned is the computation restarted. In the common case where most servers are up-to-date and can return accurate data, reading from a nearby server and starting a speculative execution can result in a reduced overall computation time. As a concrete example, consider an application that generates thumbnails for a collection of photos. Since photos are rarely edited, eventual consistency reads will almost always return the latest photo, and so the application can speculatively generate thumbnails from locally available photos but wants to ensure that the correct thumbnail is produced. Such speculative programs can use similar consistency-based SLAs.

```
CreateTable (name);
DeleteTable (name);
tbl = OpenTable (name);

s = BeginSession(tbl, sla);
EndSession(s);

Put (s, key, value);
value, cc = Get (s, key, sla);
```

Figure 2. The Pileus API

3 System API

This section examines the functionality presented to application developers. The Pileus system combines the features of a traditional key-value store offered by current cloud storage providers with new consistency choices and expanded service level agreements.

3.1 Key-value store

From the viewpoint of an application developer, Pileus manages a collection of replicated key-value *tables*. Applications can create any number of tables for holding application-specific data. A table contains one or more *data objects*. Each data object consists of a unique string-valued *key* and a *value* that is an opaque byte-sequence. Every table is created with a globally unique name and is managed independently with its own configuration of servers and replication policies; for the most part, these configuration details are transparent to applications.

Put is the method for adding or updating a data object with a given key. If a Put is performed for a key that does not already exist in the table, then a new data object is created; otherwise, a new version of the object is produced in which its value is overwritten with the new value.

Get fetches the data associated with the given key. It differs from Get operations in a traditional key-value store in that it takes a consistency-based SLA as an optional argument. Additionally, the Get method returns, along with some version of a data object, a condition code that indicates whether (or how well) the SLA was met, including the consistency of the data. Section 3.3 discusses the structure and semantics of consistency-based SLAs. If the governing SLA allows relaxed consistency, then the data object being read may not be up-to-date, i.e. clients may read a version that existed at some point in the past.

Pileus supports transactions. That is, a client may call BeginTx, perform a sequence of Get and Put operations on arbitrary objects, and then call EndTx. Each transaction runs with snapshot isolation and is committed atomically. The details of the transaction mechanisms are beyond the scope of this paper but are available in a technical report [38]. In this paper, we treat each Get and Put as single-operation transactions.

All Get and Put operations are enclosed within a *session*. Sessions serve as the scope for certain consistency guarantees such as *monotonic reads*; these guarantees are presented in the next section. The BeginSession method takes a consistency-based SLA as its only argument. This serves as the default SLA for all of the Gets within the session; however, a Get may override the default by indicating its own SLA.

The basic operations supported by Pileus (ignoring transactions) are summarized in Figure 2. This API is provided by a client library that is called by applications. The client library, as discussed in more detail in Section 4, performs operations by contacting one or more servers as needed.

3.2 Consistency guarantees

Each Get operation returns some, but not necessarily the latest, version written by a Put to the key being read. Different consistency guarantees permit different amounts of staleness in the values of the objects returned by Gets. Pileus offers several read guarantees, which we informally define as follows:

- ***strong***: A Get(*key*) returns the value of the last preceding Put(*key*) performed by any client.
- ***eventual***: A Get(*key*) returns the value written by any Put(*key*), i.e. any version of the object with the given key; clients can expect that the latest version eventually would be returned if no further Puts were performed, but there are no guarantees concerning how long this might take.
- ***read-my-writes***: A Get(*key*) returns the value written by the last preceding Put(*key*) in the same session or returns a later version; if no Puts have been performed to this key in this session, then the Get may return any previous value as in eventual consistency.
- ***monotonic***: A Get(*key*) returns the same or a later version as a previous Get(*key*) in this session; if the session has no previous Gets for this key, then the Get may return the value of any Put(*key*).
- ***bounded(t)***: A Get(*key*) returns a value that is stale by at most *t* seconds. Specifically, it returns the value of the latest Put(*key*) that completed more than *t* seconds ago or some more recent version.
- ***causal***: A Get(*key*) returns the value of a latest Put(*key*) that *causally precedes* it or returns some later version. The causal precedence relation < is defined such that *op1*<*op2* if either
  (a) *op1* occurs before *op2* in the same session,
  (b) *op1* is a Put(*key*) and *op2* is a Get(*key*) that returns the version put in *op1*, or
  (c) for some *op3*, *op1*<*op3* and *op3*<*op2*.

Consistency	U.S.	England (Primary)	India	China
strong	147	1	435	307
causal	146	1	431	306
bounded(30)	75	1	234	241
read-my-writes	13	1	18	166
monotonic	1	1	1	160
eventual	1	1	1	160

Figure 3. Average latency (in milliseconds) observed for consistency choices and client locations

This selection of guarantees was motivated by earlier work demonstrating their usefulness [39]. As noted previously, a number of cloud storage providers currently offer clients a choice between strong and eventual consistency. The read-my-writes and monotonic guarantees were part of the *session guarantees* offered by Bayou [36]. Bounded staleness has been proposed in a number of systems [9][46]. Our definition of causal consistency resembles that used in other systems [28][30] but includes the notion of sessions. Like previous storage systems, it only considers causal dependencies between Puts and Gets and does not track causality through direct client-to-client messages.

As will be evident when we discuss how these read guarantees are implemented in Section 4, selecting a guarantee may limit the set of servers that can process a Get operation. Limiting the set of suitable servers indirectly increases the expected read latency since nearby servers may need to be bypassed in favor of more distant, but more up-to-date replicas. For example, strong reads must be directed to the primary site; eventual reads can be answered by any replica thereby delivering the best possible availability and performance. The other read guarantees fall somewhere in between these two consistency extremes in terms of trading off consistency and latency.

Figure 3 shows the average latency obtained in our system when performing Gets with certain consistency choices. In this experiment, we geo-replicated data across three datacenters with the primary site in England and secondary sites in the U.S. and India, and we ran the YCSB benchmark on clients in four different locations; more details are discussed in Section 5 where we present additional evaluations. These numbers confirm that latency does indeed vary drastically for different consistencies and also differs significantly from client to client. Fortunately, when using Pileus, an application developer need not be fully aware of the performance consequences of choosing a specific consistency guarantee or commit to a particular choice; instead, the developer provides an SLA to capture the application needs and preferences.

Rank	*Consistency*	*Latency*	*Utility*
1.	read-my-writes	300 ms	1.0
2.	eventual	300 ms	0.5

Figure 4. The shopping cart SLA

3.3 Service level agreements

In Pileus, consistency-based SLAs allow an application developer to express his desired consistency and latency for `Get` operations. Importantly, an SLA indicates whether the system should relax consistency or use servers with slower response times when the application's ideal consistency and ideal latency cannot both be met given the current conditions.

An SLA is specified as a sequence of consistency and latency targets. Each consistency-latency pair is called a *subSLA*. A given SLA can contain any number of subSLAs. The first subSLA indicates the application's preferred consistency and latency. Lower subSLAs denote alternatives that are acceptable but less desirable. Generally, lower subSLAs permit lesser consistency or higher latency or both.

For example, the SLA for the shopping cart application discussed in Section 2.1 can be expressed as in Figure 4. This SLA says that the application prefers the read-my-writes guarantee but can tolerate eventual consistency and absolutely requires round-trip times of under 300 ms. For the web applications discussed in Section 2.2 that favor lower latency, the SLA can be defined as in Figure 5. The SLA for the password checking scenario discussed in Section 2.3 is shown in Figure 6.

Pileus makes a best effort attempt to provide the highest desired service. It may not always succeed in meeting the top subSLA due to the configuration of replicas and network conditions, over which Pileus has no control, or because it makes a poor decision based on inaccurate information as arising from changing load conditions. Along with the data that is returned by a Get, the caller is informed of which subSLA was satisfied. This information allows the application to take different actions based on the consistency of the returned data.

If none of the subSLAs can be met, then the Get call returns with an error code and no data. Thus, *unavailability* in Pileus is defined in practical terms as *the inability to retrieve the desired data with acceptable consistency and latency as defined by the SLA*. If an application wants maximum availability, it need only specify `<eventual, unbounded>` as the last subSLA. In this case, data will be returned as long as some replica can be reached.

Rank	Consistency	Latency	Utility
1.	bounded(300)	200 ms	0.00001
2.	bounded(300)	400 ms	0.000008
3.	bounded(300)	600 ms	0.000005
4.	bounded(300)	1000 ms	0.0

Figure 5. The web application SLA

Associated with each subSLA is the *utility* of that consistency-latency pair to the application. Lower-ranked subSLAs have lower utility than higher-ranked ones within the same SLA. The utility of a subSLA is a number that allows applications to indicate its relative importance. As will be seen in Section 4.6, these utilities are used to decide the best strategy for meeting an SLA. If Pileus were deployed as a public cloud service with a tiered pricing model, the utility of a subSLA ideally would match the price the storage provider charges for the given level of service. In this case, the storage provider has a strong incentive to meet the highest subSLA possible, the one that will generate the highest revenue.

4 Design and implementation

This section presents the design and implementation of the Pileus system. The emphasis is on the challenges faced in providing consistency-based SLAs for access to data that is partitioned and geo-replicated.

4.1 Architecture

The Pileus system contains the following major components:

Storage nodes are servers that hold data and provide a Get/Put interface. They know nothing about consistency guarantees or SLAs. The Put operation writes a new version with a given timestamp, while the Get operation returns the latest version that is known to the node. Any number of storage nodes may exist in each datacenter. As discussed in more detail below, some storage nodes are designated as primary nodes, which hold the master data, while others are secondary nodes.

Replication agents are co-located with storage nodes and asynchronously propagate updates between nodes. Any replication protocol could be used as long as updates are applied in timestamp order. In our current implementation, secondary nodes periodically pull new versions of data objects from primary nodes (though they could also receive updates from other secondary nodes). Each replication agent simply records the latest timestamp of any version it has received and periodically, say once per minute, retrieves versions with higher timestamps.

Rank	Consistency	Latency	Utility
1.	strong	150 ms	1.0
2.	eventual	150 ms	0.5
3.	strong	1 sec	0.25

Figure 6. The password checking SLA

Monitors track the amount by which various secondary nodes lag behind their primaries and measure the roundtrip latencies between clients and storage nodes. In the current system, each client has its own monitor though having a shared monitoring service could be useful (as discussed in Section 6.1).

The *client library* is linked into the application code. It exports the API presented in Section 3 and maintains state for sessions. Moreover, the client library contains the logic for directing Get operations to the storage nodes that maximize the expected utility for a given SLA.

4.2 Replication and partitioning

For scalability, a large table can be sharded into one or more *tablets*, as in other storage systems like BigTable [16]. Horizontal partitioning divides each table into tablets according to key-ranges. Tablets are the granularity of replication and are independently replicated on multiple storage nodes.

All Puts in Pileus are performed and strictly ordered at a *primary site*, a set of storage nodes within a datacenter. Different tablets may be configured with different primary sites; only the primary site accepts Put operations for keys in the tablet's key-range. This mimics the design of many commercial cloud storage systems including Windows Azure Storage [14] and PNUTS, except that PNUTS allows per-object masters [17]. The advantages of this primary-update approach, compared to a multi-master update scheme, are two-fold. First, the primary site is an authoritative copy for answering strongly consistent Gets. Second, the system avoids conflicts that might arise from different clients concurrently writing to different servers.

The primary site could consist of a fault-tolerant cluster of servers, but any such structuring is invisible to clients. The initial Pileus prototype, which is used for the evaluations in section 5, designates a single node as the primary. Clearly, this has limited fault-tolerance that could be addressed by well-known techniques such as the Paxos-based scheme used in Spanner [19] or chain replication [41]. As an example, we have built a second version of Pileus in which each "node" is a Windows Azure Storage account utilizing strongly consistent three-way replication.

Secondary nodes *eventually* receive all updated objects along with their update timestamps via an asynchronous replication protocol. Because this

protocol reliably transmits objects in timestamp order, Pileus actually provides a stronger form of eventual consistency than many systems, a guarantee that has been called *prefix consistency* [37] or *timeline consistency* [17]. No assumptions are made about the time required to fully propagate an update to all replicas, though more rapid dissemination increases a client's chance of being able to read from a nearby node and hence increases the likelihood of satisfying a latency-critical SLA.

Consistency choices and consistency-based SLAs are applicable in any system that contains a mixture of strongly consistent nodes that are synchronously updated and eventually consistent secondary nodes that are asynchronously updated, especially when there is a large variation in access times to different nodes. This is invariably the case in systems that employ world-scale geo-replication, where performing synchronous updates to large numbers of widely distributed storage nodes is costly. But systems with both primary and secondary nodes are also commonly deployed within a datacenter or geographical region.

Our Pileus prototype is manually configured by a system administrator. Currently, the system does not automatically elect primary sites, create new replicas, migrate nodes, or repartition tables. Other work has shown how to provide more dynamic reconfiguration [1][13][25], and we could adopt such mechanisms in the future as they are largely orthogonal to our new contributions.

4.3 Storage metadata

Each storage node maintains a single version of each data object in its tablet. The state managed by a node includes the following information:

- *tablet store* = set of <key, value, timestamp> tuples for all keys in a range.
- *high timestamp* = the update timestamp of the latest data object version that has been received and processed by this node.

Since all Put operations are assigned increasing update timestamps from the primary site and the replication protocol transfers updated objects in timestamp order, at any point in time, each node has received a prefix of the overall sequence of Put operations. Thus, a single high timestamp per node is sufficient to record the set of updates that have been processed at the node. When a node receives and stores a new version of some data object, it updates its high timestamp to the object's update timestamp and records this same timestamp with the stored object.

If no Puts have updated the tablet recently, secondary nodes will receive no new versions during their periodic replication. In this case, the primary sends its current time causing secondary nodes to advance their high timestamps; this permits clients to discover that these nodes are up-to-date.

In response to a Get(key) request, the node replies with its locally stored version of the object with the requested key. Included in the response is the object's timestamp as well as the node's high timestamp.

4.4 Consistency-specific node selection

When selecting the node to which a Get operation should be sent, the desired consistency guarantee, along with the previous object versions that have been read or written in the current session and the key being read, determines the *minimum acceptable read timestamp*. This read timestamp can be computed based solely on information maintained by the client. Any storage node whose high timestamp is greater than or equal to this minimum acceptable read timestamp is sufficiently up-to-date to process the Get request. In other words, the minimum acceptable read timestamp indicates how far a secondary node can lag behind the primary and still provide an answer to the given Get operation with the desired consistency. Figure 7 illustrates an example in which a node can provide any of the consistency choices except for strong and causal.

For strong consistency, the minimum acceptable read timestamp must be at least as large as the update timestamp of the latest Put to the key that is being Get. This guarantees, as expected, that each Get accesses the latest version of the object. In practice, clients do not actually need to determine the minimum acceptable read timestamp for strongly consistent Gets. The client simply sends such operations to the key's primary site.

For read-my-writes guarantees, the client's session state records the update timestamps of any previous Puts in the session. The minimum acceptable read timestamp is the maximum timestamp of any previous Puts to the key being accessed in the current Get.

For monotonic reads, the client's session state records each key along with the timestamp of the latest object version returned by previous Gets. The minimum acceptable read timestamp is the recorded timestamp for the key being accessed in the Get.

For bounded staleness, the minimum acceptable read timestamp is simply the current time minus the desired time bound. Clients and storage nodes need only have approximately synchronized clocks since staleness bounds tend to be large, often on the order of minutes. Work has shown that clocks can be tightly synchronized even across globally-distributed datacenters [19].

For causal consistency, observe that, because Puts are performed in causal order at the primary site, each secondary node always holds a causally consistent, but perhaps stale, copy of a tablet. However, causal consistency could still be violated if clients perform Gets from randomly selected nodes. Causal

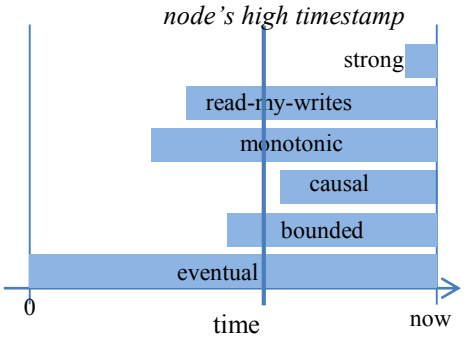

Figure 7. Acceptable read timestamp ranges for different consistency guarantees

consistency can be guaranteed, while still allowing clients a choice of servers, by setting the minimum acceptable read timestamp in a similar manner to the monotonic reads and read-my-write guarantees. Specifically, the minimum acceptable read timestamp is the maximum timestamp of *any* object that was previously read or written in this session.

For eventual consistency, the minimum acceptable read timestamp is simply time zero. Get operations are thus allowed to be sent to any storage node for the given key.

4.5 Monitoring storage nodes

Clients need information about both the network latency and high timestamp of each storage node. Monitors residing in each client record the set of nodes for each tablet along with their pertinent statistics. This information is collected as clients perform Get and Put operations on various nodes; for nodes that have not been accessed recently, the monitor may send active probes. The monitor measures the round-trip latency of each operation and records a sliding window of the last few minutes of measurements. It also records the maximum high timestamp that it has observed for each node. Monitors implement three main operations that return probability estimates based on the recorded information.

PNodeCons (node, consistency, key) returns a number between zero and one indicating a conservative estimate of the probability that the given storage node is sufficiently up-to-date to provide the given consistency guarantee for the given key. Although clients do not have perfect knowledge of each node's high timestamp, it only increases over time, and thus stale information is still useful. The local monitor returns one if the node's last known high timestamp is greater than the minimum acceptable read timestamp for the given consistency (as discussed in the previous subsection), and otherwise returns zero.

```
SelectTarget (SLA, key) =
    maxutil = -1;
    bestnodes = {};
    bestlatency = ∞;
    targetSLA = null;
    foreach subSLA in SLA
        foreach node in key.replicas
            util = PNodeSla (node, subSLA.consistency,
                    subSLA.latency, key)  * subSLA.utility;
            if (util > maxutil)
                targetSLA = subSLA;
                maxutil = util;
                bestnodes = node;
            else if (util = maxutil)
                bestnodes = bestnodes + node;
    foreach node in bestnodes
        if (node.latency < bestlatency)
            bestnodes = node;
            bestlatency = node.latency;
    return targetSLA, bestnodes;
```

Figure 8. Algorithm that selects target subSLA and node for each Get operation

PNodeLat (node, latency) returns an estimate of the probability that the node can respond to Gets within the given response time. Determining whether the node is likely to respond in time is based on past measurements of its round-trip response times; a sliding window is maintained so that the system reacts to changing latency caused by failed links or varying load. PNodeLat returns the fraction of previous times that are less than the desired latency. More recent measurements could be weighted higher than older ones.

PNodeSla (node, consistency, latency, key) returns an estimate of the probability that the given node can meet the given consistency and latency for the given key. This is obtained by multiplying PNodeCons (node, consistency, key) by PNodeLat (node, latency).

4.6 Client-side SLA enforcement

One simple, but flawed scheme for meeting an SLA is to broadcast each Get operation to all replicas, and then take the first response that provides the desired consistency and latency. However, broadcasting Gets would be wasteful of network and server resources. More importantly, such a strategy would yield a multiplicative increase in the client's financial costs since cloud service providers charge for each byte that is sent/received and for each operation performed.

The client library's main responsibility is to determine the minimum acceptable read timestamp based on the session's SLA and send each Get operation to a single node (or perhaps small set of nodes) that can provide data at the desired consistency and also can meet the target round-trip latency.

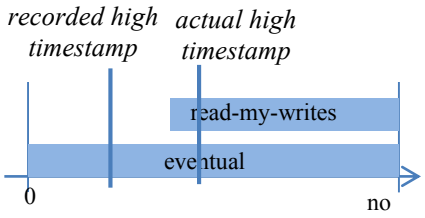

Figure 9. Meeting a higher subSLA than predicted

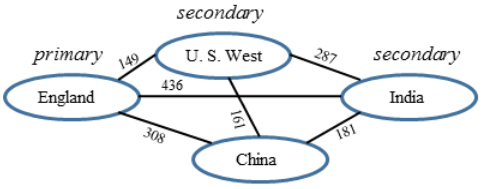

Figure 10. Experimental configuration with the average round-trip latency (in milliseconds) between datacenters in four countries

4.6.1 Choosing a target subSLA

Recall that an SLA consists of an ordered list of subSLAs where each subSLA has an application-provided utility. The client library's goal in selecting a storage node is to maximize the expected utility. Figure 8 illustrates the algorithm used when presented with a Get operation and an SLA.

For each subSLA and each node storing the key that is being accessed by the Get, the client computes the expected utility that would accrue from sending the Get to that node. This expected utility is the product of the PNodeSla function provided by the monitor and the utility associated with this subSLA. The client selects the *target subSLA* with the highest expected utility along with the set of nodes that it believes can best meet this subSLA at the current time. If multiple nodes offer the same expected utility, the client chooses the one that is closest. Alternatively, the client could choose one at random to balance the load or pick the one that is most up-to-date.

Note that the application's top subSLA is not always chosen as the target subSLA. For example, consider the shopping cart SLA specified in Figure 4. If the second subSLA has a utility that is only slightly less than that of the first subSLA and the first subSLA has a much lower chance of success, then the client will select the second subSLA as its target and choose among the nodes that can provide eventually consistent data rather than aiming for read-my-writes consistency.

4.6.2 Determining which subSLA was met

The client measures the time between sending a Get and getting a reply, and uses this round-trip latency along with timestamps included in the reply to determine whether the target subSLA was met. The client may determine that some higher or lower subSLA was satisfied.

In the response to each Get operation, along with the value of the requested data object, a storage node includes its current high timestamp. Given the minimum acceptable read timestamps for each consistency guarantee, the client can use the responding node's high timestamp to determine what consistency is actually being provided for a particular Get. The client uses this actual consistency, along with the measured round-trip latency, to determine which subSLA was satisfied and returns this indication to the Get's caller.

Interestingly, a Get may meet a higher subSLA than the target subSLA. For example, revisiting the shopping cart SLA, although the client may have chosen a node that it believed would provide only eventual consistency, the storage node may return an object whose version satisfies the read-my-writes guarantee as illustrated in Figure 9. This could very well happen in practice when the client has outdated information for storage nodes, and hence severely underestimates whether a node can meet a guarantee.

5 Evaluation

This section describes experiments we conducted to evaluate Pileus in a globally distributed datacenter environment. The goal was to verify that adapting consistency to different conditions in accordance with application-specific SLAs can yield significant benefits compared to selecting a fixed consistency.

5.1 Experimental set-up

For these experiments, Pileus was run on a research test bed connecting private datacenters in different parts of the world. As shown in Figure 10, the primary storage node was in England and secondary nodes were placed on the U.S. West Coast and in India. We evaluate configurations where the client runs in the same datacenter as one of the nodes as well as when it is in China.

The widely used YCSB benchmark [18], which was developed for evaluating the performance of cloud-based key-value stores, provided the workload that we used in our experiments. In this workload, clients perform equal numbers of Puts and Gets to a collection of 10,000 keys. We adapted this workload to add the notion of sessions. In particular, we started a session, performed 400 Gets and Puts in this session, then ended the session and started a new one.

One client performs all of the Gets and Puts in the benchmark. The origin of the Puts is irrelevant since they all are performed on the primary node, and so the

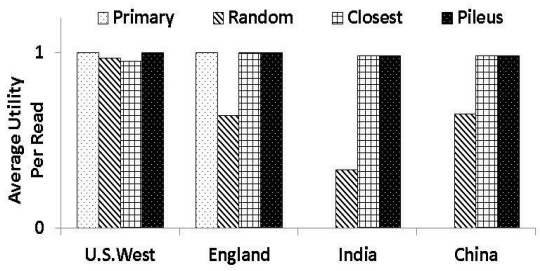

Figure 11. Utility of shopping cart SLA for clients in U.S., England, India, and China

Client	*Target SubSLA*	*Get from U.S.*	*Get from England*	*Get from India*	*SubSLA Met*	*Avg. Utility*
U.S.	1	90.9%	9.1%	0%	100%	1.0
	2	0%	0%	0%	0%	
England	1	0%	100%	0%	100%	1.0
	2	0%	0%	0%	0%	
India	1	0.2%	0%	95.9%	96.1%	0.98
	2	0%	0%	3.9%	3.9%	
China	1	95.1%	0%	0.4%	95.5%	0.98
	2	4.5%	0%	0%	4.5%	

Table 1. Breakdown of Pileus client decisions for shopping cart SLA

results are the same as if a distributed set of clients were performing concurrent updates. Once per minute, both secondary nodes pull all newly created versions from the primary.

The goal is to measure how well the client's Get operations meet a given consistency-based SLA. We use the shopping cart SLA in Figure 4 and the password checking SLA in Figure 6; the third SLA presented in Figure 5 is not evaluated since it uses a single consistency and would not provide additional insights into Pileus. We compare Pileus to three alternative systems that use simpler, fixed strategies for deciding where to send each Get operation:

- *Primary* always performs Gets at the primary node. This is equivalent to systems like Azure [14] that only offer strong consistency or choosing strong reads in SimpleDB [5].
- *Random* performs each Get at a node selected randomly from one of the three. This mimics the behavior of SimpleDB when clients choose eventually consistent reads [44].
- *Closest* always performs Gets at the node with the lowest average latency. Thus, it promises only eventual consistency.

The comparison metric used for all experiments is the average delivered utility. Recall that each SLA consists of a list of subSLAs each with their own numerical utility. For each Get operation, Pileus returns an indication of which subSLA was met. For the alternative systems, we similarly determine the subSLA that was met for each Get even though their behavior does not depend on the given SLA. The subSLA, in turn, determines the utility that is delivered for each Get, and we average this delivered utility over all operations in the workload.

5.2 Shopping cart SLA

Figure 11 shows results for the shopping cart SLA. The fixed strategy of always sending Gets to the primary works well for clients in the U.S. and England since their round-trip latencies are always below the desired 300 ms, but completely fails for clients in India and China. As expected, choosing random nodes is not ideal for anyone since a too-distant node is often chosen. Choosing the closest storage node works reasonably for this particular SLA since the read-my-writes consistency guarantee can be met at least 90% of the time. But there are times when clients (except those in England) receive only eventual consistency from their local node; these clients obtain an average utility from 0.95 for the U.S. client to 0.98 for India. Pileus's utility-driven strategy always performs well.

Table 1 provides more detail on the decisions made by the Pileus client in trying to meet this SLA; this table indicates the percentage of Gets for which Pileus selects each combination of target subSLA and storage node. When the client is in the U.S., Pileus always chooses the top subSLA as its target. This client reads from the local U.S. node 90.9% of the time, i.e. when this node can provide the read-my-writes guarantee, and otherwise sends its Gets to England. This results in a perfect average utility unlike the Closest strategy. Note that Pileus and the Primary strategy both satisfy the top subSLA 100% of the time, but Pileus obtains an average latency of 14.48 ms compared to 148 ms when accessing the primary. When the client is in England, Pileus always reads from the local node, as expected. With the client in India, Pileus targets the top subSLA if it believes that the read-my-writes guarantee can be met locally or by accessing the U.S. and otherwise targets the second subSLA. It almost always chooses to access the local secondary in India, and it obtains a high utility value since this secondary is quite often sufficiently up-to-date. The results for the client in China are similar although all of its Get operations are sent to remote nodes; it does, however, experience much higher round-trip times. The China client mostly accesses the U.S. node since it is the closest. Occasionally, 0.4% of the time, it discovers that the India node can meet the read-my-writes guarantee while the U.S. node cannot. It never accesses England since the latency is too high.

5.3 Password checking SLA

Using the password checking SLA produces the results in Figure 12 and a more detailed breakdown in

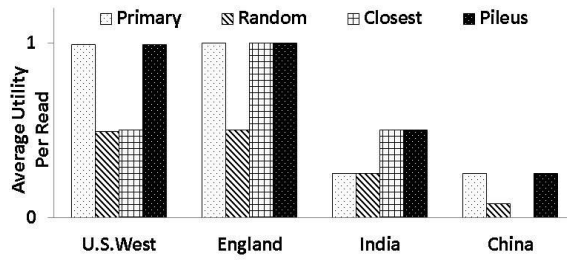

Figure 12. Utility of password checking SLA

Client	*Target SubSLA*	*Get from U.S.*	*Get from England*	*Get from India*	*SubSLA Met*	*Avg. Utility*
U.S.	1	0%	100%	0%	99.4%	0.99
	2	0%	0%	0%	0%	
	3	0%	0%	0%	0.6%	
England	1	0%	100%	0%	100%	1.0
	2	0%	0%	0%	0%	
	3	0%	0%	0%	0%	
India	1	0%	0%	0%	0%	0.5
	2	0%	0%	100%	100%	
	3	0%	0%	0%	0%	
China	1	0%	0%	0%	0%	0.25
	2	0%	0%	0%	0%	
	3	0%	100%	0%	100%	

Table 2. Breakdown of Pileus client decisions for password checking SLA

Table 2. Again, Pileus targets the top subSLA when the client is in the U.S. or England and always reads from the primary. When the client is in the U.S., choosing to read from the remote primary rather than the local secondary is almost always the correct decision, though occasionally the primary does not respond quickly enough, resulting in a .99 average utility. When the client is in India, Pileus believes the top subSLA to be unattainable since the primary is too far away, and thus it targets the second subSLA and reads from the local secondary. The client in China is so remote that Pileus forgoes the top two subSLAs and targets the third one, causing it to read from the primary but delivering only a 0.25 utility; in contrast, the Closest strategy always accesses the U.S. and receives a zero utility, which is worse than Random's 0.08 average utility.

5.4 Adaptability to network delays

To explore how well Pileus adapts to latencies that change drastically over time, we repeated our experiments for the password checking SLA while introducing artificial delays in the round-trip times for Get operations. Figure 13 shows the delivered utility for Gets performed over a period of almost six minutes. Initially, with no added delays, the client (in the U. S.) always chooses to go to the primary (in England) for the first subSLA but occasionally the primary does not respond in time (as previously indicated in Table 2). At the point labeled #1 in Figure 13, the latency to the primary node was increased by 300 ms; we simply added 300 ms to the measured round-trip times that were reported to the client. Such an increase might happen in practice if the primary or its inbound/outbound network becomes overloaded. For some small period of time (between points #1 and #2 in the figure), the client continued to choose the top subSLA and continued to send all of its Gets to the primary. None of these Gets returned in time to meet the top subSLA but they did satisfy the third subSLA, resulting in a utility of 0.25.

Eventually, the client learned that the primary was too far away, switched to the second subSLA, and started performing Gets on the local node (between points #2 and #3 in the figure). At point #3, we added 300 ms to the latency when accessing the local node. For some period (between points #3 and #4) the client continued to use the local node, but it responded too slowly and was too inconsistent to meet any of the subSLAs, resulting in a utility of zero. Note that in this case, after receiving a local response, the client could have performed the Get at the primary and still have met the third subSLA within the specified 1-second bound; we are considering adding such a strategy to the client library. At point #4, the client decided correctly that only the third subSLA could be met and resumed sending Gets to the primary.

At point #5 we reduced the access latency to the local node back to a millisecond, and at point #6 we restored the average latency to the primary to the usual 149 ms. The client eventually discovers, through periodic probes, that it can regularly access its local site with low delay, and the client switches back to choosing the second subSLA; this switch takes a while since the client probes infrequently and has some built-in hysteresis. Later, sometime after point #6, the client switches to targeting the top subSLA and resumes sending Gets to the primary. This experiment clearly demonstrates Pileus's ability to select a strategy that maximizes the delivered utility in response to varying network latencies.

5.5 Sensitivity to utility values

Finally, to study the sensitivity of our results to the utility number included in an SLA, we varied the utilities for the password checking SLA in Figure 6. We multiplied the utilities of the second and third subSLAs by a factor from 2, which places the second subSLA on par with the first, to 0.1, which makes the top subSLA considerably more valuable. These results are presented in Figure 14. Observe that different utilities affect the relative rankings of the fixed selection schemes but Pileus again outperforms them.

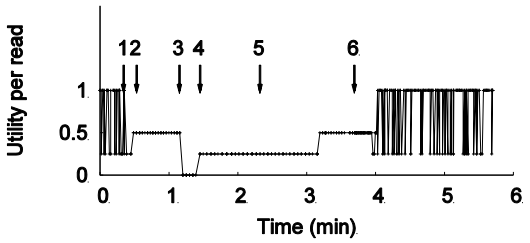

Figure 13. Behavior under varying latency

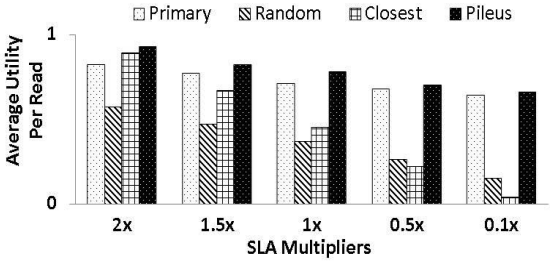

Figure 14. Behavior under varying utility

5.6 Summary

In all of the configurations that we measured, Pileus delivered the same utility as the best performing fixed consistency scheme. As expected, always requesting strong or always requesting eventual consistency yielded suboptimal service in some configurations. Pileus was able to adapt the service provided to clients in different locations to best meet the target SLA.

6 Extensions and future work

6.1 Enhanced monitoring

Although the monitoring performed in Pileus does not consume many resources, especially when it piggybacks on normal traffic, it could potentially be improved. For one thing, clients could adapt the rate at which they send periodic probes based on the data they obtain. If the latency to a node is fairly stable and the consistency predictable, then clients could probe less frequently.

Even if communication latencies are well-known, probes are used to determine the staleness of a storage node. Clients could potentially predict a node's high timestamp based on the time that it last communicated with the node as well as knowledge about the update rates for various objects and the replication protocol's propagation delay.

Additionally, clients could share monitoring information with other clients in the same datacenter. We have considered having a distributed monitoring service that is detached from individual clients, perhaps with monitors in every region of the world, or having clients gossip monitoring information among themselves. Exploring the relationship between the amount of traffic generated by various monitoring schemes and the accrued benefits is an interesting subject for future work.

6.2 SLA-driven reconfiguration

Currently, we assume that the number and placement of storage nodes is outside of Pileus's control. However, given knowledge of the SLAs being used by various clients, the system could make reasonable re-configuration decisions. For example, Pileus might automatically move the primary to a different datacenter in order to maximize the utility delivered to its clients. If one client has stringent latency requirements but loose consistency needs, a new secondary storage node could be placed nearby. Similarly, the rate at which updated data objects are propagated from the primary to secondary nodes could be adjusted based on the clients' desired consistency and proximity. We are currently investigating SLA-driven reconfiguration.

6.3 Parallel Gets

The current system sends each Get operation to a single node based on utility estimates. This policy minimizes client costs when storage providers charge for each operation. However, clients could receive more rapid responses and more up-to-date data when sending a Get in parallel to two or more nodes that are predicted to provide roughly the same service, particularly in cases where changing conditions lead to poor utility estimates. Existing methods for computing expected utilities could be used in a cost-benefit analysis to explore multi-node selection schemes.

6.4 Multi-site Puts

Our current implementations perform Put operations at a single primary site, i.e. a cluster of storage nodes within a datacenter. Generally, the cost of Put operations can be traded off against the cost of strongly consistent Get operations. If the system synchronously sends Puts to a larger collection of primary nodes, possibly nodes that are replicated across datacenters or even across regions, the expected latency of strong Gets is reduced (and the availability of such operations increases). A wider distribution of primary nodes can positively affect Gets with other consistency choices as well, except for eventual consistency. A thorough study of these Put/Get trade-offs remains future work.

7 Related work

The design of Pileus adopts and extends prior work on cloud storage systems, variable consistency, and service level agreements. However, we are not aware of other systems that combine consistency guarantees with latency targets as part of a storage service SLA.

Numerous cloud storage systems have been designed with a variety of data models, read and write operations, replication protocols, consistency, and partitioning schemes. Some key-value store probably exists with every imaginable combination of features and occupies every point in the space of consistency, scalability, availability, cost, and performance trade-offs. These include Dynamo [20], SimpleDB [5], BigTable [16], PNUTS [17][35], Cassandra [27], Windows Azure [14], Spanner [19], and many more [15]. Pileus borrows from many of these systems its simple Get/Put interface, key-range partitioning, geo-replication, and primary-update model.

Researchers have observed the need for more flexible storage designs that permit tradeoffs between consistency and availability [33]. Some cloud storage systems offer both strongly consistent and eventually consistent read operations [22][43][17][2][8], and papers have suggested switching between these options based on application classes [26][45]. Studies have shown that even eventually consistent systems frequently deliver strongly consistent data [11][44]. Researchers have proposed consistency models with guarantees that lie between these two extremes, such as session guarantees [36], continuous consistency [3][9][46], RedBlue consistency [29], and causal consistency [30], and many have been shown to be useful in diverse applications [10][36][42][23][34]. However, very few of these are being used in current systems. To the best of our knowledge, Pileus is the first cloud storage system to offer a broad choice of consistency guarantees, and allow the requested consistency to vary for each Get even when accessing the same data.

Service level agreements are an integral part of cloud services, including storage and networking. But such SLAs mainly focus on performance metrics and availability. For example, a typical SLA for an Amazon service guarantees "a response within 300ms for 99.9% of its requests" [20]. Others have suggested including consistency in SLAs [7] and developed algorithms for checking consistency [6][21], but Pileus is the first system to actually support consistency-based SLAs.

8 Conclusions

The Pileus storage system's main contribution is support for consistency-based SLAs that allow developers to declaratively specify their needs using a choice of consistency guarantees coupled with latency targets. Get operations access data that is partitioned and replicated among servers in all parts of the world while conforming to such SLAs. Consistency-based SLAs allow applications that were written to tolerate eventual consistency, as are many cloud applications today, to benefit from increased consistency when the performance cost is not excessive. When conditions are favorable, such as when the application is running in the same datacenter as up-to-date replicas, Pileus is able to deliver ideal consistency and latency to the application, and when conditions are less favorable, such as when nodes fail or become overloaded or clients are far from their frequently accessed data, the application's SLA indicates how best to adapt.

Pileus cleanly separates the mechanism for finding versions of a data object with the desired consistency from the techniques for selecting servers that can meet an SLA given existing network and server characteristics. Timestamp mechanisms support a broad range of consistencies while monitoring permits clients to independently select subSLAs that maximize the utility delivered to their local applications.

While our early experimental results show that consistency-based SLAs can indeed improve application-specific levels of service, further studies are needed to explore the full space of practical SLAs. Future work will investigate additional schemes for monitoring/predicting the lag and performance of storage nodes, expanding the choice of consistency guarantees, and automatically configuring services based on their applications' SLAs.

9 Acknowledgements

For their feedback at various stages of our research, we thank our colleagues including Paul Barham, Phil Bernstein, Michael Isard, Rebecca Isaacs, Jean-Philippe Martin, Rama Ramasubramanian, Masoud Saeida Ardekani, Mike Schroeder, Chandu Thekkath, and Yuan Yu. For letting us play with their client library, we thank our friends in Windows Azure, especially Brad Calder and Jai Haridas.

10 References

[1] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. *Proceedings USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2010.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems* 27 (3), November 2009.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems* 15(3):359-384, September 1990.

[4] Amazon Web Services. Amazon DynamoDB Pricing. http://aws.amazon.com/dynamodb/pricing/

[5] Amazon Web Services. Amazon SimpleDB. http://aws.amazon.com/simpledb/

[6] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What consistency does your key-value store actually provide? *Proceedings USENIX Workshop on Hot Topics in Systems Dependability*, 2010.

[7] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. *Proceedings Conference on Innovative Data Systems Research (CIDR)*, January 2009.

[8] J. Baker, C. Bond, J. C. Corbett, JJ Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. *Proceedings Conference on Innovative Data Systems Research (CIDR),* January 2011.

[9] D. Barbara-Milla and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *VLDB Journal* 3(3):325-353, 1994.

[10] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula and J. Zheng. PRACTI replication. *Proceedings USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[11] D. Bermbach and S. Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. *Proceedings Workshop on Middleware for Service Oriented Computing*, December 2011.

[12] E. Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, February 2012.

[13] N. Bonvin, T. G. Papaioannou, and K. Aberer. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. *Proceedings ACM Symposium on Cloud Computing (SoCC)*, June 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. Fahim ul Haq, M. Ikram ul 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. *Proceedings ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[15] R. Cattell, Scalable SQL and NoSQL data stores, *ACM SIGMOD Record* 39(4), December 2010.

[16] 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. *ACM Transactions on Computer Systems* 26(2), June 2008.

[17] B. 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. *Proceedings International Conference on Very Large Data Bases (VLDB)*, August 2008.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. *Proceedings ACM Symposium on Cloud Computing (SoCC)*, June 2010.

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

[20] 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. *Proceedings ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[21] W. Golab , X. Li , and M. A. Shah. Analyzing consistency properties for fun and profit. *Proceedings ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, June 2011.

[22] Google. Read consistency & deadlines: more control of your datastore. *Google App Engine Blog*, March 29, 2010. http://googleappengine.blogspot.com/2010/03/read-consistency-deadlines-more-control.html

[23] H. Guo, P.-Å. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: How to say "good enough" in SQL. *Proceedings ACM International Conference on Management of Data (SIGMOD)*, June 2004.

[24] J. Hamilton. The cost of latency. *Perspectives Blog*, October 31, 2009. http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx

[25] S. Kadambi, J. Chen, B. F. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam, and H. Garcia-Molina. Where in the world is my data? *Proceedings International Conference on Very Large Data Bases (VLDB)*, August 2011.

[26] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proceedings International Conference on Very Large Data Bases (VLDB)*, August 2009.

[27] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review* 44(2), April 2010.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7), July 1978.

[29] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguica, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. *Proceedings USENIX Symposium on Operating System Design and Implementation (OSDI)*, October 2012.

[30] 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. *Proceedings ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[31] R. Minnear. Latency: The Achilles heel of cloud computing. *Cloud Computing Journal*, March 9, 2011.

[32] Oracle. Oracle NoSQL Database. An Oracle White Paper, September 2011. http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf

[33] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. *Proceedings Workshop on Management of Big Data Systems*, September 2012.

[34] M. Serafini and F. Junqueira. Weak consistency as a last resort. *Proceedings ACM Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, July 2010.

[35] A. E. Silberstein, R. Sears, W. Zhou, and B. F. Cooper. A batch of PNUTS: Experiences connecting cloud batch and serving systems. *Proceedings International Conference on Management of Data (SIGMOD)*, June 2011.

[36] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. *Proceedings IEEE International Conference on Parallel and Distributed Information Systems*, 1994.

[37] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[38] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, and M. K. Aguilera. Transactions with consistency choices on geo-replicated cloud storage. *Microsoft Technical Report MSR-TR-2013-82*, September 2013.

[39] D. Terry. Replicated data consistency explained through baseball, *Microsoft Technical Report MSR-TR-2011-137*, October 2011. To appear in *Communications of the ACM*, December 2013.

[40] J. F. Van Der Zwet. Layers of latency: Cloud complexity and performance. *Wired*, September 18, 2012.

[41] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. *Proceedings USENIX Symposium on Operating System Design and Implementation (OSDI),* December 2004.

[42] W. Vogels. Eventually consistent. *Communications of the ACM*, January 2009.

[43] W. Vogels. Choosing consistency. *All Things Distributed*, February 24, 2010. http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html

[44] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. *Proceedings Conference on Innovative Data Systems Research (CIDR)*, January 2011.

[45] X. Wang, S. Yang, S. Wang, X. Niu, and J. Xu. An application-based adaptive replica consistency for cloud storage. *Proceedings IEEE International Conference on Grid and Cloud Computing*, November 2010.

[46] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems* 20(3):239-282, August 2002.