Consistency-based Service Level Agreements for Cloud Storage

Doug Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos Aguilera
Microsoft Research Silicon Valley

Hussam Abu-Libdeh
Cornell University

Microsoft Research

Cornell University

Motivation

- Storage systems make consistency-latency tradeoffs
- Eventual consistency is not always sufficient
  Strong consistency is not always efficient
- Some data stores provide options
  `StrongRead`, `WeakRead`, `ConsistentRead`, `ReadCritical`, `ReadLatest`, `ReadAny` ...

Problem with multi-consistency stores

Devs are forced to make consistency-latency tradeoffs at development time with insufficient information!

Our Contributions

We built *Pileus*, a key-value store with salient properties:

Simple `put`/`get` API

Developers do not need to choose from multiple read/write operations at development time

Multiple consistency guarantees

`get(Key)` consistency guarantees	
Strong	Return the value of the last preceeding `put(Key)`
Eventual	Return the value of any previous `put(Key)`
Read-my-writes	Return the value of the last `put(Key)` in the same session
Monotonic	Return a value not older than last `get(Key)` in this session
Bounded (t)	Return a `put(Key)` value that is stale by at most t seconds
Causal	Return the value of the last `put(Key)` that causally precedes the `get(Key)`

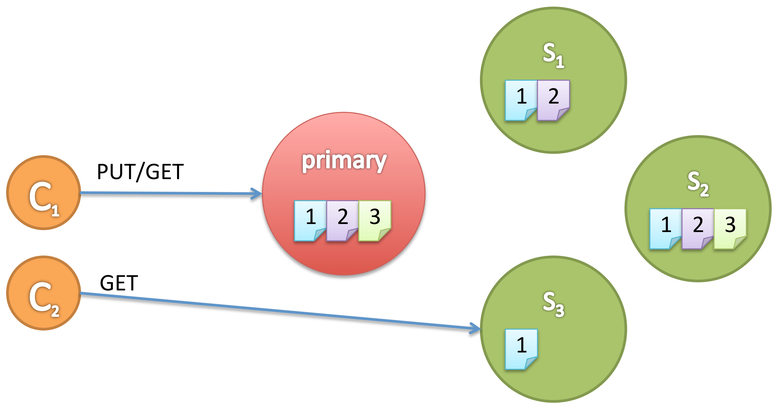
Declarative consistency/latency requirements

Consistency and latency requirements are specified in a service level agreement and enforced at runtime

Supporting Multiple Consistency Guarantees

Data Replication in Pileus

– Primary receives and orders all write requests
– Key-value pairs are timestamped according to write-order
– Primary asynchronously propagates writes to secondaries
– Secondaries apply updates according to timestamp order
– Keys can be partitioned across multiple primary/secondary groups



Consistency Guarantees with Pileus

– Replicas maintain highest timestamp of an applied update
– Clients maintain timestamp of last `put`/`get` request
– Clients route `get` requests based on local timestamps, previous session operations, and desired consistency guarantee

Guarantee	**Timestamp at replica receiveing** `get(Key)`
Strong	*(Primary only)*
Eventual	*(Any replica)*
Read-my-writes	$\geq$ timestamp of last `put` of same key in current session
Monotonic	$\geq$ timestamp of last `get` of same key in current session
Bounded (t)	$\geq$ current time - time bound
Causal	$\geq$ timestamp of last `get` of *any* key in current session

More info?

Consistency-based SLAs

Goal:
Capture developer's consistency/latency preferences and make best effort at satisfying them.

Expressing SLAs:

- Ordered list of consistency, latency bound, and utilitly triples
- `get` requests return data with information about the delivered consistency and latency

Examples:
Shopping cart application:

> *"Answer all requests with a 300 msec latency bound, but try to make responses consistent."*

⇓

Rank	Consistency	Latency	Utility
1	read-my-writes	300 msec	1.0
2	eventual	300 msec	0.5

Bound on latency, preference for consistency

Web application:

> *"I want a reply in under 150 msec 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."*

⇓

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

Prefer fast response, if not, slow but consistent

Enforcing SLAs:

- Clients enforce SLAs by monitoring storage replicas for operation latencies and highest timstamp at each replica and directing `get` requests accordingly
- Clients choose replicas that maximize expected utility
- If an SLA cannot be satisfied for a `get` request, then an error code will be returned. A catch-all consistency/latency requirement can be added to the end of the SLA to ensure that all requests are satisfied