Robust Data Sharing with Key-Value Stores
Replication over multiple cloud storage providers

Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic and Ido Zachevsky
Motivation
Cloud storage is popular.

Infrastructure for various services.
Data stored in blobs – **Binary Large OBjects**, referenced by name.
The promise: sharing, elasticity and reliability. But single cloud service providers can and do fail.
Solution: replication among multiple providers
Interlude
Classical Replication
The goal: Provide read/write semantics:
\( \nu \leftrightarrow \text{read()}, \text{write}(\nu) \)
**The goal**: Provide read/write semantics: 
\[ v \leftarrow \text{read}(), \text{write}(v) \]

- Many (\(\infty\)) clients that may crash
- A finite set of base objects, of which a minority may crash
- Asynchronous system
- Wait-free implementation
Step 1: read to and write from a majority.
Step 1: read to and write from *a majority*. Step 2: use *versioning*.
The classical solution [1]:

- **Versioning:**
  Base object stores one versioned value.

- **Client write:**
  Get largest version from a majority.
  Put next version in a majority.

- **Client read:**
  Return value with largest version in a majority.

Required: Servers that ignore old versions.

Client 1
Client 2
Client 3
Base obj. a
Base obj. b
Base obj. c
Client 4

Write
Write
Write
Read
Challenge
Cloud storage providers’ common API

All export Key-Value Store (KVS), A.k.a. associative array.

- put(k, v)  
  Associate value v with key k
- get(k)  
  Return the value associated with key k
- list()  
  Return keys
- Remove(k)  
  Dissociate the value associated with k

ABD doesn’t work in the clouds! It requires servers that ignore old versions
**Goal:** Provide regular register semantics:

\[ v \leftarrow \text{read}(), \text{write}(v) \]
Many ($\infty$) clients that may crash.

A finite set of (atomic) KVS base objects, of which a minority may crash.

Asynchronous system.

Wait-free implementation.

Readers do not put values in KVSs.
Take 1: Store in one key the register version and value.

Old-New overwrite problem:

Client 1
Client 2
Client 3
KVS a
KVS b
KVS c
Client 4

Write
Write
Write

Error! should return

Read
Take 2: Store version in KVS key and register value in KVS value. GC obsolete values.

Garbage collection racing problem:

function regRead():
    for each KVS i in a majority:
        until success:
            lst = list_i()
            val = get_i(max(lst))
    return the val of max version

Other clients write newer versions and GC max(lst). Function never returns.
Our Algorithm
On write and read access two keys:

- **Eternal key**: never GCed. Stores version and value.
- **Temporary key**: Key is version, value is register value. GCed when obsolete.

```
Eternal: <5, data>
4: data
5: data
```
**Write procedure (value):**

for each KVS i in a majority:
  list to get highest version

writerVer = highest found + 1

for each KVS i in a majority:
  put eternal key <eternal, <writeVer, value>>
  put temporary key <writeVer, value>

**Read procedure():**

for each KVS i in a majority:
  list to get highest version
  until success:
    list and get highest temporary key
    get eternal key
return value with largest version
Eternal: (0, ->)

put(v1)
Eternal: 
\langle 0, -\rangle

Eternal: 
\langle 0, -\rangle

Eternal: 
\langle 0, -\rangle

put(v1)
Eternal: <1, v1>

Eternal: <1, v1>

Eternal: <0, ->

put(v1)
Eternal: \(<1,v1>\)

1: v1

Eternal: \(<1,v1>\)

1: v1

Eternal: \(<0,->\)

put(v2)
Eternal: \{<1, v1>, <2, v2>, <2, v2>\}

put(v2)
Eternal: <1,v1>  
1: v1

Eternal: <2,v2>  
1: v1  
2: v2

Eternal: <2,v2>

put(v2)
Eternal: <1,v1>
1: v1

Eternal: <2,v2>
1: v1
2: v2

Eternal: <2,v2>
2: V2

put(v3)
Eternal: 
<1,v1>
1: v1

Eternal: 
<2,v2>
1: v1
2: v2

put(v3)

Eternal: 
<2,v2>
2: V2
Eternal: \langle 1, v1 \rangle

1: v1

Eternal: \langle 2, v2 \rangle

1: v1

2: v2

put(v3)
Eternal: \langle 3, v3 \rangle

1: v1

put(v3)

Eternal: \langle 3, v3 \rangle

2: v2

Eternal: \langle 2, v2 \rangle

2: V2
Eternal: <1,v1>
1: v1
3: v3

Eternal: <2,v2>
2: v2
3: v3

Eternal: <2,v2>
2: V2

put(v3)
Eternal: \(<1, v_1>\)

1: v_1

3: v_3

---

Eternal: \(<2, v_2>\)

2: v_2

3: v_3

---

Eternal: \(<2, v_2>\)

2: v_2

3: v_3

---

read()
Correctness

• Write returns in finite time.
  Trivial.
• Read returns a legitimate value.
  Follows from correctness definition.
• Read returns in finite time.
  1. Writes stop, reads temporary, or
  2. Concurrent writes, reads eternal.
Garbage Collection

Write procedure (value):
1. Garbage—collect old versions
2. put eternal key
3. put temporary key

- 3x space complexity minimum.
- O(# concurrent writes) space complexity max.

The key: We must GC before writing.
Simulation and Implementation
Implementation vs. simulation

The graph compares latency distributions for real and simulated operations.

- **Write latency, real**
- **Write latency, simulated**
- **Read latency, real**
- **Read latency, simulated**

The x-axis represents latency in milliseconds (ms), and the y-axis represents frequency.
Read Duration

![Graph showing the relationship between read duration and writer latency.](image)
Space Complexity

Maximal space usage vs. Maximal number of concurrent writers.
Summary

- Multi-cloud storage replication is important.
- Classical solutions do not work for KVSs.
- New algorithm:
  - With new concept: eternal and temporary keys.
  - Careful garbage collection:
    - 3x space complexity minimum.
    - $O(#\text{concurrent})$ maximum.
- Minimal overhead in common scenarios.

- Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Marko Vukolic, Robust Data Sharing with Key-Value Stores, DSN’12.
Answers to good questions
Garbage Collection (2)

Write procedure (value):
for each KVS i in a majority:
    list to get highest version
    Garbage-collect older versions
    writerVer = highest found + 1
for each KVS i in a majority:
    put temporary key <writeVer, value>
    put eternal key <eternal, <writeVer, value>>
    Garbage-collect previous version

2x space complexity in synchronous runs.