Making RAMCloud Writes Even Faster
(Bring Asynchrony to Distributed Systems)

Seo Jin Park
John Ousterhout

Stanford University
Overview

- **Goal:** make writes asynchronous with consistency.
- **Approach:** rely on client
  - Server returns before making writes durable
  - If a server crashes, client retries previous writes
- **Behavior is still consistent:** linearizable if client is alive
- **Anticipated benefits:**
  - Write latency: 15 $\mu$s $\rightarrow$ 6 $\mu$s (even with geo-replication)
  - Lower tail latency
  - Write Throughput: 2-3x higher
- **Some applications don’t need durability of last 10ms**
Bring Asynchrony to RAMCloud

RAMCloud provides linearizability (current)

- Strongest form of consistency for concurrent systems
- Write is blocked while replication
- Write: 15 µs vs. Read: 5 µs
- Wastes cycles in server

Make durability for write happen asynchronously

- Should we give up consistency?

asynchrony = weak consistency??
Consistency in Performant Systems

• **Eventual consistency** is popular in distributed storage
  - Writes are asynchronously durable for best performance
  - Ex) Redis cluster, TAO, MySQL replication

• **Problem**: difficult to reason about the state of system
  - Clients may read different values.
  - Don’t know when updates will be applied
    - Cannot check update was durably queued
    - Write may get applied long after

• **New model**: linearizable unless client crashes

  => Similar to (stronger) **asynchronous file system**
API

- `asyncWrite(tableId, key, value) → value, version`
  ... asyncCondWrite(), asyncIncrement() etc

- `sync() → NULL  <waits all updates are durable>

Possible APIs [Feedback requested: are they useful?]
- `rpc.sync() → NULL  <waits 1 update is durable>`
- `sync(CallbackFunc) → NULL`

Example

```c
ramcloud.asyncWrite(1, “Bob”, “2”);
ramcloud.asyncWrite(1, “Bill”, “2”);
ramcloud.sync();
printf(“Updated Bob and Bill”);
```
New Consistency Model

Durability for write happens asynchronously
Behavior is still consistent

1. All reads are consistent
   - Reads are blocked until data become durable

2. Writes are linearizable unless client crash
   - When a server crashes, client retries previously returned writes.

- Write is lost only if both client and server crash
- Client may wait for durability before externalization
- Conditional write is still consistent and possible
In server crash, client retries previously returned writes

- **Goal:** Restore the same state as before server crash

- **Issue 1:** Retry may re-execute the same write request
  - If a server crash, a write may or may not be recovered.
  - Client retries operations that are not yet known to be durable.
  - The retried write may get re-executed, which overwrites and reverts subsequent updates by other clients

- **Issue 2:** Retries from different clients may be out of order
  - End state of system will be different
  - Previously succeeded conditional write may fail (client sees inconsistency)
Issue 1: Retry may re-execute

- RIFL (Reusable Infrastructure for Linearizability) [SOSP15] will let server ignore already completed writes
Issue 2: Out of Order Retries

- Retries from clients may arrive with different order from original execution => linearizability in danger!

- Option 1) Use object version to decide final winner
  - Write: okay
  - Conditional write: can be handled specially.
  - Append? Not possible.

- Option 2) Allow only 1 not-replicated write: overwrites wait for durable
  - Any deterministic operations are okay.
  - Weakness: continuously overwritten object can be bad.

Feedback requested:
Is it common and real problem?
**Issue 2: Out of Order Retries**

```
if x.version == 2
  then x.value = 20

if x.version == 2
  then x.value = 20

(Always) success
```

- **Client**
- **Master**
- **Backups**
- **Recovery Master**
- **Crash Recovery**

```
X.val: 1
X.ver: 2
```

```
X.val: 20
X.ver: 3
```

```
X.val: 1
X.ver: 2
```

```
X.val: 1
X.ver: 2
```

```
X.val: 20
X.ver: 3
```

```
X.val: 1
X.ver: 2
```
Anticipated Benefits

- Reduces RAMCloud write latency
- Completely decouples write latency and replication latency
  - Consistent *geo-replication* becomes practical
  - Reduced *tail latency*: not affected by 3 backup servers
- More efficient threading model in servers
  - No need to spin wait for replication
  - Dedicated replication thread is possible
  - Improves *write throughput* of RAMCloud *2-3x*
Possible Applications?

1. Don’t care about durability
   - Durability of last 10ms may not be important
   - Ex) Real-time doc sharing: user cannot distinguish from typo

2. Split of update / validate clients
   - End-user can check write was failed. If failed, retry.
   - No surprise resurrection! Validation by read is possible.
   - Ex) Purchase item, redirect to order confirmation page, which is rendered by different web server. Human notices and retries.

3. Many updates before externalization
   - Simply sync() before externalizing the success of writes.
   - Any experiences on this?
Questions

- **Applications?**
  - How does current web applications use no-sql DB?

- **How useful is ordering guarantee?**
  - Is it important to have some ordering for durability?

- **Callback based API?**
  - Is a single final response to request the only externalization?

Challenges

- **Client-side threading model for accurate timer**
Conclusion

- Rely on client retry if server crash → strong consistency with asynchronously durable writes
- Decoupling durability from critical path can improves performance (latency ↓, throughput ↑)
- RIFL (Reusable Infrastructure for Linearizability) eases design and reasoning of consistency