A locking service based on Cassandra’s light-weight transactions for MUSIC

Bharath Balasubramanian

MUSIC locking service

- MUSIC maintains client state in Cassandra but requires a locking service to provide stronger guarantees over a key — entry consistency.

- Currently locking service implemented using Zookeeper's sequentially consistent (strictly ordered writes) file system where each write is done using distributed consensus (specifically, RAFT) that allows for atomic writes

Listing 1: Example use of core MUSIC abstractions.

```
lockRef = createLockRef(key);
while(acquireLock(lockRef, key)!=true)
        skip;
// enter critical section
v1=criticalGet(lockRef, key);
//v1 is guaranteed to be the latest value of the
    key
v2=v1+1;
criticalPut(lockRef, key, v2);
releaseLock(lockRef);
//exit critical section
```

Current Zookeeper-based Solution

- lockRef = createLockRef (key): (i) Atomically creates the “Key” node if it does not already exist. (ii) Atomically creates a new child node with a new unique number and returns that as the lock reference.

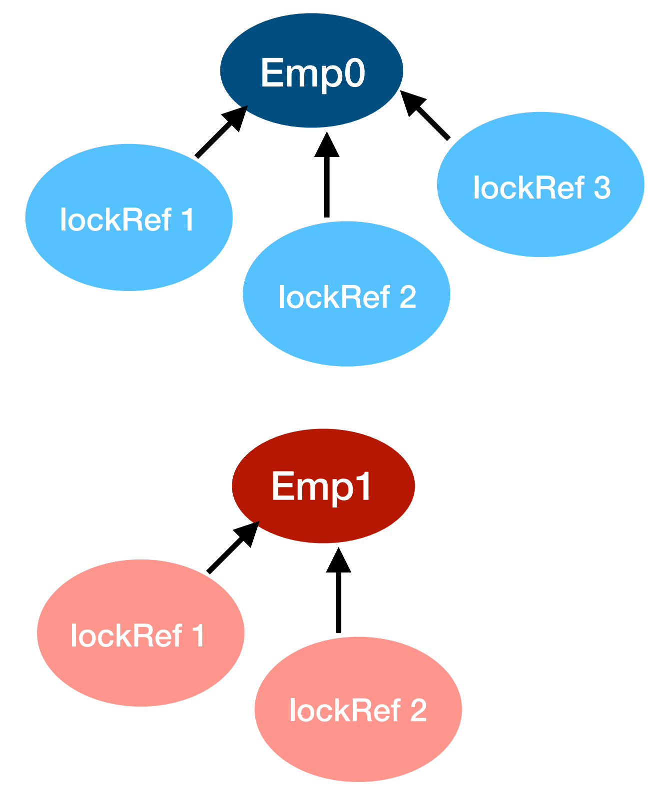
- Boolean result = acquireLock (lockRef, key): (i) Retrieves all the children of key with a non-atomic read (MUSIC algorithms ensure this is sufficient). (ii) Sorts all the children and returns true if lockReference is the youngest.

- releaseLock (lockReference): Atomically deletes the child node corresponding to lockReference from the key if it exists.

Cassandra State

Employee Name	Salary
Emp0	5000
Emp1	2000
Emp3	1000
Emp4	6000

Zookeeper State

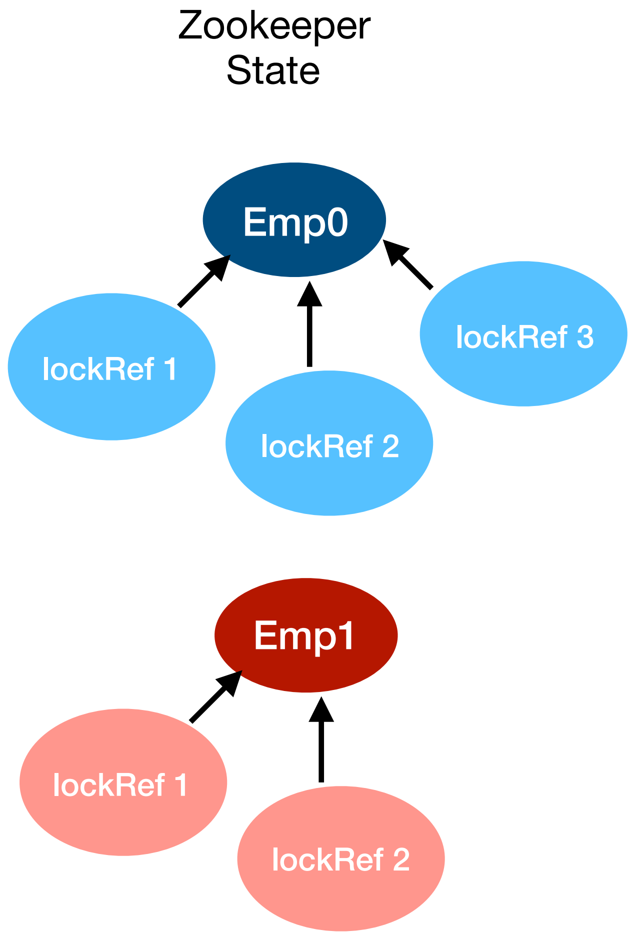


Main Problems with this solution

OPs Tooling: Requires the MUSIC team and Operations to deploy and manage two completely independent tools, especially Zookeeper which is relatively less trusted.

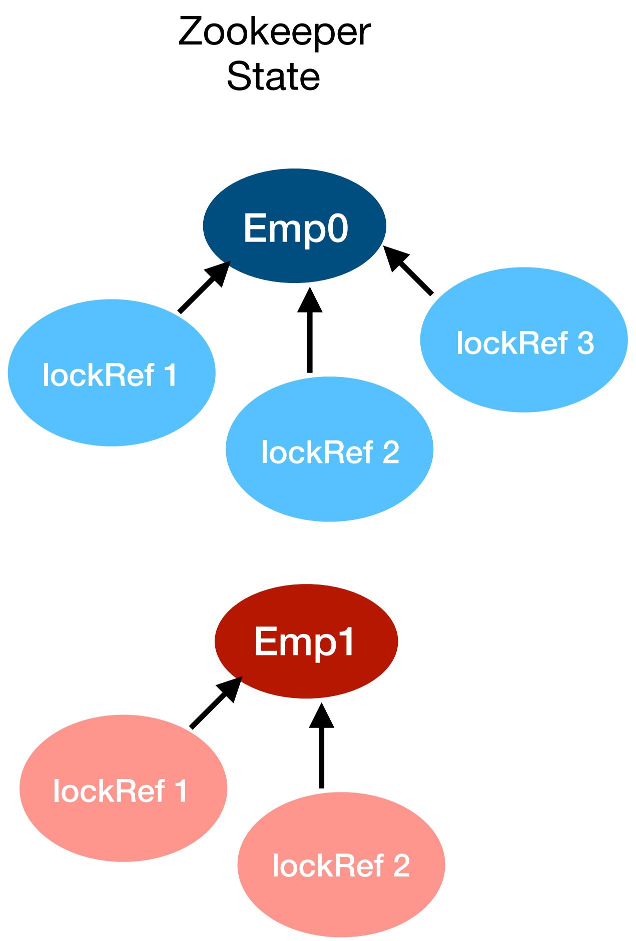
One ring to rule them all: Zookeeper guarantees that ALL writes are atomic and hence ordered — this requires that there is one global consensus ring. E.g. Adding a child to Emp0 has to be ordered behind some operation to Emp1 despite this being utterly unnecessary. Has performance and fault-tolerance implications.

No sharding: All data is replicated on all nodes leading to the standard space, scale-out issues of a non sharded system.



Other issues

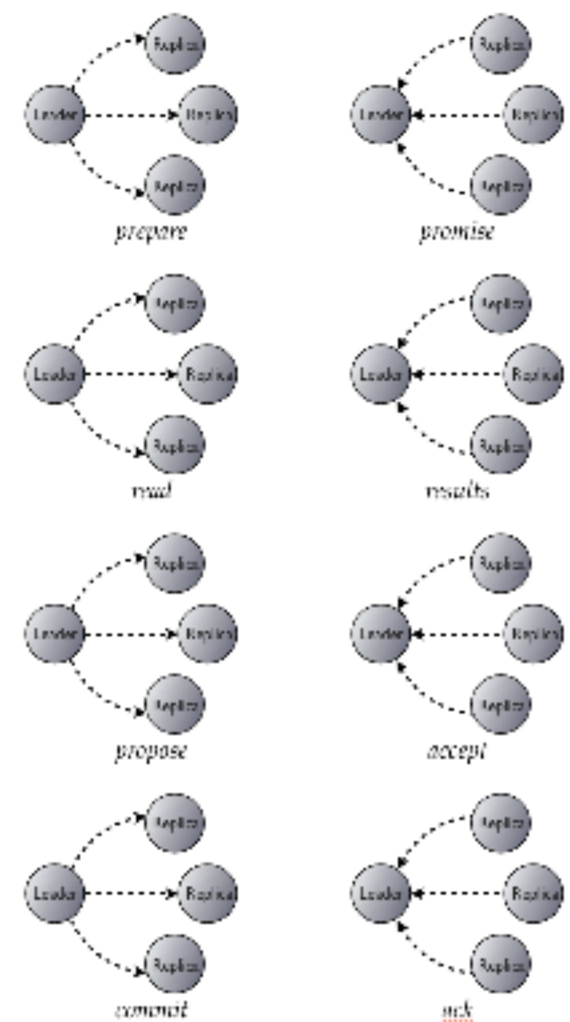
- Zookeeper is not meant to store a huge number of nodes — however, in use-cases like Conductor every row might have a “Key” node created for it. Could be in the order of thousands.
  - Hard to automatically garbage collect childless “Key” lock objects — might have consistency issues and that necessitates hand written clean up scripts in production.
- The sorting for an acquire lock could also be expensive if there are many clients waiting for a lock — maintaining a sorted queue in a sequentially-consistent write store is hard.
- All problems across last two slides exist with the Zookeeper cousins like etcd, Consul etc. — in general sequentially consistent stores.



Key Question

Can we build a locking service using Cassandra's light-weight transactions (LWT)?

- Atomic insert if it does not exist of a row (CAI)
- Atomic delete if it exists of a row (CAD)
- Atomic update if condition matches



Internally maintains paxos group per partition (e.g. key) and each of these operations use the following rounds (4 round trips)

A minor digression: does Cassandra's LWTs render MUSIC irrelevant?

NO.

LWTs != entry consistency.

- LWTs have been around for sometime (more than 5 years at least) — in fact they inspired the design of MUSIC. So they are not a surprise.
- Not even good enough for our current production use-cases
    - Conductor and portal (mdbc) needs atomic selects
    - Conductor needs more funky atomic inserts: if value is x do something and if it is y do something else — easy to build on MUSIC since locking is decoupled from the actions to the key.
    - SDN-C needs explicit locking for failover through Prom.
- Not sufficient for future use-cases: (1) multiple operations after acquiring a lock, (2) locks across multiple keys, (3) federation etc. all of which require explicit locking.

Cassandra-based Locking Service

- For every table in MUSIC that maintains client state, create a lock table (key, UUID) that partitions according to key and sorts according to UUID.
- lockRef = createLockRef (key): (i) Create a unique time-based UUID for this key (ii) Use CAI to insert into lock table and return the UUID as the lockRef.
- Boolean result = acquireLock (lockRef, key): (i) Simply perform a select of the top most row for the key in the lock table (since it is sorted) and return true if the lockRef UUID matches it.
- releaseLock (lockReference): Atomically deletes the row in lock table corresponding to the lockReference.

Cassandra State

Employees Table

Employee Name	Salary
Emp0	5000
Emp1	2000
Emp3	1000
Emp4	6000

Sorted lock_Employees Table

Key/Lock Name	Lock Reference
Emp0	1
Emp0	2
Emp0	3
Emp1	1
Emp1	2

Problems with the zk solution- addressed by the Cassa solution

- Zookeeper requires the MUSIC team and Operations to deploy and manage two completely independent tools. [Only one tool: Cassandra that has far more production exposure at scale — MUSIC can now be upstreamed into Cassandra]

- Zookeeper uses One ring to rule them all: [Cassandra maintains paxos rings at a per partition/key level]

- Zookeeper is not meant to store a huge number of nodes — however, in use-cases like Conductor every row might have a “Key” node created for it. Could be in the order of thousands. [Cassandra is built to manage millions of rows.]

    - Hard to automatically garbage collect childless “Key” lock objects — might have consistency issues and that necessitates hand written clean up scripts in production. [No such objects by definition — no lock references for a key implies no row in lock table.]

- The Zookeeper sorting for an acquire lock could also be expensive if there are many clients waiting for a lock — maintaining a sorted queue in a sequentially-consistent write store is hard. [Keys sorted according to time UUID: order of creation]

- Zookeeper is not a sharded file system — all data will be replicated on all nodes. [Lock table partitioned across nodes according to key. Hence all rows for same key in lock table will be replicated and sorted on same node]

Analysis

- Correctness - ***Hopefully*** should not be a problem: (1) both solutions essentially maintain a list of ordered lock references for each key where inserts and deletes to the list are performed atomically (2) Neither requires atomic reads. Hence semantically the same.

- Qualitative Performance -

Operation	Cassa Locking Cost	Zk Locking Cost	Comment
createLockRef	4 round trips, O (nlogn) local sorting where n = no of lock references for the key	4 round trips	While Zk is locally more efficient this operation is called only once per critical section
acquireLock	O(1) local operation	O (no. of lockRefs) local operation	This operation is typically called in a loop and hence the gains are crucial in Cassa!
releaseLock	4 round trips	4 round trips	

Conclusion

While we await the benchmarking results, the gains from a Cassandra locking service seem to far outweigh that of a Zk/etcd/Consul based locking service.