Cluster Coordination

Lukasz Antoniak

26.01.2022

Agenda

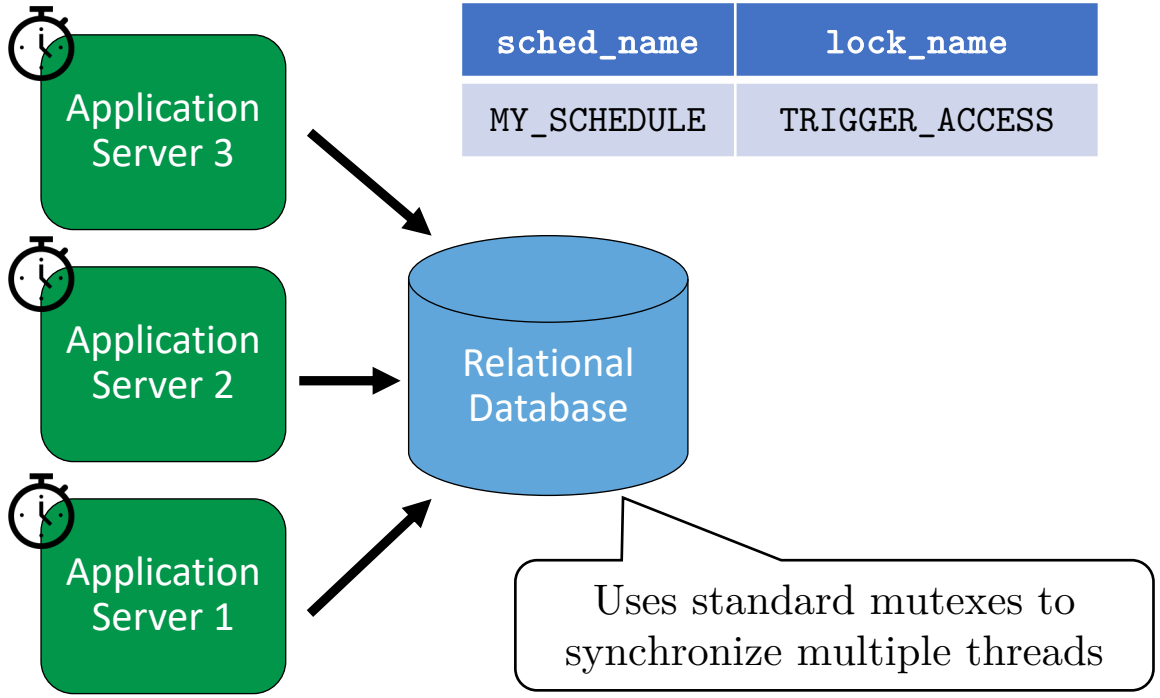
- The need of cluster-wide coordination
- RAFT consensus algorithm
- Short introduction to Etcd
- Implementation of distributed mutex
- Leader election design pattern
- Deployment requirements for strongly consistent distributed systems
- ACID properties in distributed system

Cluster Coordination with Central System

- Example of Java Quartz Scheduler that requires RDBMS backend
  - Multiple application servers can execute a periodic job (fault-tolerance). At any point in time, only one instance can execute the job
  - Implementation: https://github.com/quartz-scheduler/quartz/blob/master/quartz-core/src/main/java/org/quartz/impl/jdbcjobstore/StdRowLockSemaphore.java
  - ```
    SELECT *
      FROM qrtz_locks
     WHERE sched_name = ?
       AND lock_name = ?
       FOR UPDATE
    ```
  - ```
    INSERT INTO
    qrtz_locks(sched_name, lock_name)
    VALUES (?, ?)
    ```
  - RDBMS holds an exclusive lock whenever session updates or inserts the record. Concurrent write operations from other sessions to the same row wait, until first transaction completes

Table: qrtz_locks

sched_name	lock_name
MY_SCHEDULE	TRIGGER_ACCESS



Pessimistic and Optimistic locking

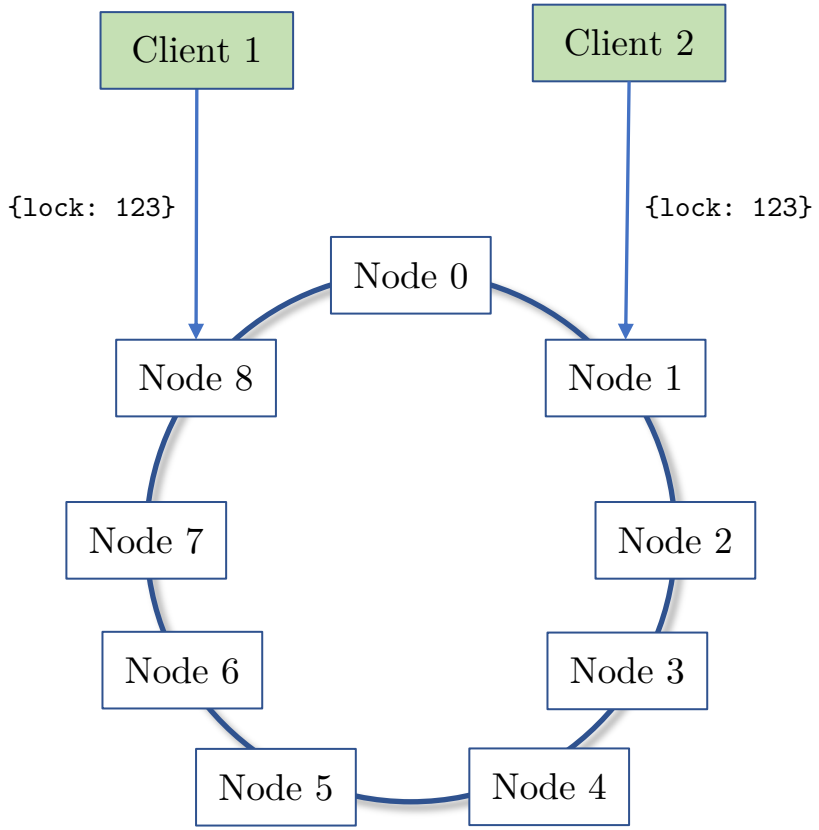
- Pessimistic locking
  - Explicitly lock the record at database level (e.g. using `SELECT ... FOR UPDATE` statement)
  - Lock is held as long as database transaction is not committed or rolled back
  - Use only for short duration of synchronized block
  - Extensive usage of connections from connection pool

- Optimistic locking
  - Introduce version column in the target table
  - Select version number of given row and preserve it until executing update
  - Update statement:

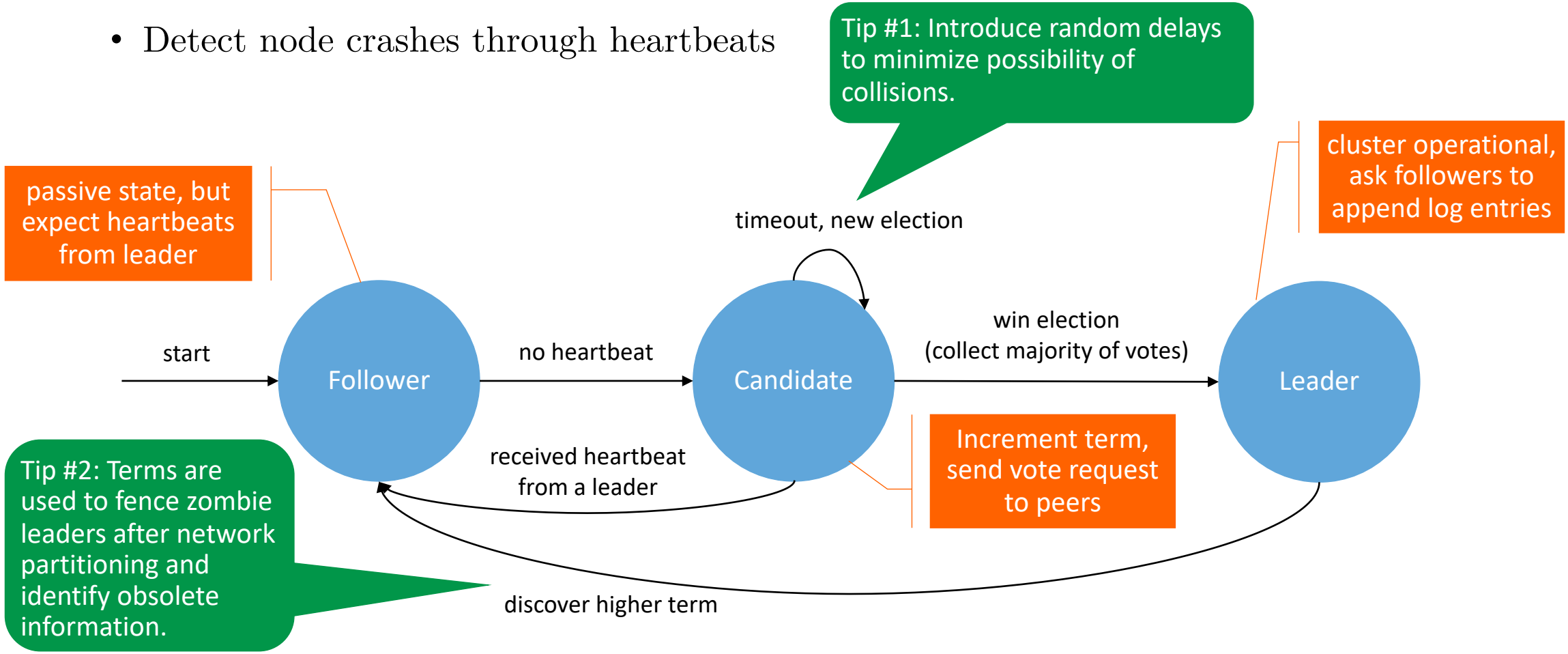
    ```
    UPDATE ... SET ..., version = version + 1 WHERE ... AND version = ?
    ```

  - JDBC update method returns number of affected rows
  - If affected rows equals zero, somebody modified the row concurrently

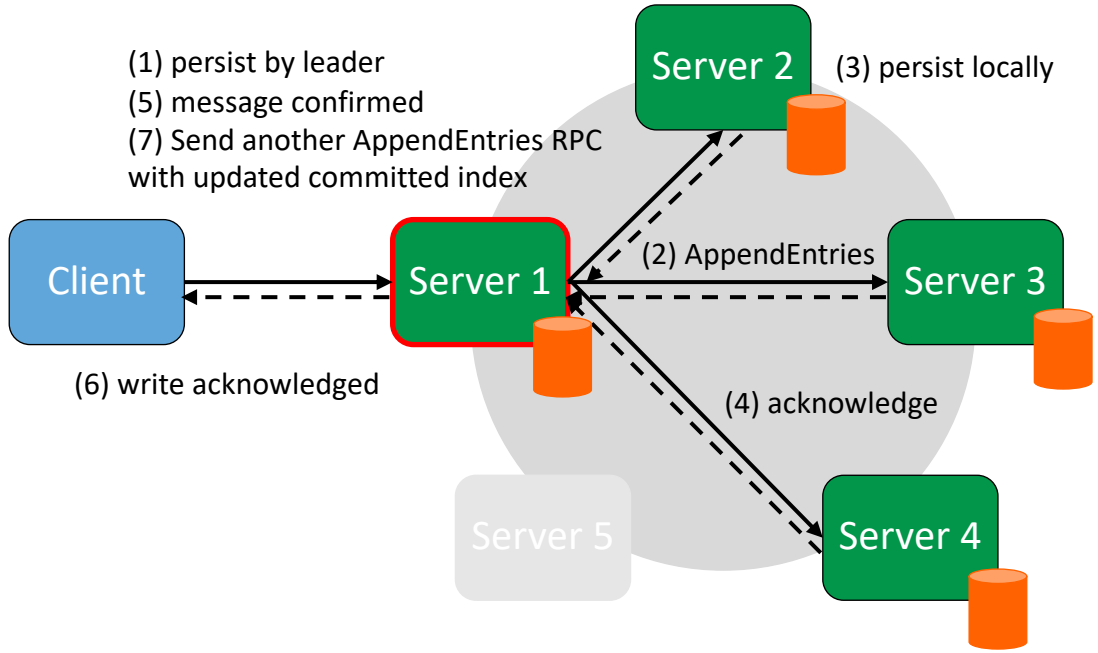
- Cluster Coordination requires a dedicated algorithm in distributed system
- Consensus algorithms
  - Paxos
  - ZAB (ZooKeeper Atomic Boardcast)
  - RAFT
- Use cases for cluster coordination
  - Synchronize access to resource shared amongst multiple remote nodes
  - Elect a leader node who distributes work across other nodes



- Leadership Election
  - Select always one server to act as a leader
  - Detect node crashes through heartbeats



- Log replication
  - Leader accepts commands from clients and appends data to its log
  - Leader replicates its log to other servers
  - Only server with up-to-date log can become a leader
- RAFT Visualization: https://raft.github.io
- Multi-RAFT



Introduction to Etcd

- Unix "/etc" distributed
- Strongly consistent key-value store
- High available and fault-tolerant
  - Gracefully handles leader election during network partitioning
  - Tolerates machine failures
  - Sacrify availability for consistency (CAP theorem)
- Requires majority of nodes to be running and connected
  - 3 node cluster, supports unavailability of 1 node
  - 5 node cluster, supports unavailability of 2 nodes
  - 2 node cluster does not make sense – both needs to be connected at all time
- Online play console: http://play.etcd.io/play

- Benchmarked at 50,000 writes and 140,000 reads per second on commodity servers
  - Environment: 3 machines of 8 vCPUs + 16GB RAM + 50GB SSD
  - Source: https://etcd.io/docs/v3.4/op-guide/performance/

Operation	# of keys	Key size	Value size	# of clients	Avg read QPS	Avg latency per request
Write	100,000	8B	256B	1000	50,104	20 ms
Read (quorum)	100,000	8B	256B	1000	141,578	5.5 ms

- Use cases
  - Key-value store for critical metadata shared between nodes in distributed system
  - Not big-data key-value store. Recommended maximum 8GB data size
  - Service Registry and Discovery for gRPC (competitor of Consul)
  - Coordination of distributed work
  - Centralized configuration management

- Write key-value pair
  - Command: `$ etcdctl put foo bar`
  - For composed (non-atomic) values, consider JSON format. Apache Kafka example:

```
[zk: localhost:2181(CONNECTED) 1] get /brokers/ids/0
{
    "endpoints": [ "PLAINTEXT://192.168.1.8:9092" ],
    "host": "192.168.1.8",
    "jmx_port": -1,
    "listener_security_protocol_map": { "PLAINTEXT": "PLAINTEXT" },
    "port": 9092,
    "timestamp": "1548914820968",
    "version": 4
}
```

- Read values of single key, or a range of keys
  - Single key lookup: `$ etcdctl get foo`
  - Range of keys ("foo3" excluding): `$ etcdctl get foo foo3`
  - All keys with "foo" prefix: `$ etcdctl get --prefix foo`
  - Read key at certain revision: `$ etcdctl get --prefix --rev=3 foo`
  - Etcd exposes previous versions of key-value pair to support "time travel queries"
  - Version numbers start with 1 and are monotonically increasing
  - Old versions of key are removed by background compaction process
- Delete values of single key, or a range of keys
  - Single key removal: `$ etcdctl del foo`
  - Range of keys: `$ etcdctl del foo foo3`
  - All keys with "foo" prefix: `$ etcdctl del --prefix foo`
  - Delete operation writes a tombstone marker. Data is removed during compaction process (similar to Apache Cassandra)

- Keys (one or many) can be attached to a lease, which defines their lifetime
  - Lease has a declared TTL
  - Lease can be periodically refreshed by client application to keep it alive
  - Once lease's TTL expires, all attached keys are removed
  - Create lease and attached it to a key:

```
$ etcdctl lease grant 60
lease 34585310baa0db06 granted with TTL(60s)
$ etcdctl put --lease=34585310baa0db06 foo bar
```

  - Explicitly revoke lease: `$ etcdctl lease revoke 34585310baa0db06`
  - Keep-alive lease: `$ etcdctl lease keep-alive 34585310baa0db06`
  - Get information about the lease and all associated keys:

```
$ etcdctl lease timetolive --keys 34585310baa0db06
```

- Watch updates to given key, or a range of keys
  - Continuous single key watch: `$ etcdctl watch foo`
  - Range of keys: `$ etcdctl watch foo foo3`
  - All keys with "foo" prefix: `$ etcdctl watch --prefix foo`
  - Watch changes from given revision: `$ etcdctl watch --prefix --rev=3 foo`
  - Return previous and most recent key value: `$ etcdctl watch --prev-kv foo`

Console 1:	Console 2:

`$ etcdctl put foo bar`
`OK`
 	
`$ etcdctl put foo bar2`
`OK`
	`$ etcdctl watch --prefix foo`
	`PUT`
	`foo`
	`bar`
	`PUT`
	`foo`
	`bar2`

- Mini-transaction support, multi-object compare-and-set
  - Atomic if-then-else block over key-value store
  - "If" clause can define comparison of multiple keys (absence, presence, value comparison, revision check)
  - When all comparisons succeed, Etcd applies all (one or many) "success requests"
  - When comparison fails, Etcd applies all "failed requests"

Console 1:

```
$ etcdctl txn --interactive
compares:
value("foo") = "bar2"

success requests (get, put, del):
put foo bar3

failure requests (get, put, del):
get foo

SUCCESS
OK
```

Console 2:

```
$ etcdctl txn --interactive
compares:
value("foo") = "bar2"

success requests (get, put, del):
put foo bar4

failure requests (get, put, del):
get foo

FAILURE

foo
bar3
```

- Service Discovery server:
  - On start-up, service provider requests a lease and creates background thread that will refresh it periodically
  - Service providers register themselves as "/${service-name}/${unique-instance-id}" key and associate it with lease. Value contains endpoint address, e.g. https://server1:8443
  - When service provider terminates gracefully, lease is removed explicitly by application and all associated keys deleted
  - When service provider terminates unexpected, associated keys are removed when lease TTL expires
- Service Discovery client:
  - On start-up, service consumers read all keys with prefix "/${service-name}/", and register watch to receive immediate notifications when service providers join and leave the cluster.
  - Service consumer maintains in-memory set of all currently available endpoints of given service

Distributed Lock Implementation

- Distributed lock implementation
  - Provide functions to obtain a lock (with thread blocking) and unlock
  - Only one node from distributed system can access the lock at a time
  - Try to implement fair algorithm, no starvation
  - In case of application failure, lock should not be held forever
- Source: https://github.com/etcd-io/etcd/blob/main/client/v3/concurrency/mutex.go

Distributed Lock Implementation

```
107  func (m *Mutex) tryAcquire(ctx context.Context) (*v3.TxnResponse, error) {
108          s := m.s
109          client := m.s.Client()
110
111          m.myKey = fmt.Sprintf("%s%x", m.pfx, s.Lease())
112          cmp := v3.Compare(v3.CreateRevision(m.myKey), "=", 0)
113          // put self in lock waiters via myKey; oldest waiter holds lock
114          put := v3.OpPut(m.myKey, "", v3.WithLease(s.Lease()))
115          // reuse key in case this session already holds the lock
116          get := v3.OpGet(m.myKey)
117          // fetch current holder to complete uncontended path with only one RPC
118          getOwner := v3.OpGet(m.pfx, v3.WithFirstCreate()...)
119          resp, err := client.Txn(ctx).If(cmp).Then(put, getOwner).Else(get, getOwner).Commit()
120          if err != nil {
121                  return nil, err
122          }
123          m.myRev = resp.Header.Revision
124          if !resp.Succeeded {
125                  m.myRev = resp.Responses[0].GetResponseRange().Kvs[0].CreateRevision
126          }
127          return resp, nil
128  }
```

"myKey" contains lock name (prefix) concatenated with lease ID, e.g. my-lock/14ab54

If the key does not exist, create one, and associate it with session lease

Get key with lock prefix and the oldest create revision

"myRev" contains global revision number, when my record was created

Distributed Lock Implementation

```
$ etcdctl get --prefix lock-1
lock-1/3f9c7c4ae273e704
lock-1/3f9c7c4ae273e708
lock-1/3f9c7c4ae273e70c
```

Lock acquired

Waiting list

Global Etcd revisions when key was created and last modified

Current, global Etcd revision

Local key revision

```
$ etcdctl lease timetolive --keys 3f9c7c4ae273e704
lease 3f9c7c4ae273e704 granted with TTL(10s), remaining(8s), attached keys([lock-1/3f9c7c4ae273e704])
$ etcdctl lease timetolive --keys 3f9c7c4ae273e708
lease 3f9c7c4ae273e708 granted with TTL(10s), remaining(9s), attached keys([lock-1/3f9c7c4ae273e708])
$ etcdctl lease timetolive --keys 3f9c7c4ae273e70c
lease 3f9c7c4ae273e70c granted with TTL(10s), remaining(9s), attached keys([lock-1/3f9c7c4ae273e70c])

$ etcdctl get lock-1/3f9c7c4ae273e704 --write-out=json
{"header":{"cluster_id":18396408571102523720,"member_id":11047135559872888732,"revision":14,"raft_term":8},"kv
s":[{"key":"bG9jay0xLzNmOWM3YzRhZTI3M2U3MDQ=","create_revision":12,"mod_revision":12,"version":1,"lease":45836
75181824993028}],"count":1}

$ etcdctl get lock-1/3f9c7c4ae273e708 --write-out=json
{"header":{"cluster_id":18396408571102523720,"member_id":11047135559872888732,"revision":14,"raft_term":8},"kv
s":[{"key":"bG9jay0xLzNmOWM3YzRhZTI3M2U3MDg=","create_revision":13,"mod_revision":13,"version":1,"lease":45836
75181824993032}],"count":1}

$ etcdctl get lock-1/3f9c7c4ae273e70c --write-out=json
{"header":{"cluster_id":18396408571102523720,"member_id":11047135559872888732,"revision":14,"raft_term":8},"kv
s":[{"key":"bG9jay0xLzNmOWM3YzRhZTI3M2U3MGM=","create_revision":14,"mod_revision":14,"version":1,"lease":45836
75181824993036}],"count":1}
```

Distributed Lock Implementation

```
71  func (m *Mutex) Lock(ctx context.Context) error {
72          resp, err := m.tryAcquire(ctx)
73          if err != nil {
74                  return err
75          }
76          // if no key on prefix / the minimum rev is key, already hold the lock
77          ownerKey := resp.Responses[1].GetResponseRange().Kvs
78          if len(ownerKey) == 0 || ownerKey[0].CreateRevision == m.myRev {
79                  m.hdr = resp.Header
80                  return nil
81          }
82          client := m.s.Client()
83          // wait for deletion revisions prior to myKey
84          // TODO: early termination if the session key is deleted before other session keys with smaller revisions.
85          _, werr := waitDeletes(ctx, client, m.pfx, m.myRev-1)
86          // release lock key if wait failed
87          if werr != nil {
88                  m.Unlock(client.Ctx())
89                  return werr
90          }
91
92          // make sure the session is not expired, and the owner key still exists.
93          gresp, werr := client.Get(ctx, m.myKey)
94          if werr != nil {
95                  m.Unlock(client.Ctx())
96                  return werr
97          }
98
99          if len(gresp.Kvs) == 0 { // is the session key lost?
100                 return ErrSessionExpired
101         }
102         m.hdr = gresp.Header
103
104         return nil
105 }
```

Try to acquire the lock

We are the owner of the lock.
Compare owner GET operation with our revision

Wait until the revision prior to our has been removed from oldest key with given prefix

Distributed Lock Implementation

```
48  // waitDeletes efficiently waits until all keys matching the prefix and no greater
49  // than the create revision.
50  func waitDeletes(ctx context.Context, client *v3.Client, pfx string, maxCreateRev int64) (*pb.ResponseHeader, error) {
51          getOpts := append(v3.WithLastCreate(), v3.WithMaxCreateRev(maxCreateRev))
52          for {
53                  resp, err := client.Get(ctx, pfx, getOpts...)
54                  if err != nil {
55                          return nil, err
56                  }
57                  if len(resp.Kvs) == 0 {
58                          return resp.Header, nil
59                  }
60                  lastKey := string(resp.Kvs[0].Key)
61                  if err = waitDelete(ctx, client, lastKey, resp.Header.Revision); err != nil {
62                          return nil, err
63                  }
64          }
65  }
```

Query for key with given prefix and most recent create revision, but not greater than revision from parameter

Wait for queried key to be removed

Distributed Lock Implementation

```
26  func waitDelete(ctx context.Context, client *v3.Client, key string, rev int64) error {
27          cctx, cancel := context.WithCancel(ctx)
28          defer cancel()
29
30          var wr v3.WatchResponse
31          wch := client.Watch(cctx, key, v3.WithRev(rev))
32          for wr = range wch {
33                  for _, ev := range wr.Events {
34                          if ev.Type == mvccpb.DELETE {
35                                  return nil
36                          }
37                  }
38          }
39          if err := wr.Err(); err != nil {
40                  return err
41          }
42          if err := ctx.Err(); err != nil {
43                  return err
44          }
45          return fmt.Errorf("lost watcher waiting for delete")
46  }
```

Watch for changes to given key, starting with certain revision

Break when DELETE event noticed, finished waiting

Never perform active waiting like:

```
while (true) {
   if (getKey(key) == null) {
      break;
   }
   Thread.sleep(1000);
}
```

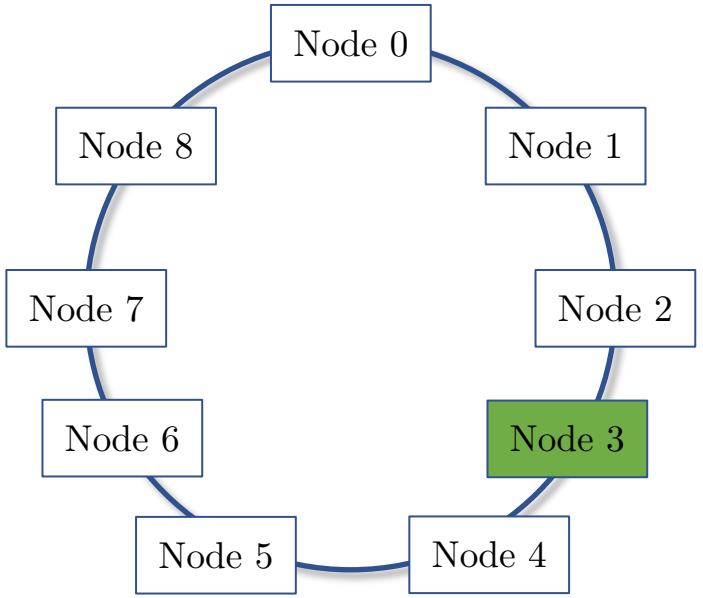
Distributed Lock Implementation

```
130  func (m *Mutex) Unlock(ctx context.Context) error {
131          client := m.s.Client()
132          if _, err := client.Delete(ctx, m.myKey); err != nil {
133                  return err
134          }
135          m.myKey = "\x00"
136          m.myRev = -1
137          return nil
138  }
```

Delete the key, potentially allowing other instance to acquire the lock

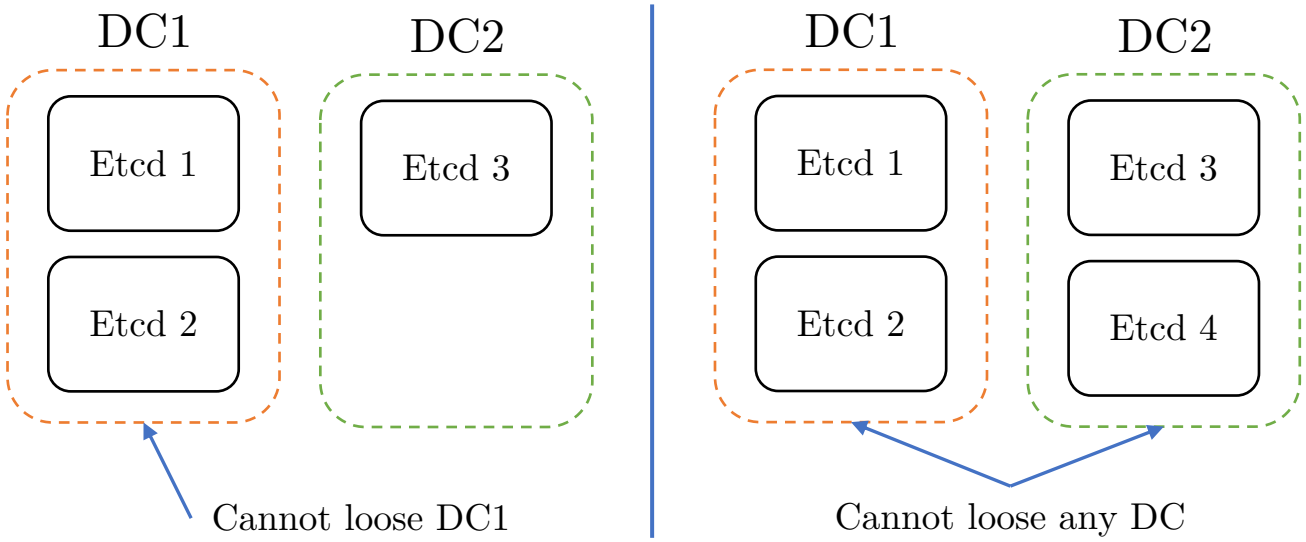
Leader Election Design Pattern

- Only one node form the cluster can become a leader at a time
- Every node from the cluster is eligible to become a leader
- Whenever leader terminates, new leader is elected immediately
- Tasks that require shared resource access are executed only by the leader
- Leader assigns and distributes work to other nodes
- Advantages
  - Cluster needs to run consensus once per election
  - For distributed lock, cluster needs to achieve consensus whenever any node tries to obtain the lock



Deployment Requirements

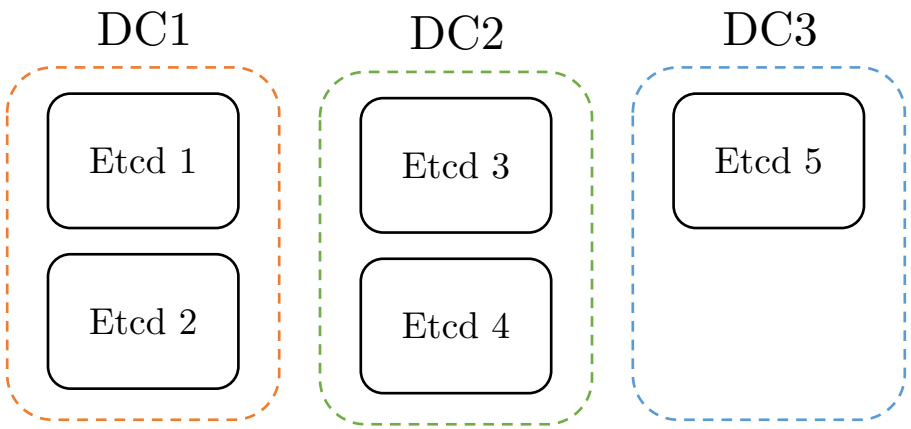
- Typical number of nodes in Etcd cluster: 3, 5, 7, 9
  - Higher number of nodes increases request latency, and possibility of split-votes
  - High availability and fault-tolerance achieved
- Nodes distributed across 3 availability-zones / data centres
  - In two data centres architecture, we cannot support lose of complete DC
  - In three data centre scenario, we can accommodate lose of complete DC



Cluster size	Majority	Failure Tolerance
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

Deployment Requirements

- Deploy quorum-based distributed systems across odd number of failable components (servers, availability-zones)



Deployment Requirements

- Distance between nodes
  - Heartbeat interval (--heartbeat-interval) – interval at which leader will notify follower that he is still a leader (default: 100 ms.). Configure around 0,5 – 1,5x round-trip time between DCs.
  - Election timeout (--election-timeout) – timeout for a node to operate without receiving heartbeat from leader until it becomes candidate (default: 1000 ms.). Configure at least 10x heartbeat interval.
  - Round-trip within AWS US: ~70 ms (https://www.cloudping.co/grid)
    - Heartbeat interval: 200 ms.
    - Election timeout: 3000 ms.
    - If Etcd leader terminates abnormally, cluster will be unavailable for little more than 3 seconds.

ACID Properties in Distributed System

Capability	Overview	Proposed approach	Difficulty
Atomicity	Multiple database modifications composing single transaction (unit of work) either succeed or fail as a whole	Idempotent operations and retries	hard
Consistency	Guarantees that database is brought from consistent state to consistent state. Triggers and foreign key constraints apply	Guaranteed by application, not data store	easy
Isolation	Ensures that concurrent updates of data leaves database in consistent state, as if all modifications were applied sequentially. If two conflicting transactions modify same entity, one succeeds, the other fails	Requires coordination protocol, e.g. Paxos, RAFT. Access to same data from multiple nodes has to be synchronized	hard
Durability	Confirmed data modifications are durable and will remain in case of system failures, e.g. power outage	Multiple data replicas by design	easy

Summary

- The need of cluster-wide coordination
- RAFT consensus algorithm
- Short introduction to Etcd
  - Basic PUT and GET operations
  - Leases, watches and “mini-transactions”
- Implementation of distributed mutex
- Leader election design pattern
- Deployment requirements for strongly consistent distributed systems