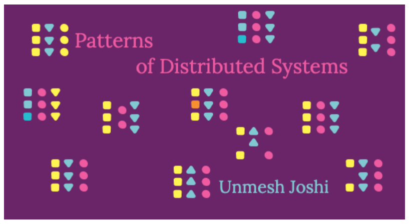
Leader and Followers

Have a single server to coordinate replication across a set of servers.



This pattern is part of Patterns of Distributed Systems

06 August 2020

Unmesh Joshi

CONTENTS

Problem

To achieve fault tolerance in systems which manage data, the data needs to be replicated on multiple servers.

It's also important to give some guarantee about consistency to clients. When data is updated on multiple servers, a decision about when to make it visible to clients is required. Write and read Quorum is not sufficient, as some failure scenarios can cause clients to see data inconsistently. Each individual server does not know about the state of data on the other servers in the quorum, It's only when data is read from multiple servers, the inconsistencies can be resolved. In some cases, this is not enough. Stronger guarantees are needed about the data that is sent to the clients.

Solution

Select one server amongst the cluster as leader. The leader is responsible for taking decisions on behalf of the entire cluster and propagating the decisions to all the other servers.

Every server at startup looks for an existing leader. If no leader is found, it triggers a leader election. The servers accept requests only after a leader is selected successfully. Only the leader handles the client requests. If a request is sent to a follower server, the follower can forward it to the leader server.

Leader Election

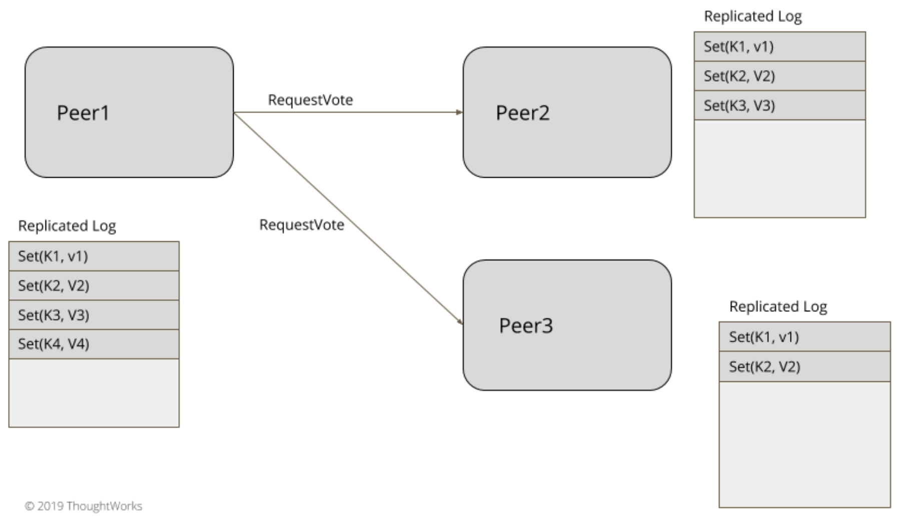


Figure 1: Election

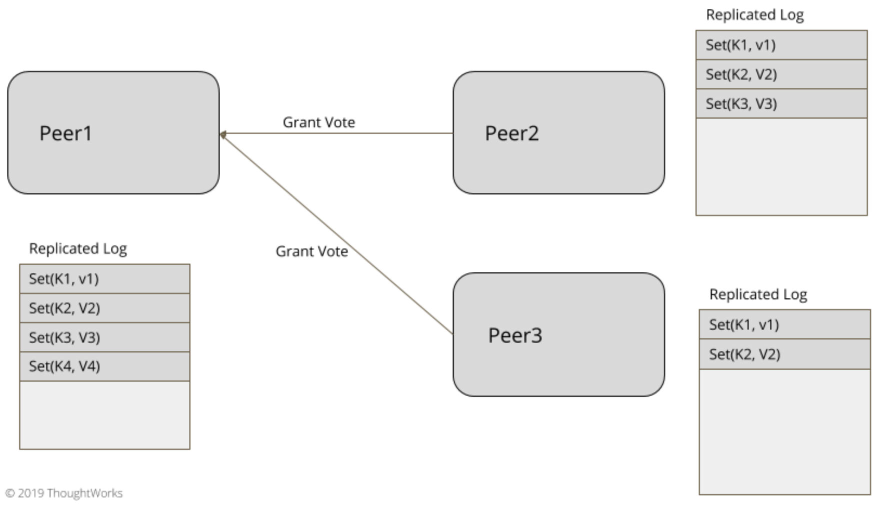


Figure 2: Votes

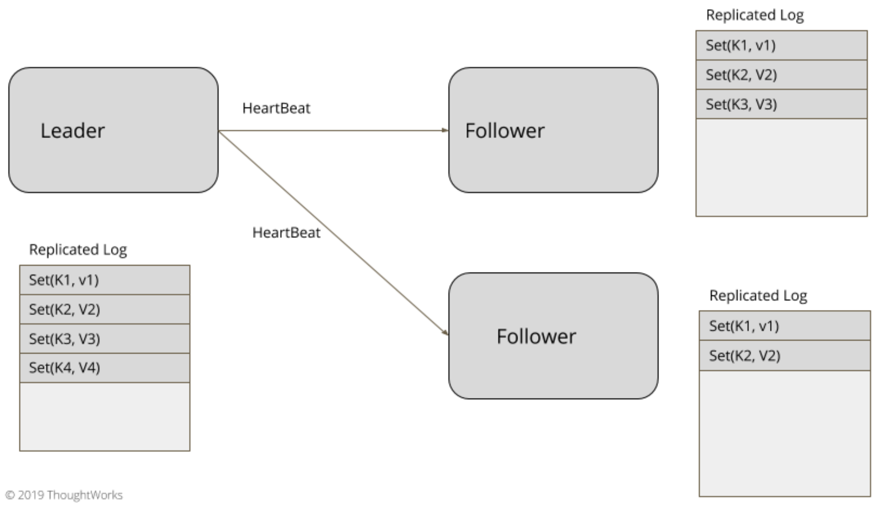


Figure 3: Leader Heartbeats

For smaller clusters of three to five nodes, like in the systems which implement consensus, leader election can be implemented within the data cluster itself without dependending on any external system. Leader election happens at server startup. Every server starts a leader election at startup and tries to elect a leader. The system does not accept any client requests unless a leader is elected. As explained in the Generation Clock pattern, every leader election also needs to update the generation number. The server can always be in one of the three states, Leader, Follower or Looking For Leader (or Candidate)

```
public enum ServerRole {
    LOOKING_FOR_LEADER,
    FOLLOWING,
    LEADING;
}
```

HeartBeat mechanism is used to detect if an existing leader has failed, so that new leader election can be started.

Concurrency, Locks and State updates

State updates can be done without any hassle of manipulating synchronization and locking by using Singular Update Queue

New leader election is started by sending each of the peer servers a message requesting a vote.

class ReplicationModule…

```
  private void startLeaderElection() {
      replicationState.setGeneration(replicationState.getGeneration() + 1);
      registerSelfVote();
      requestVoteFrom(followers);
  }
```

Election Algorithm

ZAB and RAFT

There are two popular mainstream implementations which have leader election algorithms with few subtle differences. Zab as part of Zookeeper implementation and leader election algorithm in Raft

There are subtle differences in things like the point at which the generation number is incremented, the default state the server starts in and how to make sure there are no split votes. In Zab, each server looks for the leader at the startup, generation number is incremented only by a leader when its elected and split vote is avoided by making sure each server runs the same logic to choose a leader when multiple servers are equally upto date. In the case of RAFT, servers start in the follower state by default, expecting to get heartbeats from the existing leader. If no heartbeat is received, they start election by incrementing the generation number. The split vote is avoided by using randomized timeouts before starting the election.

There are two factors which are considered while electing a leader.

- Because these systems are mostly used for data replication, it puts some extra restrictions on which servers can win the election. Only servers, which are the

'most up to date' can be a legitimate leader. For example, in typical consensus based systems, The 'most up to date' is defined by two things:
  - The latest Generation Clock
  - The latest log index in Write-Ahead Log
- If all the servers are equally upto date, then the leader is chosen based following criterias:
  - Some implementation specific criteria, like which server is ranked better or has higher id. (e.g. Zab)
  - If care is taken to make sure only one server asks for a vote at a time, then whichever server starts the election before others. (e.g. Raft)

Once a server is voted for in a given Generation Clock, the same vote is returned for that generation always. This makes sure that some other server requesting a vote for the same generation is not elected, when a successful election has already happened. The handling of vote request happens as following:

class ReplicationModule...

```
  VoteResponse handleVoteRequest(VoteRequest voteRequest) {
      //for higher generation request become follower.
      // But we do not know who the leader is yet.
      if (voteRequest.getGeneration() > replicationState.getGeneration()) {
          becomeFollower(LEADER_NOT_KNOWN, voteRequest.getGeneration());
      }

      VoteTracker voteTracker = replicationState.getVoteTracker();
      if (voteRequest.getGeneration() == replicationState.getGeneration()) {
              if(isUptoDate(voteRequest) && !voteTracker.alreadyVoted()) {
                  voteTracker.registerVote(voteRequest.getServerId());
                  return grantVote();
              }
              if (voteTracker.alreadyVoted()) {
                  return voteTracker.votedFor == voteRequest.getServerId() ?
                          grantVote():rejectVote();

              }
      }
      return rejectVote();
  }

  private boolean isUptoDate(VoteRequest voteRequest) {
      boolean result = voteRequest.getLastLogGeneration() > wal.getLastLogEntryGeneration()
              || (voteRequest.getLastLogGeneration() == wal.getLastLogEntryGeneration() &&
              voteRequest.getLogIndex() >= wal.getLastLogEntryId());
      return result;
  }
```

The server which receives votes from the majority of the servers, transitions to leader state. The majority is determined as discussed in Quorum. Once elected, the leader continuously sends HeartBeat to all the followers. If followers do not get a heartbeat in specified time interval, a new leader election is triggered.

Leader Election using External [Linearizable] Store

Running a leader election within a data cluster works well for smaller clusters. For large data clusters, which can be upto few thousand nodes, it's easier to use an external store like Zookeeper or etcd. (which internally use consensus and provide linearizability guarantees). These large clusters typically have a server which is marked as a master or a controller node, which makes all the decisions on behalf of the entire cluster. There are three functionalities needed for implementing a leader election:

- A compareAndSwap instruction to set a key atomically.
- A heartbeat implementation to expire the key if no heartbeat is received from the elected leader, so that a new election can be triggered.
- A notification mechanism to notify all the interested servers if a key expires.

For electing the leader, each server uses the compareAndSwap instruction to try and create a key in the external store, and whichever server succeeds first, is elected as a leader. Depending on the external store used, the key is created with a small time to live. The elected leader repeatedly updates the key before the time to live value. Every server can set a watch on this key, and servers get notified if the key expires without getting updated from the existing leader within the time to live setting. e.g. etcd allows a compareAndSwap operation, by allowing a set key operation only if the key does not exist previously. In Zookeeper there is no explicit compareAndSwap kind of operation supported, but it can be implemented by trying to create a node, and expecting an exception if the node already exists. There is no explicit time to live either, but zookeeper has a concept of ephemeral node. The node exists until the server has an active session with zookeeper, else the node is removed and everyone who is watching that node is notified. For example, Zookeeper can be used to elect leader as following:

class ServerImpl…

```
public void startup() {
    zookeeperClient.subscribeLeaderChangeListener(this);
    elect();
}

public void elect() {
    var leaderId = serverId;
    try {
        zookeeperClient.tryCreatingLeaderPath(leaderId);
        this.currentLeader = serverId;
        onBecomingLeader();
    } catch (ZkNodeExistsException e) {
        //back off
        this.currentLeader = zookeeperClient.getLeaderId();
    }
}
```

All other servers watch for the liveness of the existing leader. When it is detected that the existing leader is down, a new leader election is triggered. The failure detection happens using the same external linearizable store used for the leader election. This external store also has facilities to implement group membership and failure detection mechanisms. For example, extending the above Zookeeper based implementation, a change listener can be configured with Zookeeper which is triggered when a change in the existing leader node happens.

class ZookeeperClient…

```
public void subscribeLeaderChangeListener(IZkDataListener listener) {
    zkClient.subscribeDataChanges(LeaderPath, listener);
}
```

Every server in the cluster subscribes for this change, and whenever the callback is triggered, a new election is triggered again the same way shown above.

class ServerImpl…

```
  @Override
  public void handleDataDeleted(String dataPath) throws Exception {
      elect();
  }
```

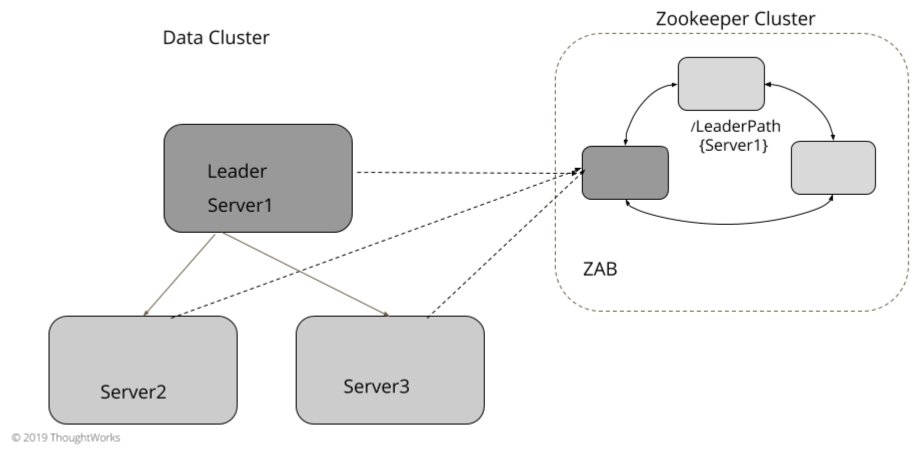


Figure 4: Zookeeper Based Election

Systems like etcd or Consul can be used the same way to implement leader election.

Why Quorum read/writes are not enough for strong consistency guarantees

It might look like Quorum read/write, provided by Dynamo style databases like Cassandra, is enough for getting strong consistency in case of server failures. But that is not the case. Consider the following example. Let's say we have a cluster with three servers. Variable x is stored on all three servers. (It has a replication factor of 3). Value of x = 1 at startup.

- Let's say writer1 writes x = 2, with replication factor of 3. The write request is sent to all the three servers. The write is successful on server1 but fails for server2 and server3. (either a network glitch or writer1 just went into a long garbage collection pause after sending the write request to server1.).
- Client c1 reads the value of x from server1 and server2. It gets the latest value of x=2 because server1 has the latest value.
- Client c2 triggers a read for x. But Server1 goes down temporarily. So c1 reads it from server2, server3, which have old values for x, x=1. So c2 gets the old value even when it read it after c1 read the latest value.

This way two consecutive reads show the latest values disappearing. Once server1 comes back up, subsequent reads will give the latest value. And assuming the read repair or anti entropy process is running, the rest of the servers will get the latest value as well 'eventually'. But there is no guarantee provided by the storage cluster to make sure that once a particular value is visible to any clients, all subsequent reads will continue to get that value even if a server fails.

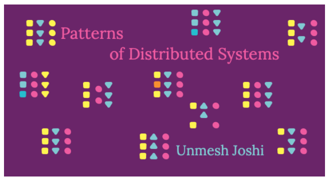
Examples

- For systems which implement consensus, it's important that only one server coordinates activities for the replication process. As noted in the paper Paxos Made Simple, it's important for the liveness of the system.
- In Raft and Zab consensus algorithms, leader election is an explicit phase that happens at the startup or on the leader failure
- Viewstamp Replication algorithm has a concept of Primary, similar to leader in other algorithms
- Kafka has a Controller which is responsible for taking all the decisions on behalf of the rest of the cluster. It reacts to events from Zookeeper and for each partition in Kafka, there is a designated leader broker and follower brokers. The leader and follower selection is done by the Controller broker.

This page is part of:

Patterns of Distributed Systems

by **Unmesh Joshi**



Main Narrative Article

Patterns

† pattern in progress

▶ Significant Revisions