Efficient Geographic Replication & Disaster Recovery

Tom Pantelis
Brian Freeman
Colin Dixon

The Problem: Geo Replication/Disaster Recovery

- Most mature SDN controllers run in a local cluster to tolerate failures
    - Many use strongly consistent replication for at least some data
    - ONOS and OpenDaylight both use RAFT
- What happens when the a whole data center fails?

Service Provider needs

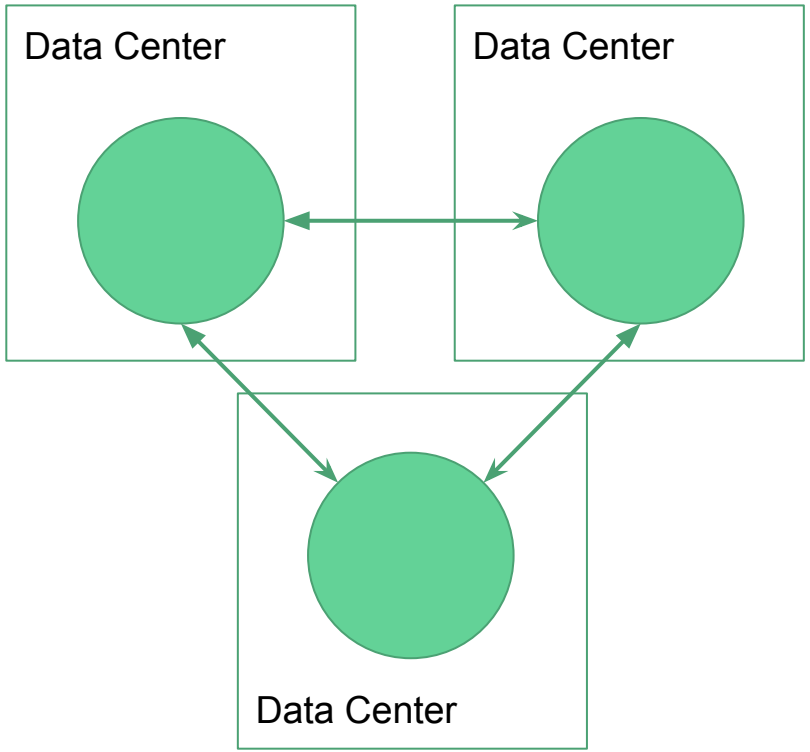
- Geographic-redundancy for disaster recover with no loss of data
- Minimal outage duration while moving to failover site
- Minimal manual intervention
- Ability to easily recover the failed site after repair

Outline

- The Problem
  - Service Provider needs
- **Naïve solutions**
- Our Solution: Non Voting Nodes
- Disaster Recovery
- Operational Feedback
- Future Work

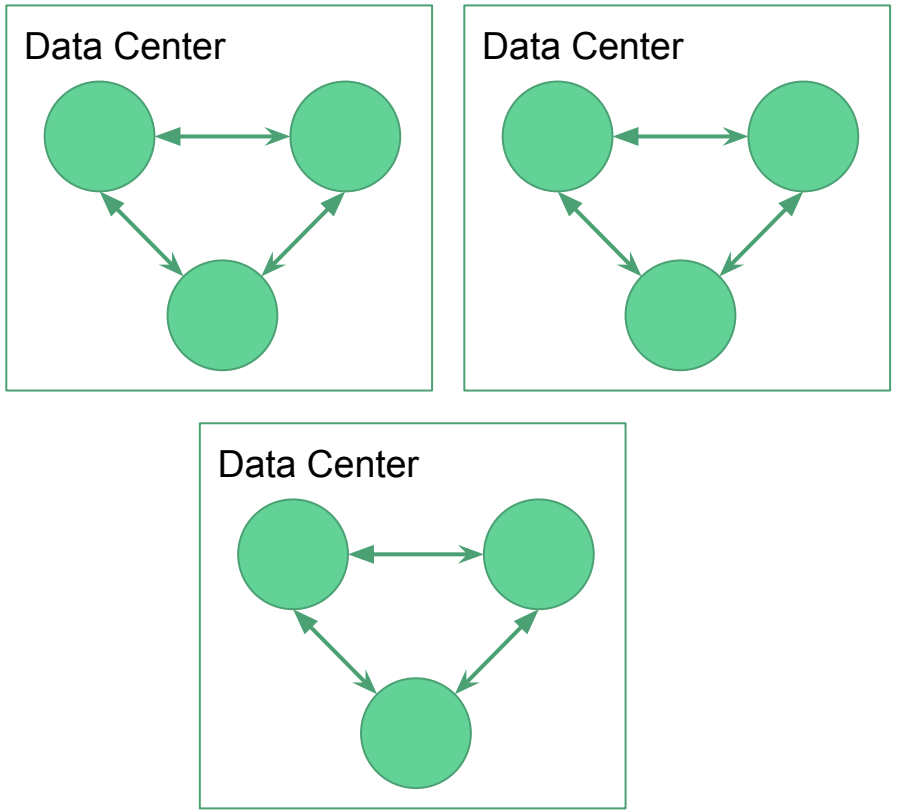
Naïve solution 1: nodes in different data centers

- Latency between locations is often high
- Strongly consistent systems typically require at least one round trip between a majority of the nodes
    - Transaction throughput can be limited to 1/latency if applications serialize their operations, which does is common
- Requires at least 3 locations if using a strongly consistent system



Naïve solution 2: Have 2 or 3 clusters

- With two data centers, either one failing results in strongly consistent systems becoming unavailable
- Adding a third cluster makes it is the same as one node per data center
- Still suffers from high latency for operations and three location



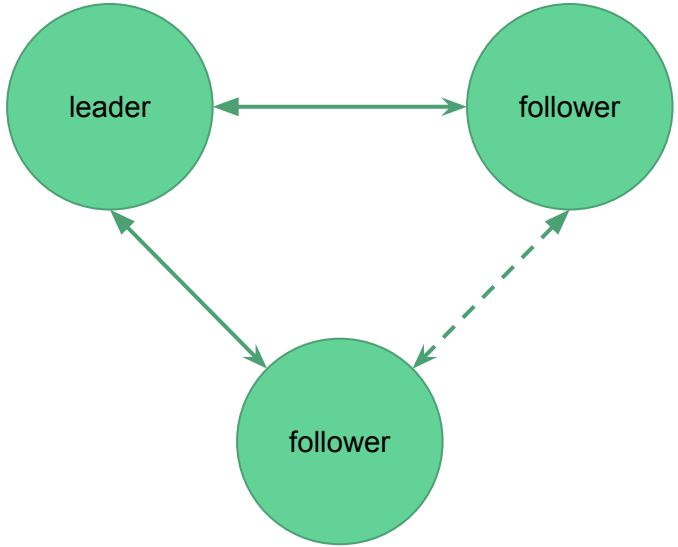
Outline

- The Problem
  - Service Provider needs
- Naïve solutions
- **Our Solution: Non Voting Nodes**
- Disaster Recovery
- Operational Feedback
- Future Work

A very fast introduction to RAFT

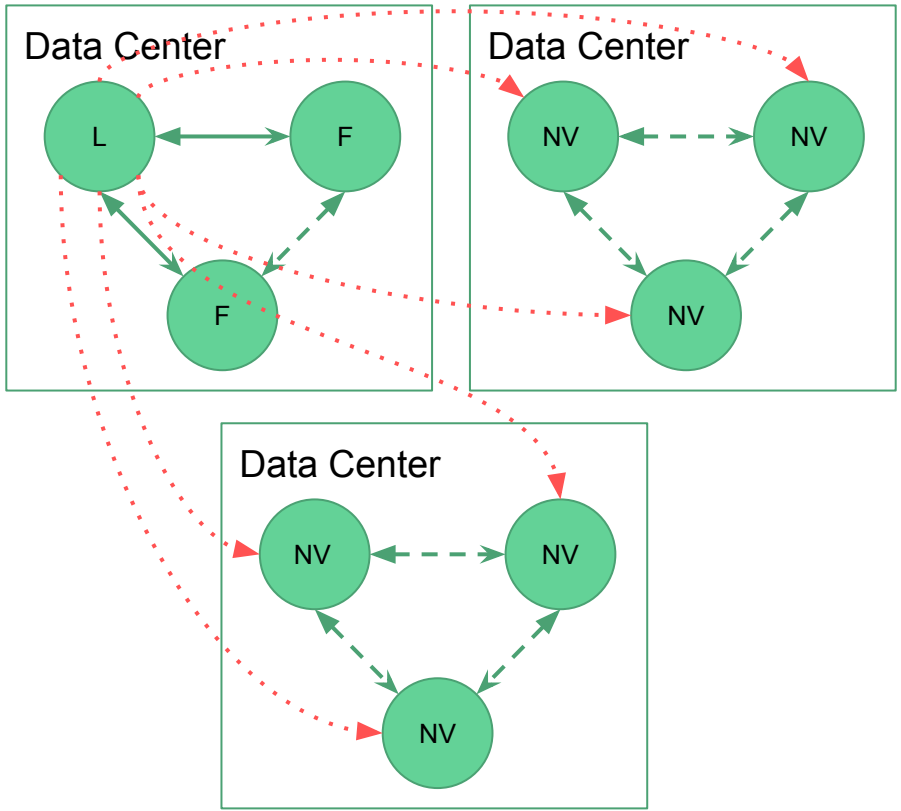
- Produces a consistent replicated log

- Elect a leader for a term
- Leader takes operations (AKA log entries; AKA transactions) and replicates them to followers
- Followers “vote” on operations
- Operations “commit” when the leader + followers voting yes is a majority



Our Solution: Non-voting Nodes

- New state for nodes (non-voting)
- Replicate data to non-voting nodes, but don't block on their responses
  - Avoid latency between locations on the critical path
- Mechanically:
  - Only voting nodes can become the leader
  - Only voting nodes participate in leader election
  - All nodes participate in replication but only voting nodes count toward a majority



Non-voting Nodes: Changing Voting States

- All nodes start out as voting
- Voting states can be modified via RESTCONF RPCs
- All voting state change requests are forwarded to the leader which applies the changes and replicates to followers
- If the current leader is changed to non-voting it steps down as leader
  - The remaining voting nodes elect a voting follower to become the new leader
- There must be at least one voting node, requests that don't do this fail

`change-member-voting-states-for-all-shards` RPC to any node specifying the 3 redundant members as non-voting

```
{"input": {
   "member-voting-state": [{
      "member-name": "member-4",
      "voting": "false"
   },{
      "member-name": "member-5",
      "voting": "false"
   },{
      "member-name": "member-6",
      "voting": "false"
   }]
}}
```

Fail over to non-voting clusters

- Issue the **`flip-member-voting-states-for-all-shards`** RPC to any node in the redundant cluster
  - All voting nodes become non-voting and vice-versa

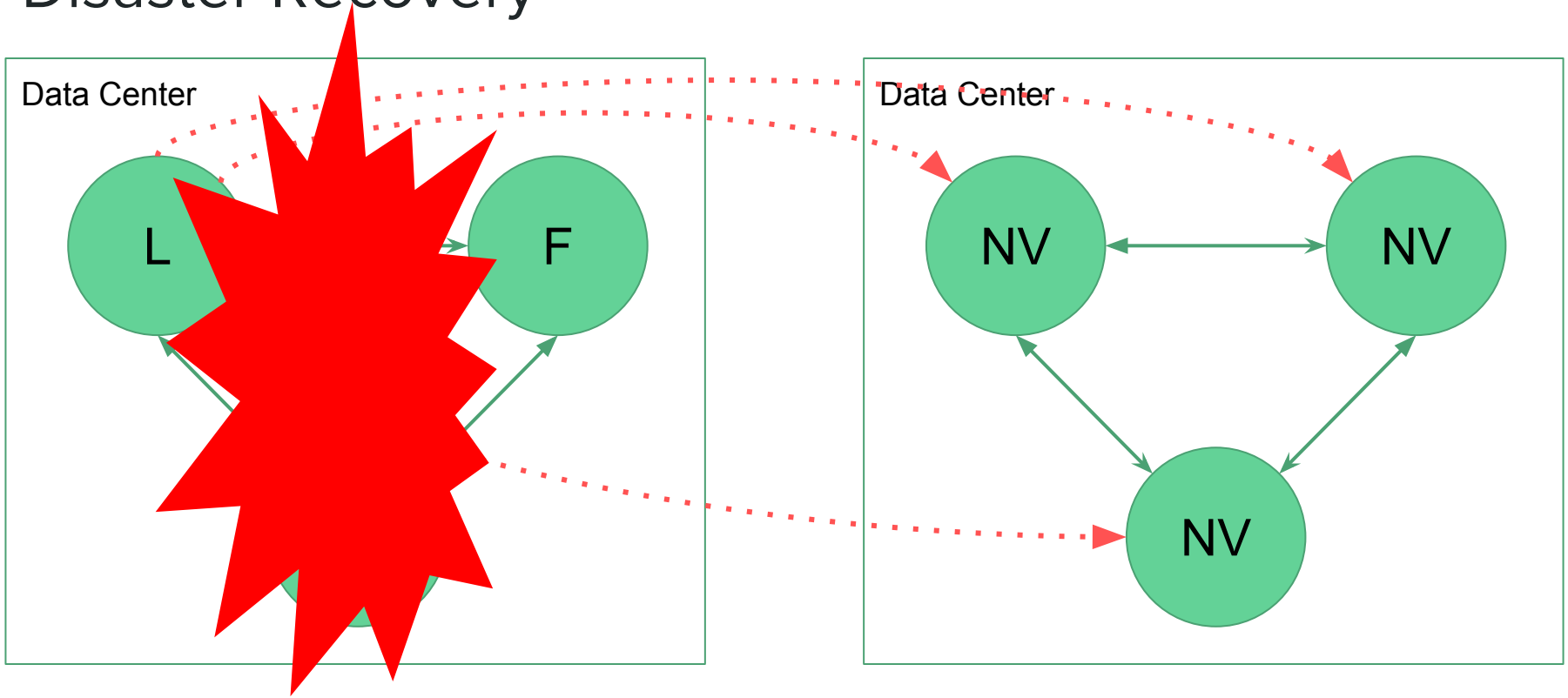
- If the (voting) leader is unavailable, e.g., the primary data center failed
  - the non-voting node to which the request was issued attempts to become the leader by applying the changes locally and starting an election among the non-voting nodes
  - if it successfully gathers enough votes, it becomes the leader and replicates the changes

- This change is not guaranteed to be consistent
  - In practice, there is a very narrow window of operations that can be lost
  - Then the new cluster is strongly consistent again

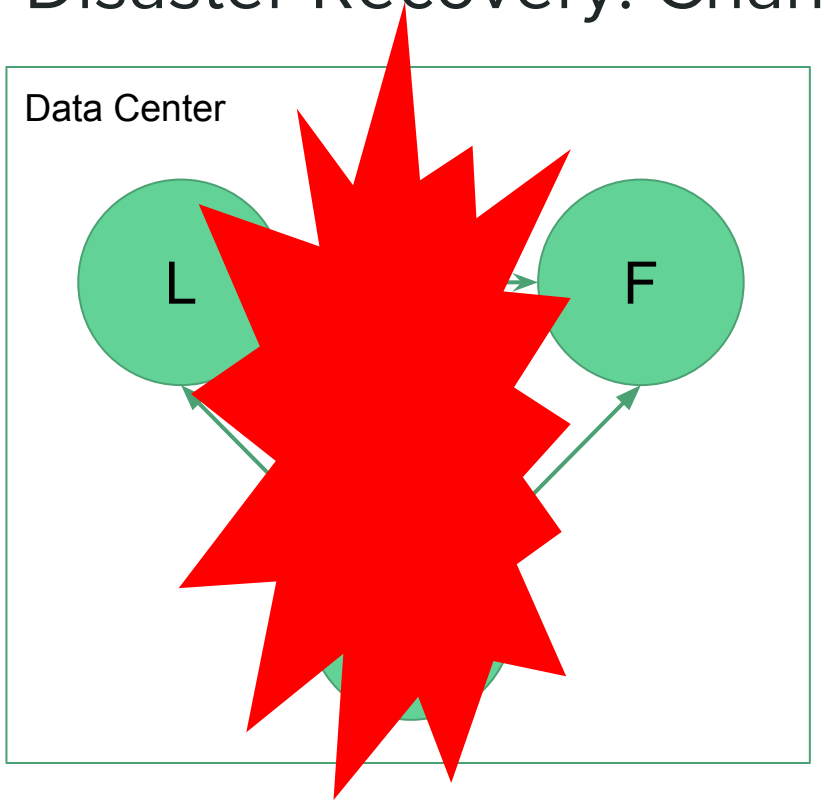
Outline

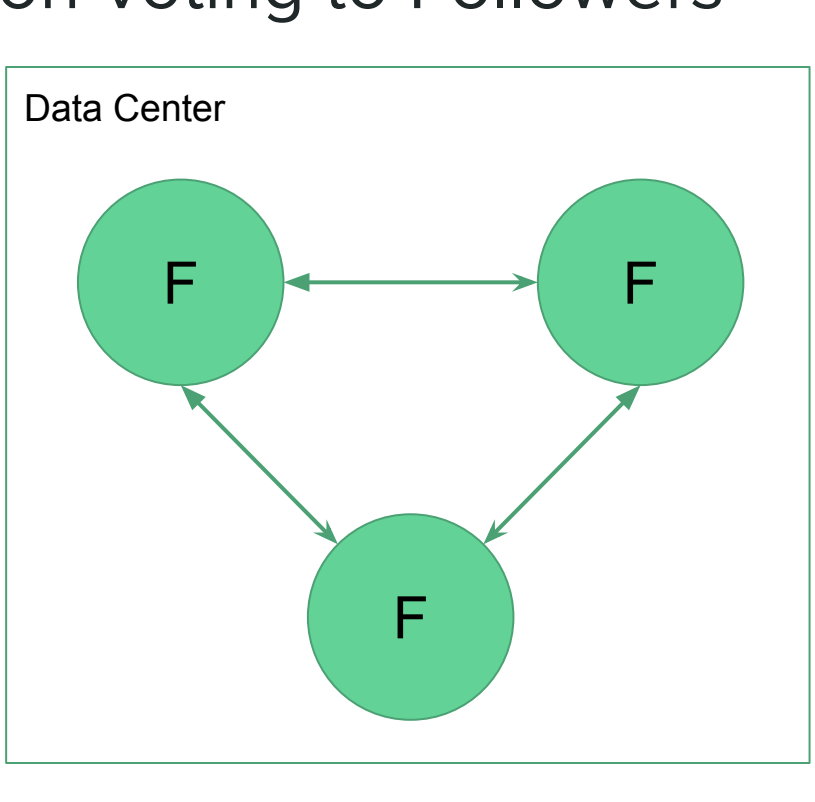
- The Problem
  - Service Provider needs
- Naïve solutions
- Our Solution: Non Voting Nodes
- **Disaster Recovery**
- Operational Feedback
- Future Work

Disaster Recovery

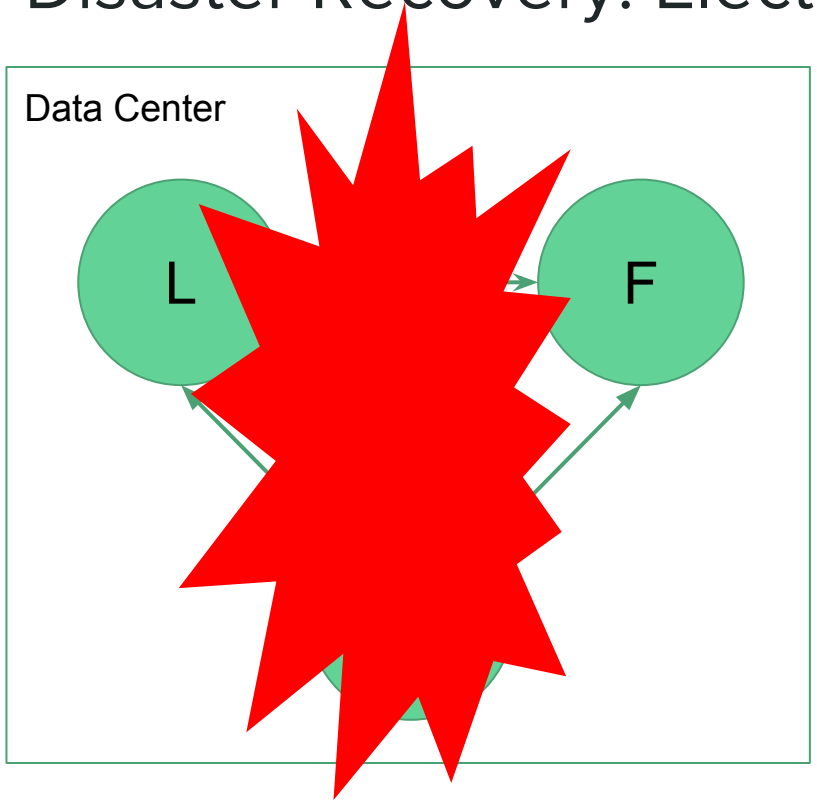


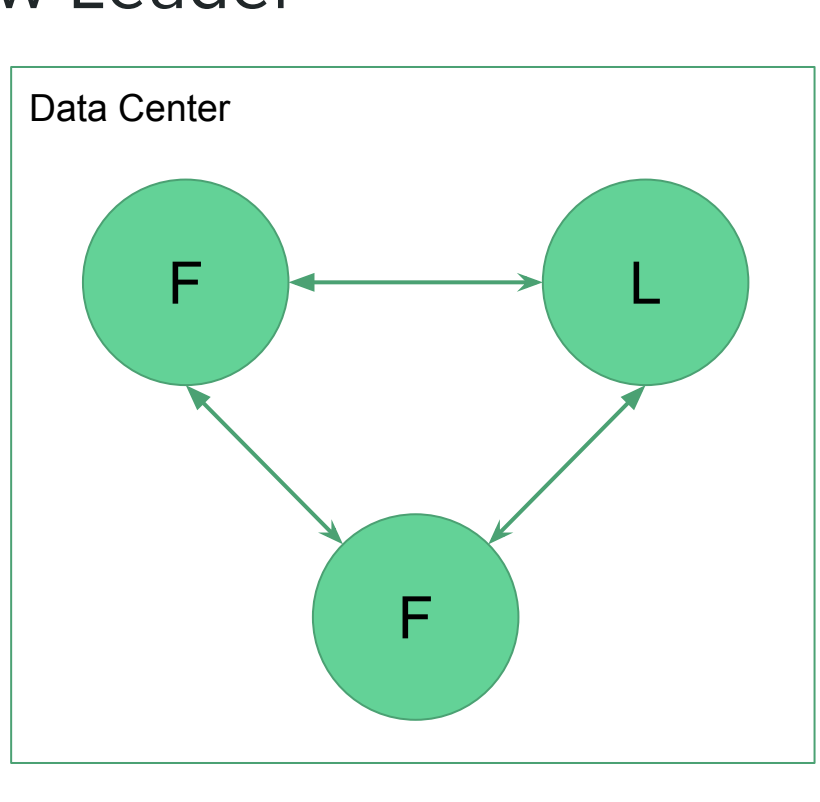
Disaster Recovery: Change Non-voting to Followers





Disaster Recovery: Elect a new Leader





Disaster Recovery: Failing Back

- When the primary cluster comes back, they think they’re voting nodes
- They discover the term has advanced with the election of one of the non-voting nodes as a leader
- Then will switch to being non-voting nodes and catch up from the new primary cluster
- Can then use the same RPC to flip voting and non-voting states as needed

Outline

- The Problem
  - Service Provider needs
- Naïve solutions
- Our Solution: Non Voting Nodes
- Disaster Recovery
- **Operational Feedback**
- Future Work

Operational Feedback

- Current solution includes both OpenDaylight and surrounding infrastructure (RDBMS, iDNS)

- Failover of OpenDaylight is about the same time as failing over the RDBMS
  - Two to three minutes each
  - Switching the iDNS for the northbound interfaces takes additional time.

- Creating geo-redundant clusters and splitting geo-redundant clusters works but can be error prone due to the manual steps

Areas for Improvement

- Automation of the detection and execution of the failover upon two node failure in the primary cluster
- Ability to use the non-voting node solution to support in service upgrades
- Script reports iptables updates to block/unblock access. It should do that automatically. Would be nice to not even use iptables.

Outline

- The Problem
  - Service Provider needs
- Naïve solutions
- Our Solution: Non Voting Nodes
- Disaster Recovery
- Operational Feedback
- **Future Work**

Future Work

- Leverage geographic replication for federation
  - Replicate only a subset of the state
  - Replicate the state to a “shadow” data store
  - Potentially mutate the state before replication
  - Allows for clusters to share state in loosely coupled ways

- Move beyond simple primary and backup clusters

Conclusions

- Naive geographic replication of strongly-consistent clusters hurts hurts performance and is expensive
- Adding non-voting nodes in remote locations trades off small windows on inconsistency for performance and efficiency
- This meets major service providers’ requirements in practice

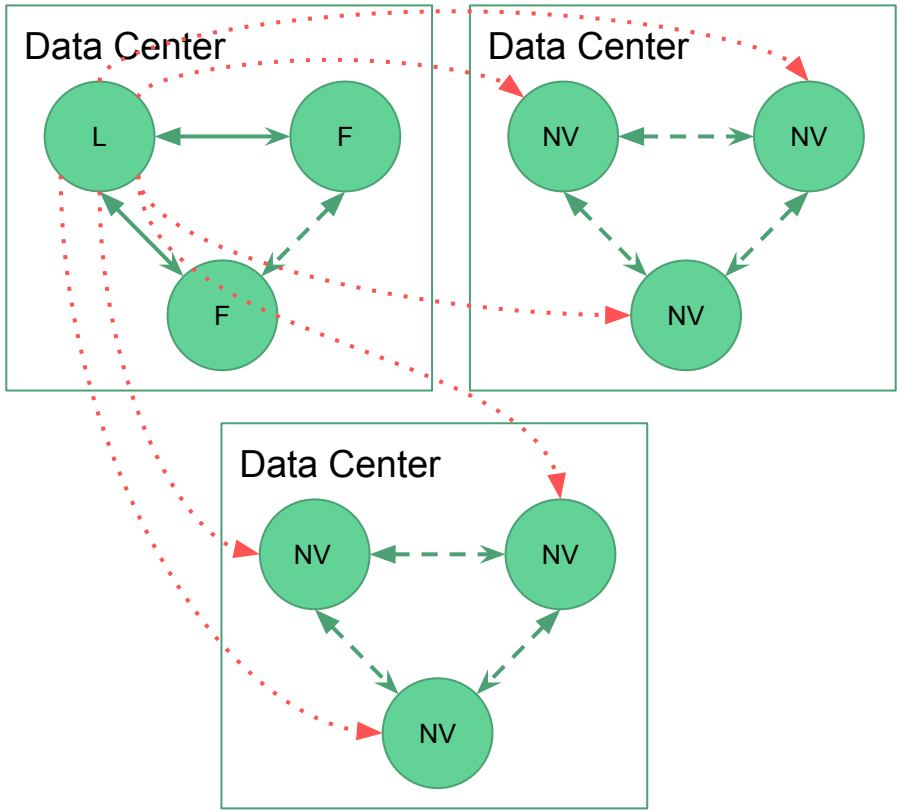
Backup Slides

Abstract

Most, if not all, modern network controllers acknowledge that they are distributed systems and must deal with how to replicate state. There is a trade-off between strong consistency (with easier debugging, programming models, and understandability) on the one hand and tolerance for high latency between controllers and possible partitions on the other hand. To address this, we have extended a strongly-consistent data store (based on RAFT) to also allow for best-effort replication to additional nodes. This allows there to be a small number of nodes (with low latency between them) providing a strongly-consistent data store, while efficiently replicating state to other entities including other controllers acting as warm standbys to handle disaster recovery. Future extensions can allow simple federation by leveraging selective best-effort replication to exchange state without tight coupling.

Our Solution: Non-voting Nodes

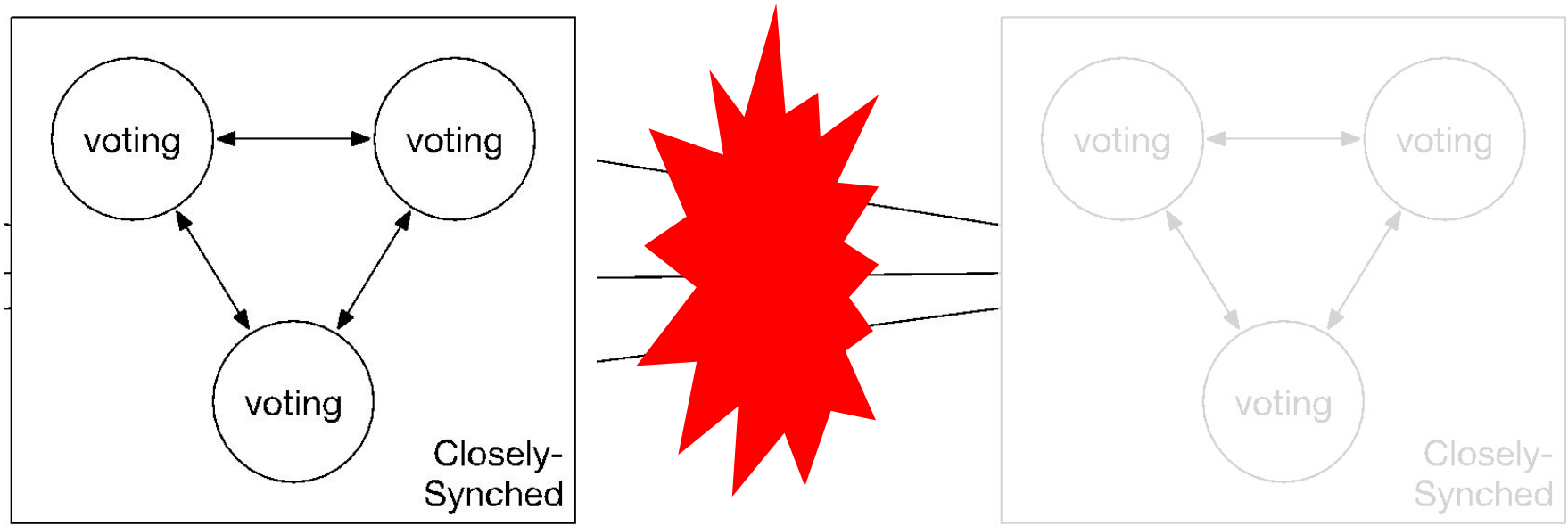
- Each node is assigned a boolean flag indicating whether or not is a voting node
- Rules for voting state
  - Only voting nodes can become the leader
  - Only voting nodes participate in leader election
  - All nodes participate in replication but only voting nodes count toward a majority
- Node voting states are stored as an entry in the journal which is replicated and persisted in the same manner as all data



Disaster Recovery

Lazy replication to second cluster

Primary cluster is tightly coupled



Configuring a redundant 3-node cluster

- Add the 3 redundant nodes to the primary 3 node cluster normally to form a 6-node cluster
- Issue the *change-member-voting-states-for-all-shards* RPC to any node specifying the 3 redundant members as non-voting, eg

```
{"input": {
    "member-voting-state": [{
       "member-name": "member-4",
       "voting": "false"
     },{
       "member-name": "member-5",
       "voting": "false"
     },{
       "member-name": "member-6",
       "voting": "false"
}]}}
```