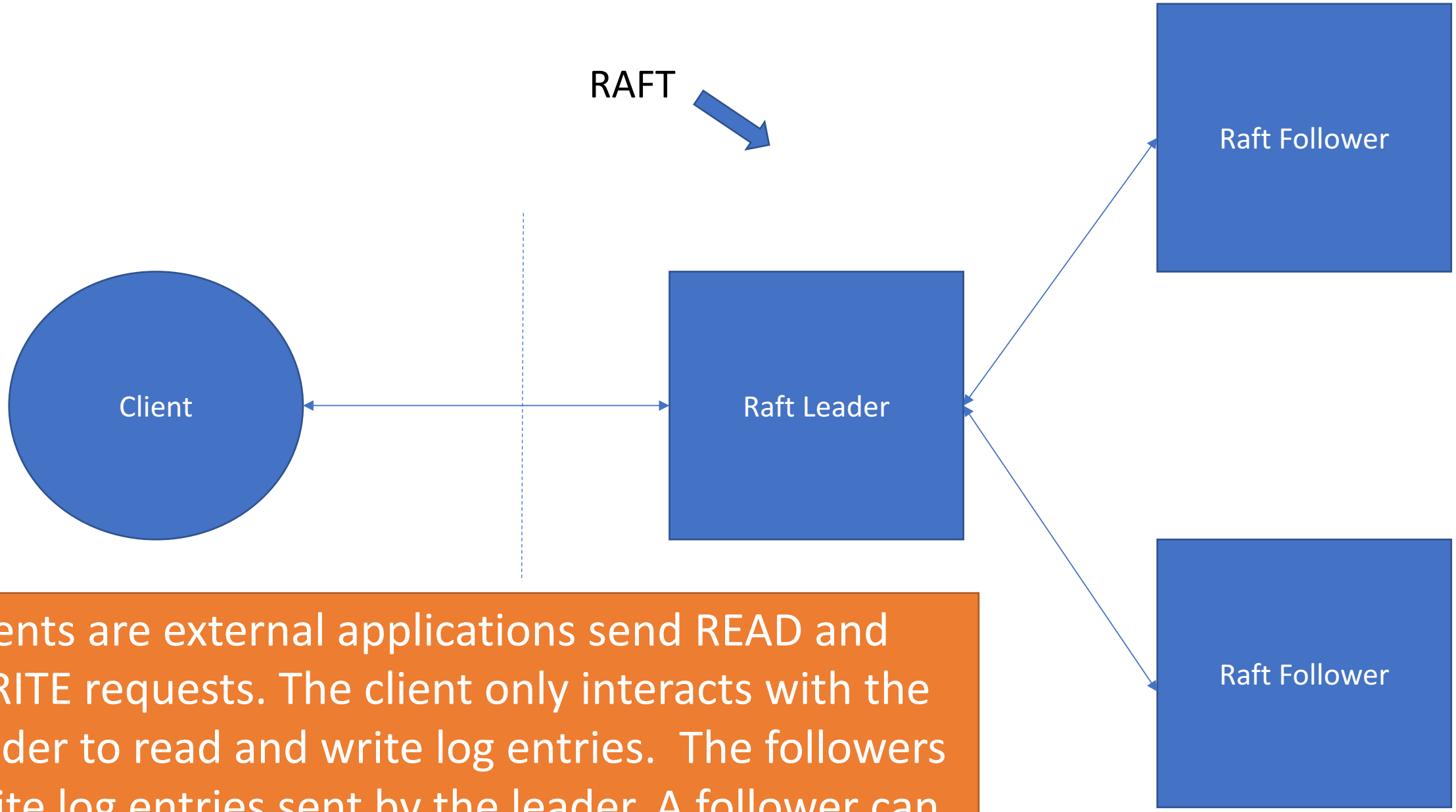
Advanced Raft Topics for Science Gateways

Continuous Delivery for Stateful Services and Security Considerations

Raft Recap

- Log-centric design is a powerful way to build fault tolerant distributed systems.
  - Logs act as a state machine
- Raft is a protocol for maintaining log consistency across multiple servers through consensus
- It uses a strong leader model, is very similar to Zookeeper's Zab protocol
- You could (probably) implement Raft with Zookeeper
  - I have not verified this....



Clients are external applications send READ and WRITE requests. The client only interacts with the leader to read and write log entries. The followers write log entries sent by the leader. A follower can become a new leader.

Logs and Airavata

What is a log in Airavata terms? That is, what is Airavata's state machine?

Airavata State Machine Entries

Data Model	Description and States	Components
Experiment	Records metadata for one or more job submissions. These evolve through states (submitted, executing, completed). Experiments can be shared.	Registry, Task Executor, Sharing Service
Project	These contain experiments. Projects can be shared.	Registry, Sharing Service
Application Description	Machine independent information on how the application is run. Can be added and updated. Sharing coming soon.	Registry, Sharing Service
Application Interface	Deployment specific information on how an application is executed on a computing resource. Can be added and updated. Sharing coming soon.	Registry, Sharing Service
Compute Resource	Information on a particular computing resource. Can be added and updated. Sharing coming soon.	Registry, Sharing Service

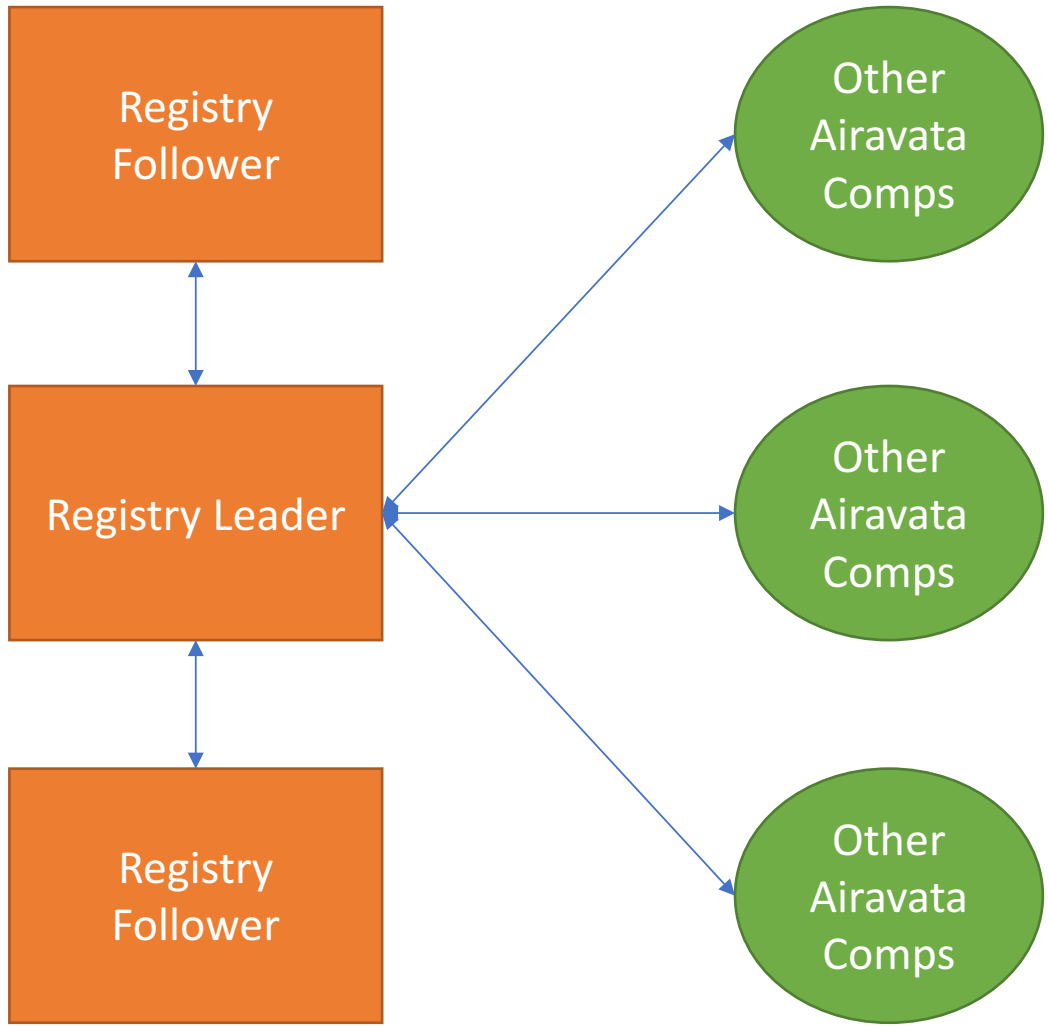
Microservices can split state data across many different services. How do you update these services in a continuous deployment scenario?

Airavata Components Are Log Clients

- Note a log system is just recording logs.
  - It is not implementing other services.
- The Airavata Registry stores a lot of info, but it is not the Airavata's state machine.
  - We should think of it as a client to the log system
  - Overall system state is split between multiple services
- Kafka, for instance, is a good implementation for a logging service.
  - We could use Kafka as our state machine
- But stateful clients may also want to implement Raft's consensus protocol as a way of selecting new leaders.

A Raft-ful Registry

- The Registry today is a frontend to a MariaDB database.
- Imagine a dockerized Registry with multiple instances on various VMs.
- The Registry could use a Raft-like approach for committing incoming messages, determining leaders, and interacting with other Airavata components.
- This has other interesting applications besides fault tolerance



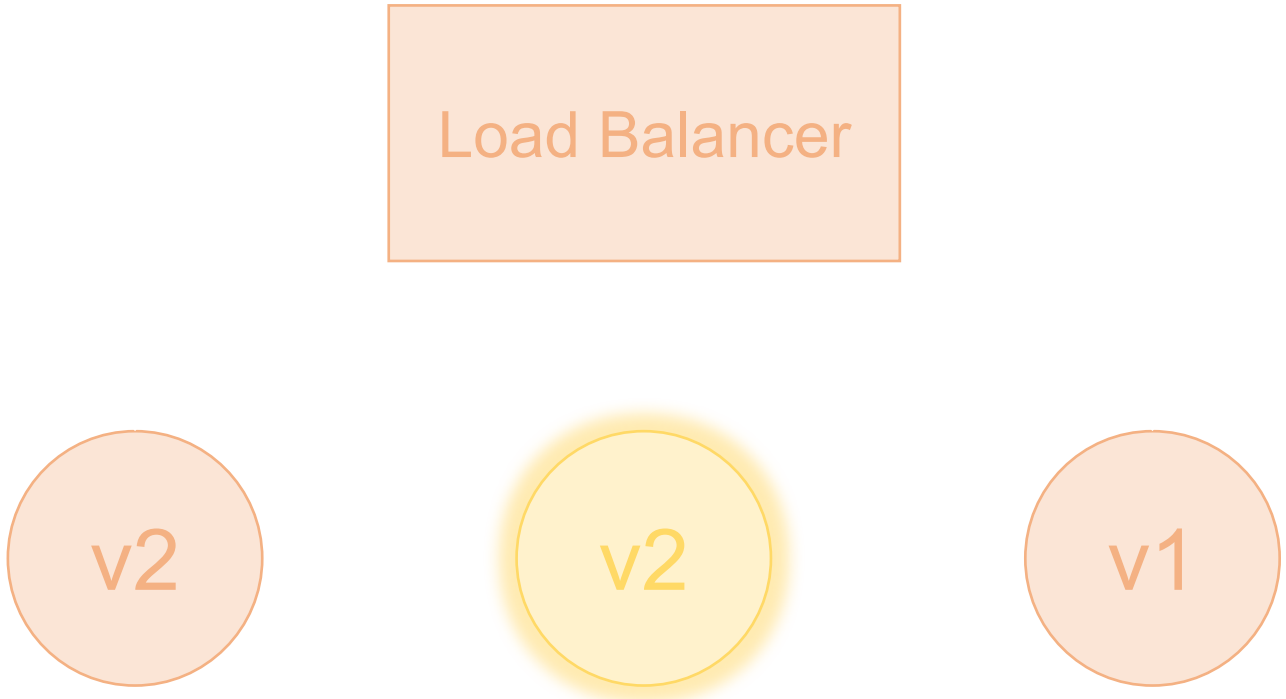
Raft Cluster Membership Changes

Moving from one set of members to another

Updating Raft Clusters

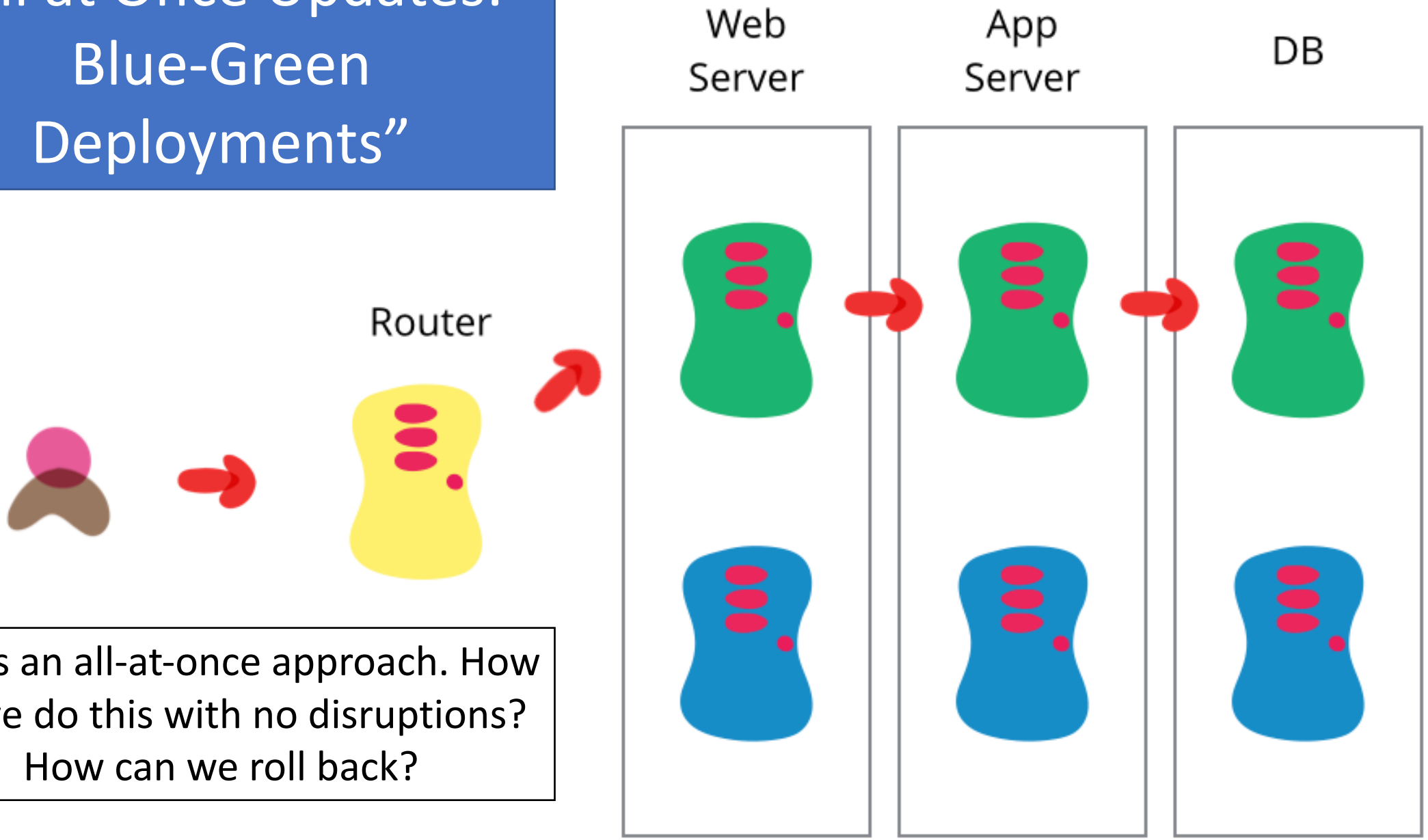
- Raft clusters are quasi-static.
- During operations, all members know about all other members.
- Membership in the cluster is fixed.
- Consensus is based on the number of all members, even if they are unresponsive.
  - You need an odd number of members to commit logs and elect leaders
  - 2N+1 servers can withstand N failed servers
- But Raft clusters need to be updated, of course
  - Migrate to a new version of the service
  - Migrate to a new hosting system
  - Expand or contract the number of cluster members
- We'd like for this to happen without disrupting operations
- **Continuous Deployment Problem!**

“Rolling update – Deploy without downtime”



This is fine for simple, stateless services, but many Airavata services are stateful. We need to make sure both old and new parts of the systems can access messages.

“All at Once Updates: Blue-Green Deployments”

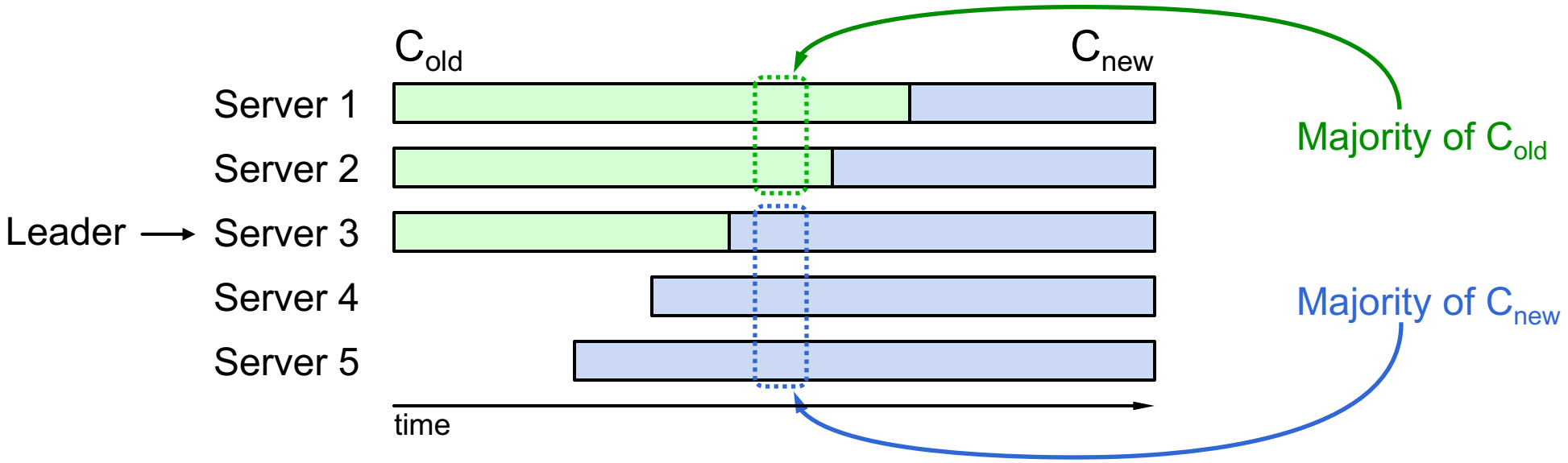


This is an all-at-once approach. How do we do this with no disruptions? How can we roll back?

https://martinfowler.com/bliki/BlueGreenDeployment.html

Configuration Changes

Cannot switch directly from one configuration to another: conflicting majorities could arise during the process if the leader fails



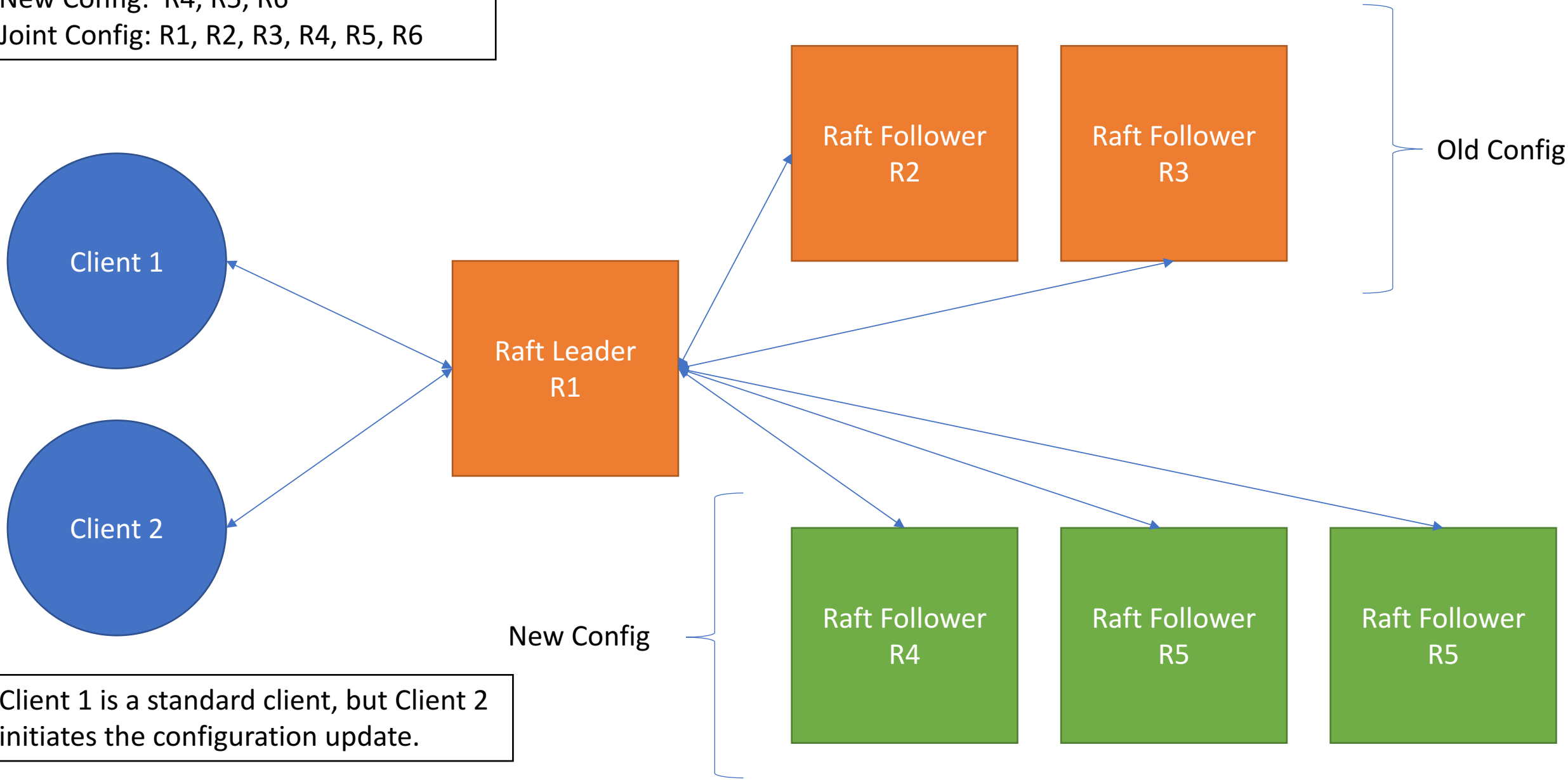
The Raft Configuration Update Approach

“Soft”updates without disruption to clients or major switchovers, and with ability to roll back (by rolling forward).

Joint Consensus in Raft

- Imagine bringing up the servers in the new configuration.
  - They all must know about each other.
- During the transition, all the servers (old and new) must belong to a joint configuration
  - You could imagine using Raft to record and commit the state changes to the configuration
  - The Leader of the old configuration is leader of the joint configuration
- The Leader will update the new followers' logs until they are up to date.
- Log entries from clients are replicated across the entire joint configuration.

Old Config: R1, R2, R3
New Config: R4, R5, R6
Joint Config: R1, R2, R3, R4, R5, R6



Client 1 is a standard client, but Client 2 initiates the configuration update.

Raft Uses Raft to Update Raft

- Let’s assume the leader survives while we are updating from C_old to C_new.
- When the leader receives a message to update the configuration, it stores this message as a log entry.
- It then replicates the log entry to all members of C_old and C_new
  - C_old,new is the joint group
- It commits the entry once it has majority consensus from the C_old,new

Making the Switch to C_new

- When that a majority of C_old,new have committed the update message, it means that a consensus of old and new servers are aware of the update
  - If the leader crashes, another eligible candidate can be come the leader of the joint configuration
- It is now safe for the leader to send the C_new configuration information to just the members of C_new
  - In the process of accepting C_new, the follower also accepts all previous messages as described last time
- Once a majority of C_new members have committed this information, C_old can be shut down.
- This may include the leader of C_old,new, so a new leader is elected.

Issue #1: Bringing New Servers Up to State

- New servers may begin as blank slates with no logs.
- The Raft leader updates them just like it would update any other server
- Problem: C_old,new will be larger than C_old, so majority consensus commits and leader changes may not be achievable or take much longer.
  - The servers in C_new are still busy catching up.
- Raft's simple solution: the servers in C_new are **non-voting** members until they catch up with C_old
  - Raft paper doesn't specify what exactly it means by "catch up"

Issue #2: C_old,new Leaders May Not Be Part of C_new

- The leader steps down after it has committed C_new
  - That is, a majority of C_new members have logged C_new configuration information
- Until this happens, the leader is interacting with clients and writing logs to the C_new servers but not its own logs
  - So it doesn't include itself in majorities

Final Thoughts on Continuous Deployment

- Raft's configuration update protocol may be interesting to apply to Airavata's stateful components such as the Registry
- This requires redesigning Airavata to be log-centric instead of message-queue centric.
- The proposed Kafka-centered approach described on Airavata's Dev list would provide this.
  - We would need to consider how Kafka client groups would change under this scenario.

Log Compaction

Managing old logs and rapidly updating new followers

Raft Log Compaction

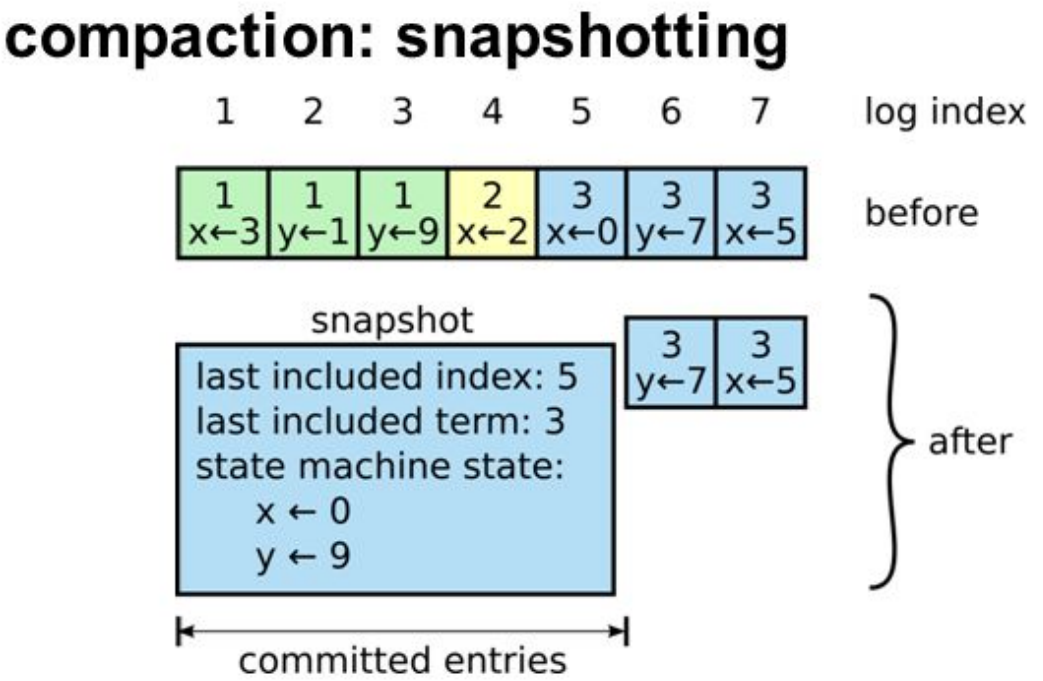
- As we saw with Kafka, logs grow without bound
- Kafka discards logs after a timeout period
  - An SLA understood by all clients
- Replicating entire logs is important in two scenarios
  - Bringing a new server up to date during configuration changes
  - Updating a follower server that may have been offline
- We need a strategy...

Snapshotting

- Used by Zookeeper, Chubby and other systems
- The entire log up to a snapshot point is written to stable, long term storage
  - This is external to the Raft system
- All the logs up to the snapshot point are then discarded
- The snapshot is a log entry that replaces the old entries.

Raft Log Compaction

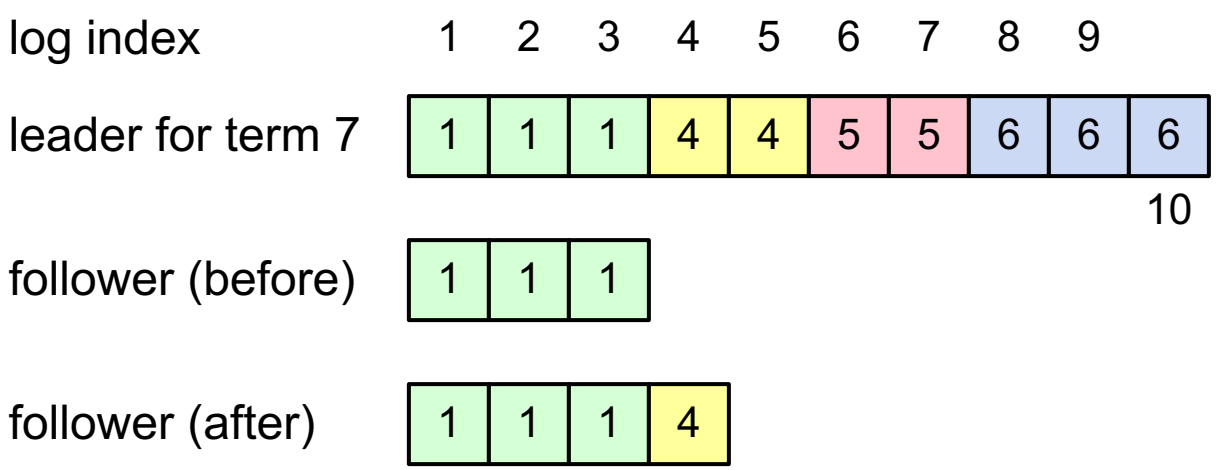
- In Raft, each server can snapshot independently
  - Only committed entries are snapshotted
- The snapshot contains
  - The last included log index
  - The last included term
  - The system state at the snapshot point in the log



Note “state machine state” may be application specific or determiend through some other strategy. Raft does specify what you store in the snapshot log entry. This is a tricky bit.

Leader-Specified Snapshots

- Recall the way Raft works:
  - Leaders instruct followers to append entries.
  - The leader walks backward through its log until it finds an entry that the follower doesn't have
- What if a follower doesn't have entries that the leader has already snapshotted?
- The Raft API includes a method InstallSnapshot that the leader invokes on the lagging follower



This is the way Raft would normally update a logging follower, one entry at a time. What if the leader had already snapshotted entry (4,4)? Or what if this will take too long?

Snapshot Updates Situation #1

- The follower is so far behind that the leader’s snapshot term and index are far in advance of the follower’s logs
- The follower can just discard any of its logs

Snapshot Updates Situation #2

- The snapshot instructions cover a portion but not all of the follower's logs
- The follower snapshots its entries up to the entry in the the leader's snapshot
  - Follower saves the snapshotted data to persistent storage
- Follower writes the snapshot entry to its log and discards all previous entries.

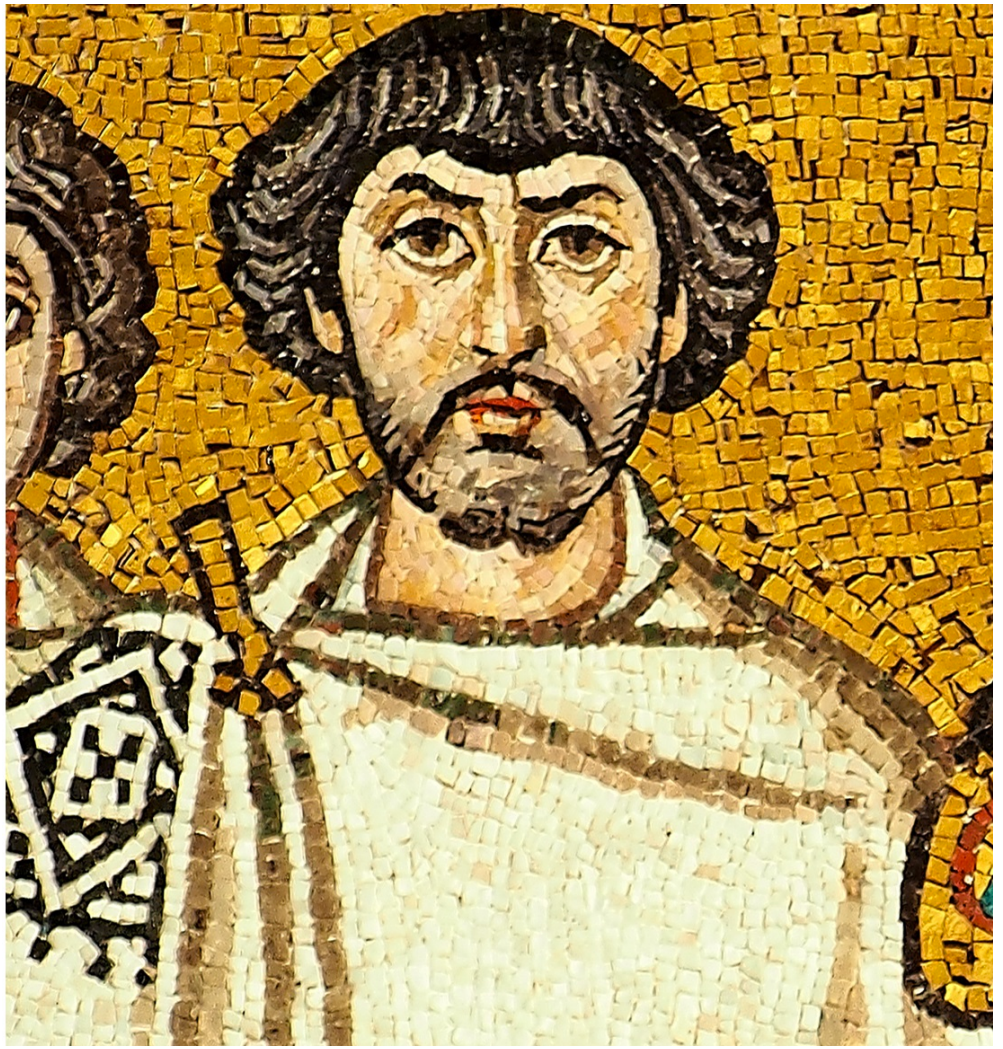
Byzantine Failures and Raft

What if followers don’t play by the rules?

Slides based on “Copeland, C. and Zhong, H., 2016. Tangaroa: a byzantine fault tolerant raft.”

Byzantine Failures

- What if a Raft cluster member doesn't behave like it is supposed to?
- Byzantine failure sources
  - Bugs in software
  - Hardware and networking issues
  - Malicious cluster members
- These types of problems are known academically as "the Byzantine Generals Problem"



Byzantine Leader Election Failures

- In Raft, any follower can attempt to become a leader at any time
- This should only be triggered by heartbeat failures from the leader
- But a malfunctioning follower can keep the system in a state of near-perpetual election

Byzantine Log Replication Failures

- Raft leaders are solely responsible for interacting with external clients and sending log entries to followers.
- Logs are committed once the leader detects consensus
  - Followers trust the leader
  - They don’t know what other followers are doing
- A Byzantine leader could send incorrect log entries
  - Different followers may receive different values for a given log entry
- A Byzantine leader could also send the “commit” message before consensus has been obtained.
- A Byzantine leader may return incorrect results to a client

The solution to Byzantine failures looks a lot like security infrastructure

Authentication, message integrity, etc.

Strategies for Byzantine Fault Tolerance (BFT)

Strategy	Description
Message Signatures	Digital signing can be used to authenticate the message source and verify integrity. Public keys are the standard way to do this.
Client Intervention	Clients can interrupt continuous elections
Incremental Hashing	Each replica contains a cryptographic hash calculated over the new log entry and the previous hash value. By sending the hash, a follower can prove that it has a correct copy of the log
Election Verification	The leader must prove to the other servers that it won by sending a message containing the votes that it received.
Commit Verification	Followers broadcast their AppendEntries response message to the entire cluster, not just the leader. Followers also decide for themselves when to commit
Lazy Voters	Votes are only granted if a follower believes the leader is faulty: the follower also hasn't received leader heartbeats, the client intervenes, or the leader is detected as being byzantine.

BFT Raft Quora

- In normal Raft, consensus is based on all the members of the cluster, even if some are no longer responsive
  - 2N+1 members (odd number) can withstand *N* failures
- In BFT Raft, Byzantine nodes are not considered when determining consensus
  - 3f+1 nodes can withstand *f* byzantine failures; the *f* nodes are not counted
  - So N=4 can withstand f=1 byzantine member (3 functioning members)
  - N=7 can withstand f=2 byzantine members (5 functioning members)
  - N=10 can withstand f=3 byzantine failures (7 functioning members)

Public Key Infrastructure

- In BFT Raft, the cluster members all have the public keys of the other nodes and any clients.
  - Zookeeper can be used for this.
- Public-private key pairs are used as follows:
  - Cryptographically sign messages with the private key
  - Send the signature along with the message
  - Recipients use the public key to verify that the message came from the signer
- This works as long as the private keys are kept private
- Public keys of compromised private keys need to be revoked
  - This is another well known distributed systems problem

Cryptographic Hashing

- A *hash* algorithm is a fast mathematical function that generates a unique, hard-to-guess numerical value from a given input
- Two messages differing by a single character generate completely different hashes.
- Hashes are not reversible: given a value, you can't easily guess the original input
- Hashes are a simple way to verify that data hasn't been corrupted or modified during transmission
- Hashing is often combined with signing
- BFT Raft members hash their entries to prove they are correct
  - Hash of Log=Hash(Entry_n + Hash(Entry_(n-1))

Client-Triggered Timeouts

- Raft elections normally occur when a follower has not received a heartbeat from the leader for a timeout period.
- Since any follower can trigger an election, byzantine followers can disrupt the system by continuously sending “vote for me” messages.
- Clients can detect this because Raft is unable to commit its messages during an election.
  - The old leader is unresponsive
- In BFT Raft, clients can also trigger elections.
- Client-triggered elections go to all cluster members
  - “The leader has failed. There is no leader. You must elect a new leader”
- Leader election proceeds as in Raft.
  - Nodes start sending “Vote for me—I did!” messages to the rest of the cluster.

Note this assumes the client knows how to contact all the members of the Raft cluster

Giving Your Vote

- Nodes receiving a RequestVote message from another node will
  - Validate the signature
  - Ignore if they have received a valid heartbeat from the leader
    - Or else have received a client-instigated message
  - Check to see if the new term proposed by the candidate is higher than the recipient's term
  - Check to see if the candidate sends the term and index of the last committed log
- Followers will only vote once during a given term

Challenges and Further Directions

- Some general challenges
  - Too much security can impact performance
  - Increasing the complication of standard operations like leader election can decrease availability and have unintended consequences
- “Practical Byzantine Fault Tolerance” is the place to get started.
- Bitcoin and Blockchain
  - A scalable approach for peers to keep a ledger of transactions
  - Obviously must be tolerant of forgers and other miscreants
- OAuth2 is the basis for many web security operations today

Some Thoughts on Spring 2018 Topics

- Help with prototyping log-centric Airavata
  - See Dimuthu's postings
  - Gourav Shenoy will give a guest lecture November 14th on his prototypes
- Develop and evaluate prototypes that use Raft-like strategies for updating stateful services like the Airavata Registry
- Develop and evaluate prototype approaches like Practical Byzantine Fault Tolerance and Blockchain that can be used in log-centric Airavata
  - Decide if this is worth the effort