Advanced Raft Topics

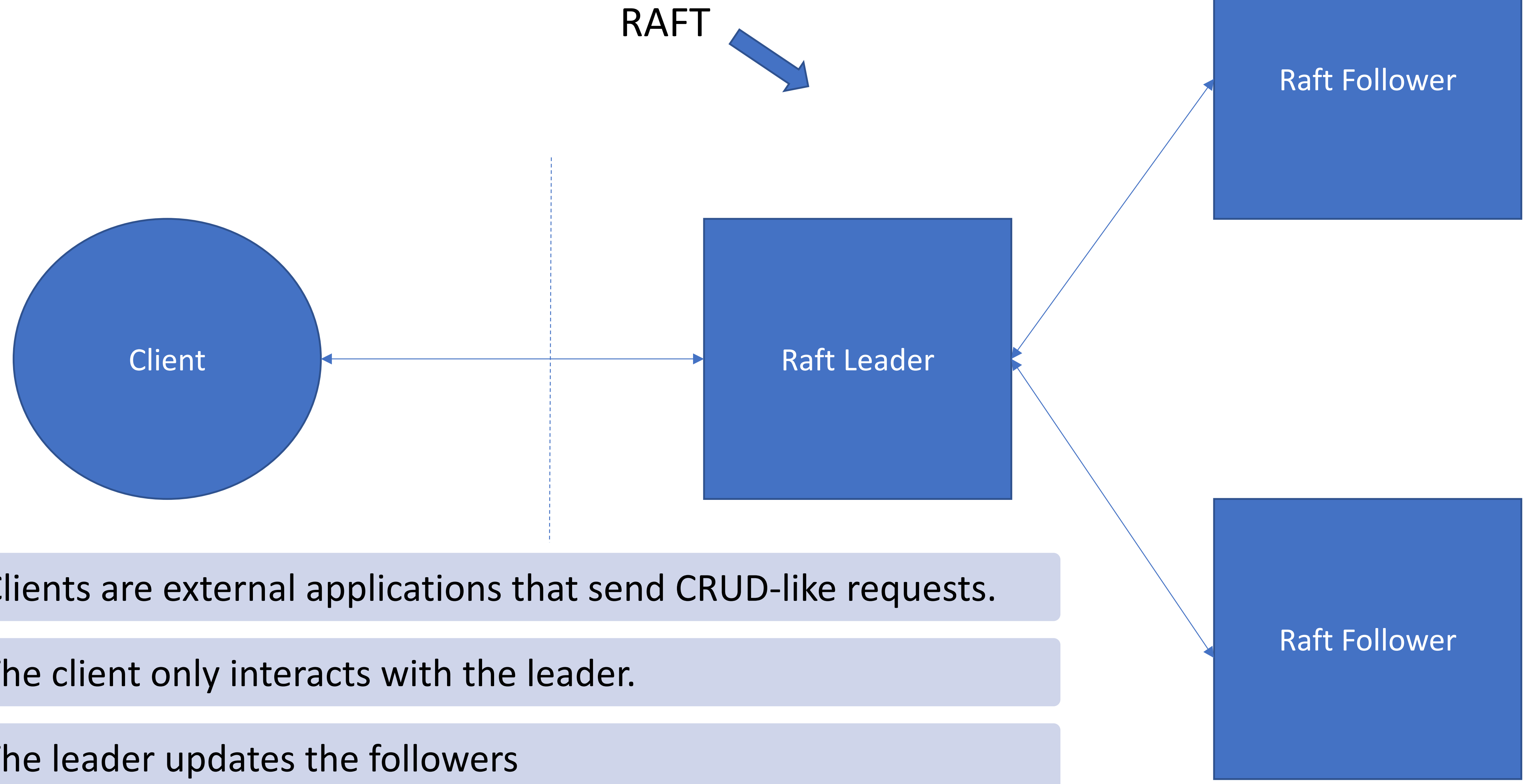
Continuous Delivery for Stateful Services, Log Management, and Security Considerations

Log-Centric Architecture and Raft Recap

Log-centric design is a way to build fault tolerant distributed systems by recording state changing messages

Replay messages in order to recover a particular state

Raft is a popular algorithm for maintaining state log consistency across multiple servers through consensus



- Clients are external applications that send CRUD-like requests.
- The client only interacts with the leader.
- The leader updates the followers
- A follower can become a new leader.

Raft Cluster Membership Changes

Moving from one set of members to another

Raft Clusters Are Quasi-Static

During operations, all members know about all other members.

This information can be in a log message

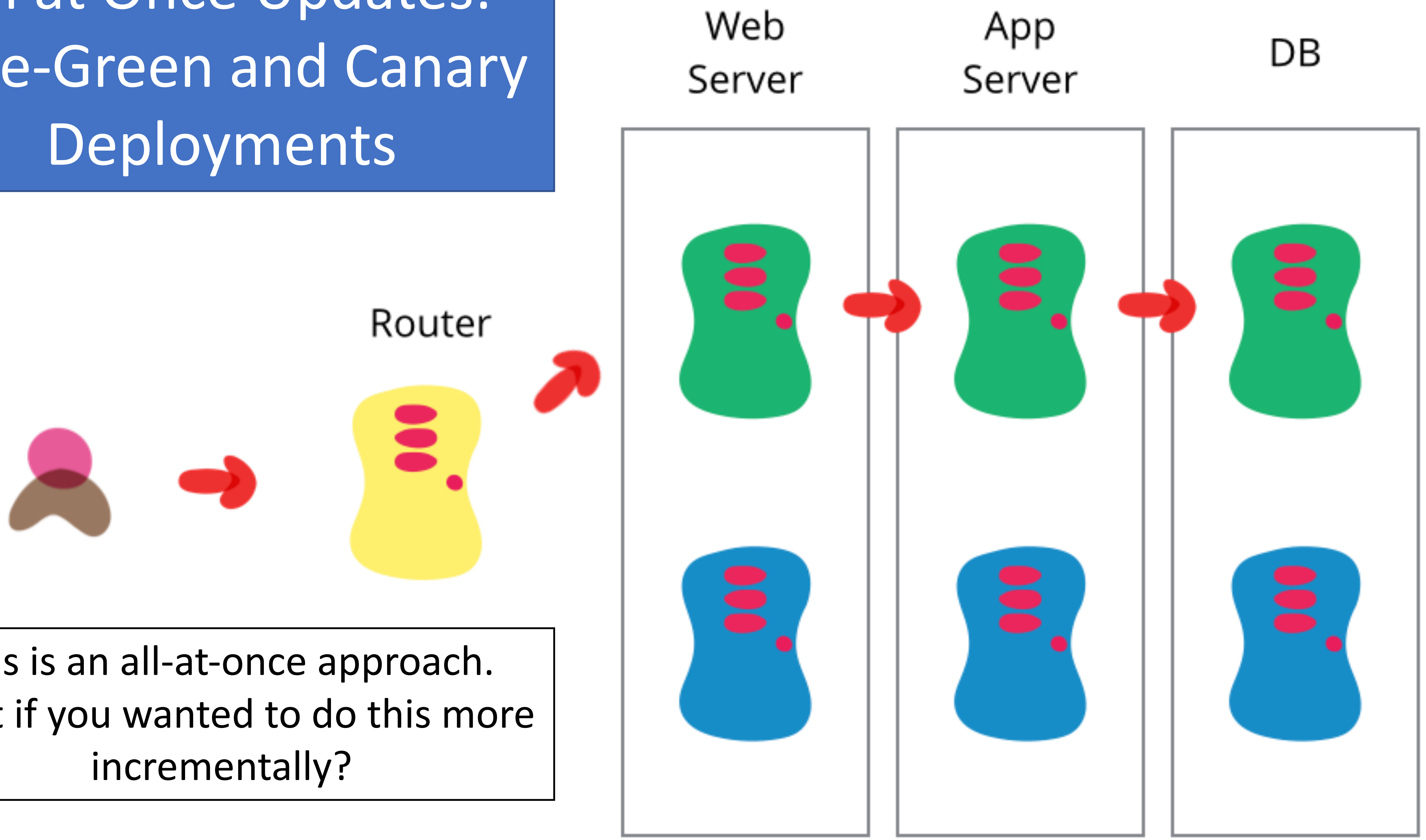
Membership in the cluster is fixed.

Consensus is based on the number of all members, even if they are unresponsive.

Changing a Raft Cluster

- Imagine if you need to grow or shrink the Raft cluster
- Or update the Raft server versions
- Or move to new host servers
- Can you do this without taking the system down?

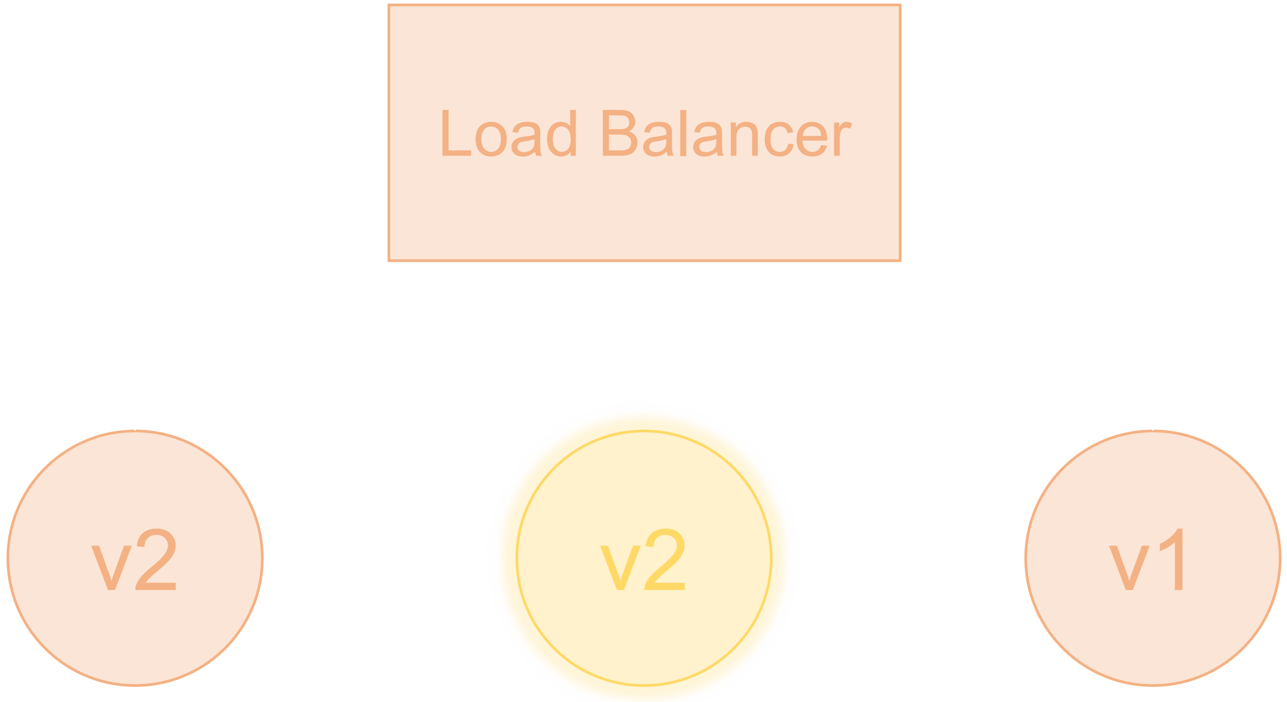
All at Once Updates: Blue-Green and Canary Deployments



This is an all-at-once approach. What if you wanted to do this more incrementally?

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

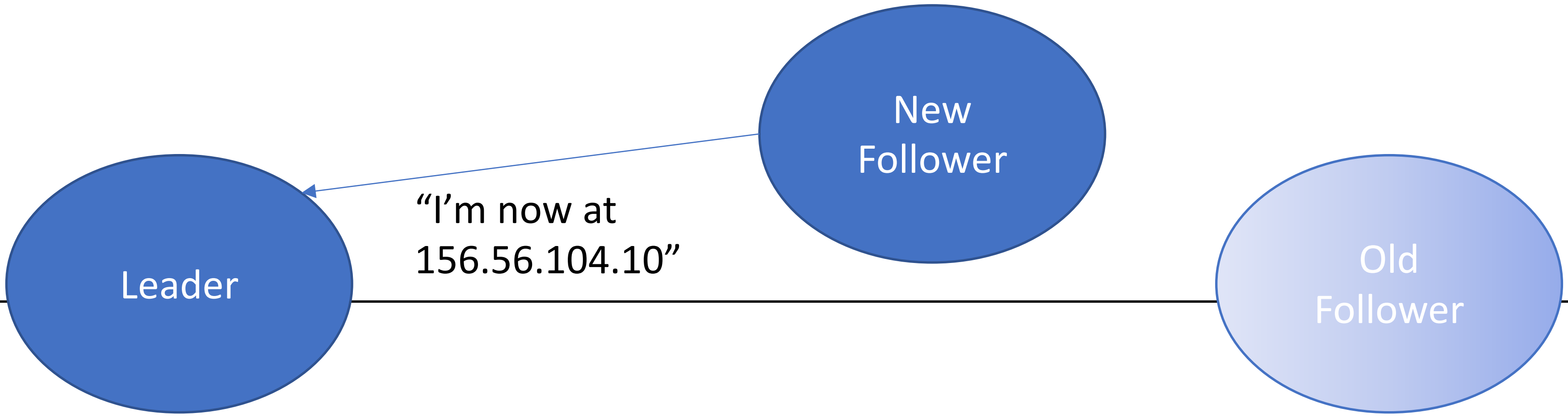
“Rolling update – Deploy without downtime”



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

Updating RAFT Configurations

- In RAFT, all the members of cluster know about each other so that they can run elections
- You can use RAFT messages to update configuration information while the system is operating.



But You Have to Be Careful!

Accidental Elections: RAFT leaders will abdicate and start an election if they lose contact with too many followers. Followers will try to become leaders if they lose contact with the leader

Consensus Problems: If you are growing or shrinking the cluster, this can become even more complicated because the number of servers required for a majority changes.

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

During the transition, all the servers (old and new) must belong to a **joint configuration**

If the leader fails, the new leader can take over the cluster changing process

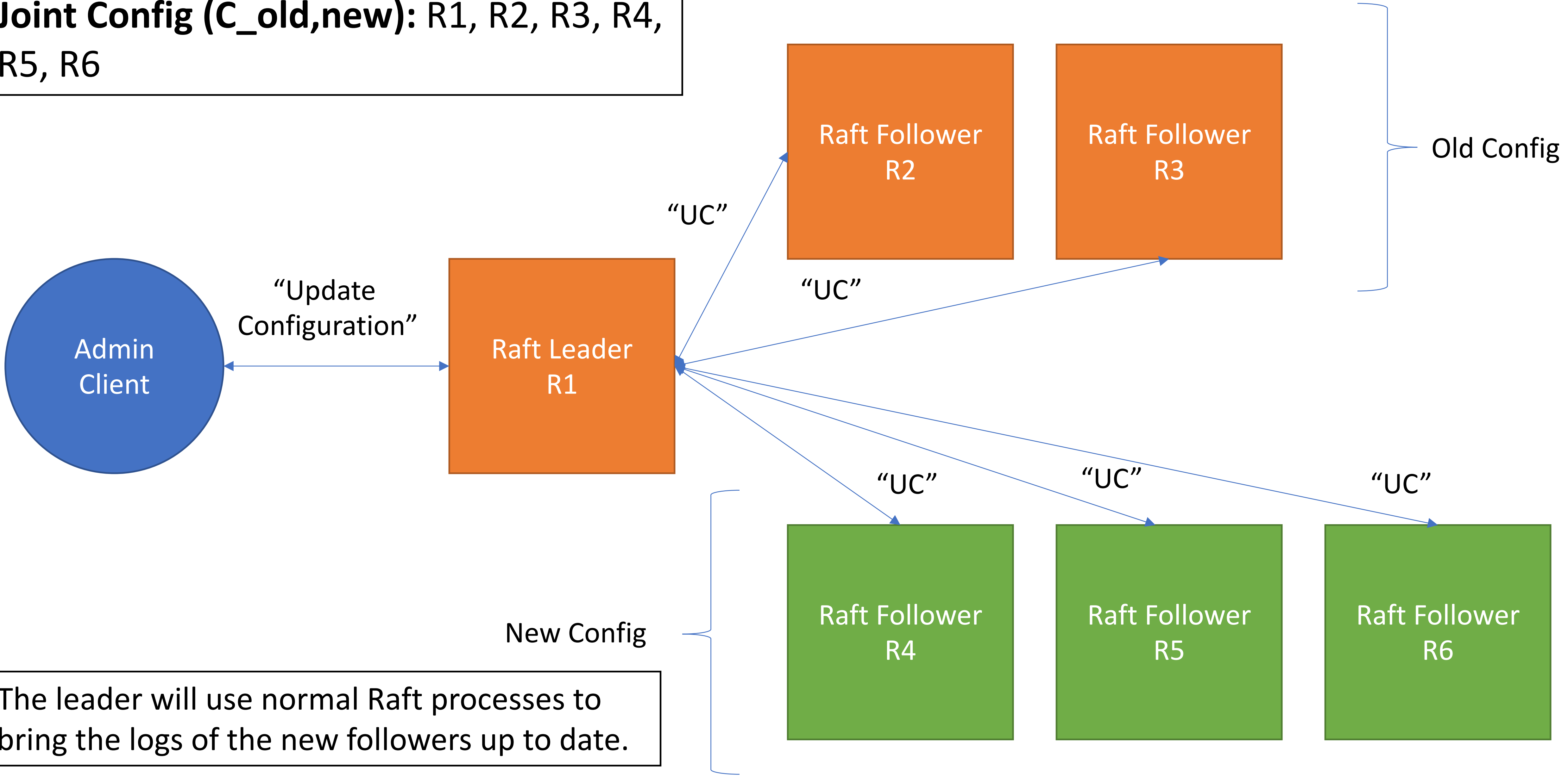
The leader will update the new followers' logs until they are up to date.

Log entries from clients are replicated across the entire joint configuration.

Raft Updates Take Two Steps

- Each stage is signaled by a special log message
- **Step 1:** “Update Configuration” tells everyone that new members are joining
- **Step 2:** “New Configuration” tells the members of the new cluster that the transition is complete

Joint Configuration Phase (Step 1)
Joint Config (C_old,new): R1, R2, R3, R4, R5, R6



The leader will use normal Raft processes to bring the logs of the new followers up to date.

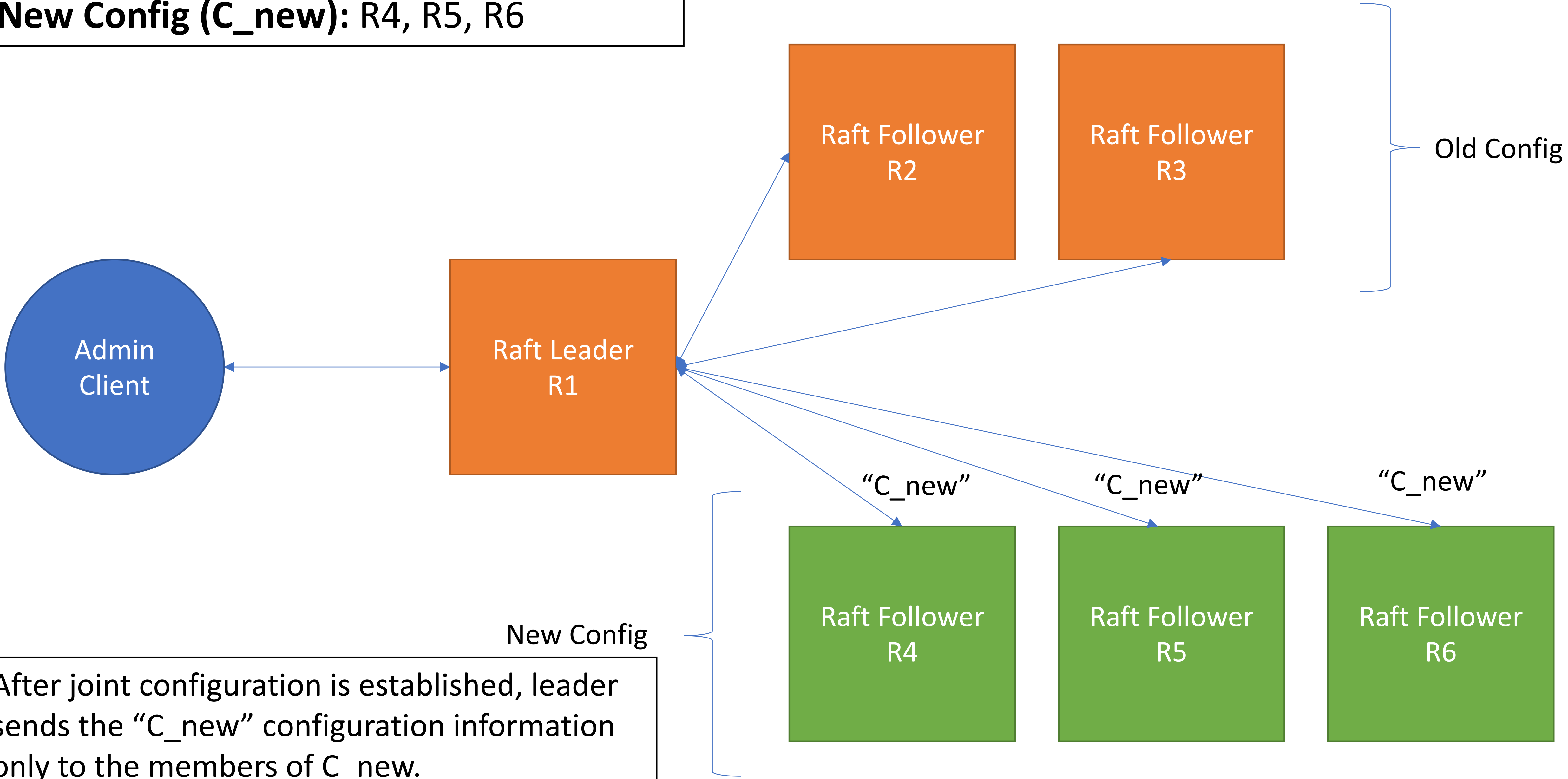
Raft Uses Raft to Update Raft

- 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.
- This message would contain information on the new configuration, including how to reach the new members.

Joint Configuration Stage

- The leader replicates the “update configuration” log entry to all members of C_old and C_new
- C_old,new is the combined group
- The leader commits the entry once it has majority consensus from the combined C_old,new group
- If the leader crashes, another eligible candidate can be come the leader of the joint configuration

New Configuration Phase (Step 2)
New Config (C_new): R4, R5, R6



After joint configuration is established, leader sends the “C_new” configuration information only to the members of C_new.

Switching from the Joint to the New Configuration

Joint configuration protects the system from leader crashes during the update phase.

Consensus for the C_new message is only needed from the members of the C_new cluster.

The leader may not be a member of C_new!

After the “C_new” message is committed, members of C_old can be shut down

The ”C_new” message can only be committed when a majority of the C_new members have up-to-date logs.

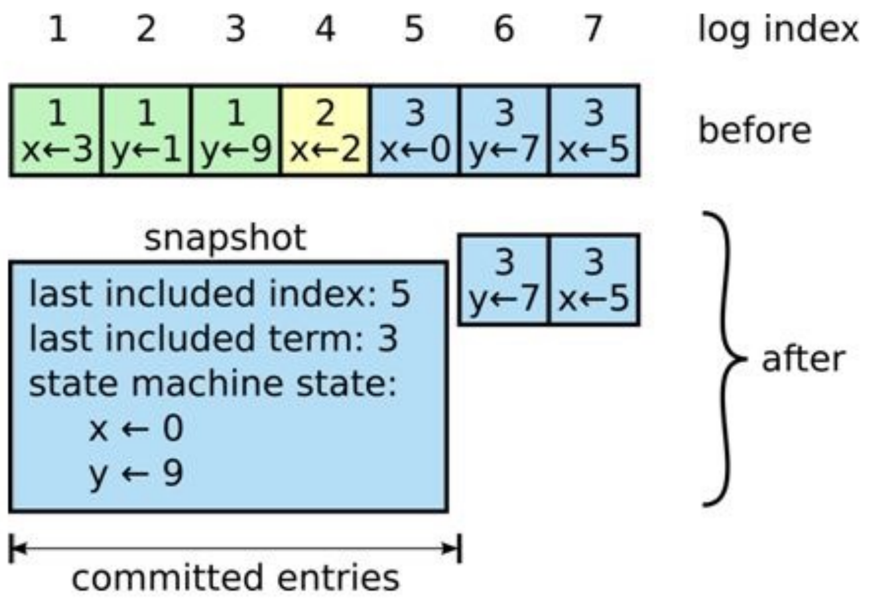
Continuous Deployment and Rolling Back

- If you needed to rollback, you could just bring C_old back up, form a new joint configuration, and reverse the process
- C_old and C_new reverse roles.

A Comment on Raft Log Snapshots and Compaction

- Raft logs can grow very large, increasing the time to bring new members up to state.
- Raft replaces older entries with a single snapshot entry
- The snapshot entry contains
  - The last included log index
  - The last included term
  - The system state at the snapshot point in the log

- **Log compaction: snapshotting**



Raft does not specify what you store in the snapshot log entry. Raft suggests using **Log Structured Merge (LSM) Trees**

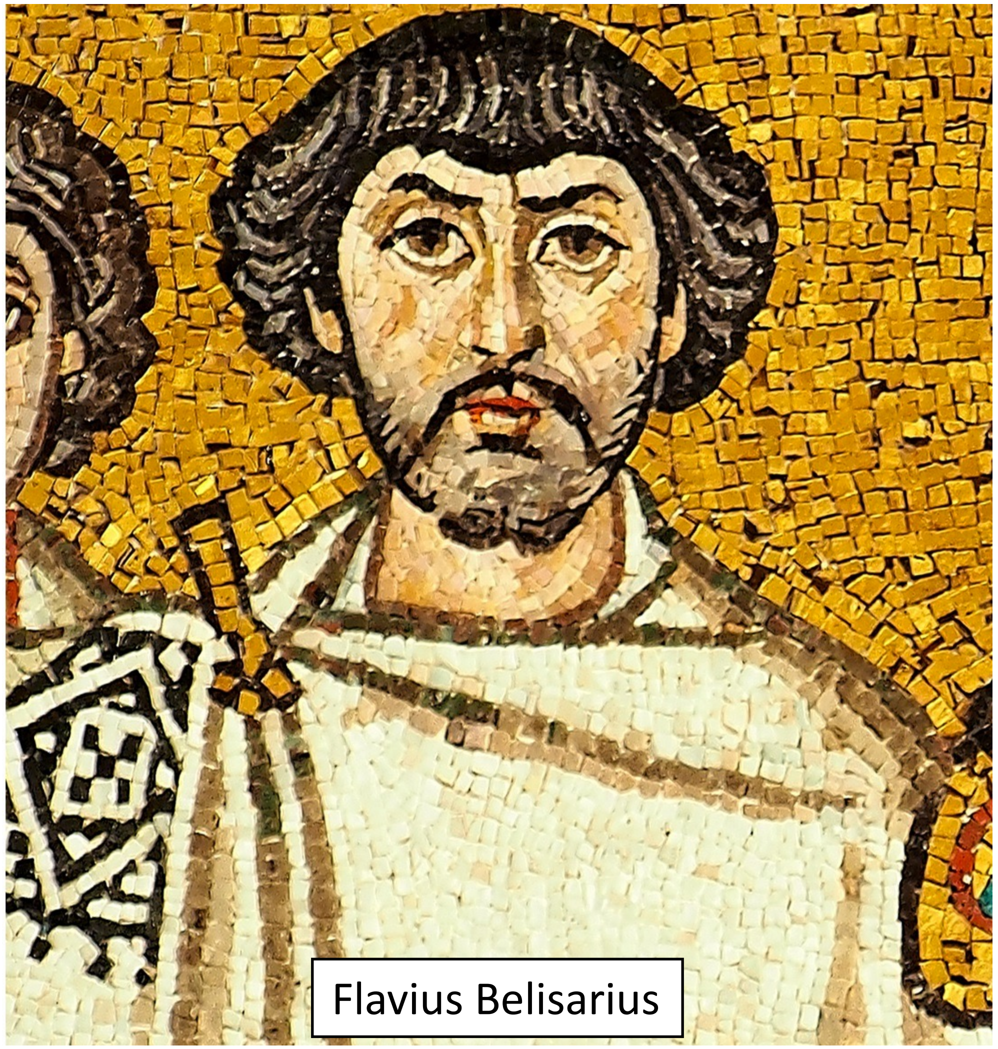
Byzantine Failures and Raft

What if followers don’t play by the rules? What if the leader can’t be trusted?

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

Byzantine Failures

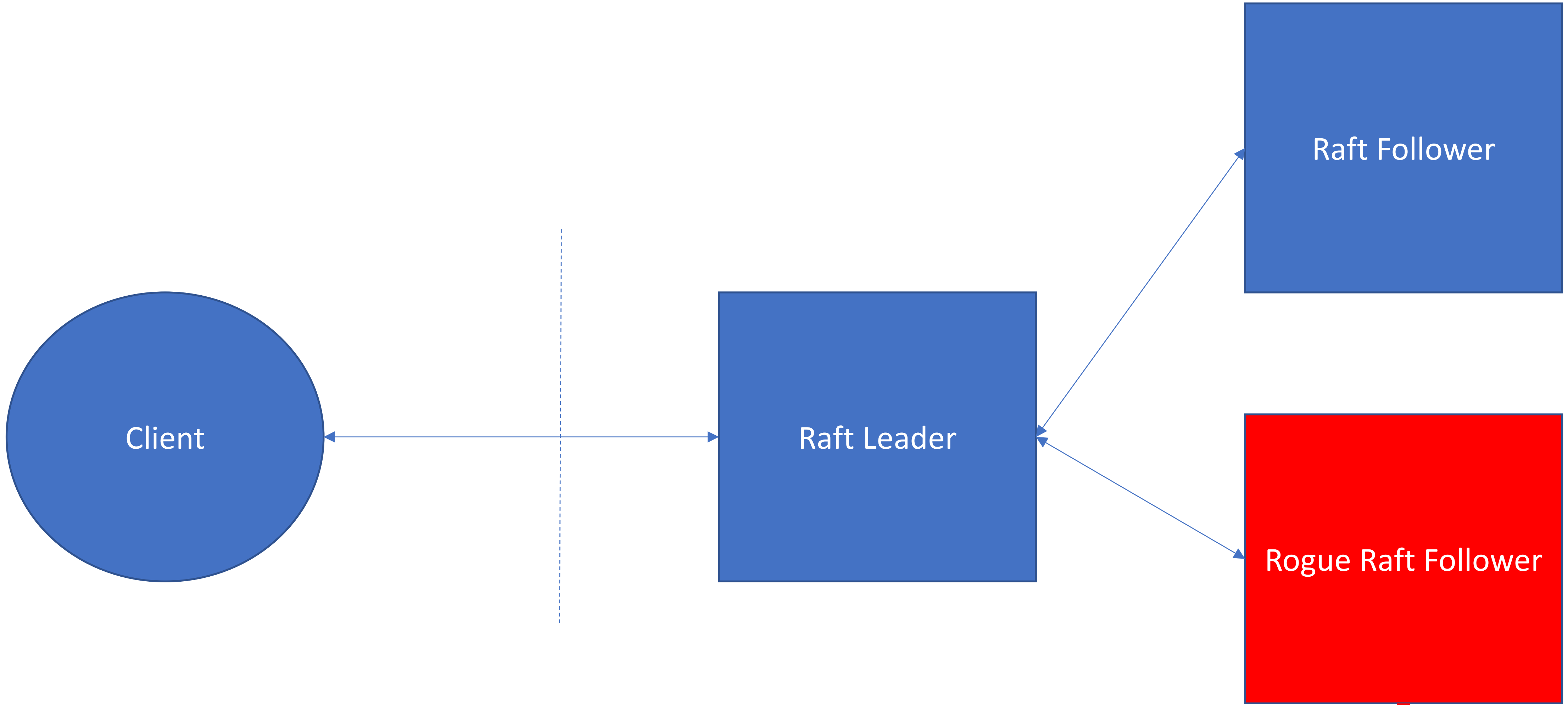
- What if a Raft cluster member doesn't behave like it is supposed to?
  - Corrupts or changes log files
  - Calls elections all the time
  - Leaks logs to third parties
- Byzantine failure sources
  - Bugs in software or configurations
  - Hardware and networking issues
  - Malicious cluster members
- These types of problems are known as "the Byzantine Generals Problem"


Flavius Belisarius

Problem #1: Leaky Follower Problem

What if a malicious follower joins the pool, masquerading as a legitimate follower?

It could leak log messages to a third party

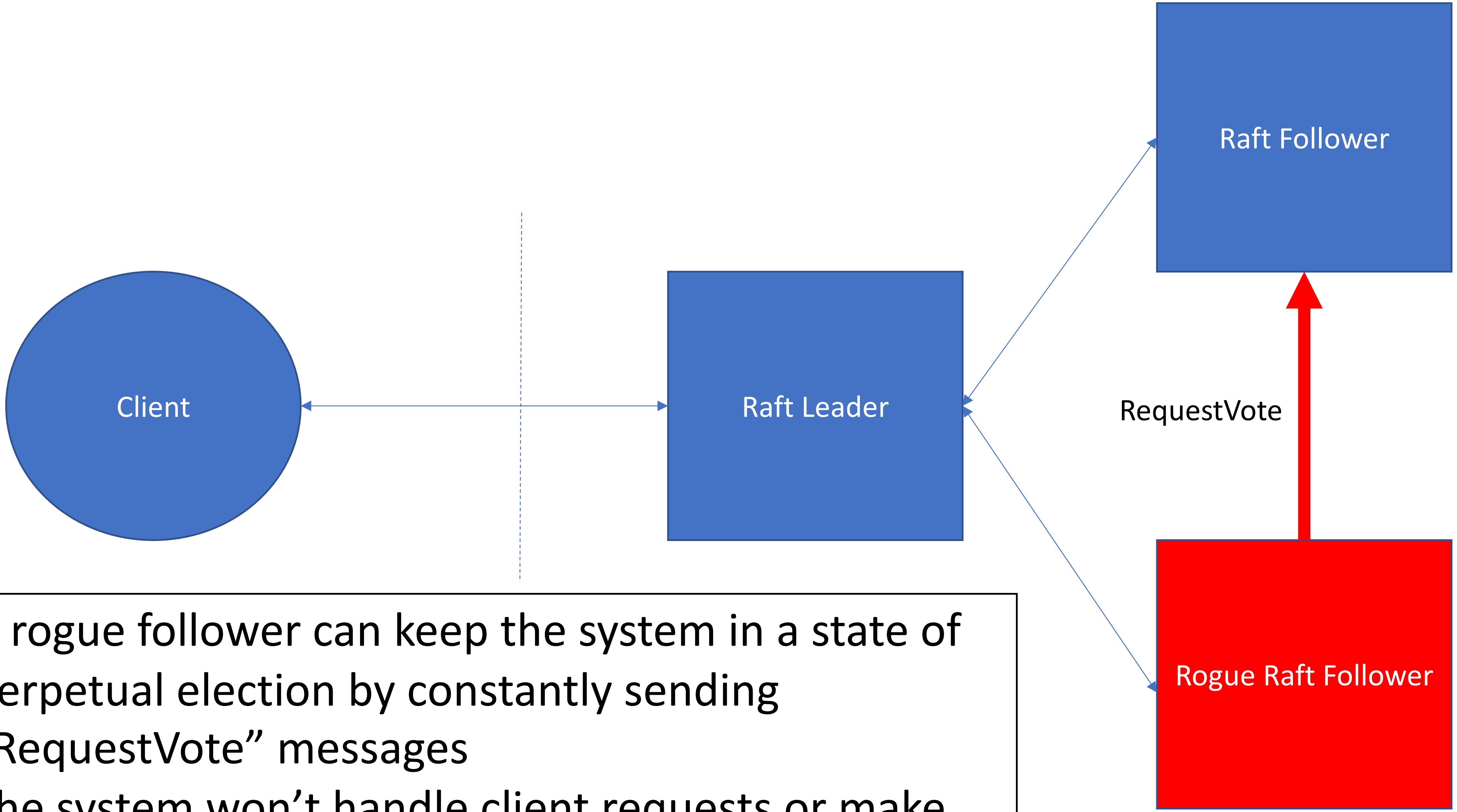


- How can the system know that the rogue follower has been injected into the cluster

Problem #2: Disruptive Follower Problem

In Raft, any follower can attempt to become a leader at any time

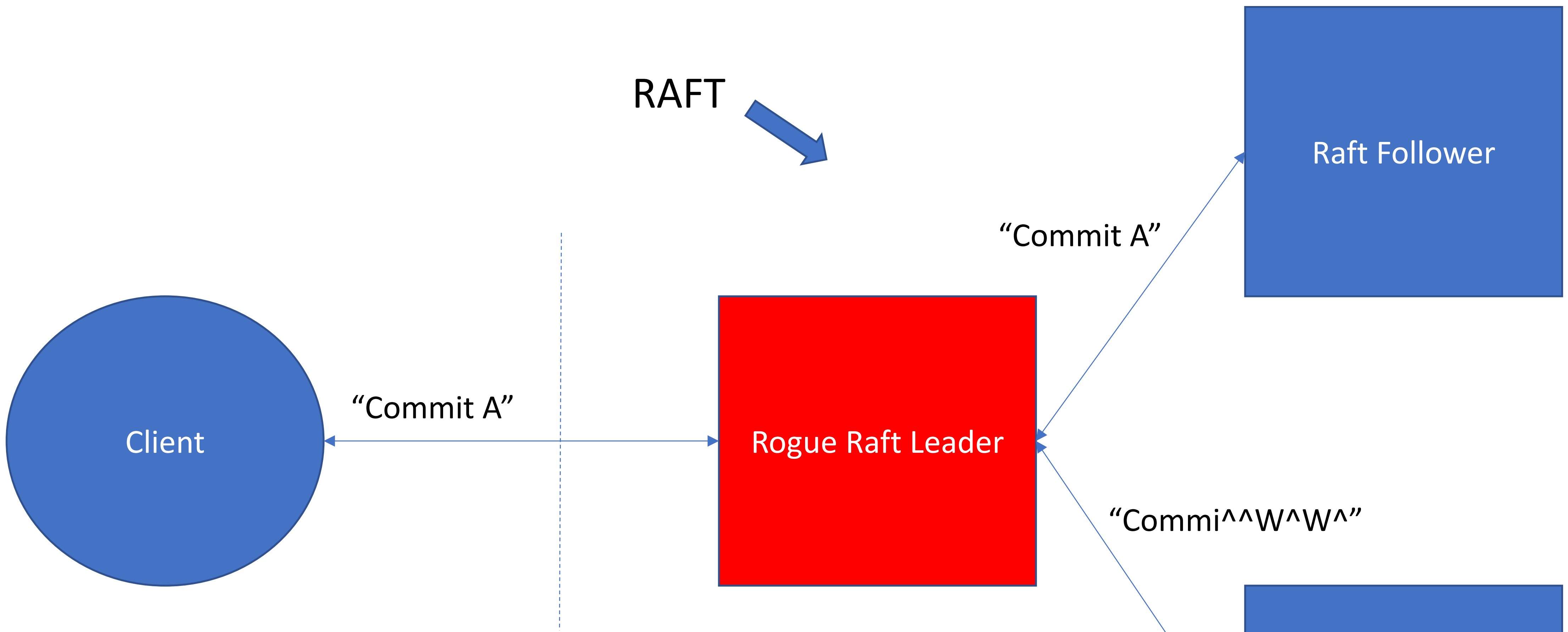
This should only be triggered by followers not receiving heartbeat messages from the leader within a set timeout period



- A rogue follower can keep the system in a state of perpetual election by constantly sending "RequestVote" messages
- The system won't handle client requests or make commits during an election

Problem #3: The Bad Boss Problem

- 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
- Followers 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

Strategies for Byzantine Fault Tolerance look a lot like network security

Authentication, message integrity, nonces, etc.

Strategy #1: Public Key Infrastructure

- Sender uses **private key** to cryptographically sign messages
- 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

Message Signatures with Key Pairs

Digital signing can be used to authenticate the message source and verify integrity.

Public/private keys are the standard way to do this: TLS and mutual authentication

Public Key Infrastructure and Messaging Signing Can Stop Impersonation Problems

Strategy #2: 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

Strategy #3: Election Verification

The would-be leader must prove to the other servers that it won the election

It does this by sending a secure message containing the signed, hashed votes that it received.

A follower can verify the signatures on the votes and the message integrity hash

Strategy #4: Commit Verification

Followers broadcast their AppendEntries response message to the entire cluster, not just the leader

Followers can ensure that everyone is getting the same message

Followers can confirm consensus

Strategy #5: Lazy Voters

The follower doesn’t blindly trust the RequestVote method from a candidate

Followers only vote in elections if they believe the leader is faulty.

Example: the follower also hasn’t received leader heartbeats

Example: the leader is detected as being rogue

BFT Raft Challenges

- Too much security can impact performance
- Increasing the complication of standard operations like leader election can decrease availability and have other unintended consequences
- “Practical Byzantine Fault Tolerance” is the place to get started if you want to learn more.
    - Castro, M. and Liskov, B., 1999, February. Practical Byzantine fault tolerance. In *OSDI* (Vol. 99, No. 1999, pp. 173-186).

BFT Raft Takeaways

- Hashing, signing, and encryption are ubiquitous in distributed system security.
- When choosing a Control Plane technology, know which questions to ask about security
- Security isn't free, so know the performance costs
- Security isn't foolproof, so think through your risks and have a plan for emergencies