Why is Texas having such a hard time restoring power?

What's different about the way Perseverance's landing method compared to previous rover missions? Why is it different?

Mental Models vs. Tutorials

- People who have good mental models ask better questions
- Mental models are always simplifications of real things
- Be ready to replace your mental model with a better one when it no longer answers your questions.

Log-Centric Distributed Systems

Motivations and an overview of the Raft protocol

Some Highly Recommended References

- “The Log: What every software engineer should know about real-time data's unifying abstraction”
  - Jay Kreps
  - https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
- “In search of an understandable consensus algorithm”
  - Diego Ongaro, John K Ousterhout
  - https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf
- “The RAFT Consensus Algorithm”
  - http://www.andrew.cmu.edu/course/14-736/applications/ln/riconwest2013.pdf
  - Diego Ongaro and John Ousterhout

What Are Some Properties of Cloud-Native Distributed Systems?

What's your mental model?

Some Properties of Cloud-Native Services

They are fault-tolerant, can keep working even if part of the system is down.

They can smoothly scale up or down to handle different loads

They are dynamic: minimal static configuration and hard coding

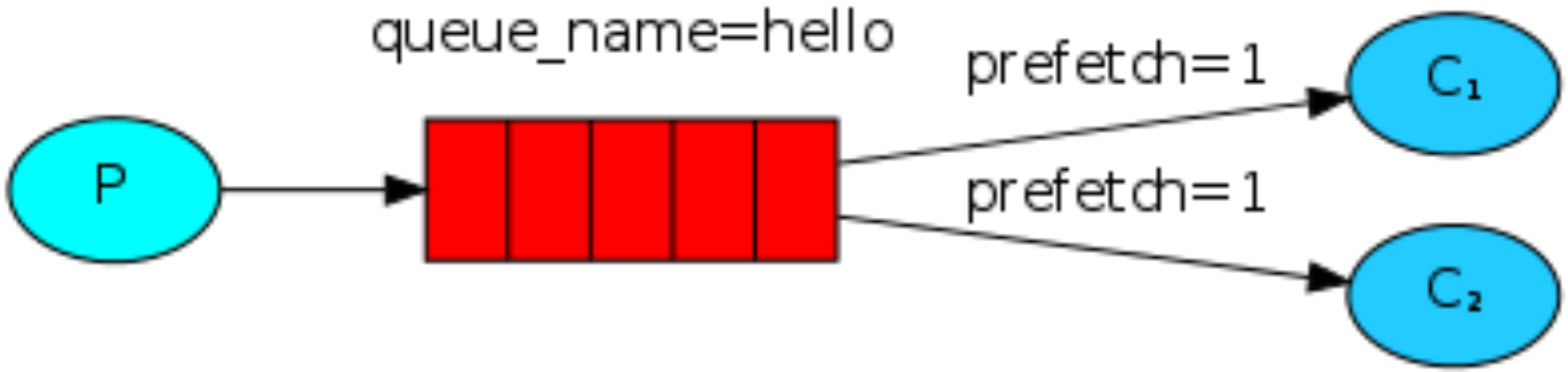
Two General Classes of Cloud-Native Services

Stateless

Stateful

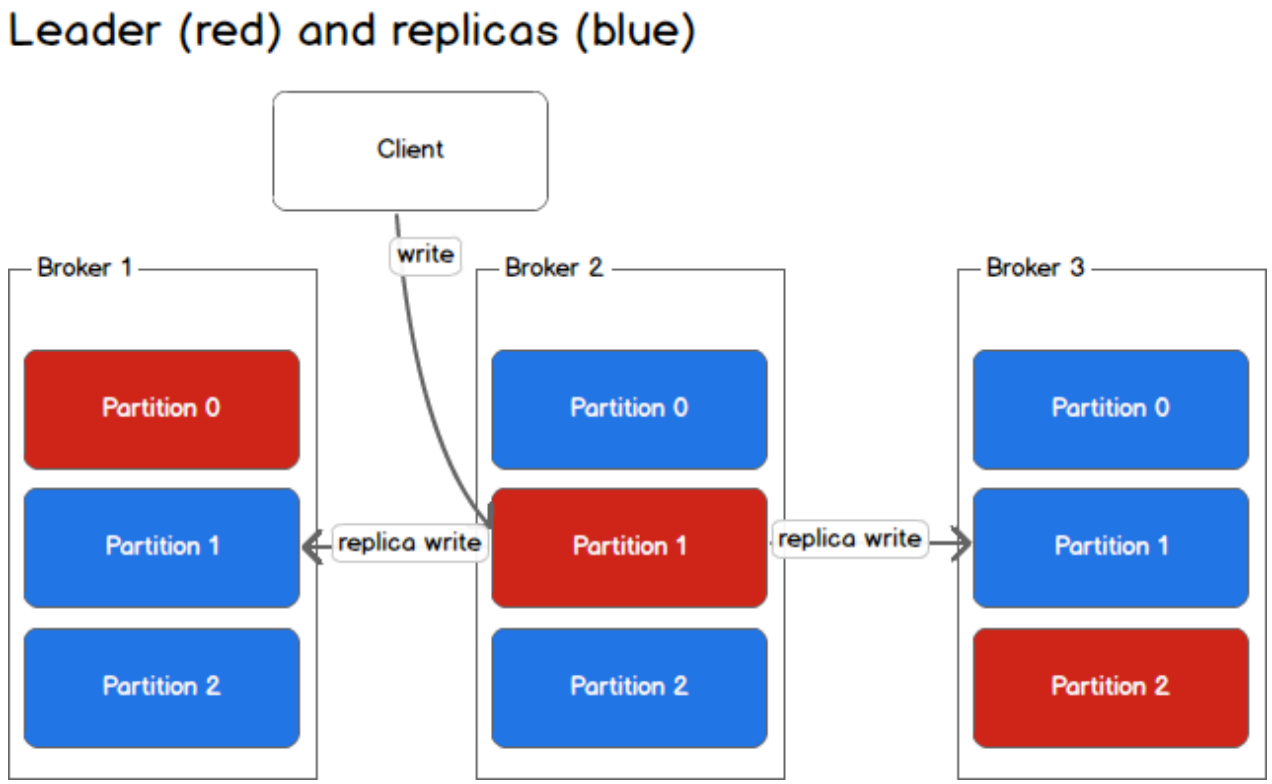
A Stateless Example: Work Queue Producers and Consumers

- Producers are stateless: fire and forget
- Consumers are mostly stateless: after C1 finishes a job, it can do new work with no memory of the previous job.
- **Scaling out producers and consumers is trivial, as long as the broker scales**
- There is state; it's in the broker



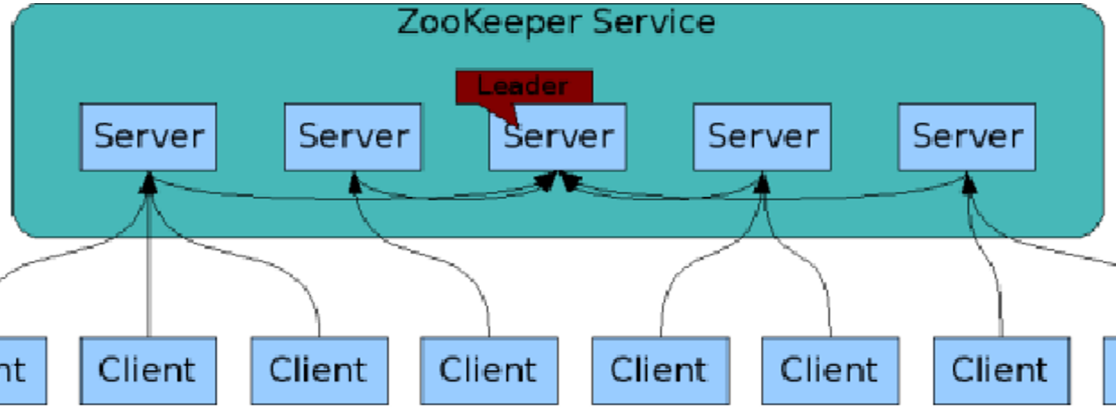
A Stateful Example: Leader-Follower

- The broker service has memory (state)
- It must recover from failures
- It should resist failures
- Strategies?
  - Use logs to capture system state
  - Store logs carefully
  - Distribute logs to potential replacements



This is not as scalable as stateless services. Why not?

The ZooKeeper Service



Zookeeper is itself an interesting distributed system

These are microservices

- ZooKeeper Service is replicated over a set of machines
- A leader is elected on service startup
- Client can **read** from any Zookeeper server.
- **Writes** go through the leader & need majority consensus.
- If a leader goes offline, the other servers can pick a new leader

https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription

Logs and Service Mesh Control Planes

Consul, ETCD, and Zookeeper manage information in distributed systems

You can use them to implement a control plane for your service mesh (microservices)

Consul and ETCD use a protocol called RAFT

Let’s Contrast Log-Centric and Queue-Centric Approaches

Queues vs. Logs

- These are similar data structures: First in, first out
- Important difference: logs are persistent while queue entries are transient

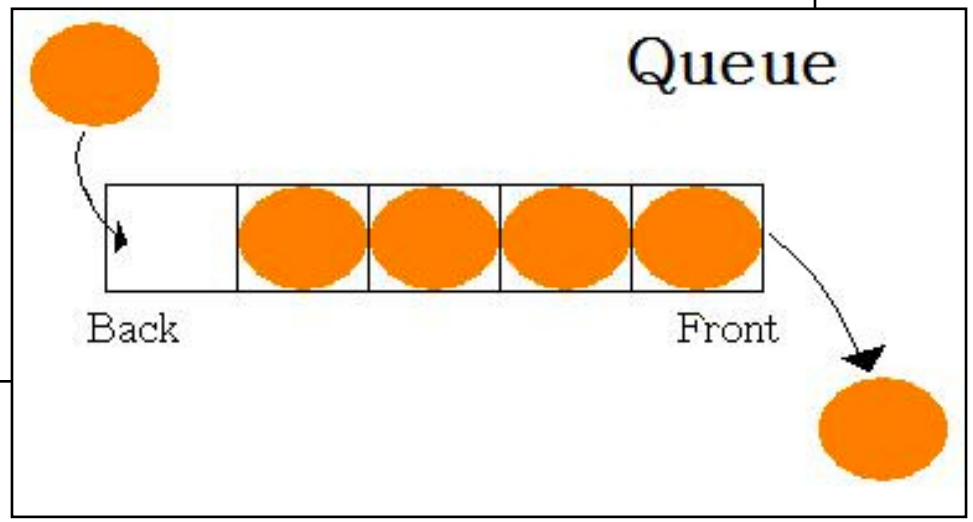
Log-Based Systems

- Client is responsible for knowing the last entry it consumed
- The broker just keeps track of the log records
- Stateless, or REST-like, interactions with clients (idempotent)

Message Queues

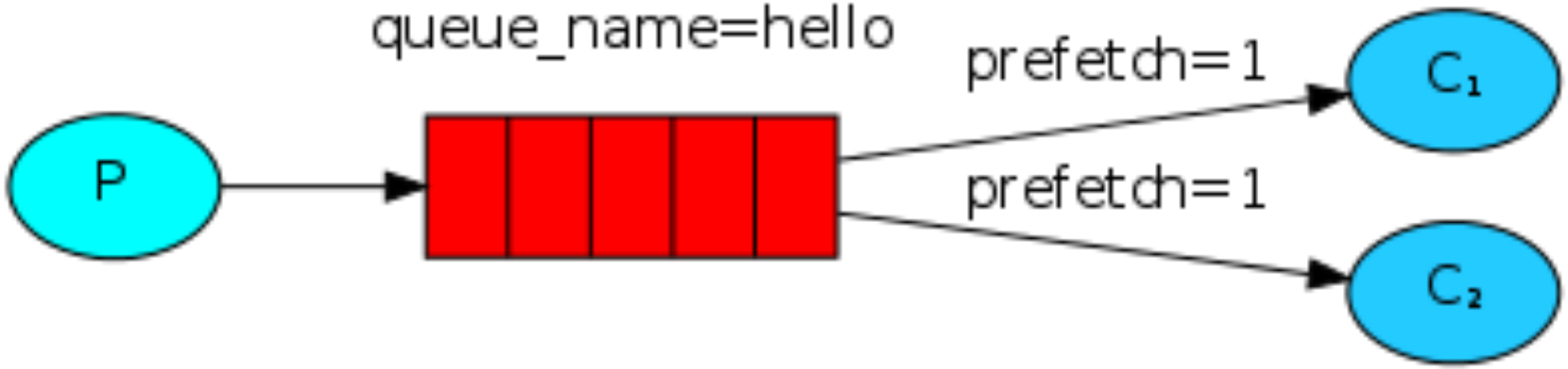
- Message Queue: a data structure containing a **consumable** list of entries.
  - Publishers create entries
  - Brokers manage the queues
  - Consumers consume entries
  - Entries are removed when they are consumed.

Queues are not logs. The state of the queue is something that the broker must carefully manage.



A Queue-Centric Design

- RabbitMQ's Work Queue Tutorial Example
- A publisher puts work in a queue
- The broker distributes it to consumers in round-robin fashion
- The consumers send ACKs after they have processed the message
- Consumers can limit the number of messages they receive.



Queue-Based Systems

- The broker needs to know if the message was delivered and processed correctly
- Is it safe to delete an entry? Must use ACKs nor NACKs
- The broker needs to keep careful track of its consumers.

Issue	Solution
Consumer crashes	Broker detects the crash using a **heartbeat.**
Consumer is very slow	Heartbeat detects that the consumer is alive but taking a very long time to send an ACK. Solution: use a **time out**.
Consumer is temporarily inaccessible	Consumer A doesn't crash but the heartbeat fails. The broker resends the message to Consumer B. Then network returns and Consumer A sends the ACK. The message got processed twice.
Broker is temporarily inaccessible	The broker's host server is temporarily off the network. The broker thinks all un-ACK'd messages are lost and so re-queues them. It will want to redeliver them when it detects consumers are available again, but then a cascade of ACKs will arrive. How do you handle this?

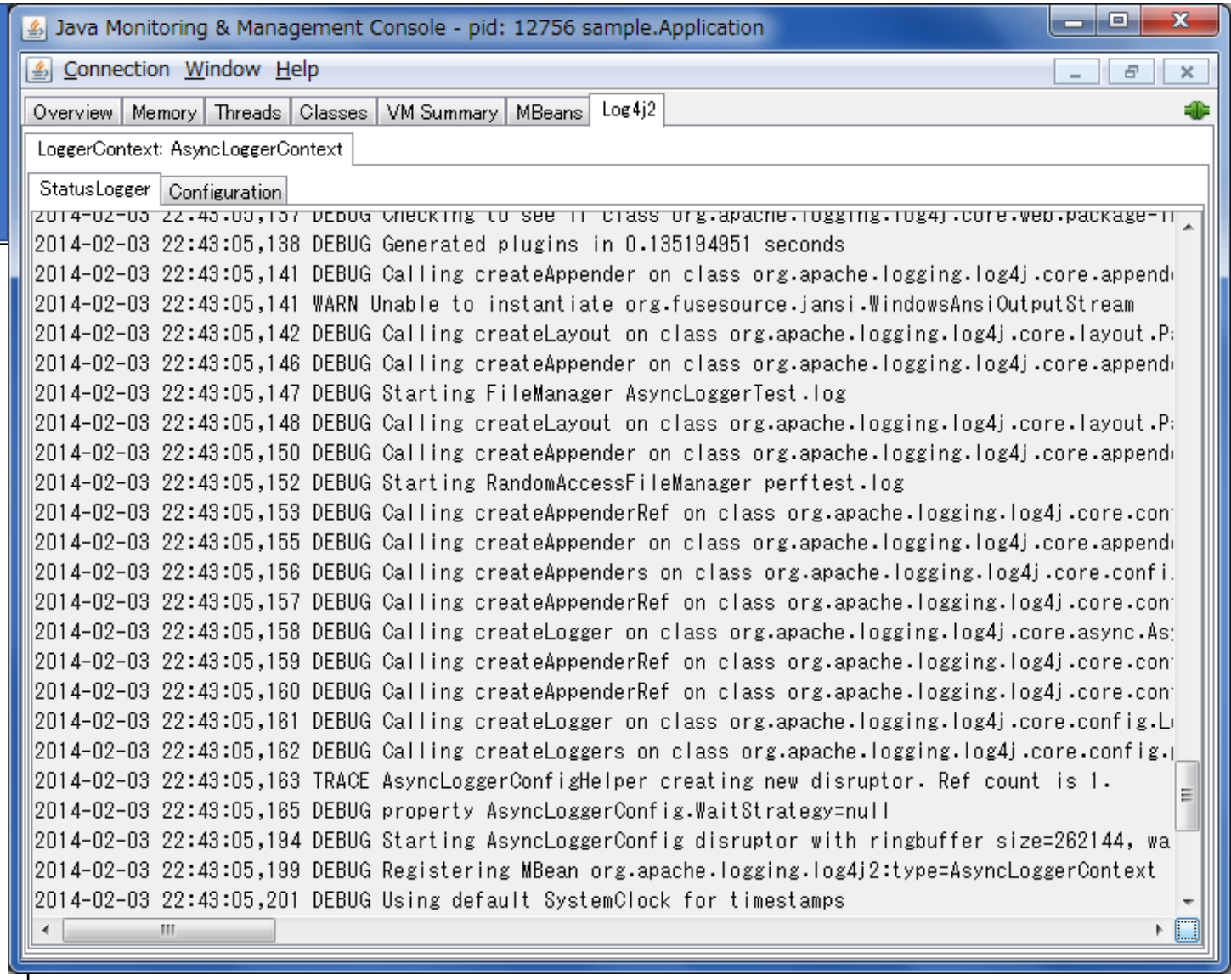
Some issues with detecting failed message delivery

Towards a Log-Centric Architecture:

But first, what do we mean by logs? Application Logs, Queues, and State Logs

Application Logs

- The info, warning, error, and other debugging messages you put into your code.
- Very useful for detecting errors, debugging, etc.
- Human readable, unstructured format
- **This is NOT a state log**



The Other Type of Log: System State Records

- Logs: records of state changes in a system
  - **Machine readable**
  - Distributed logs -> distributed state machines
- We typically want to consolidate these stateful logs in one part of the system
  - One very robust, scalable part of the system

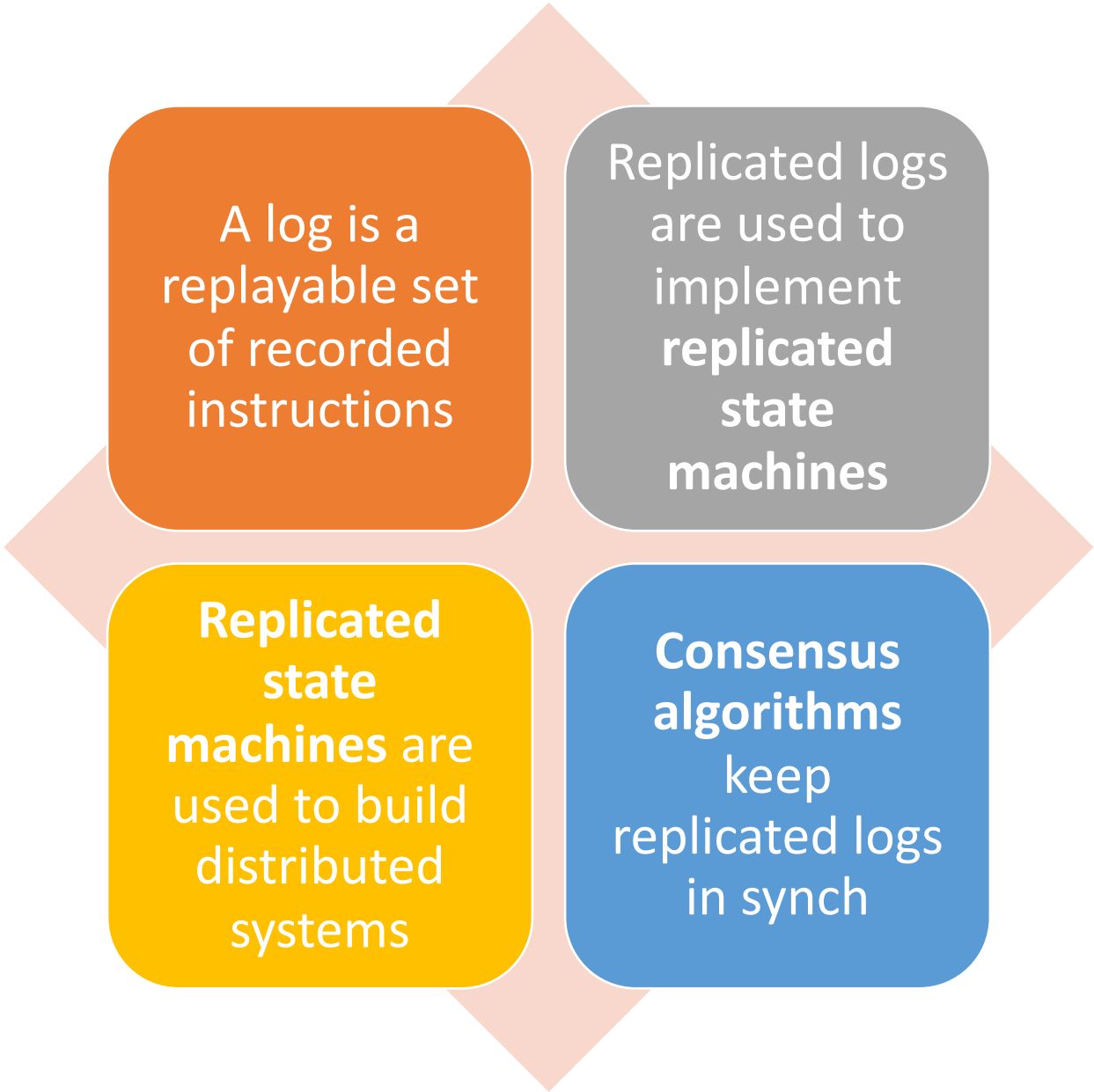
Kirk records a message in case Spock needs to take over.

Example Log: MySQL Dump

- You can use MySQL's dump command to create a restorable version of your DB.
  - These are logs
- What if you needed to restore lots of replicated databases from the same dump?

```
CREATE TABLE IF NOT EXISTS `mg_oro_analytics_data` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT 'Id',
  `period` datetime NOT NULL DEFAULT '0000-00-00 00:00:00' COMMENT 'Period',
  `store_id` int(11) DEFAULT NULL COMMENT 'Store_id',
  `customer_id` int(11) DEFAULT NULL COMMENT 'Customer_id',
  PRIMARY KEY (`id`,`period`),
  KEY `IDX_MG_ORO_ANALYTICS_DATA_ID` (`id`),
  KEY `IDX_MG_ORO_ANALYTICS_DATA_CUSTOMER_ID_PERIOD` (`customer_id`,`period`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COMMENT='mg_oro_analytics_data'
/*!50100 PARTITION BY RANGE (TO_DAYS(period))
(PARTITION p_2014_01_04 VALUES LESS THAN (735602) ENGINE = MyISAM,
 PARTITION p_2014_02_04 VALUES LESS THAN (735633) ENGINE = MyISAM,
 PARTITION p_2014_03_04 VALUES LESS THAN (735661) ENGINE = MyISAM,
 PARTITION p_2014_04_04 VALUES LESS THAN (735692) ENGINE = MyISAM,
 PARTITION p_2014_05_04 VALUES LESS THAN (735722) ENGINE = MyISAM,
 PARTITION p_2014_06_04 VALUES LESS THAN (735753) ENGINE = MyISAM) */ AUTO_INCREMENT=358 ;
```

Logs and State Machines



Property	Description
Ordered	Logs record state changes, so they must be in sequence (indexed)
Correct	We are confident that a log entry was recorded correctly
Complete	There are no missing records between the first and last entry
Machine Readable	Log entries are serialized data structures, operations
Persistently Stored	The logs are stored on highly stable media
Available	Applications that depend upon the logs can get them

Some Desirable Properties of State Logs

High Availability Requires Log Replication

- There must be a failover source of logs if the primary source is lost.
- Weak consistency may be OK: a replica may be behind the primary copy, but otherwise, they match exactly
- **Consensus algorithms** address this problem
- **Paxos** is the most famous consensus algorithm
  - Lamport, L., 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, *16*(2), pp.133-169.
- But it is hard to understand and implement

Raft Consensus Protocol

- Raft has been developed to provide a more comprehensible consensus protocol for log-oriented systems
- Several implementations
  - https://raft.github.io/
  - See also http://thesecretlivesofdata.com/raft/
- It resembles but is simpler than Zookeeper's Zab protocol
- Ongaro, D. and Ousterhout, J.K., 2014, June. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference* (pp. 305-319)

I'll use some slides from https://raft.github.io/

Remember: Raft is a protocol, not a piece of software

Software like Consul and ETCD use the protocol.

Zab, Viewstamped Replication, and Paxos are alternative protocols

Properties of Consensus Systems

Property	Description
Safety	Never return incorrect results to queries
Availability	The system functions as long as a majority of servers are operational
Ordered Messages	Message order does not depend on system clocks; slow networks are not a problem
Majority Commits	Logs are recorded if a majority of members accepts the write. Don’t need to wait on complete consensus.

Raft Protocol Guarantees: Always True (1/5)

Election Safety: at most one leader can be elected in a given term.

Raft Protocol Guarantees: Always True (2/5)

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries.

Raft Protocol Guarantees: Always True (3/5)

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Raft Protocol Guarantees: Always True (4/5)

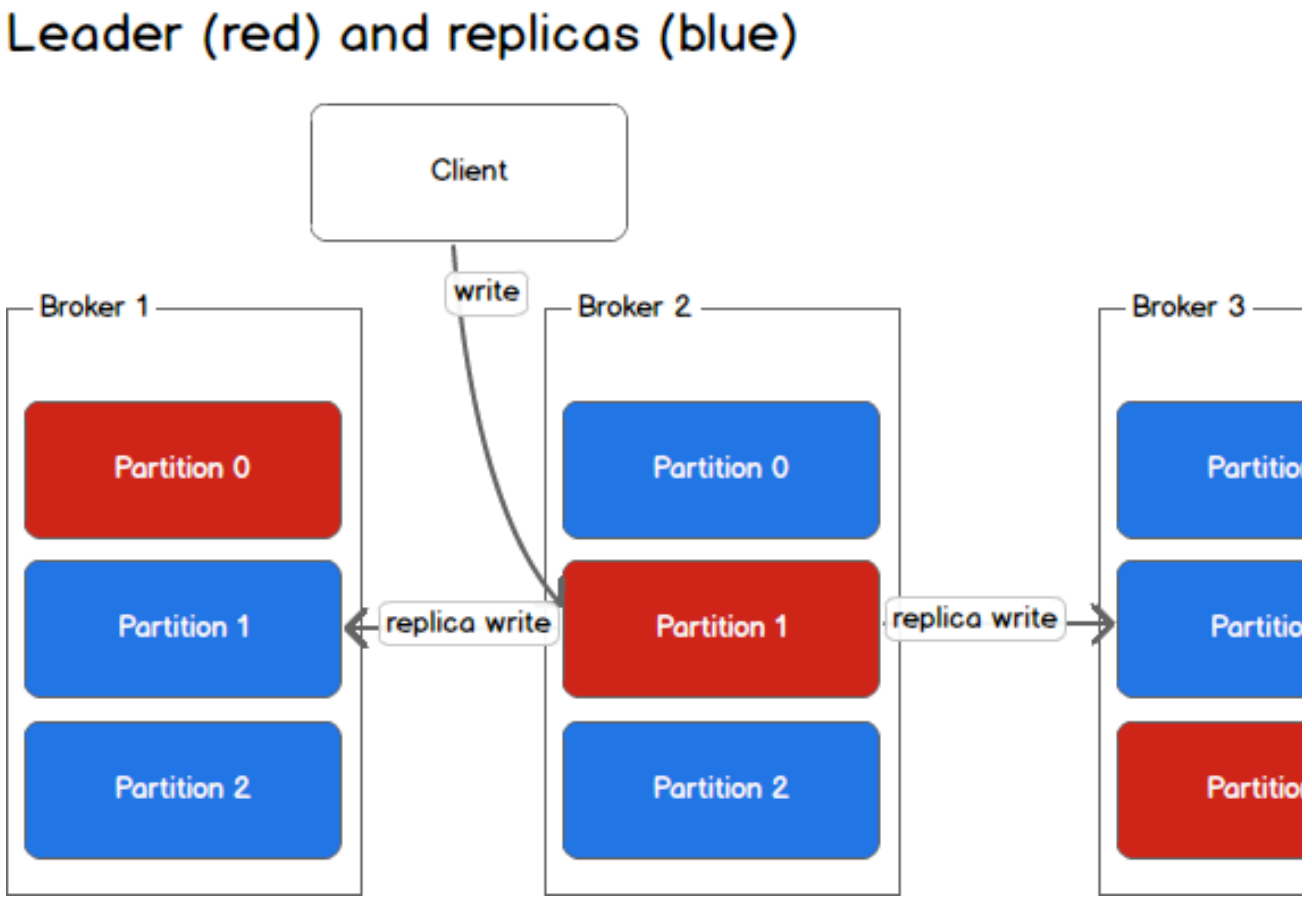
Leader Completeness: If a log entry is committed in a given term, then the entry will appear in the logs of leaders of future

Raft Protocol Guarantees: Always True (5/5)

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

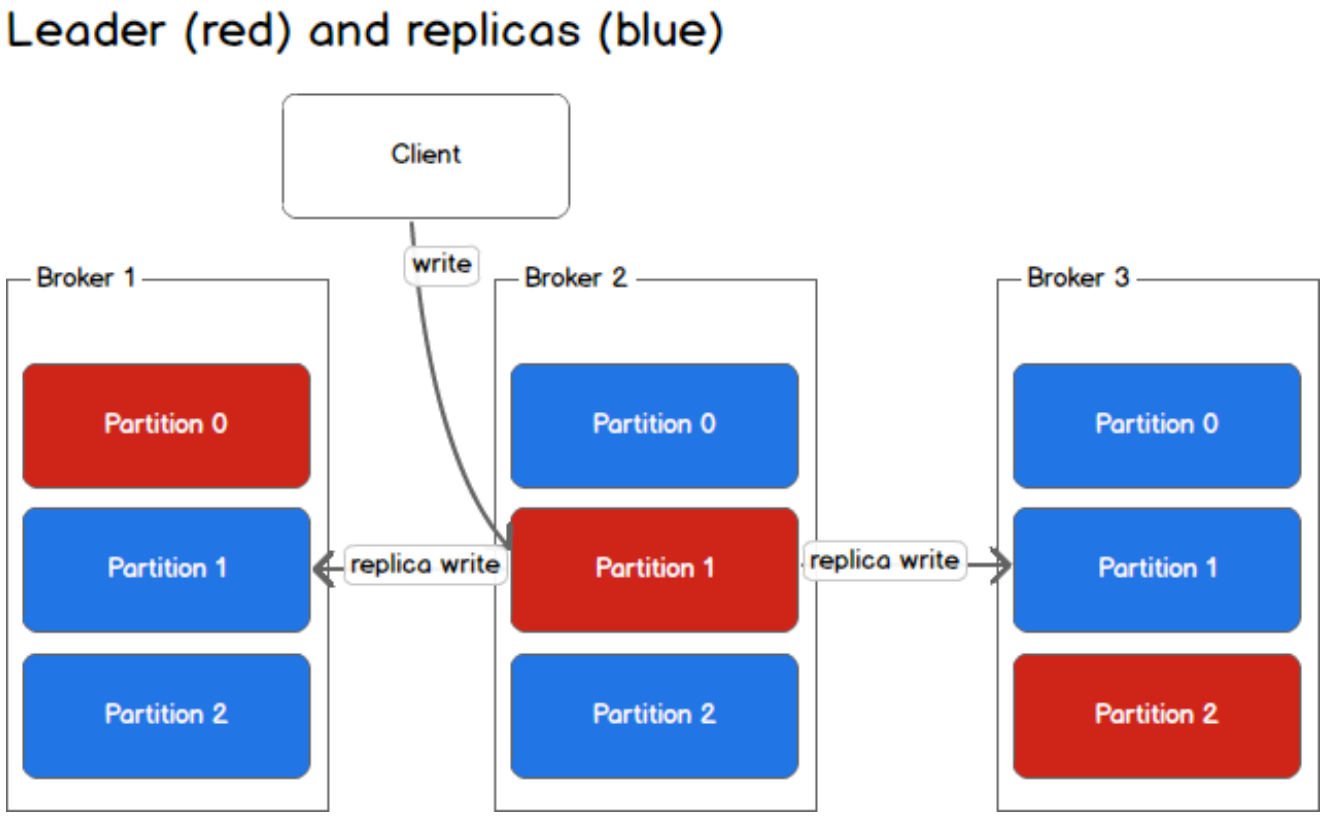
Raft Basics: Strong Leader and Passive Followers

- In Raft, the leader supervises all write operations
- The leader service/broker accept write requests from clients
- Follower-brokers redirect write requests from clients to the leader broker
- We saw this with Kafka and ZK



Raft Basics: Leaders Need Consensus

- The leader sends log updates to all followers
- If a majority of followers accept the update, the leader instructs everyone to commit the message.
- If a leader can't get consensus, it may abdicate
- Members choose a new leader through an election



Raft Achieves Eventual Consistency

A minority of followers can have fewer committed messages than the majority at any given time

But lagging members will have a subset of committed messages

Summary of Part 1

Use logs to record changes to your system.

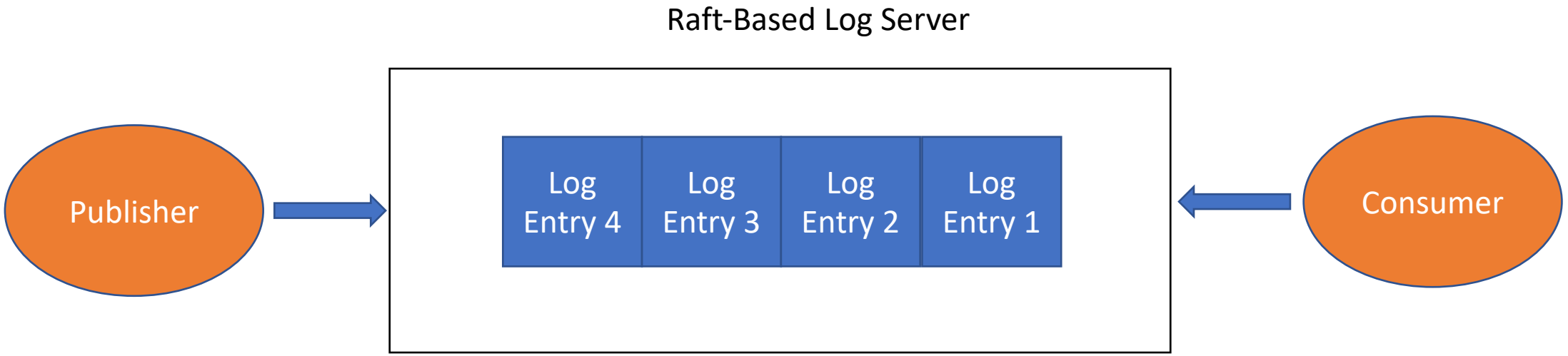
Centralized logs make it easy for the system have a universal, consistent, replayable record of how it evolved over time

Services that manage the central logs need to be correct, reliable, recoverable, and fault tolerant

The Raft protocol is a popular way to provide these guarantees

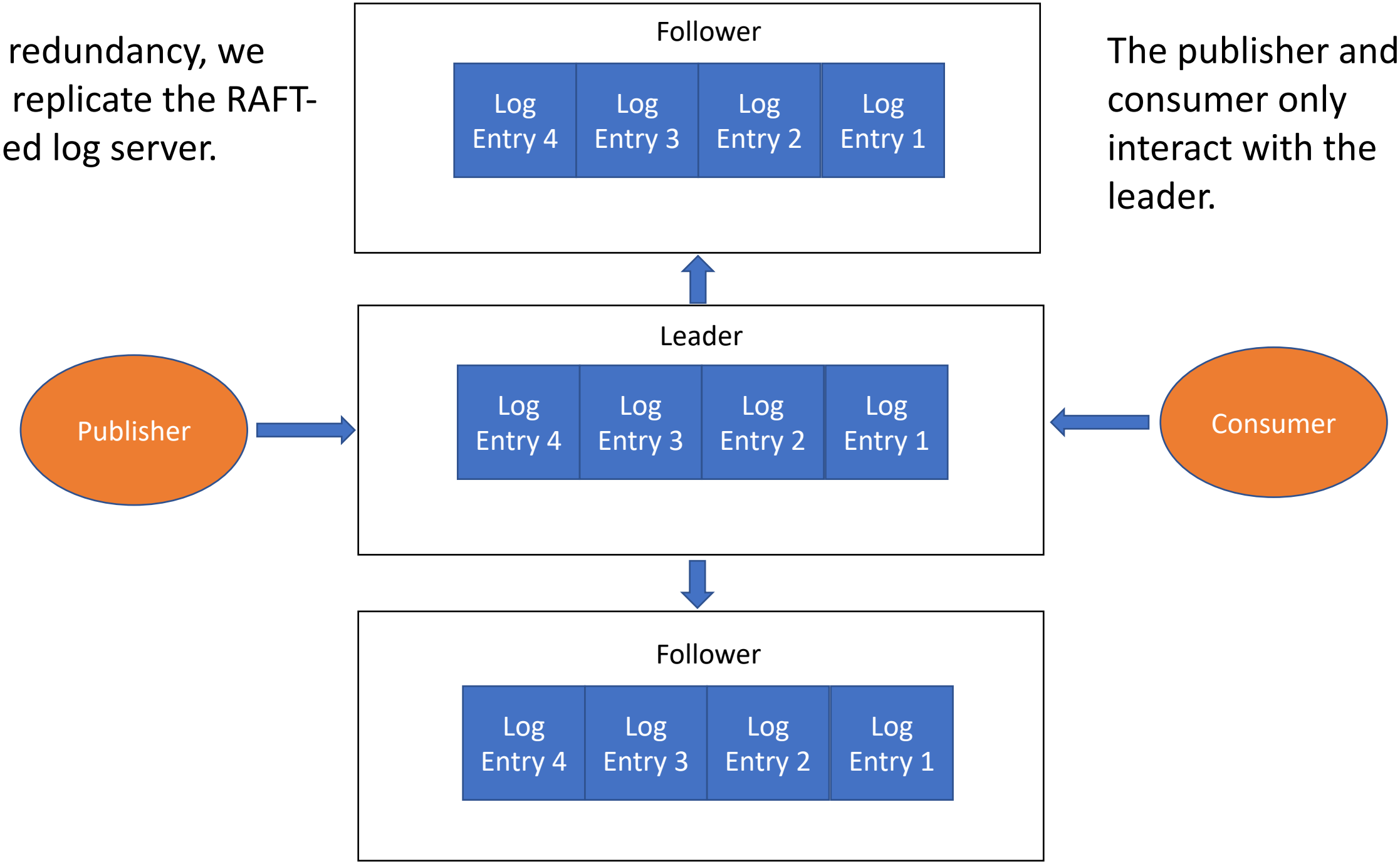
Part 2: Raft, in Detail

A Simple Raft Scenario



- A publisher appends entries to the Raft-based log server
- Each log entry contains commands, data, etc
- The log server stores the entries in order that they are received
- A consumer reads entries from the server

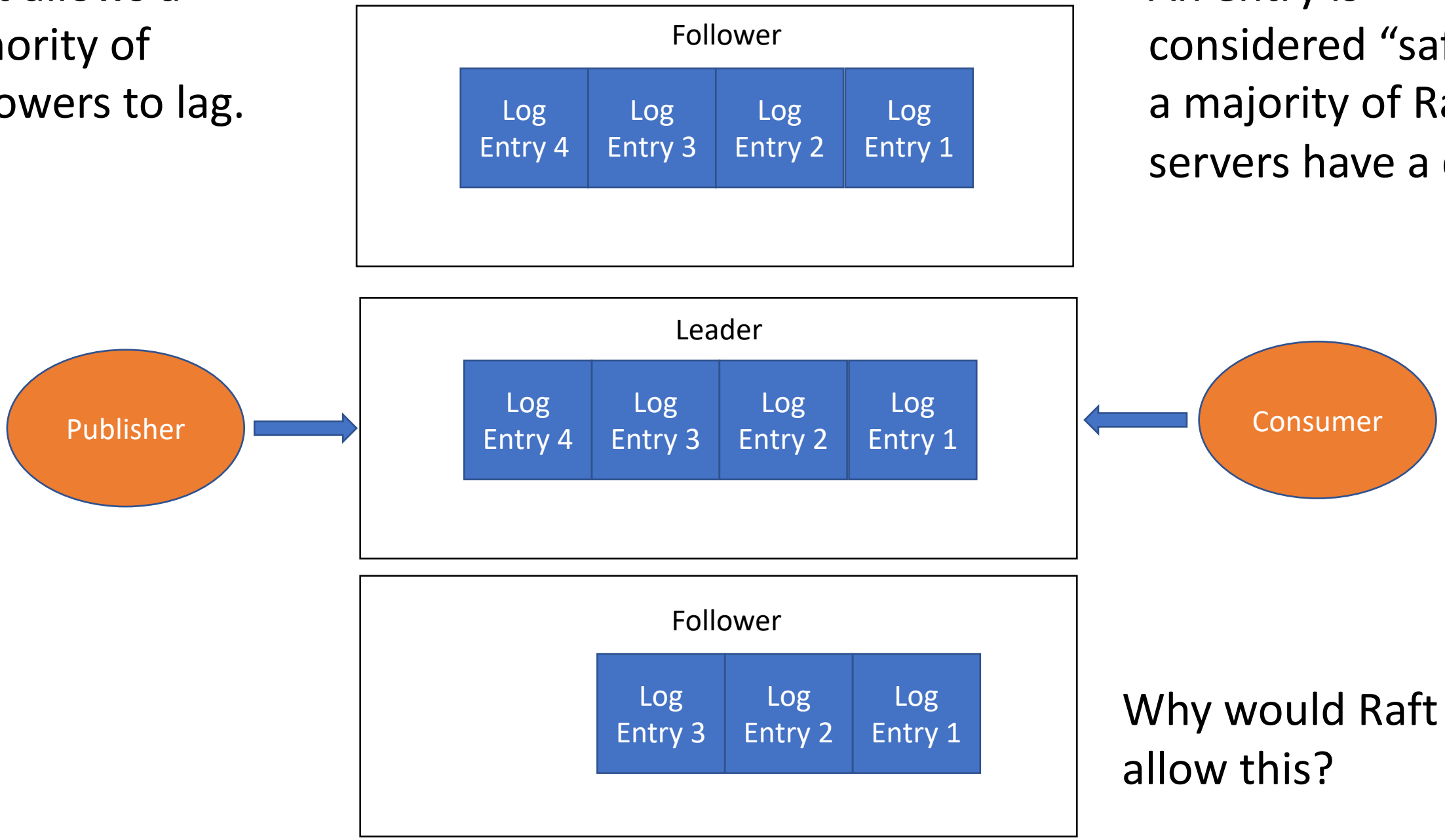
For redundancy, we can replicate the RAFT-based log server.



The publisher and consumer only interact with the leader.

Raft allows a minority of followers to lag.

An entry is considered “safe” if a majority of Raft servers have a copy



Why would Raft allow this?

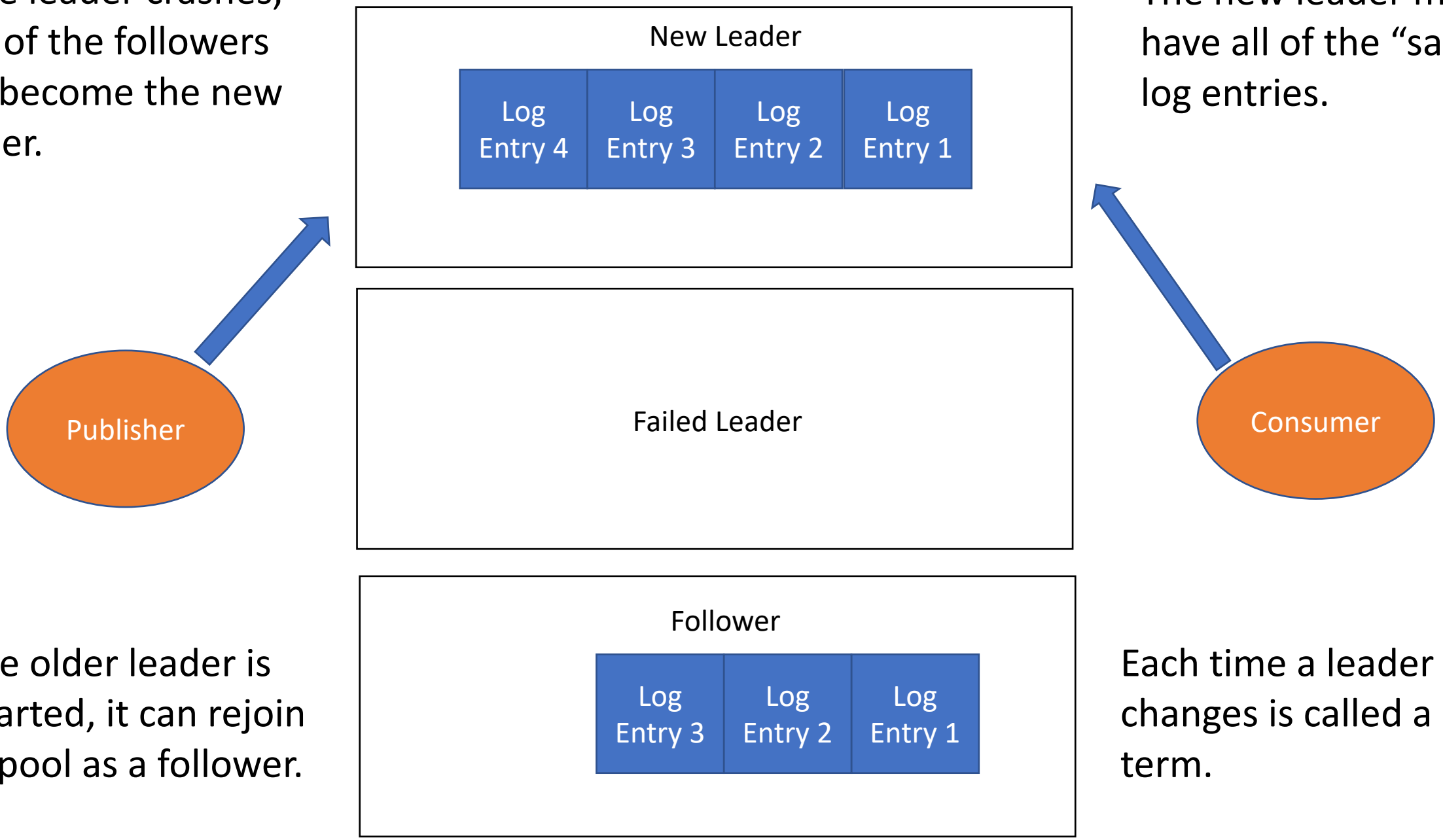
Why Allow Followers to Lag?

You can have more servers in the Raft cluster.

Publishers don’t have to wait for the log to be replicated on all followers, just most of them.

Having more servers in the Raft cluster makes it more fault tolerant.

If the leader crashes, one of the followers can become the new leader.



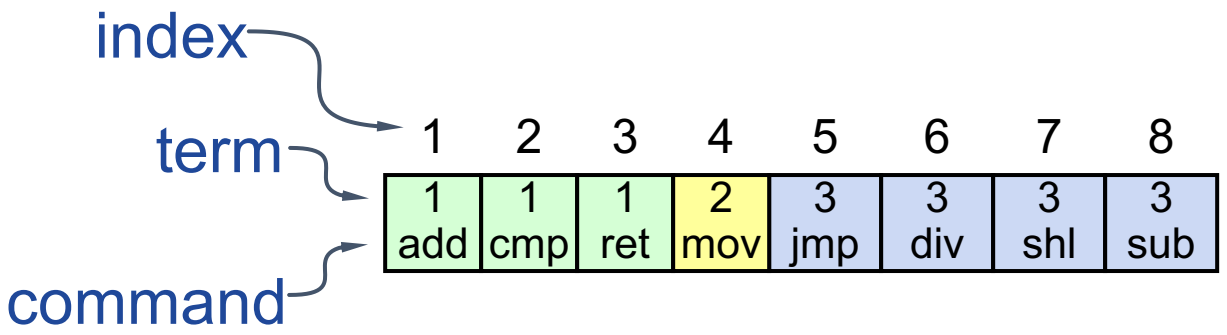
The new leader must have all of the “safe” log entries.

If the older leader is restarted, it can rejoin the pool as a follower.

Each time a leader changes is called a term.

This is called “consensus”

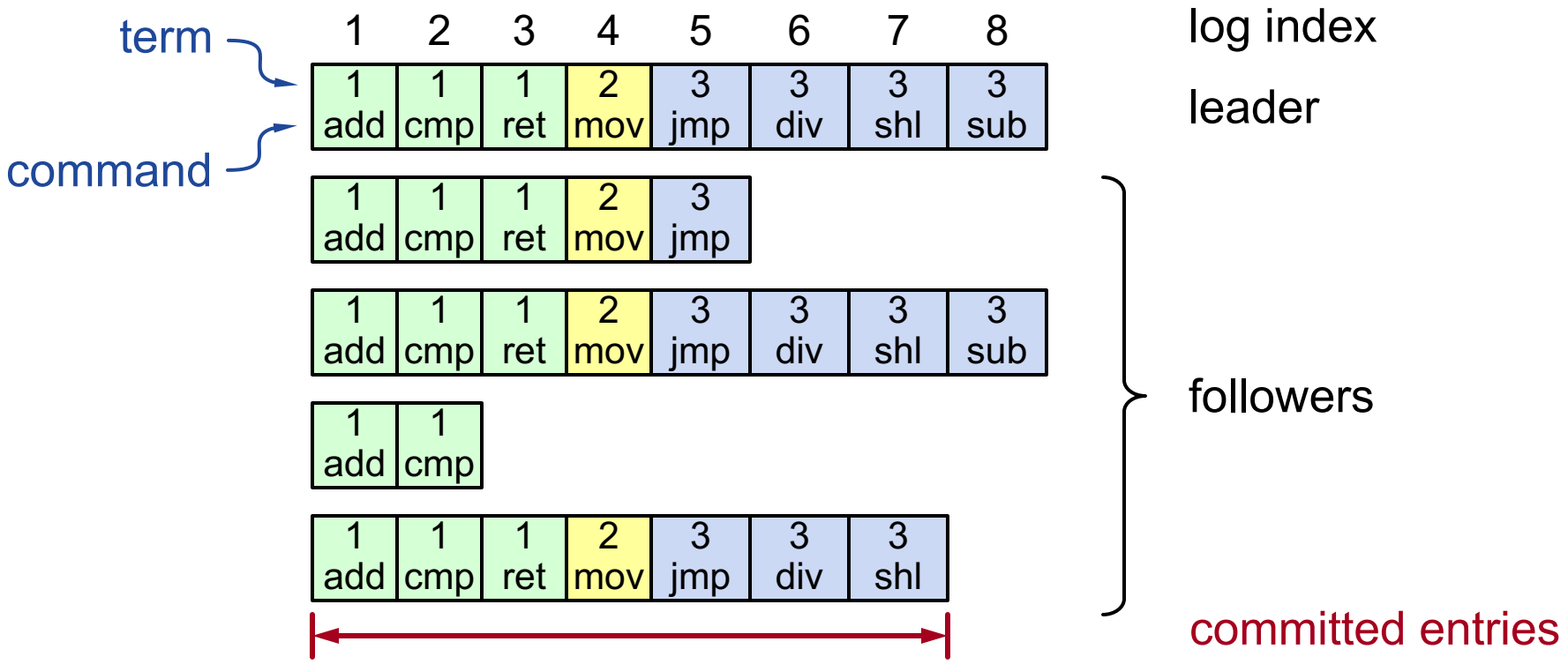
A RAFT Log Looks Like This



- Log entry = index, command, term
- Logs are stored on stable storage (disk).
- If the server crashes, re-read the log from disk

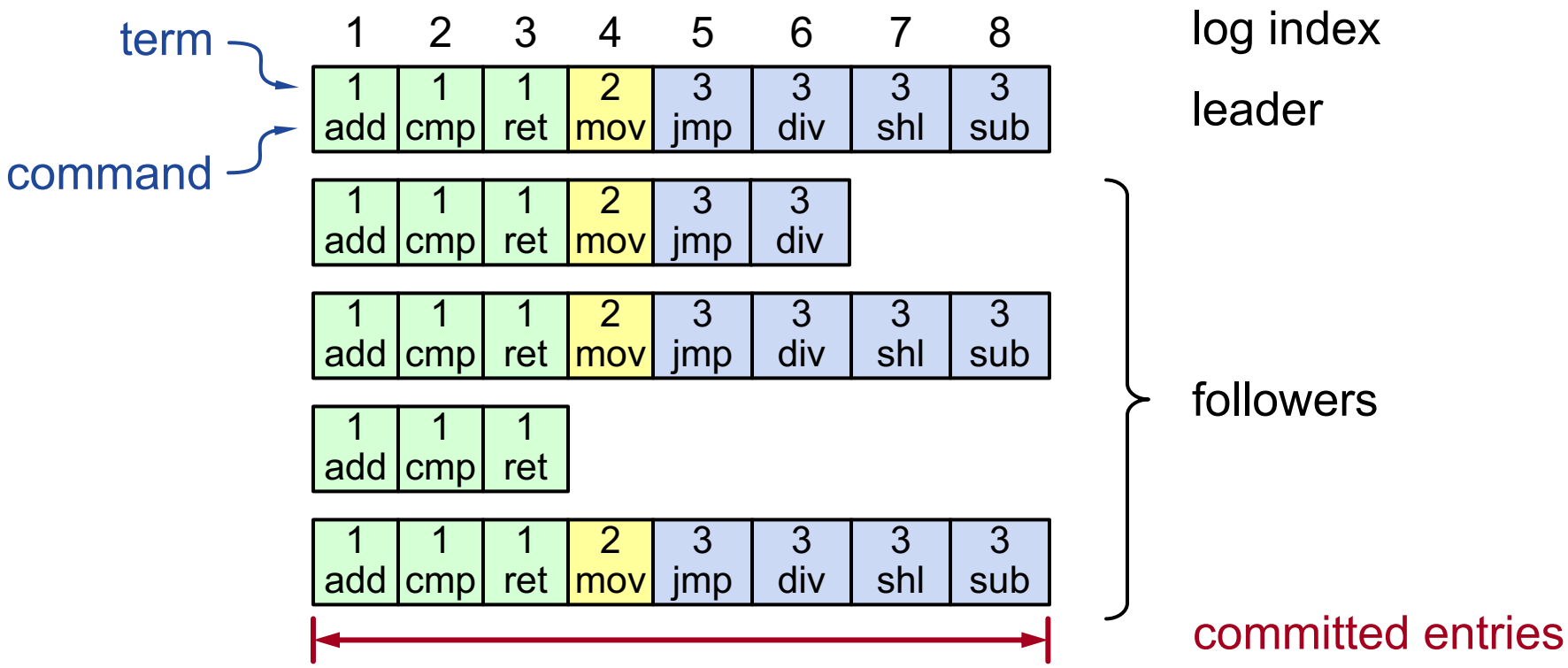
Let’s see what this looks like for five servers

Log Structure Snapshot



- **Log entry = index, term, command**
- **Log stored on stable storage (disk); survives crashes**
- **Entry committed if known to be stored on majority of servers**
  - Durable, will eventually be executed by state machines

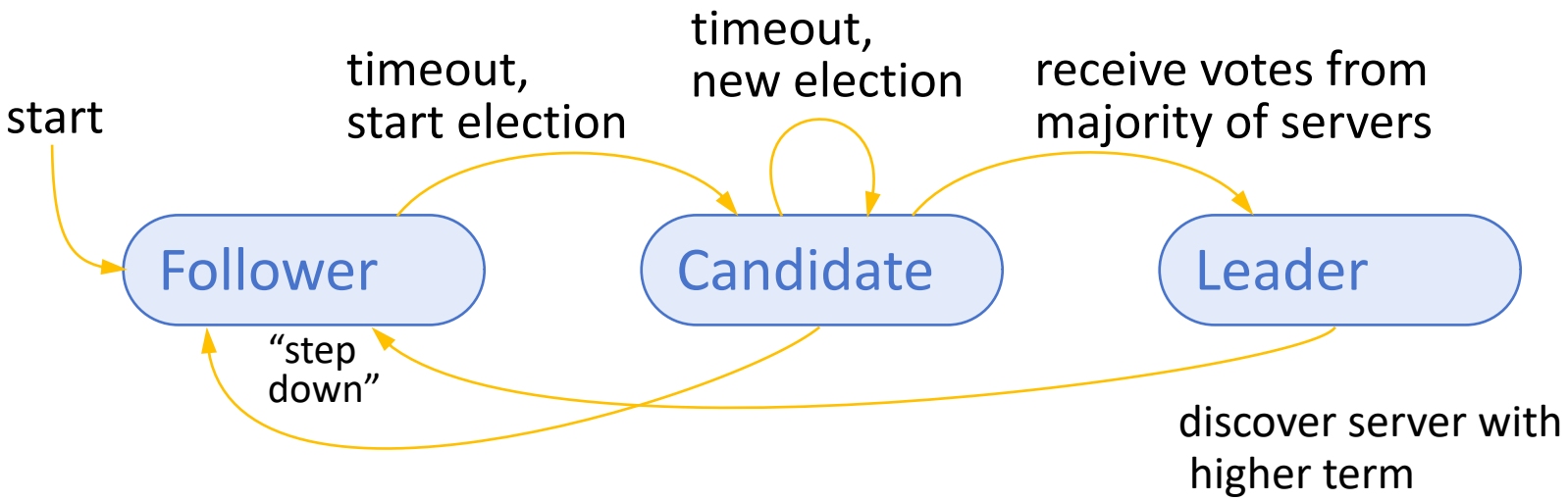
Log Structure Snapshot+1



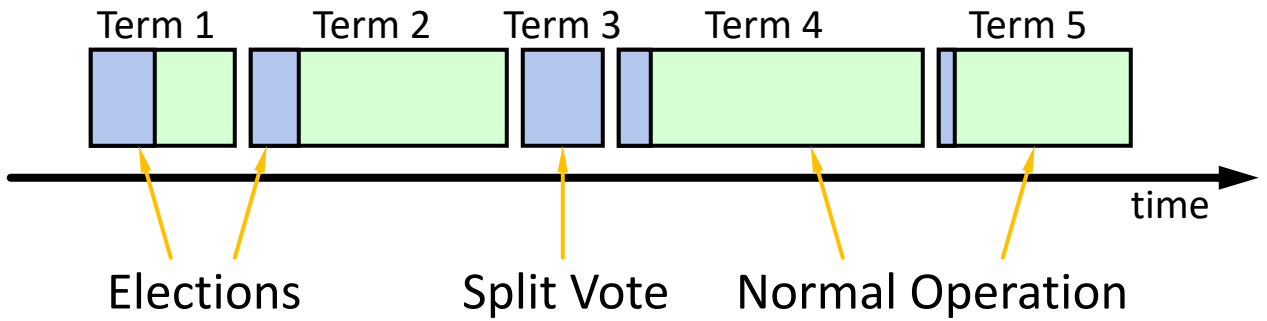
Leaders and Leadership Changes

- There is at most one leader at any time
- A system needs a new leader if the leader fails or becomes disconnected from a majority of followers: **heartbeat failures**
- New leaders are chosen by election from among the followers
  - Followers become candidates if they detect that a leader has failed
  - Only one candidate can win
- A **term** is the time period that a particular leader is in charge

Raft Server States:



Raft Time Evolution:



Key Raft Properties

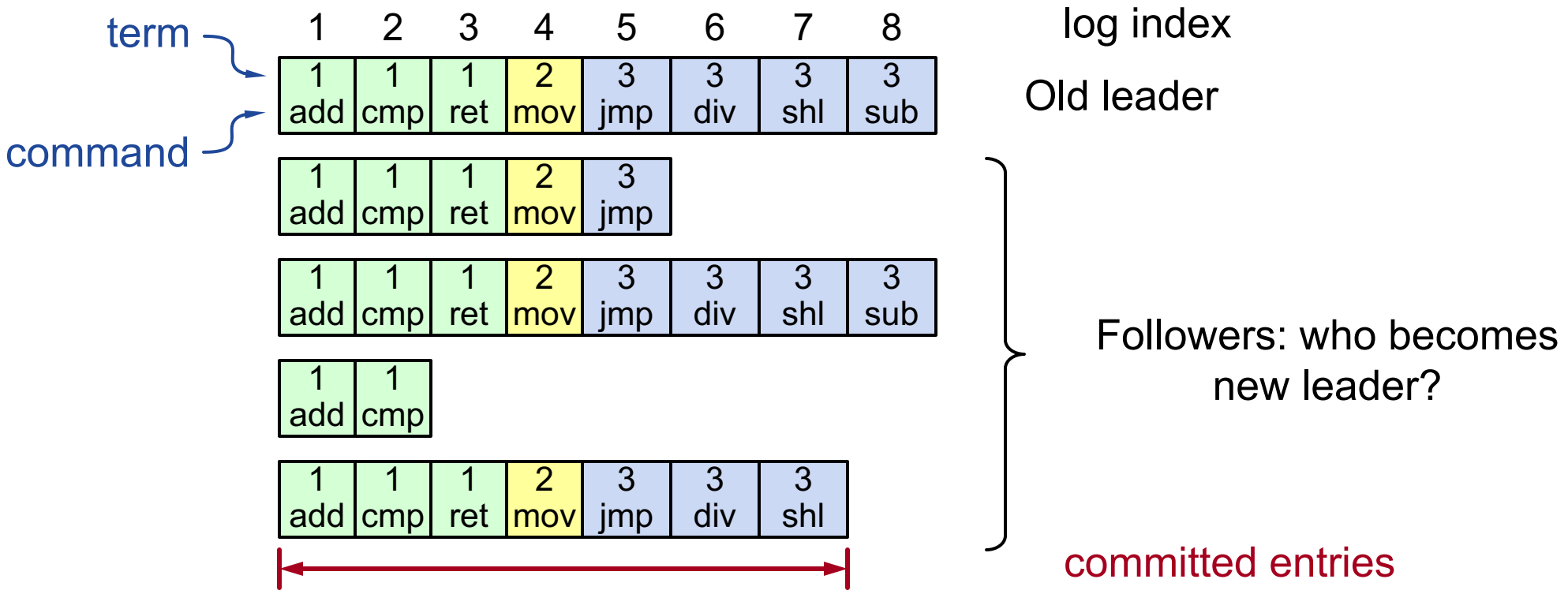
- **Election Safety:** At most, only one leader can be elected
- **Leader Append-Only:** Leaders can only append entries to the logs. They never change committed (safe) entries.
- **Log Matching**: if two logs have the same index and same term, then the logs are identical in all entries up to that index
- **Leader Completeness:** If a log entry is committed in a given term, then the entry will appear in the logs of leaders of future terms
- **State Machine Safety:** if a server has applied an entry, no server will ever overwrite this entry

Desirable qualities, but how do we implement it?

Logs and Committed (Safe) Logs

- Servers can have log entries that are volatile
- Log entries are only **committed** after a majority of servers have accepted the log message.
- **Committed logs are guaranteed**
- External clients only get acknowledgements about committed log entries

Run an Election



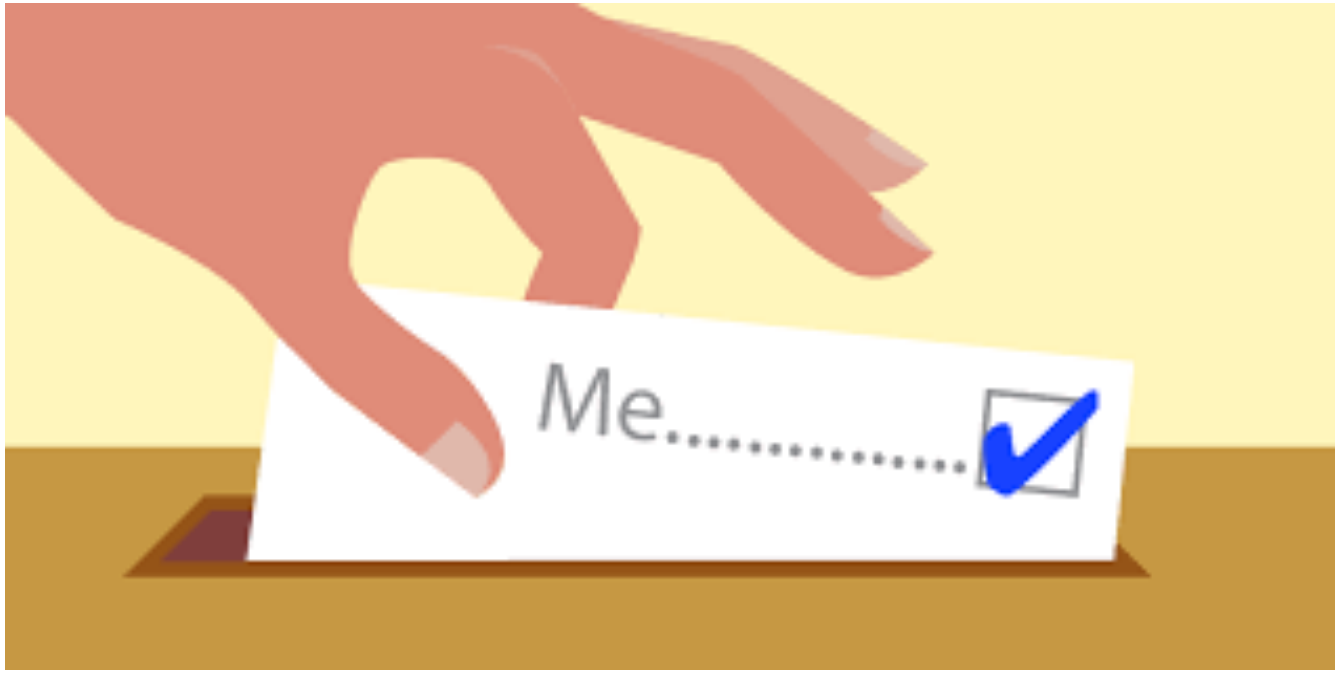
- **Log entry = index, term, command**
- **Log stored on stable storage (disk); survives crashes**
- **Entry committed if known to be stored on majority of servers**
  - Durable, will eventually be executed by state machines

Raft Election Process (1/4)

- Leaders send heartbeat messages periodically
- If a **follower** doesn't receive a heartbeat during an "election timeout" period, it changes to a **candidate.**

Raft Election Process (2/4)

The candidate increments its term value and votes for itself



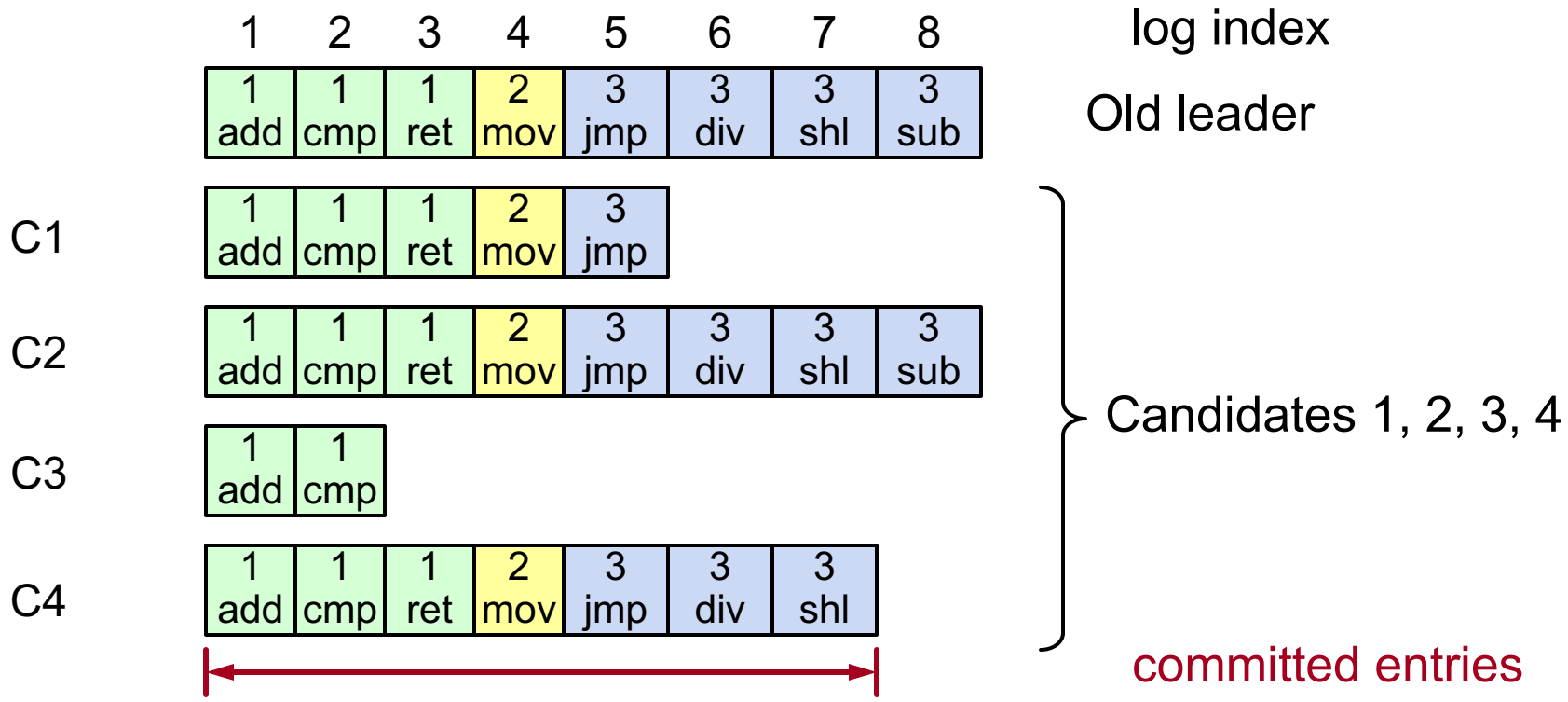
Raft Election Process (3/4)

- Candidate sends its **term** and **index** to all the other servers in a RequestVote message
- The recipient server is either in “follower” state or “candidate” state
- They compare **term** and **index** values of last logs
- Highest wins
- A server only votes at most once per term

Raft Election Process (4/4)

- When a candidate receives votes from a majority of servers, it wins the election and becomes the new leader
- It establishes this by sending an ”I Won!” log message to the clulster with its term.

Election



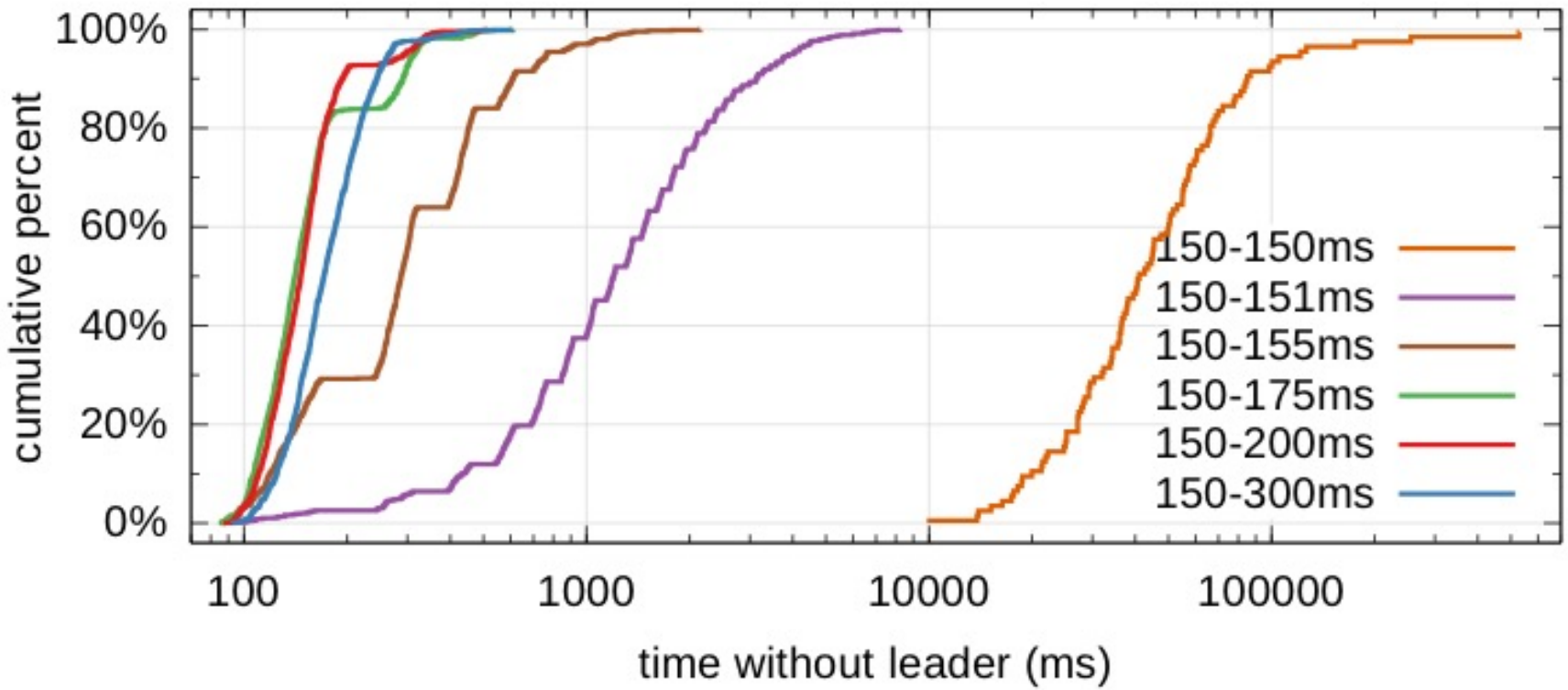
- **Candidates C2 and C4 can win the election with 3 votes**
  - Candidates C1 and C3 will vote for either
  - Candidate C4 can win if gets votes from C1 and C3 before contacting Candidate 2
- **Note a split election is possible: Candidate 1 votes for Candidate 2, and Candidate 3 votes for Candidate 4, for example**
- **Log entry 8 was not committed, but that's ok: the client (not shown) hasn't received an acknowledgement**

Split Elections

- If no candidate receives a majority of votes, the election fails and must be held again.
- Raft uses random election timeouts.
- If a leader hasn’t been elected within the election time out, a candidate increments its term and starts a new election.
- Each server has a different time out, chosen at random.
- This randomness is key for the system to find a new leader quickly

Randomized Timeouts

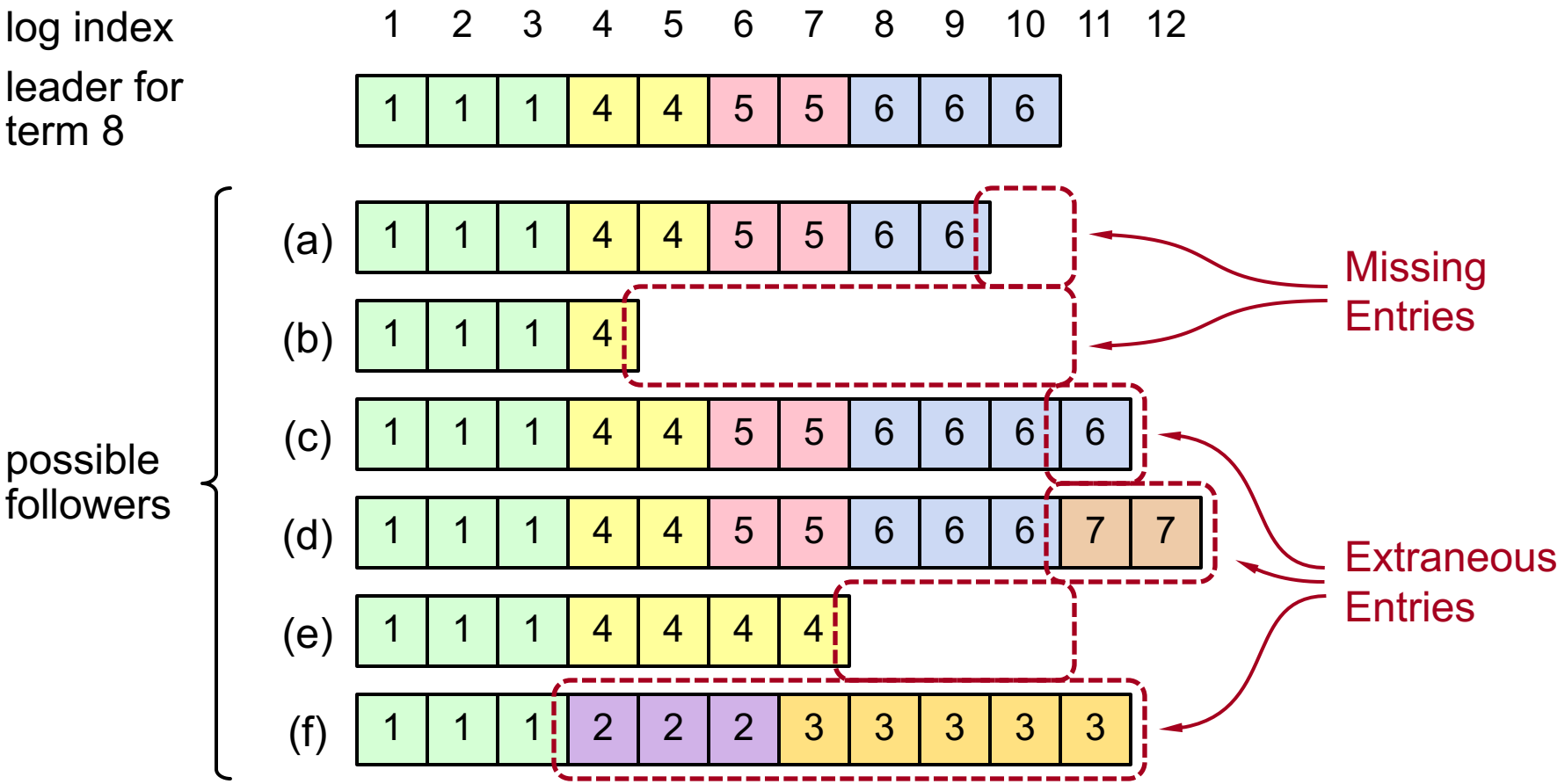
- How much randomization is needed to avoid split votes?



- Conservatively, use random range ~10x network latency

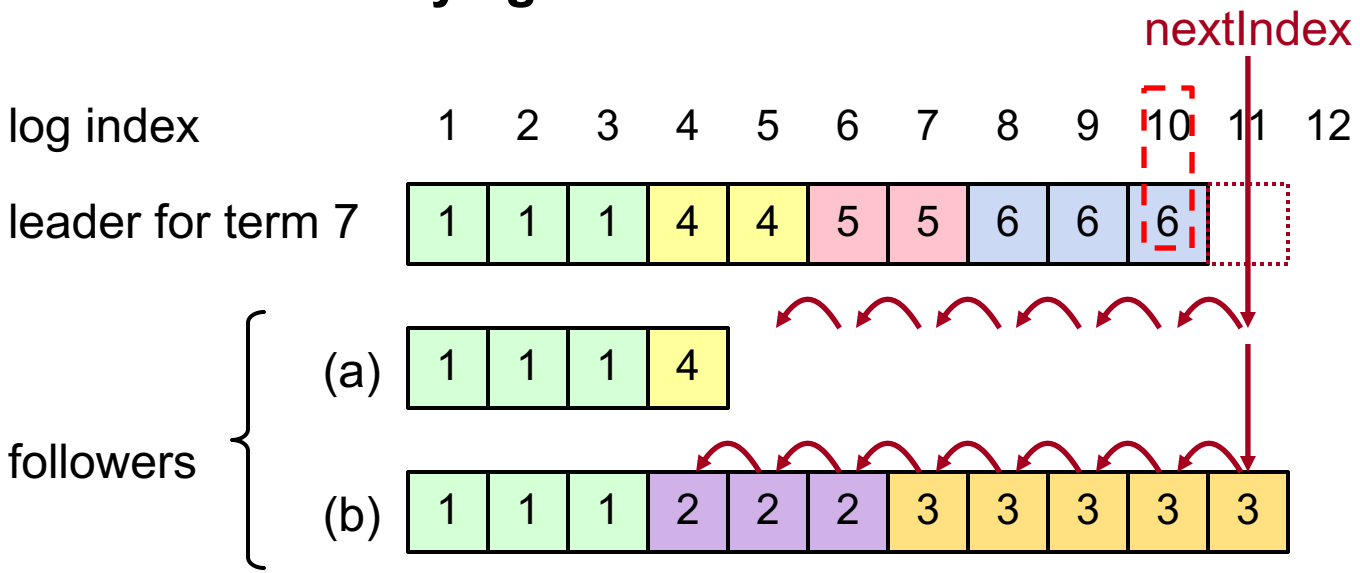
Log Inconsistencies

Leader changes can result in log inconsistencies:



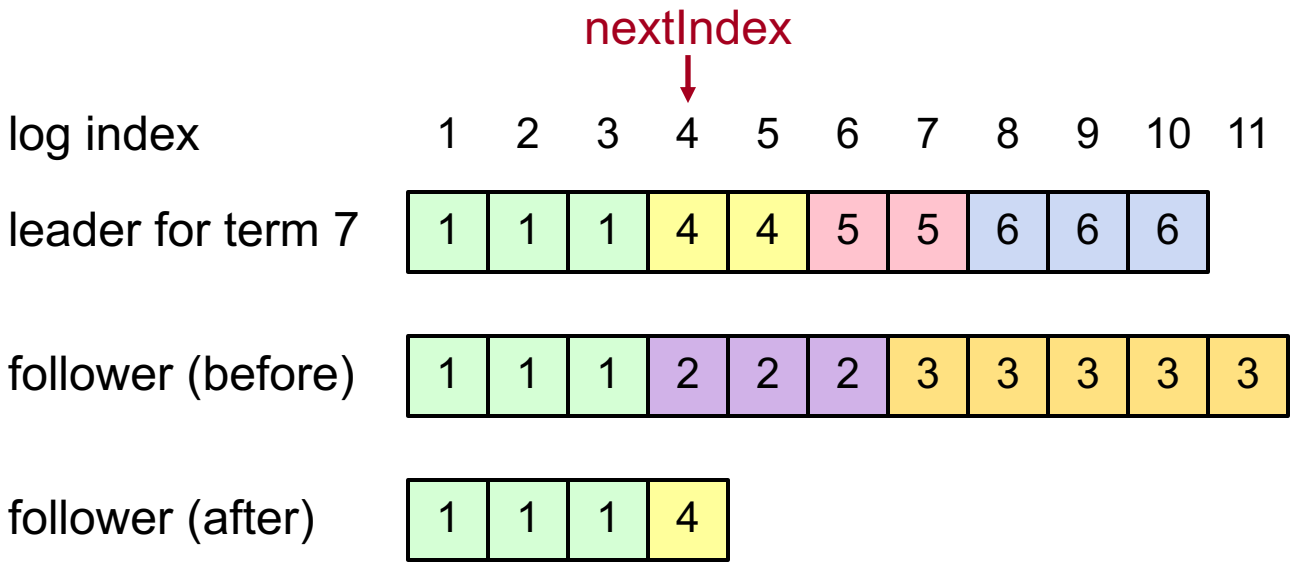
Repairing Follower Logs

- **New leader must make follower logs consistent with its own**
  - Delete extraneous entries
  - Fill in missing entries
- **Leader keeps nextIndex for each follower:**
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader's last index)
- **When AppendEntries consistency check fails, decrement nextIndex and try again:**



Repairing Logs, cont'd

- **When follower overwrites inconsistent entry, it deletes all subsequent entries:**



Neutralizing Old Leaders

- **Deposed leader may not be dead:**
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries
- **Terms used to detect stale leaders (and candidates)**
  - Every RPC contains term of sender
  - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally
- **An election updates the terms of majority of servers, by definition**
  - Deposed server cannot commit new log entries

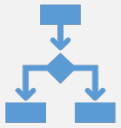
Some Previews of Upcoming Lectures

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?

Raft and Configuration Changes (Brief)

- Raft clusters can update themselves while continuously operating.
- They do this by requiring **joint consensus** between the old and new collections.
- This is an interesting approach for handling **Continuous Deployment** scenarios.

Some Limits of Raft (1/3)

All READ requests also go to the leader

The followers are just there as backups

When the Raft leader is stable, Raft provides strong consistency to its clients

But this can limit throughput

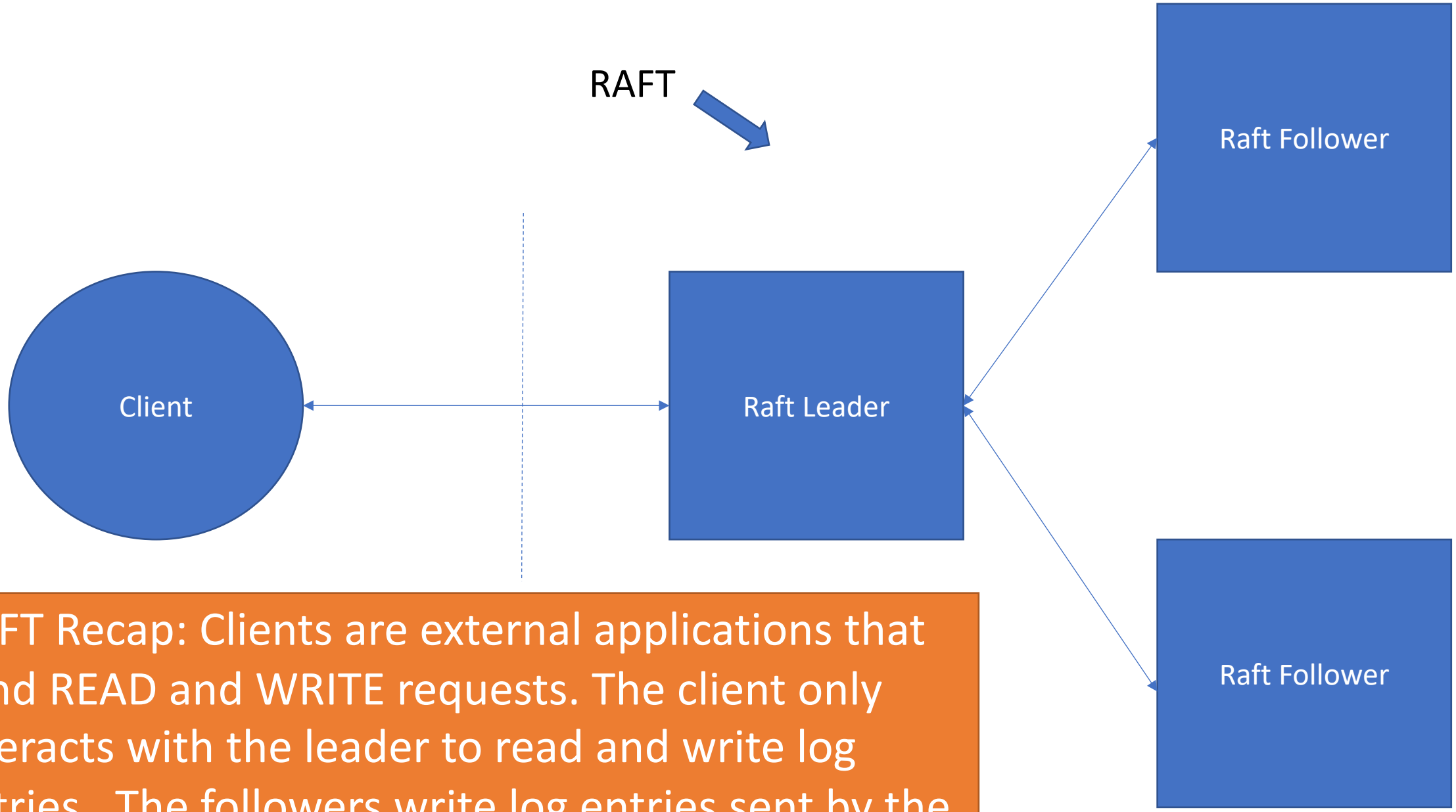
Zookeeper has weaker consistency but is more scalable for READs

Some Limits of Raft (2/3)

- Raft may not scale across geographically distant data centers or cloud regions
  - Consul supplements Raft with another protocol called SWIM
- The problem: network communication speed and network unreliability will make the system unwieldy
- Blockchain is a similar logging protocol that works at larger scales

Some Limits of Raft (3/3)

- Raft members communicate with each other using RPC calls.
- What if a malicious or faulty server is in the cluster?
  - It could tamper with logs
  - It could inappropriately share logs
  - It could disrupt the system by calling elections
- These are called Byzantine failures



RAFT Recap: Clients are external applications that 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.

What You Should Really Remember About Raft

Raft is a consensus algorithm for maintaining consistent state in distributed systems

It is the basis for Control Plane (information) services

Don’t try to invent something like this yourself. It is tricky.

It works best within a single data center with low latencies.