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.

Mental Model Goals

- What are logs and how do they relate to system state?
- What is a consensus algorithm?
- What is leader election and how does it work?
- How do these contribute to fault tolerance?
- What are the limits of leader-based consensus algorithms?

Log-Centric Distributed Systems

Motivations and an overview of the Raft algorithm

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

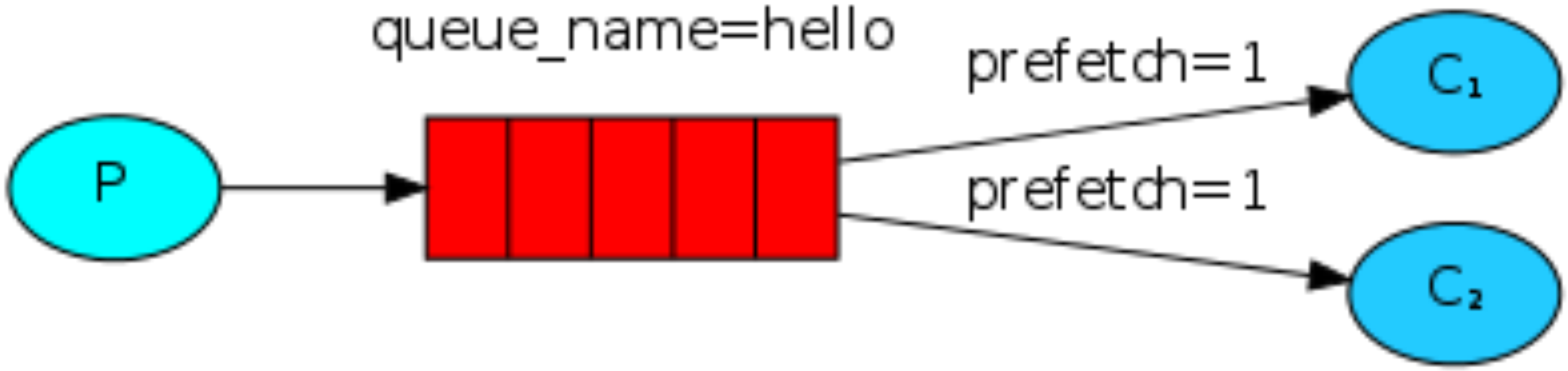
- They are fault-tolerant, can keep working even if part of the system is down.
- They can smoothly scale up or down to handle different loads
- They are dynamic: minimal static configuration and hard coding

Two General Classes of Cloud-Native Services

- Stateless
- Stateful

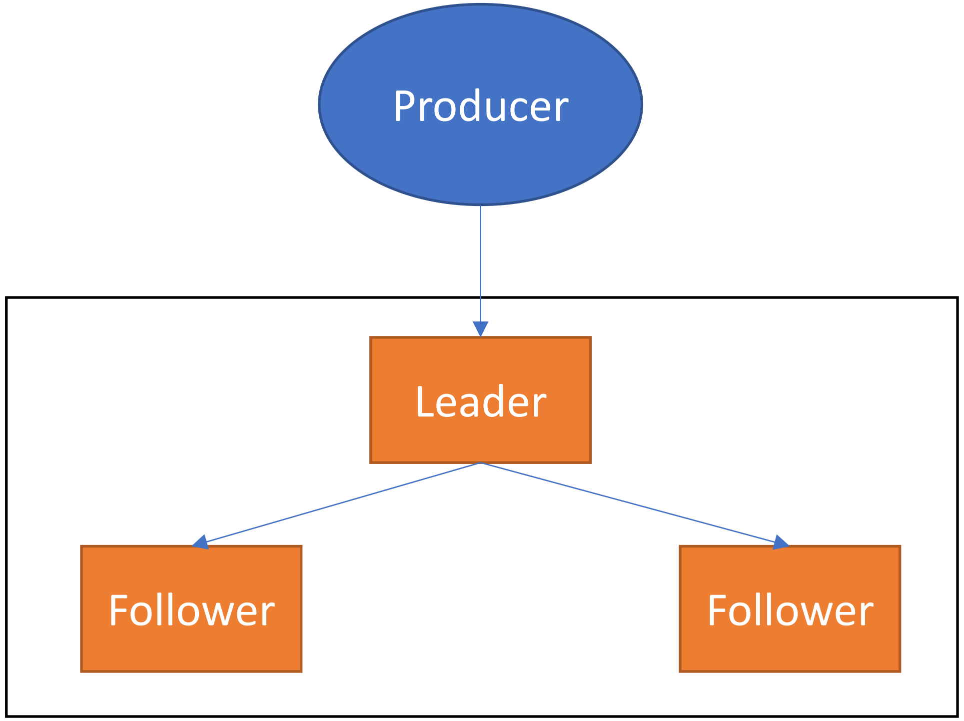
A Stateless Example: Work Queue Producers and Consumers

- 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 IF the broker scales**
- There is state; it's in the broker



A Stateful Example: Leader-Follower

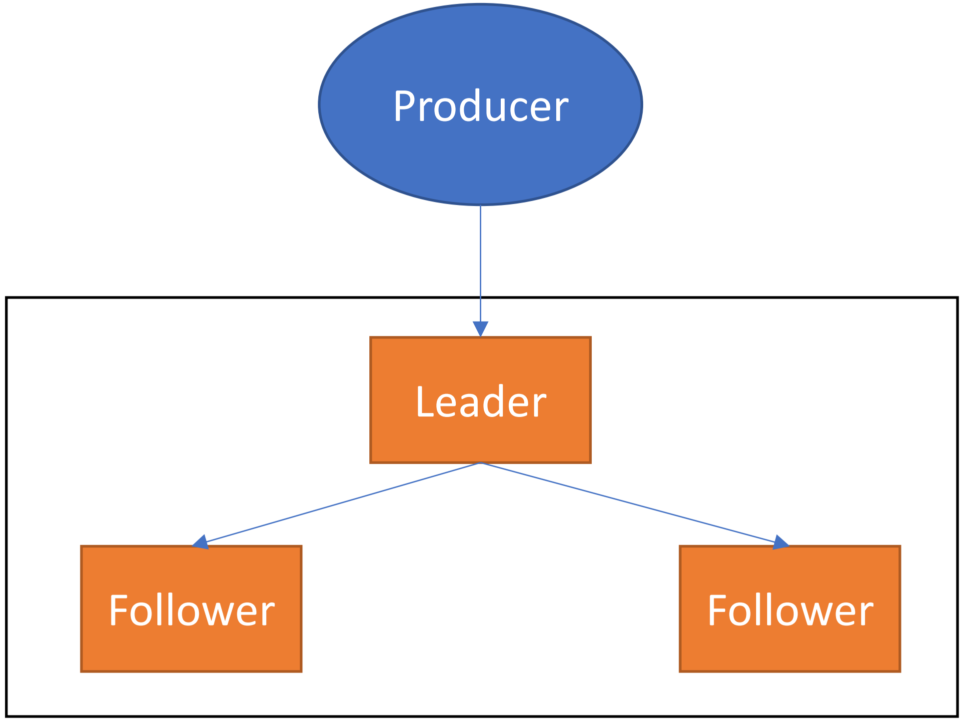
- State-changing events (Create, Update, Delete) go to a leader
- The leader updates followers.
- A follower can take over if the leader fails



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

Logs and State

- Use logs to capture system state
- Store logs carefully
- Distribute logs to potential replacements

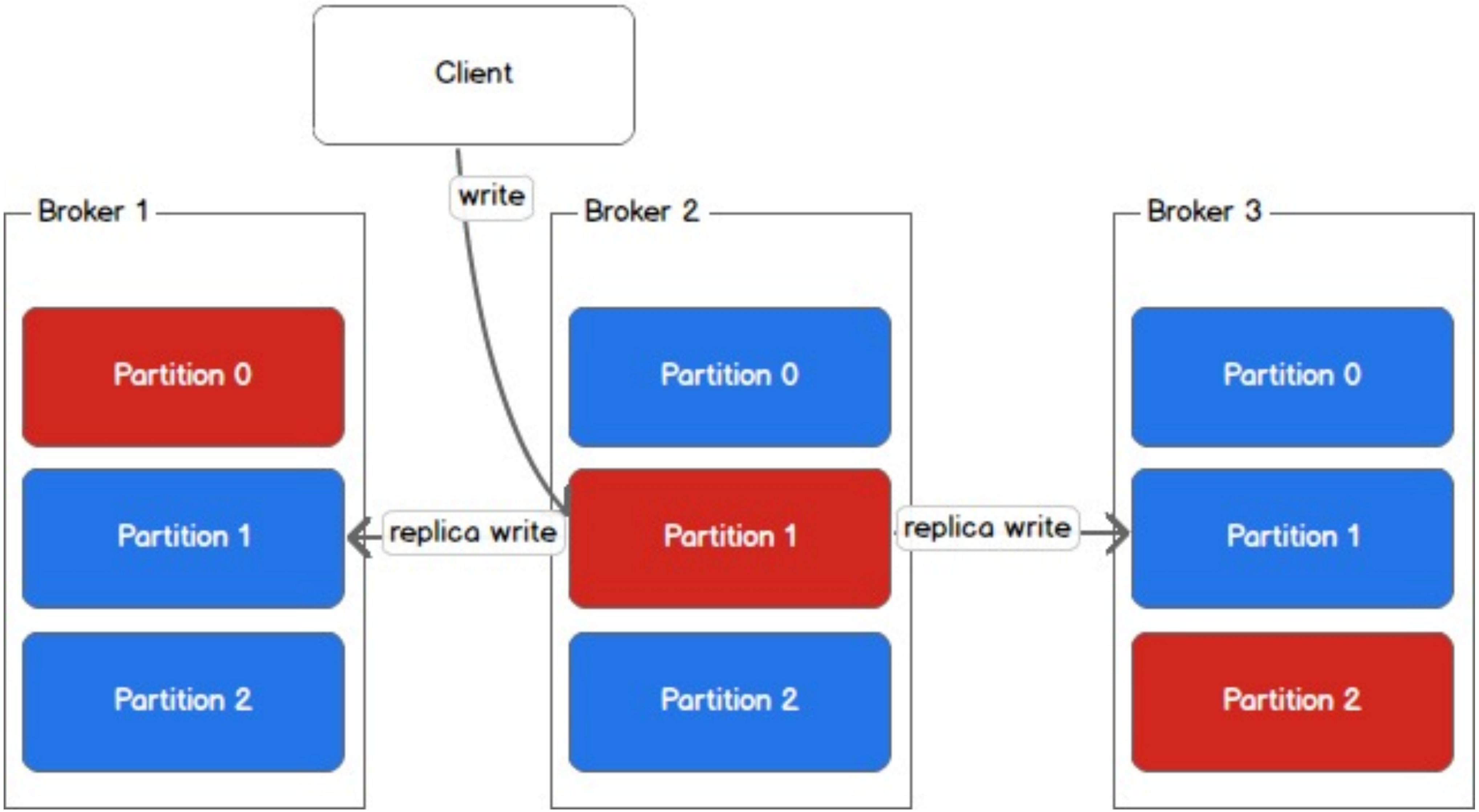


A log is a just a record of time-ordered state change events

Back up logs so that they can be replayed to recover.

Copy logs to following processes so that they can stay in synch with the leader.

Leader (red) and replicas (blue)



Kafka uses distributed logs and a leader-follower model

Logs and Service Mesh Control Planes

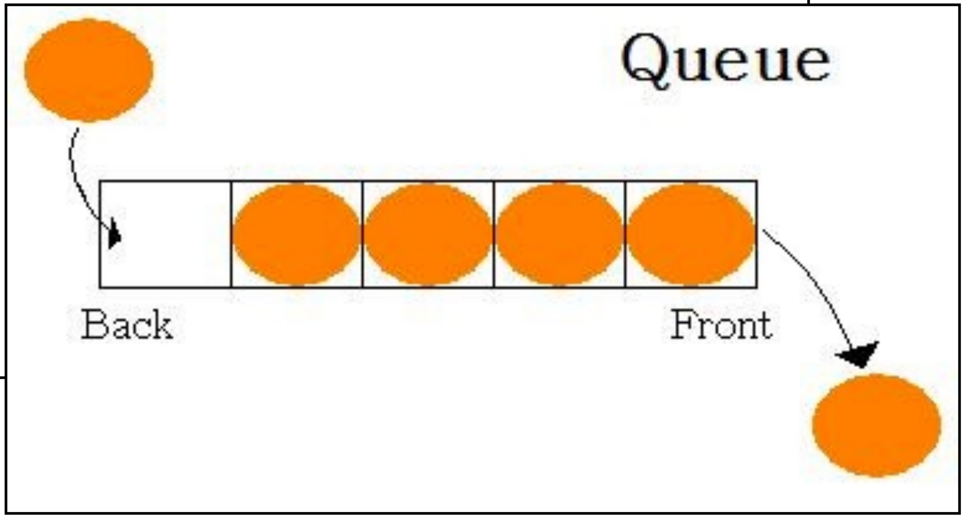
- Consul, ETCD, and Zookeeper manage information in distributed systems
- You can use them to implement a control plane for your service mesh (microservices)
- Consul and ETCD use a protocol called RAFT, as does RabbitMQ when clustered

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

Message Queues

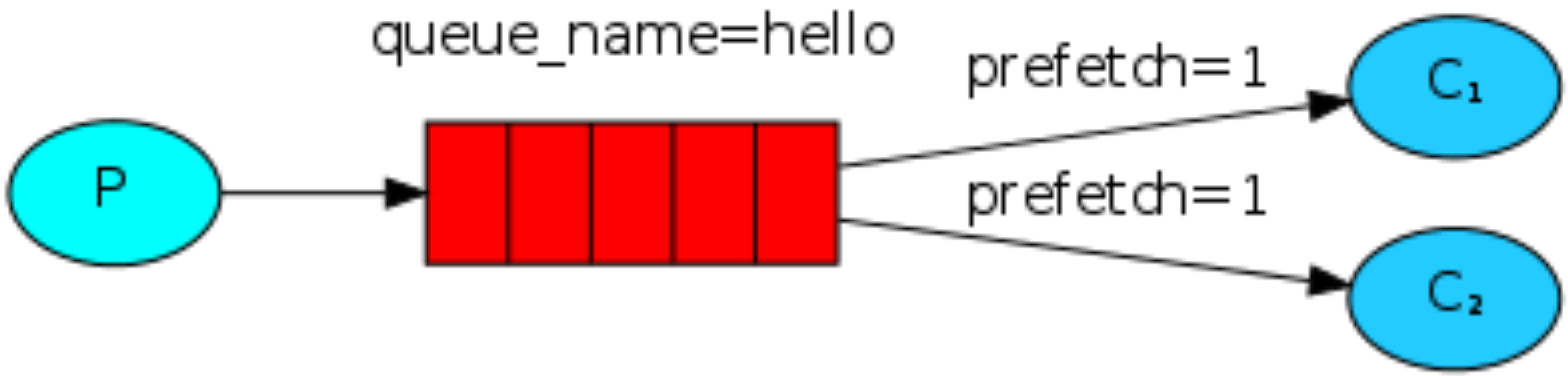
- Message Queue: a data structure containing a **consumable** list of entries.
  - Publishers create queue entries
  - Brokers manage the queues
  - Consumers consume queue entries
  - Entries are removed when they are consumed.

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 the queue.
- Queue entries are supposed to be ephemeral.

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?

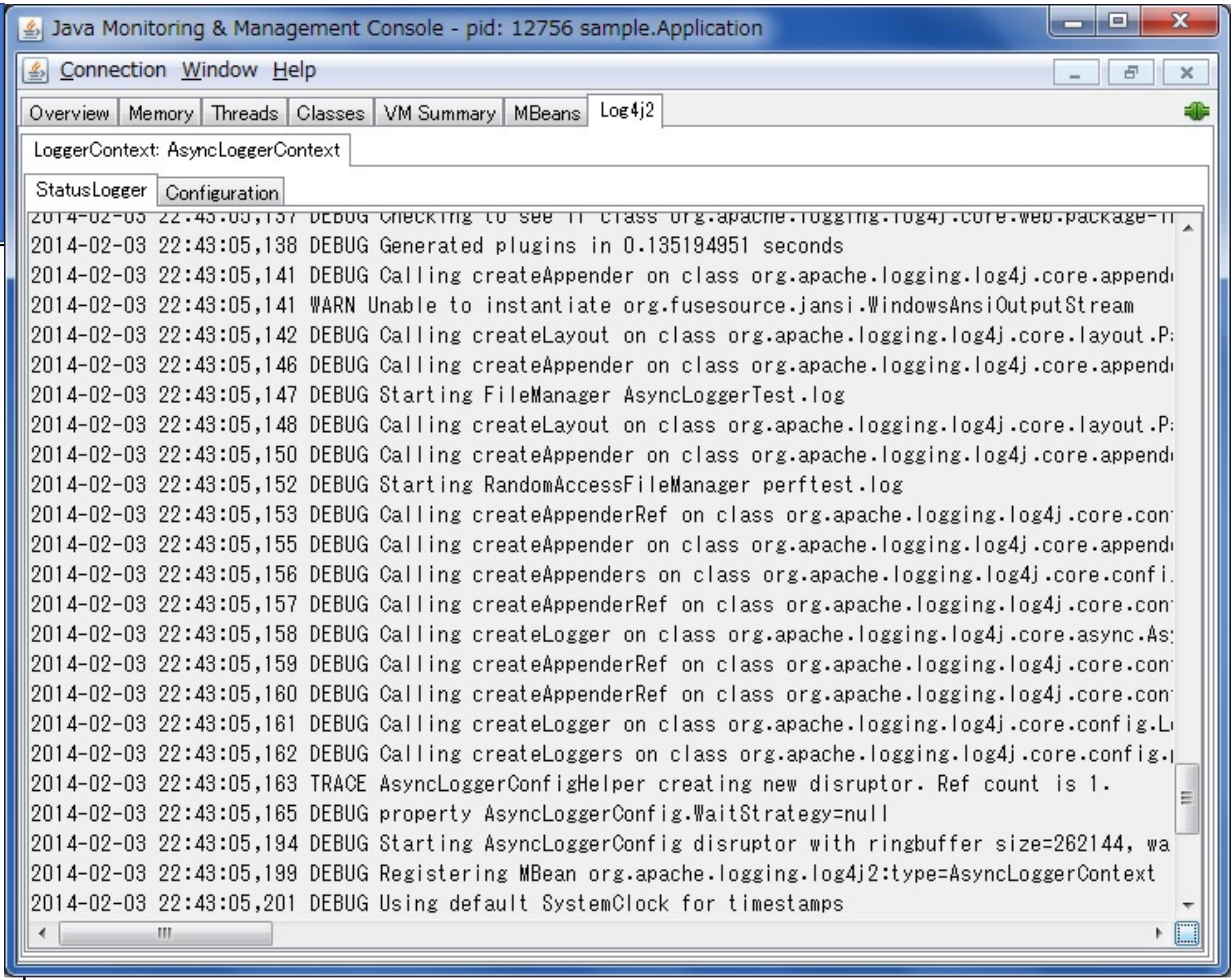
Some issues with detecting failed message delivery

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**



Example System State 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

- A log is a replayable set of recorded instructions
- Replicated logs are used to implement **replicated state machines**
- **Replicated state machines** are used to build distributed systems
- **Consensus algorithms** keep replicated logs in synch

Some Desirable Properties of System State Logs

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

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, it matches the primary copy 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 Algorithm

- 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 an algorithm, not a piece of software

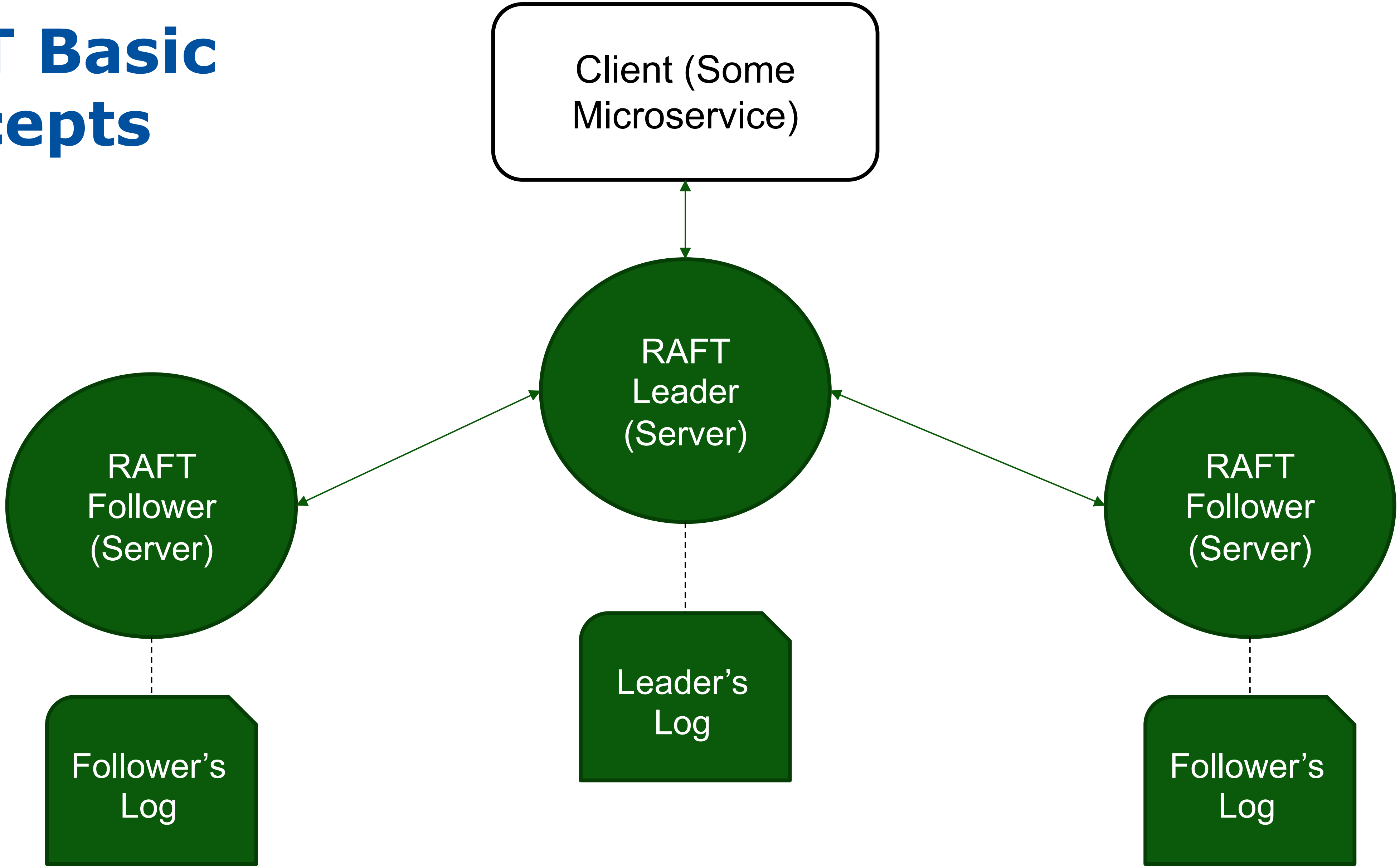
Software like Consul and ETCD use the protocol.

Zab, Viewstamped Replication, and Paxos are alternative protocols

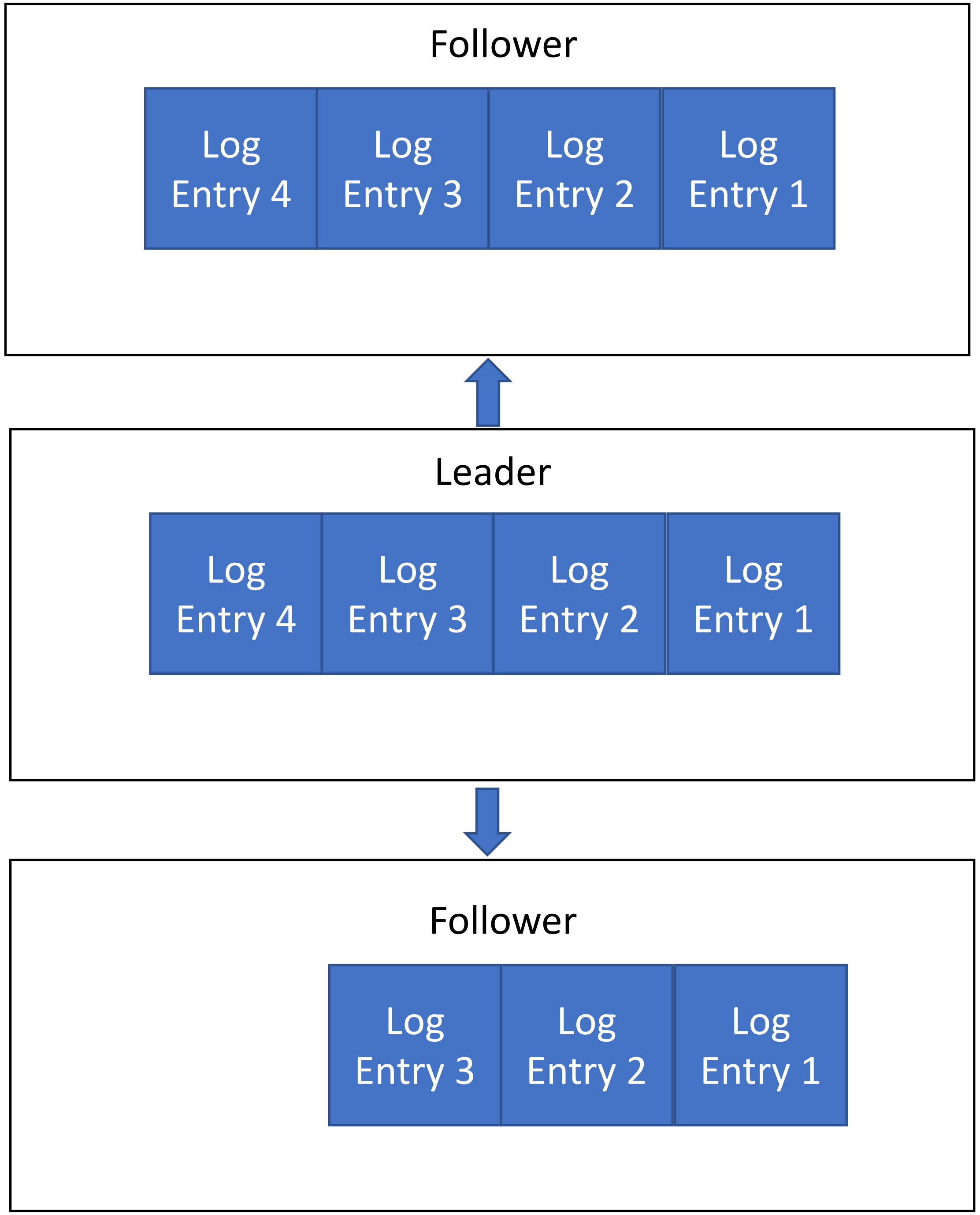
Properties of Consensus Systems

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 Basic Concepts



Structure of a Log



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)

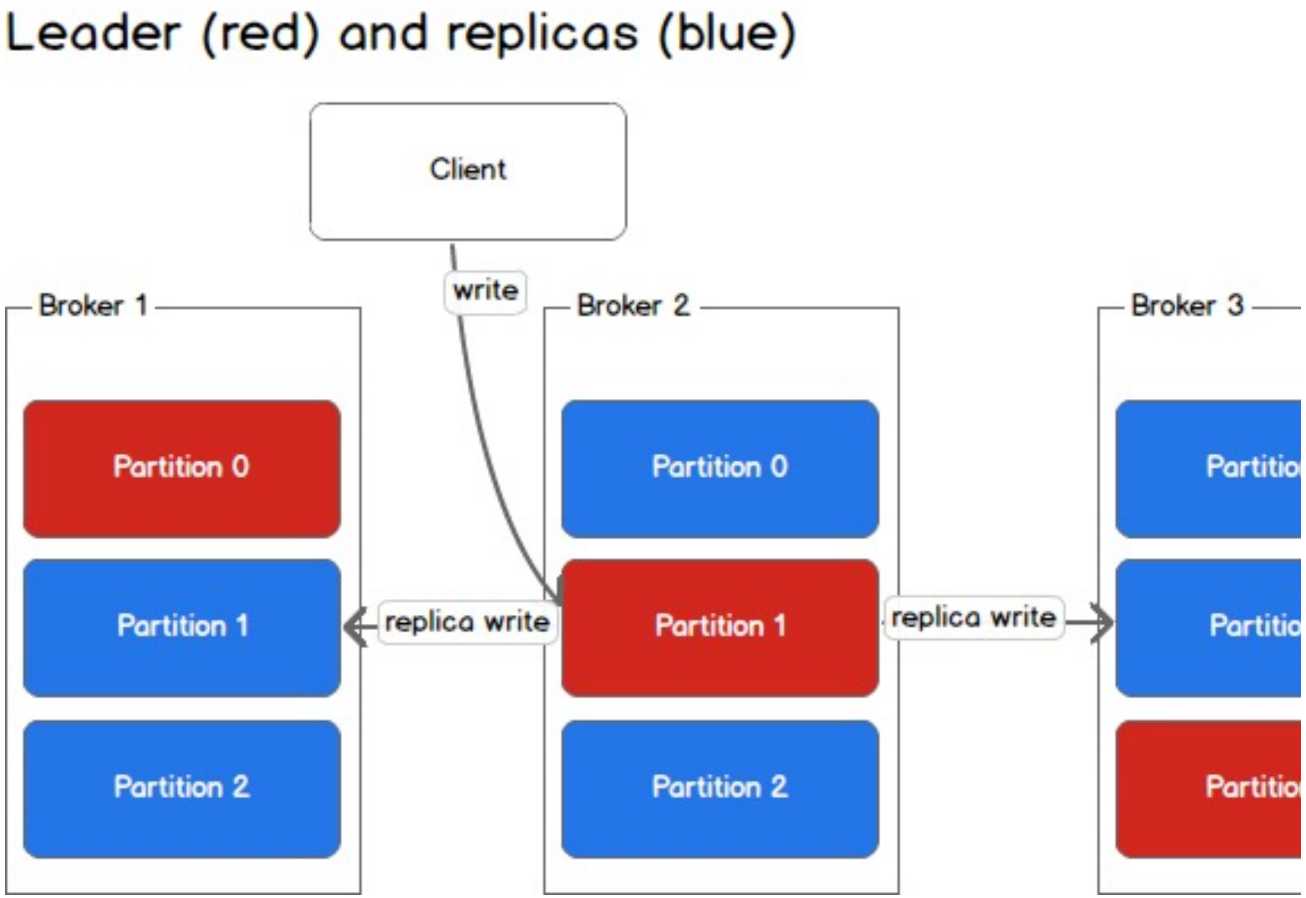
Leader Completeness: If a log entry is *committed* in a given term, then the entry will appear in the logs of leaders of future

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.

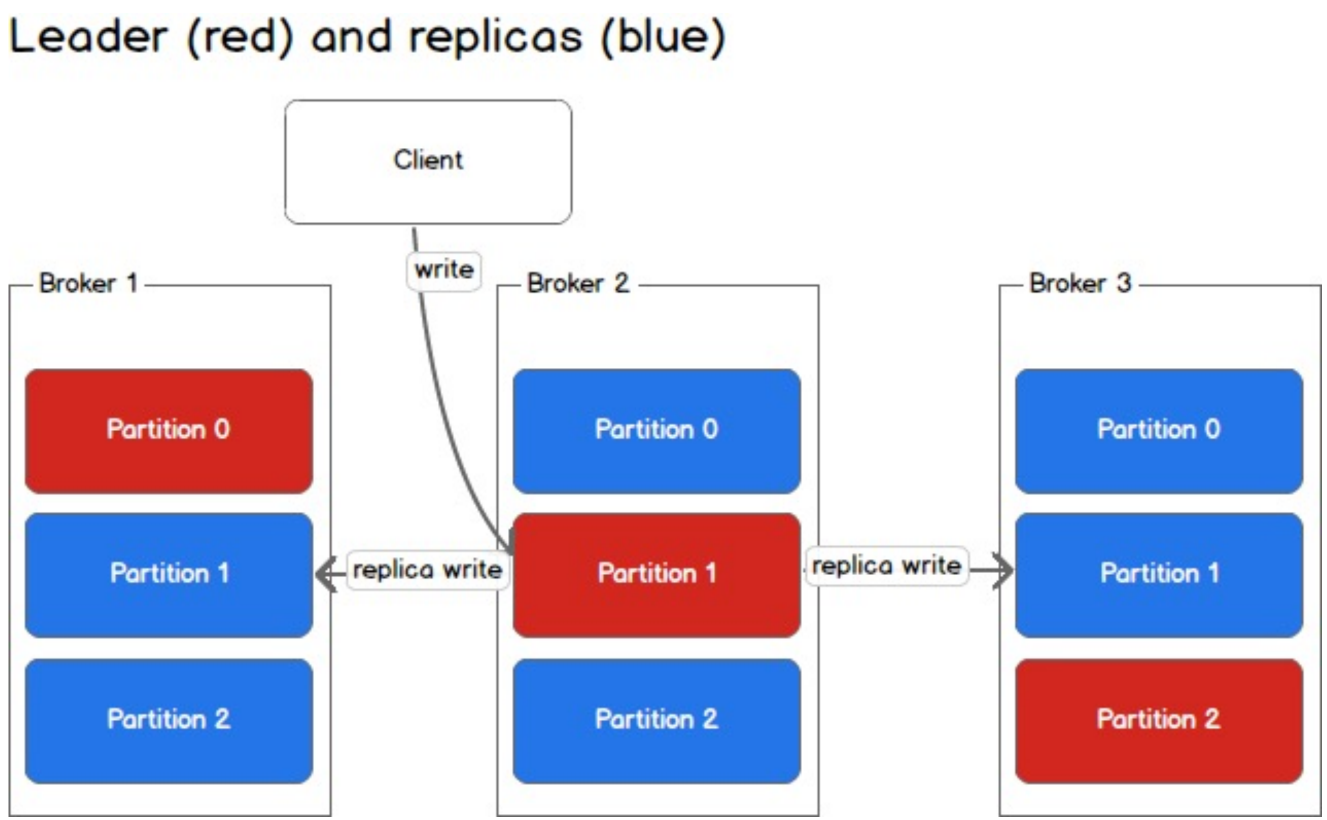
Raft Basics: Strong Leader and Passive Followers

- 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