

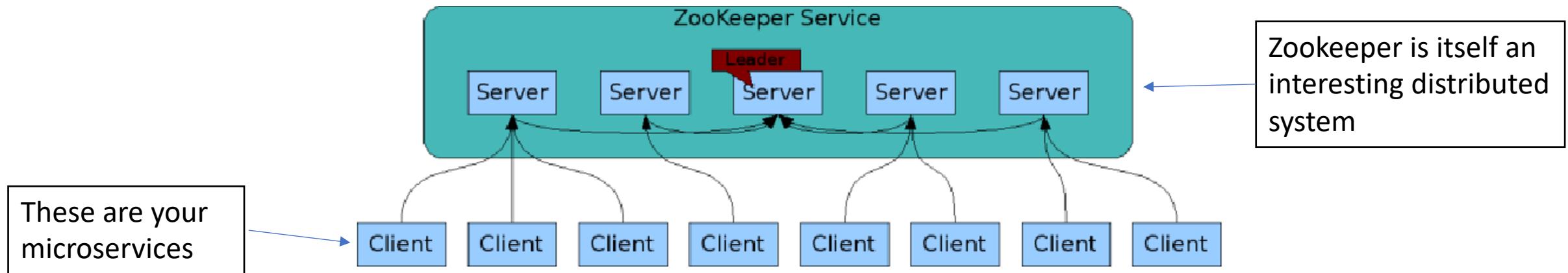
Log-Centric Distributed Systems

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

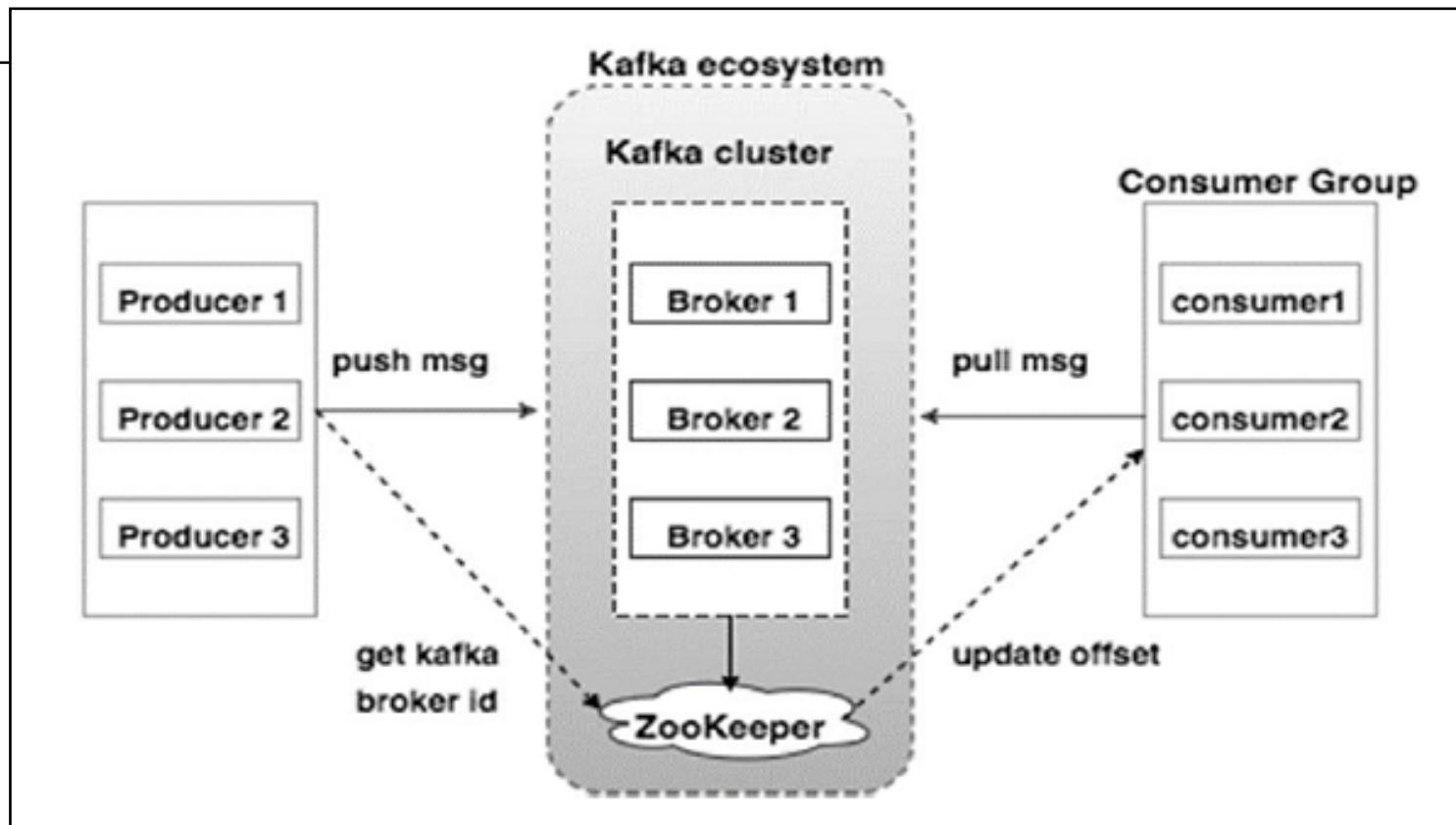
The ZooKeeper Service



- ZooKeeper Service is replicated over a set of machines
- All machines store a copy of the data in memory (!)
- A leader is elected on service startup
- Clients only connect to a single ZooKeeper server & maintains a TCP connection.
- Client can read from any Zookeeper server.
- Writes go through the leader & need majority consensus.

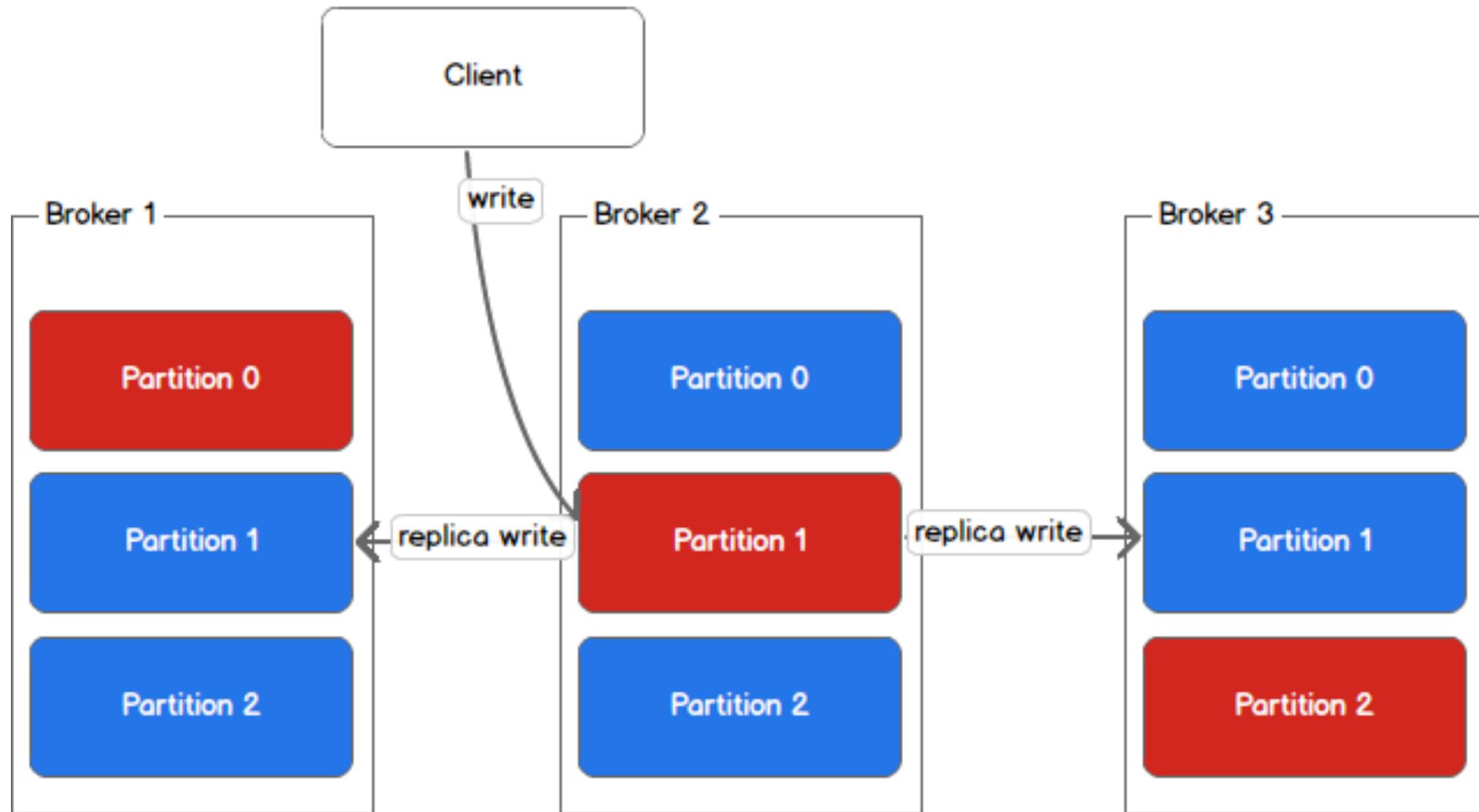
Kafka Uses Clusters of Brokers

- Kafka is run as a cluster of brokers on one or more servers.



Kafka uses Zookeeper to manage state of clients and leadership of brokers.

Leader (red) and replicas (blue)



A producer writes to Partition 1 of a topic. Broker 2 is the leader. It writes replicas to Brokers 1 and 3

Logs and Service Mesh Control Planes



Consul and ETCD resemble Zookeeper



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

Ex: see
<https://www.consul.io/intro/vs/proxies.html> for Consul + Envoy

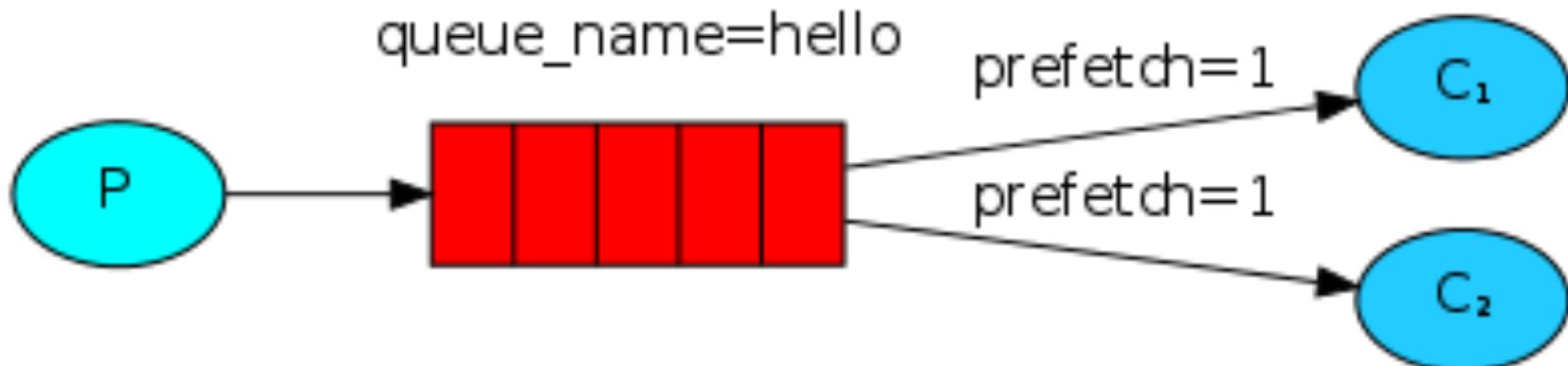


They use a protocol called RAFT

Simpler than Zookeeper's Zab protocol

But First, 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.



Some Features and Considerations for Queue-Centric Systems

- If a consumer crashes before sending an ACK, the broker will resend the message to another consumer, if one is available
- But detecting a failed consumer is a fundamentally hard problem
 - See next slide
 - These are corner cases until you have a major network disruption, then these cascade all at once
 - A resent \$1B transaction is a catastrophe
- It is possible that the broker can acquire a backlog of un-ACK'd messages, hindering performance
- What if the broker crashes?
 - RabbitMQ provides some persistency mechanisms, but are these first class design considerations?

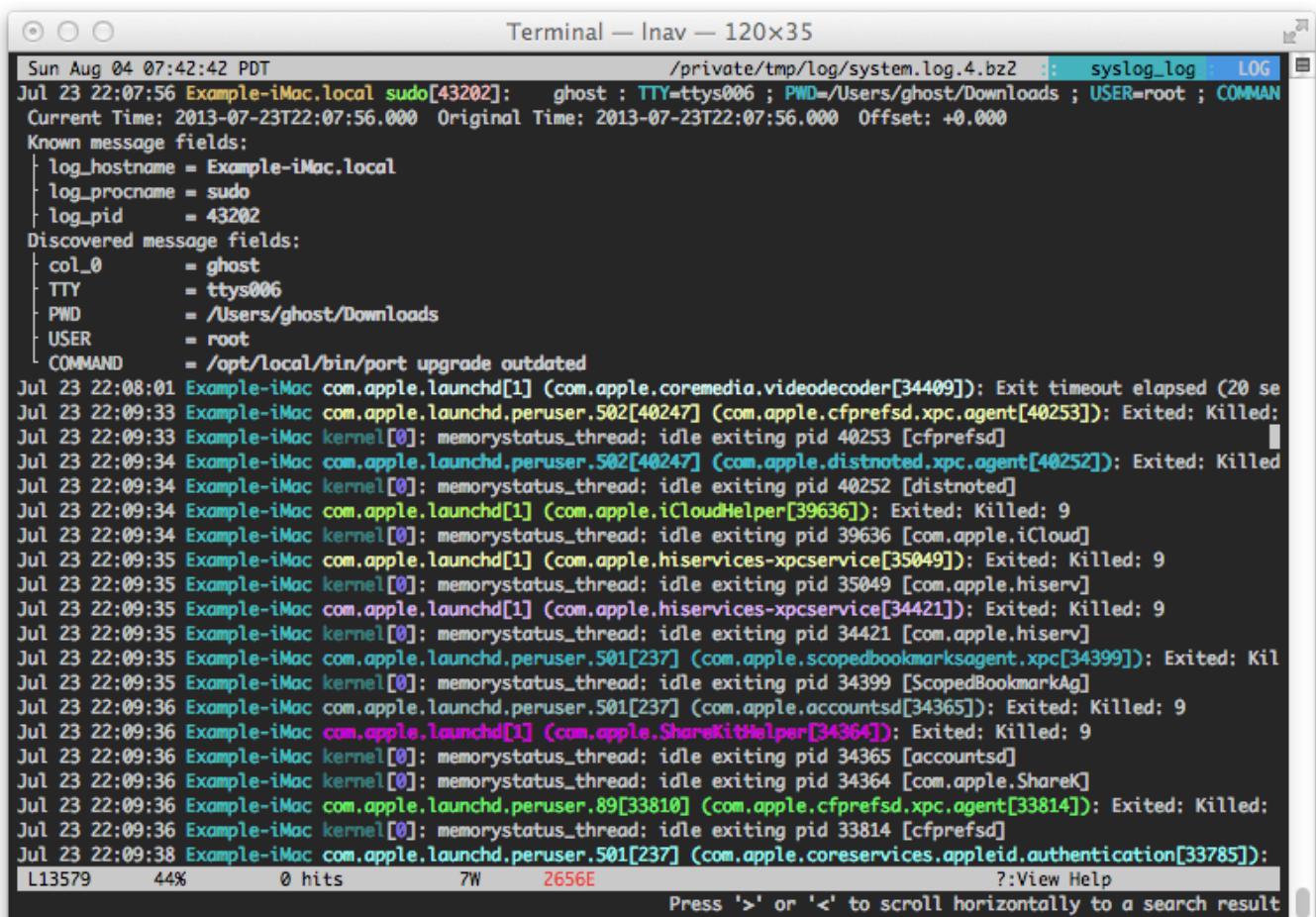
Issue	Solution
Consumer crashes	Broker detects the crash using a heartbeat . This works well for small systems (Zookeeper) but doesn't scale. See Gossip Protocol
Consumer is very slow	Heartbeat detects that the consumer is alive but taking a very long time to send an ACK. Solution: use a time out . The broker needs to tell the consumer to stop processing the message, complicating the implementation. Time outs need to be selected correctly.
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: 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.
 - Consolidation is critical in distributed systems: ELK stack → Service Mesh
- Human readable, unstructured format



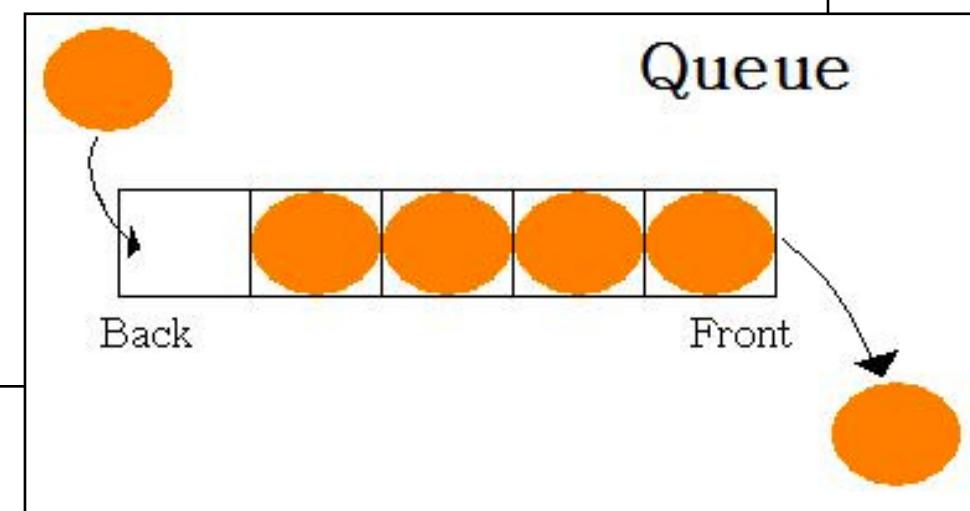
A screenshot of a Mac OS X Terminal window titled "Terminal — lnav — 120x35". The window displays a log file named "/private/tmp/log/system.log.4.bz2" with the file type "syslog_log" and a log level of "LOG". The log output shows system events from July 23, 2013, at 22:07:56 PDT. The log includes known message fields (log_hostname, log_procname, log_pid) and discovered message fields (col_0, TTY, PWD, USER, COMMAND). Many entries show processes like launchd, kernel, and com.apple.* exiting or killed. The log ends with a timestamp of Jul 23 22:09:38 and a process ID of L13579.

```
Sun Aug  4 07:42:42 PDT /private/tmp/log/system.log.4.bz2 :: syslog_log LOG
Jul 23 22:07:56 Example-iMac.local sudo[43202]:  ghost : TTY=tty006 ; PWD=/Users/ghost/Downloads ; USER=root ; COMMAND
Current Time: 2013-07-23T22:07:56.000 Original Time: 2013-07-23T22:07:56.000 Offset: +0.000
Known message fields:
| log_hostname = Example-iMac.local
| log_procname = sudo
| log_pid     = 43202
Discovered message fields:
| col_0      = ghost
| TTY        = ttys006
| PWD        = /Users/ghost/Downloads
| USER       = root
| COMMAND    = /opt/local/bin/port upgrade outdated
Jul 23 22:08:01 Example-iMac com.apple.launchd[1] (com.apple.coremedia.videodecoder[34409]): Exit timeout elapsed (20 se
Jul 23 22:09:33 Example-iMac com.apple.launchd.peruser.502[40247] (com.apple.cfprefsd.xpc.agent[40253]): Exited: Killed:
Jul 23 22:09:33 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 40253 [cfprefsd]
Jul 23 22:09:34 Example-iMac com.apple.launchd.peruser.502[40247] (com.apple.distnoted.xpc.agent[40252]): Exited: Killed
Jul 23 22:09:34 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 40252 [distnoted]
Jul 23 22:09:34 Example-iMac com.apple.launchd[1] (com.apple.iCloudHelper[39636]): Exited: Killed: 9
Jul 23 22:09:34 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 39636 [com.apple.iCloud]
Jul 23 22:09:35 Example-iMac com.apple.launchd[1] (com.apple.hiservices-xpcservice[35049]): Exited: Killed: 9
Jul 23 22:09:35 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 35049 [com.apple.hiserv]
Jul 23 22:09:35 Example-iMac com.apple.launchd[1] (com.apple.hiservices-xpcservice[34421]): Exited: Killed: 9
Jul 23 22:09:35 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 34421 [com.apple.hiserv]
Jul 23 22:09:35 Example-iMac com.apple.launchd.peruser.501[237] (com.apple.scopedbookmarksagent.xpc[34399]): Exited: Kil
Jul 23 22:09:35 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 34399 [ScopedBookmarkAg]
Jul 23 22:09:36 Example-iMac com.apple.launchd.peruser.501[237] (com.apple.accounts[34365]): Exited: Killed: 9
Jul 23 22:09:36 Example-iMac com.apple.launchd[1] (com.apple.ShareKitHelper[34364]): Exited: Killed: 9
Jul 23 22:09:36 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 34365 [accounts]
Jul 23 22:09:36 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 34364 [com.apple.ShareK]
Jul 23 22:09:36 Example-iMac com.apple.launchd.peruser.89[33810] (com.apple.cfprefsd.xpc.agent[33814]): Exited: Killed:
Jul 23 22:09:36 Example-iMac kernel[0]: memorystatus_thread: idle exiting pid 33814 [cfprefsd]
Jul 23 22:09:38 Example-iMac com.apple.launchd.peruser.501[237] (com.apple.coreservices.appleid.authentication[33785]): L13579 44% 0 hits 7W 2656E ?View Help
Press '>' or '<' to scroll horizontally to a search result
```

Message Queues

- Message Queue: a data structure containing a **consumable** list of entries.
 - Publishers create entries
 - Brokers receive and deliver entries
 - Consumers consume entries
 - Entries are machine readable.
 - Entries are removed when they are consumed.

We discussed these limits in the Kafka lectures: queues are not good for replays --> state management



The Other Type of Log: System State Records

- Logs: records of state changes in a system
 - **Machine readable**
 - Distributed logs -> distributed state machines
- Logs can be for internal consumption
 - Enables a system to recover state on restart or failover
 - Ex: Zookeeper and Kafka internals
- Logs can be for external consumption
 - Different types of processes share a state machine: same logs are processed differently
 - Ex: Clients to Zookeeper, Kafka



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

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

Example: 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 Queues

- These are similar data structures
 - Basics: First in, first out
- Important difference is that logs are persistent while queue entries are transient
 - Persistence can have limits: Kafka quality of service
- Message queue: the broker often needs to know if the message was delivered and processed correctly
 - Is it safe to delete an entry?
 - ACK or NACK
 - Stateful
- Log system: 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)

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

Zookeeper and Chubby, for instance, don't use it

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

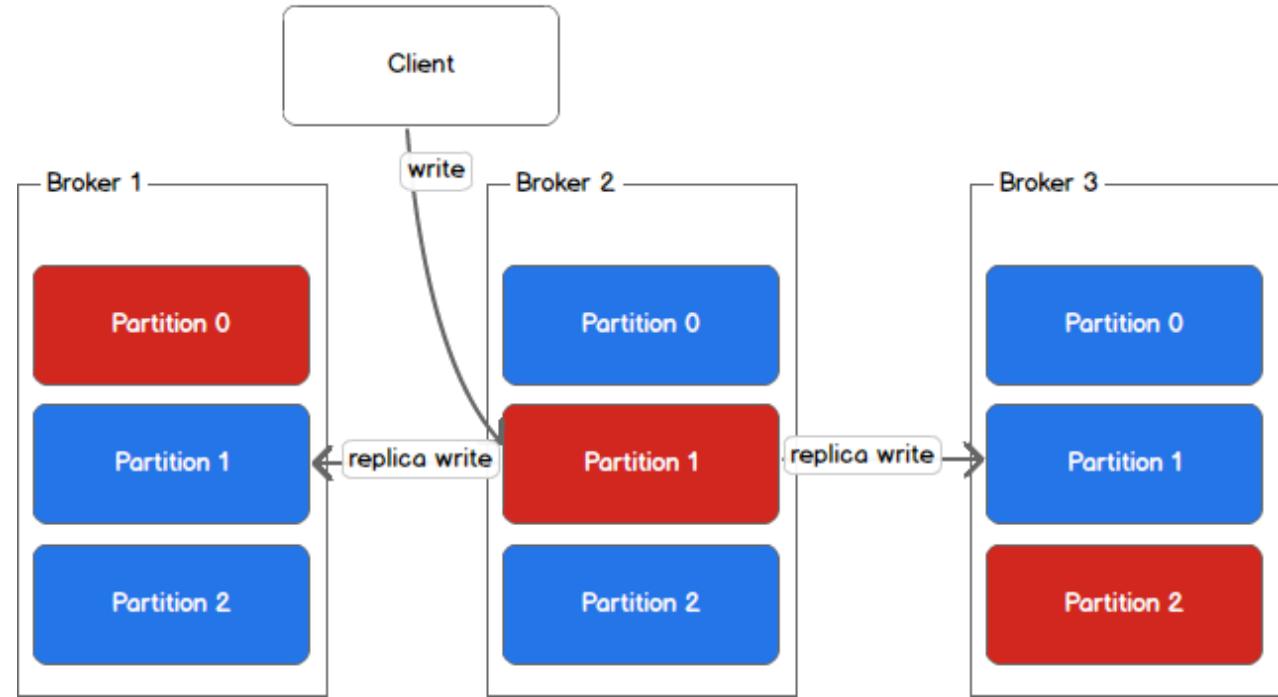
Raft Protocol Guarantees: Always True (5/5)

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

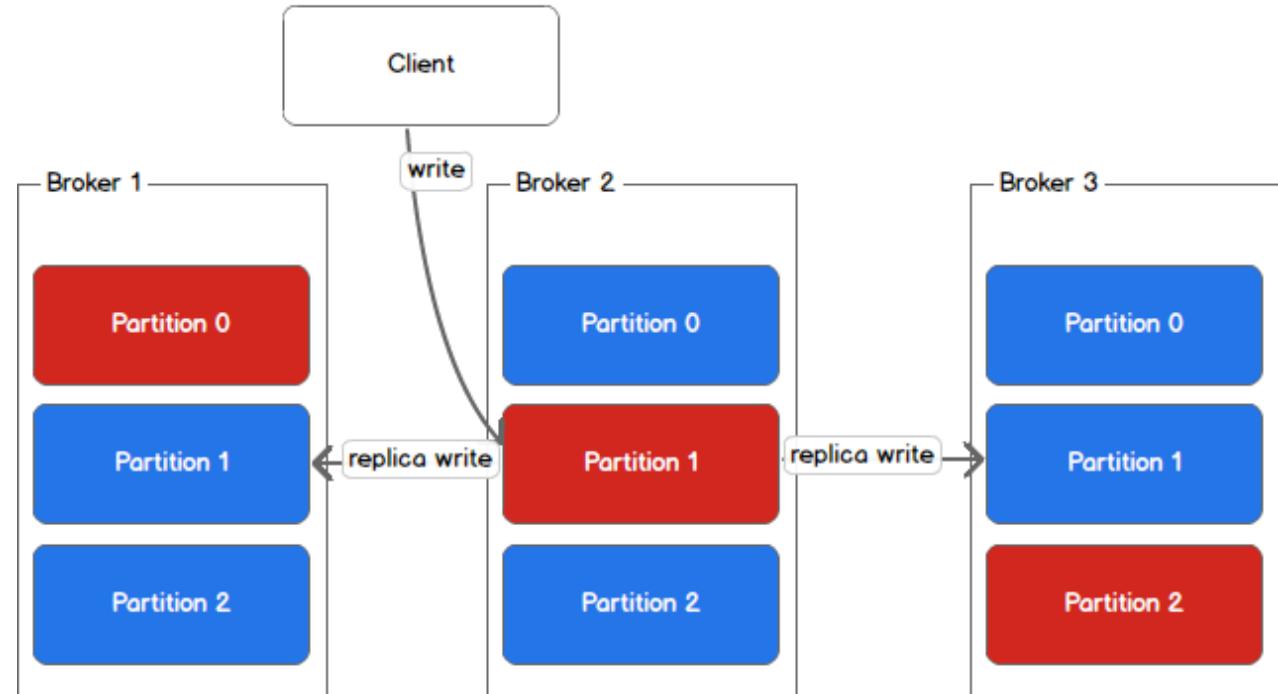
Leader (red) and replicas (blue)



Raft Basics: Strong Leader and Passive Followers

- The leader sends log updates to all followers
- If a majority of followers accept the update, the leader instructs everyone to commit the message.

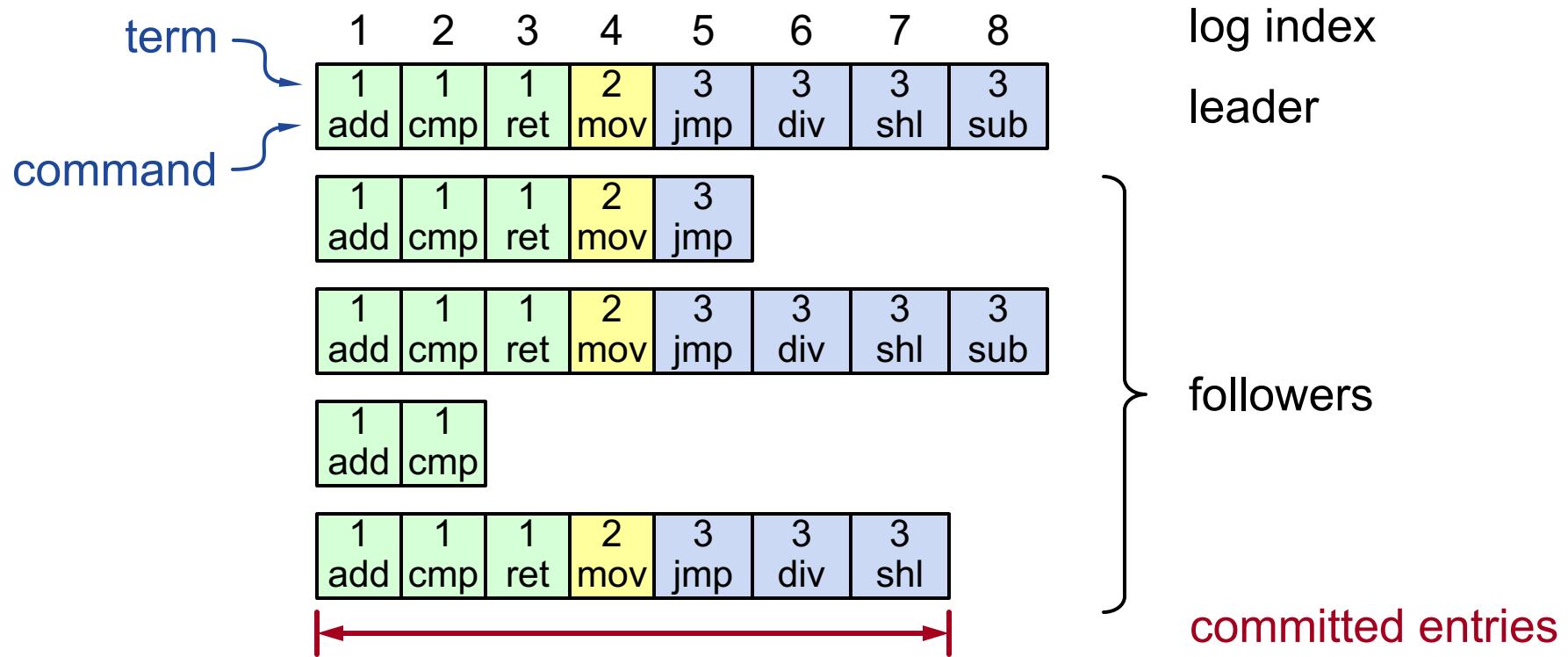
Leader (red) and replicas (blue)



Raft Achieves Eventual Consistency

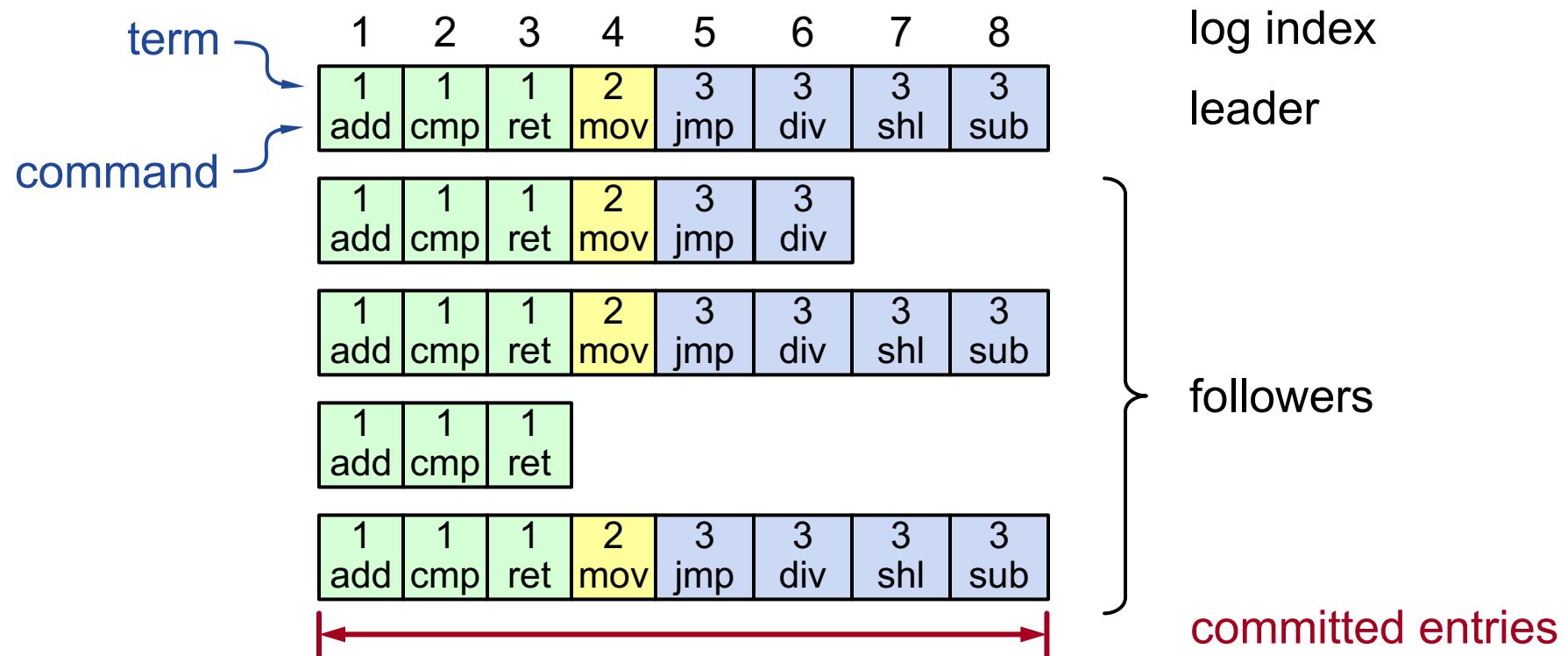
- A minority of followers can have fewer committed messages than the majority at any given time
- But lagging members will have a subset of committed messages

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

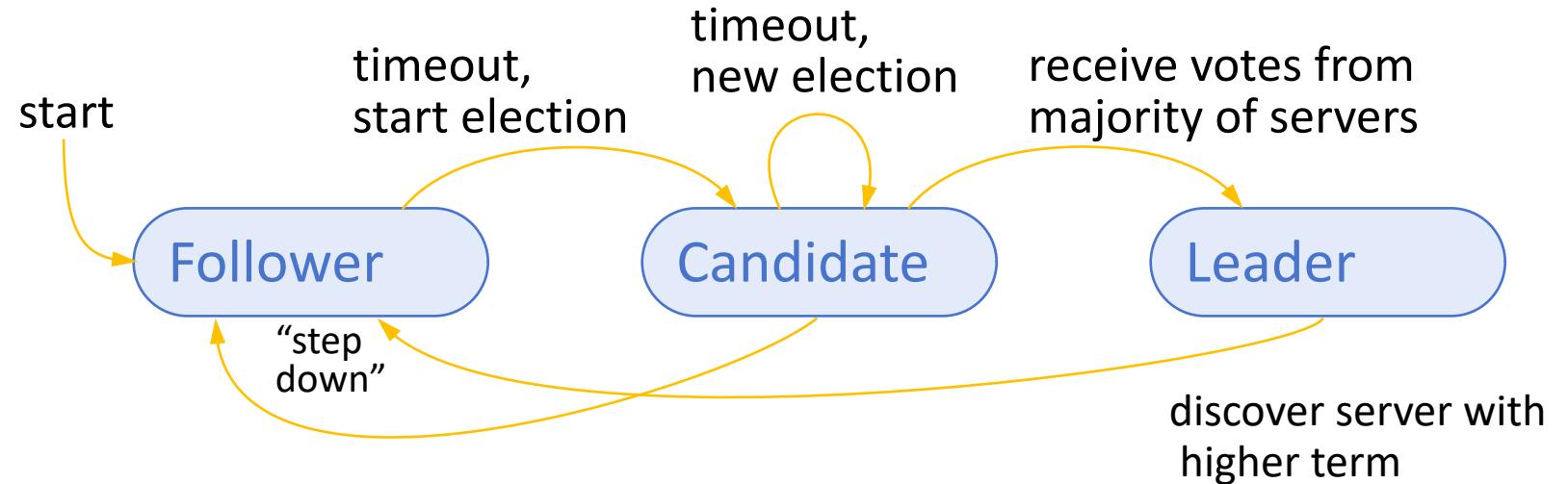
Log Structure Snapshot+1



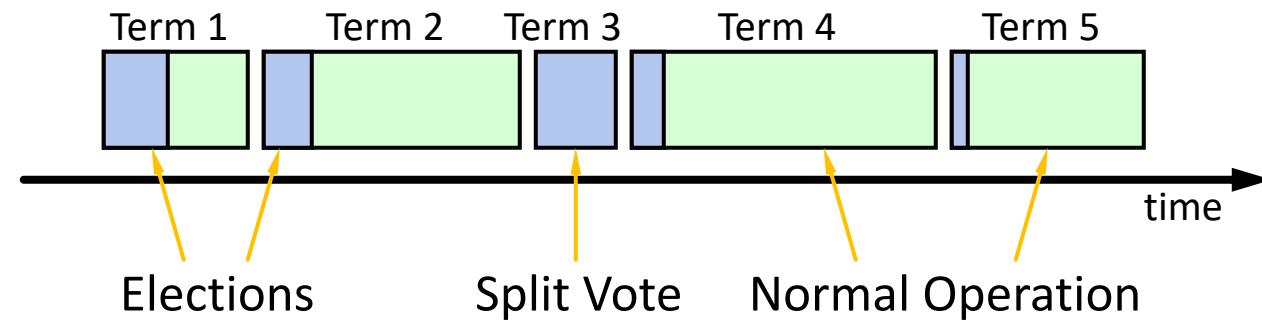
Leaders and Leadership Changes

- 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

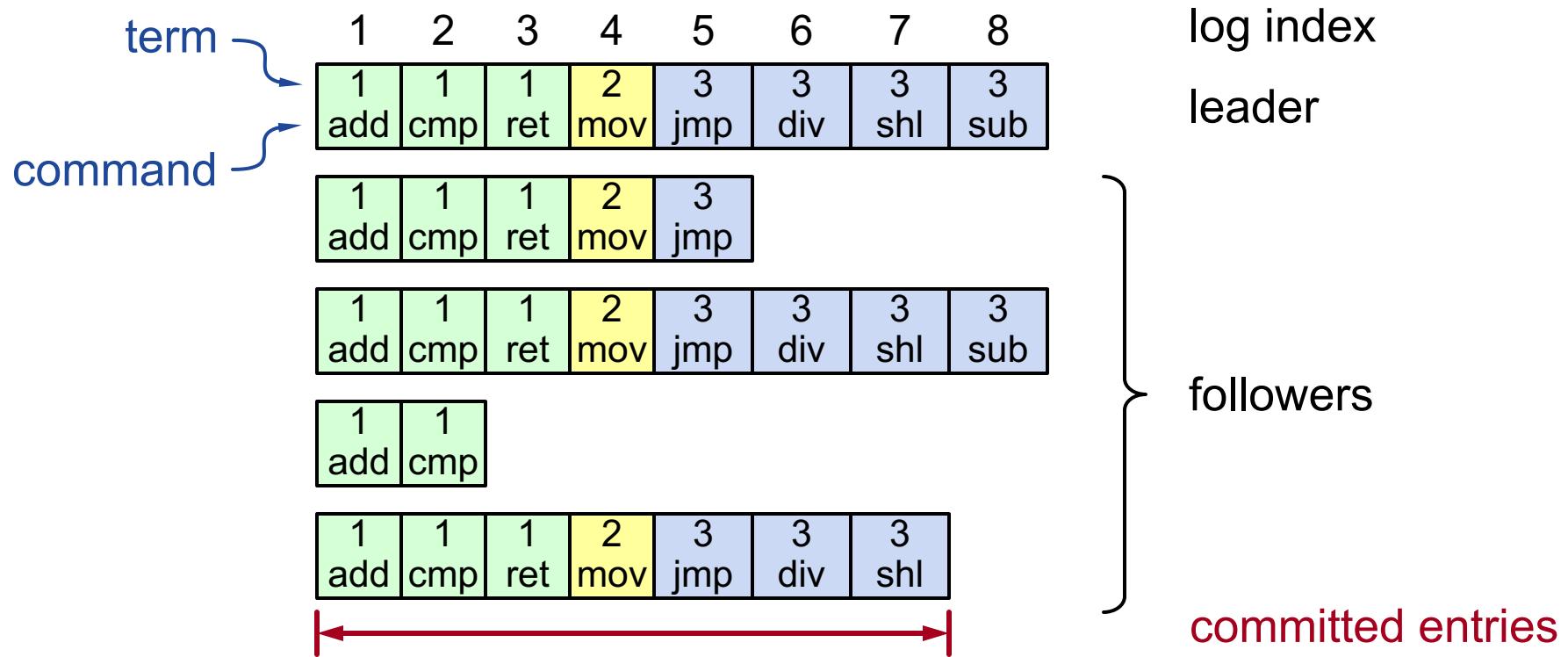
- **Election Safety:** At most, only one leader can be elected
- **Leader Append-Only:** Leaders can only append entries to their logs. They never overwrite old entries.
- **Log Matching:** if two logs have the same index and same term, then the logs are identical in all entries up to that index
- **Leader 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 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**
- Clients only get acknowledgements about committed log entries

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

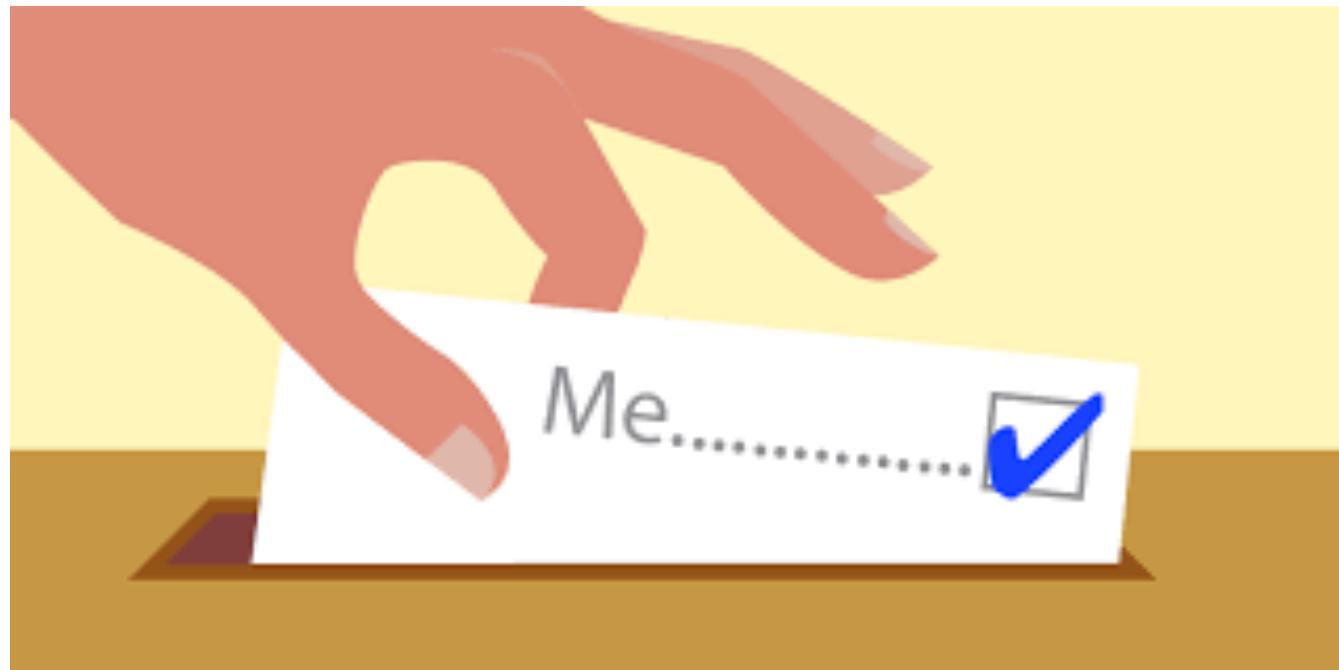
Raft Election Process (1/5)

- 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 (2/5)

The candidate
votes for itself



Raft Election Process (3/5)



- The candidate sends RequestVote messages to all other servers
 - Candidate sends its **term** and **index** to all the other servers
 - If a server is in follower state and receives a RequestVote, it votes for the candidate if the candidate's last term and index are \geq its own committed messages
 - If the server is in candidate state and has a term and index $<$ the other candidate's, it reverts to follower state and votes for the candidate

Raft Election Process (4/5)

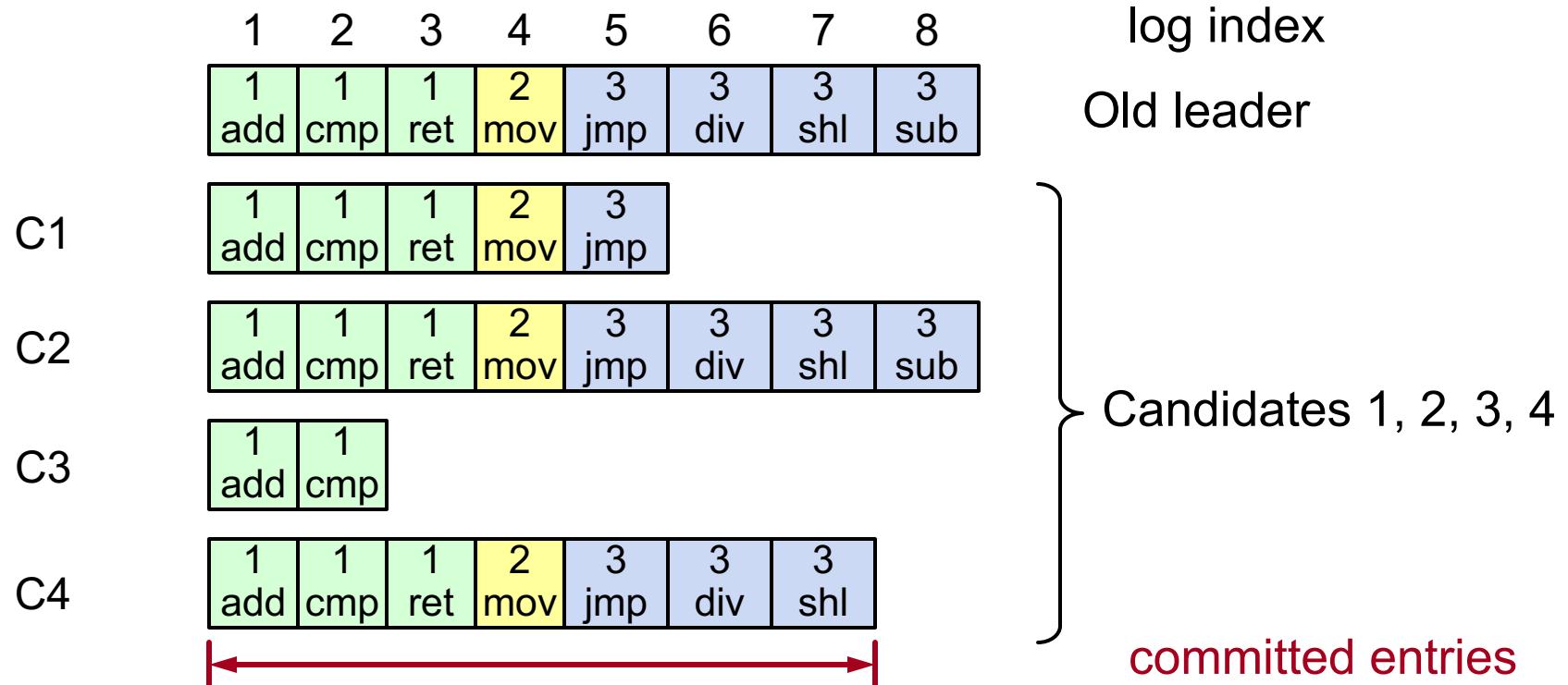
- When a candidate receives votes from a majority of servers, it wins the election and becomes the new leader



A person with long blonde hair tied back in a ponytail is standing at a whiteboard, writing with a marker. The whiteboard has a grid of 5 columns and 6 rows. The first column contains names: 'Pete Bear', 'Peter Fox', 'Frank Lion', 'Robert Wolf', and 'Diana Deer'. The second column contains binary strings: '110101010101', '110101010101', '110101010101', '110101010101', and '110101010101'. The third column contains the text 'To win 4/5' and 'Exacted 0'. The fourth column contains '27 27'. The fifth column is partially visible.

Pete Bear	110101010101	To win 4/5	27
Peter Fox	110101010101	Exacted 0	27
Frank Lion	110101010101		
Robert Wolf	110101010101		
Diana Deer	110101010101		

Election



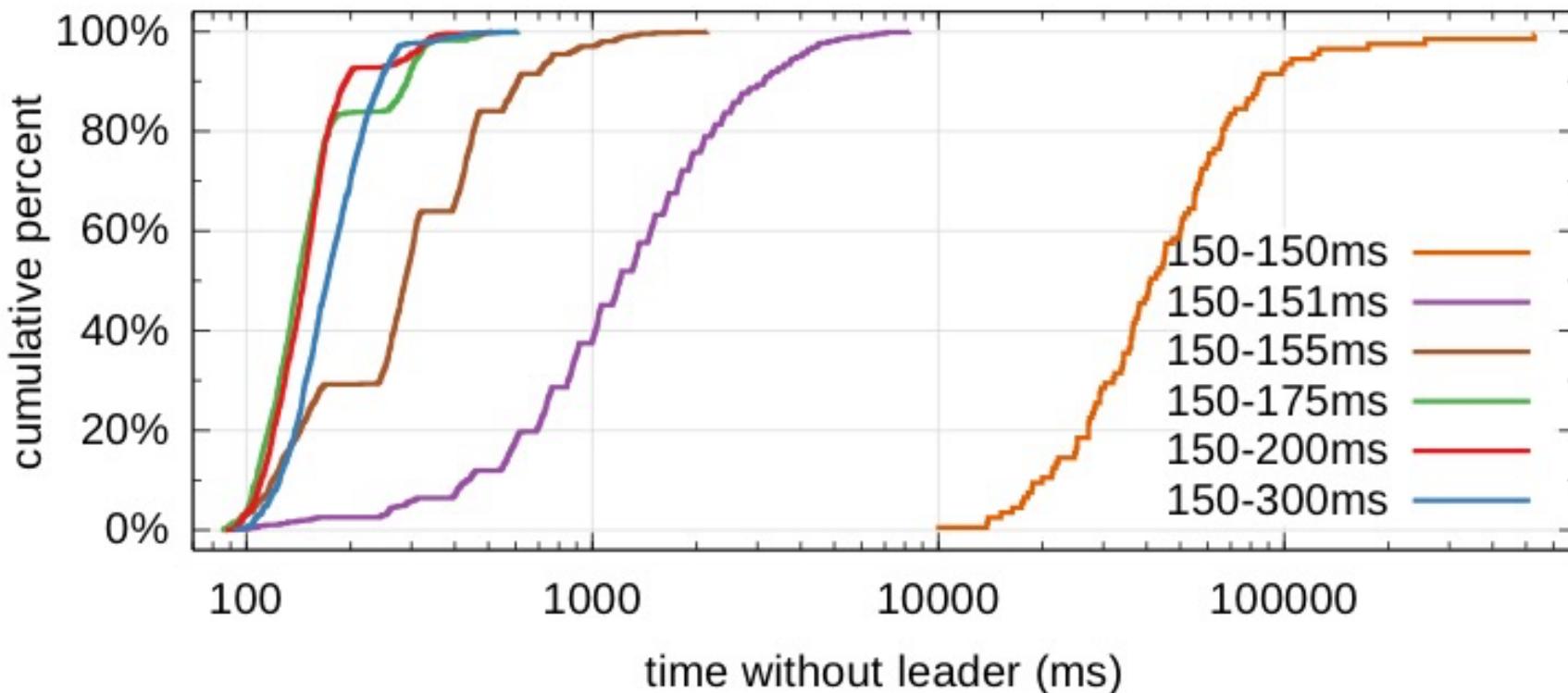
- **Candidates C2 and C4 can win the election with 3 votes**
 - Candidates C1 and C3 will vote for either
 - Candidate C4 can win if gets votes from C1 and C3 before contacting Candidate 2
- **Note a split election is possible: Candidate 1 votes for Candidate 2, and Candidate 3 votes for Candidate 4, for example**
- **Log entry 8 was not committed, but that's ok: the client 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.
 - Typically, only one server at a time will be in candidate state
- This randomness is key for the system to find a new leader quickly

Randomized Timeouts

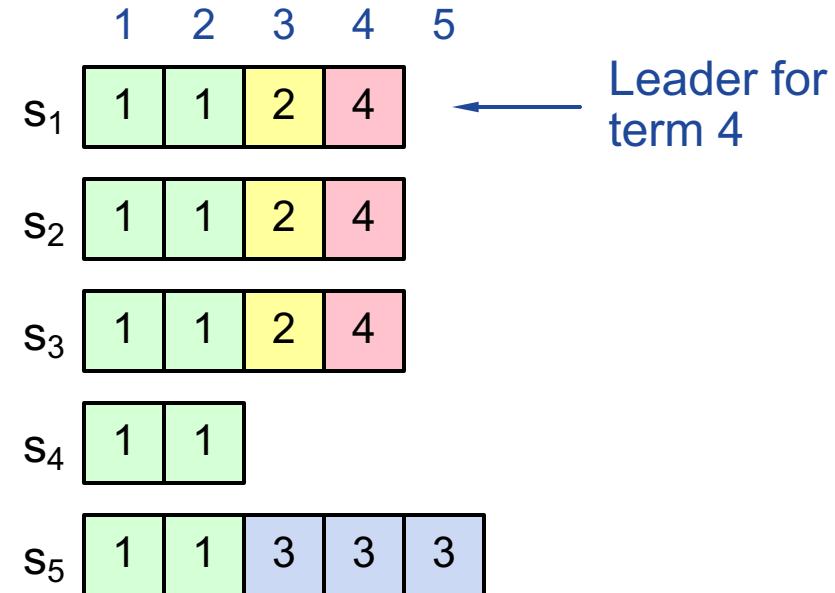
- How much randomization is needed to avoid split votes?



- Conservatively, use random range ~10x network latency

New Commitment Rules

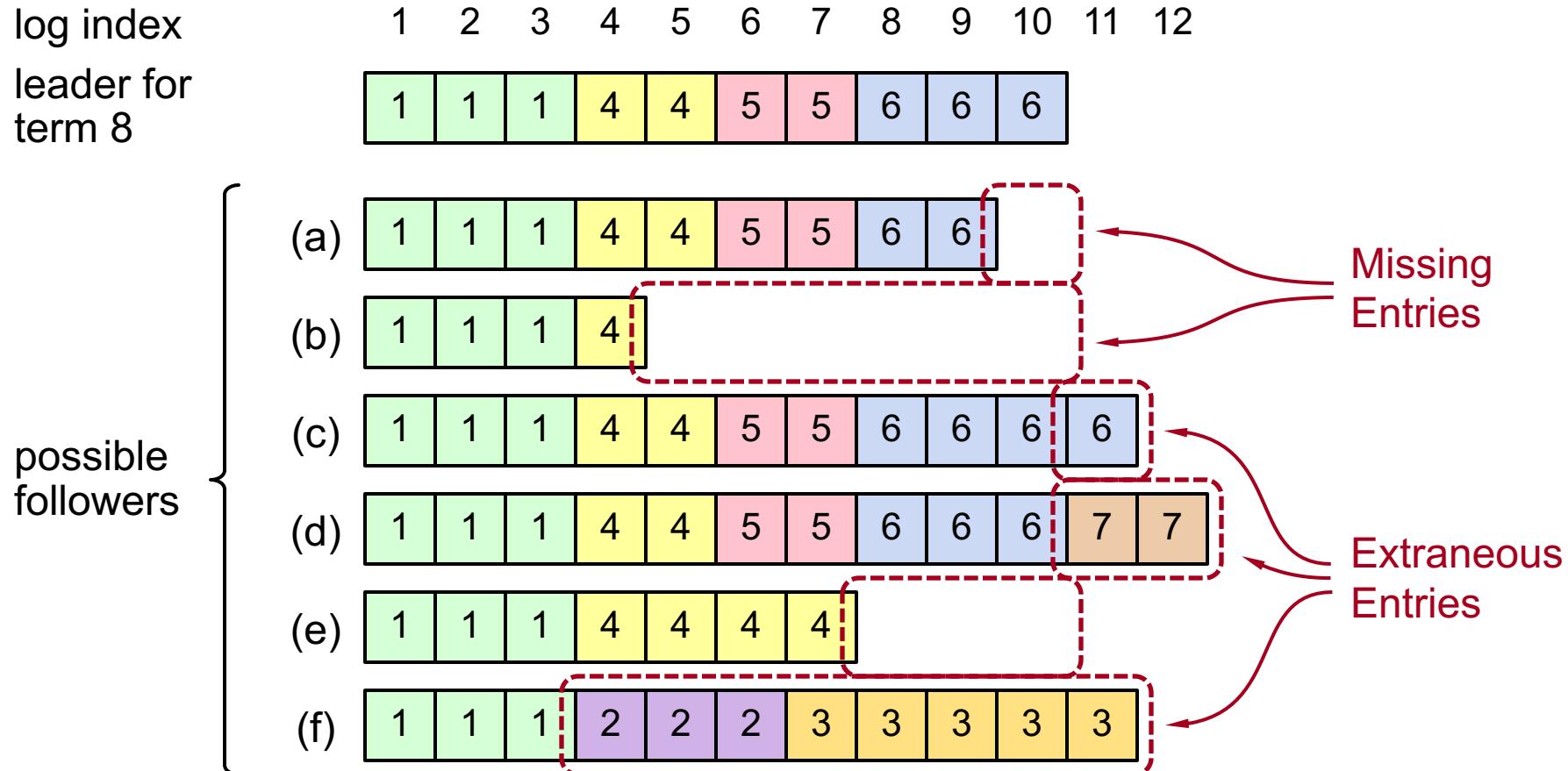
- **For a leader to decide an entry is committed:**
 - Must be stored on a majority of servers
 - **At least one new entry from leader's term must also be stored on majority of servers**
- **Once entry 4 committed:**
 - s_5 cannot be elected leader for term 5
 - Entries 3 and 4 both safe



Combination of election rules and commitment rules makes Raft safe

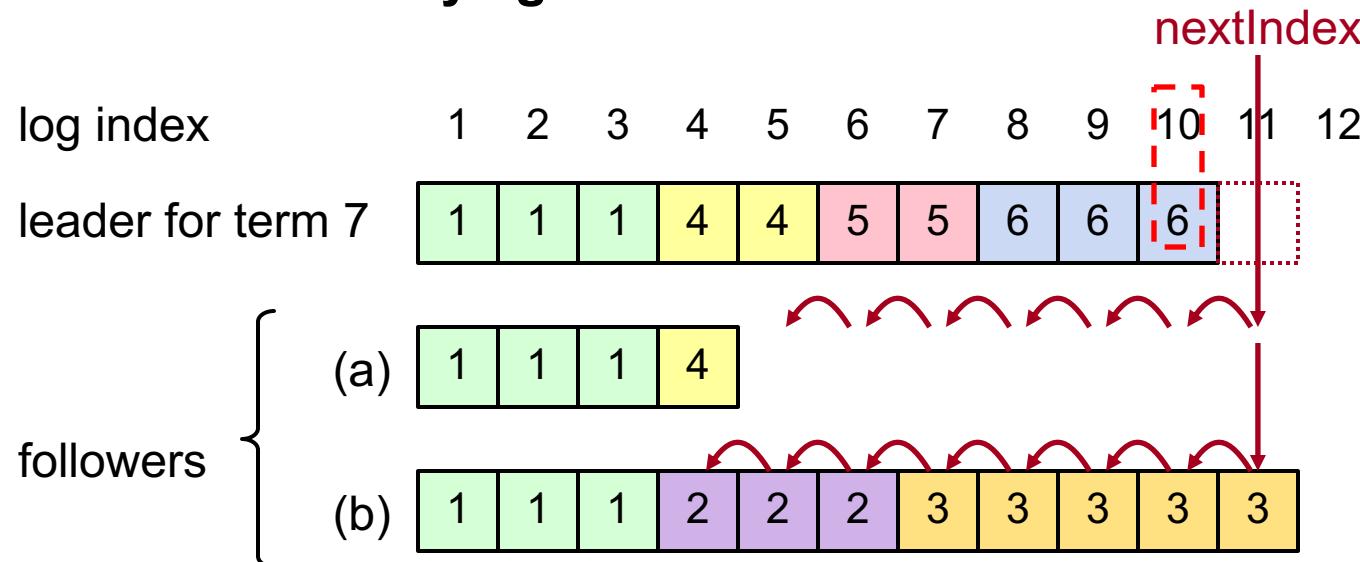
Log Inconsistencies

Leader changes can result in log inconsistencies:



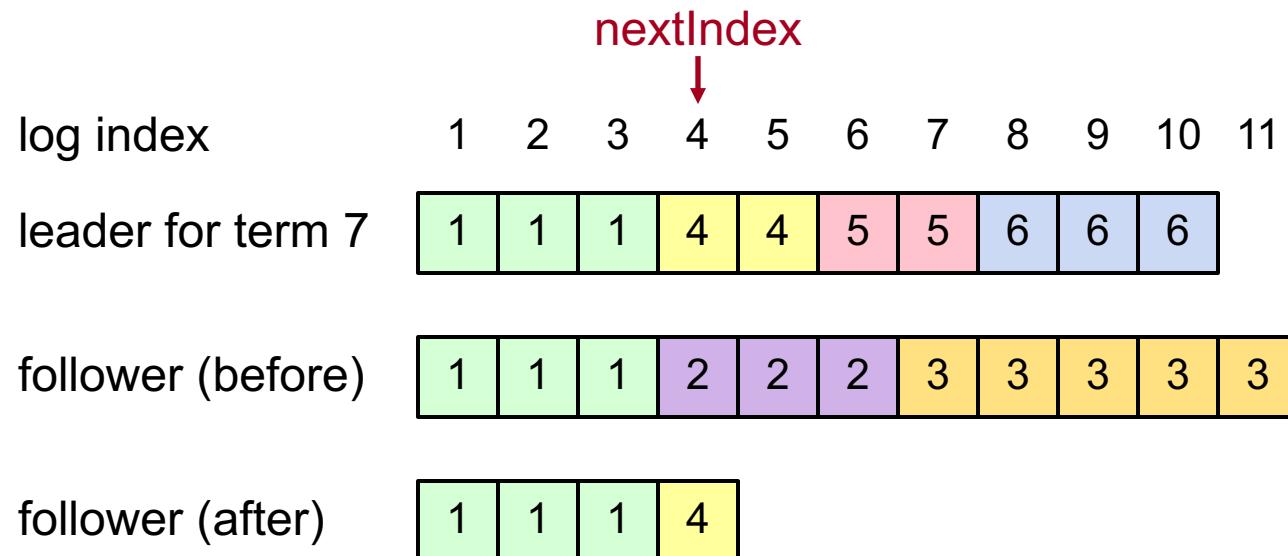
Repairing Follower Logs

- 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
- **Election updates terms of majority of servers**
 - Deposed server cannot commit new log entries

Client Protocol

- **Send commands to leader**
 - If leader unknown, contact any server
 - If contacted server not leader, it will redirect to leader
 - Even for READS
 - Compare with Zookeeper
 - Client guaranteed to get latest committed log state, but Raft doesn't scale as well
- **Leader does not send the response to the client until the command has been logged, committed, and executed by leader's state machine**
- **If request times out (e.g., leader crash):**
 - Client reissues command to some other server
 - Eventually redirected to new leader
 - Retry request with new leader

Raft and Configuration Changes (Brief)

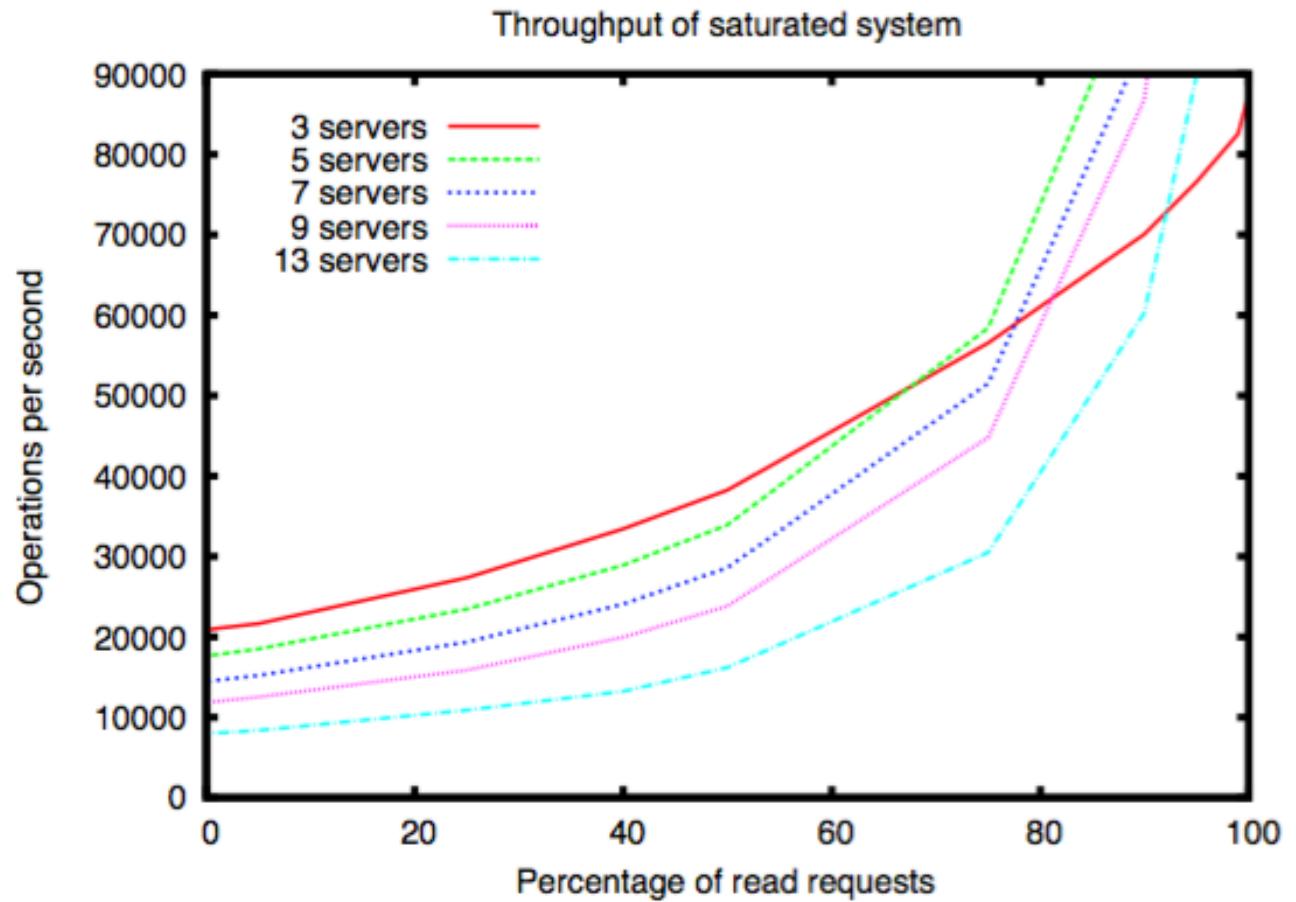
- Raft's consensus approach requires that a leader never gives up on unreachable servers.
 - Quasi-static collections of servers
- Raft can also handle situations in which new servers are added to the system and old servers are cleanly removed.
- It does this by requiring **joint consensus** between the old and new collections.
- New servers are non-voting members until they have come up to speed.
- This is an interesting approach for handling **Continuous Deployment** scenarios.
 - More in a future lecture

Some Limits of Raft (1/4)

- All READ requests also go to the leader
 - Guarantee: all clients get the same response to READ operations
 - The followers are just there as backups
- When the Raft leader is stable, Raft provides strong consistency to its clients
 - Every client request gets the same answer
- But this can limit throughput compared to Zookeeper
 - Zookeeper has weaker consistency but is more scalable for READs

Some Limits of Raft (2/4)

- All the log protocols are designed for clients that primarily READ.
- Strong leader systems will not scale well if there are a lot of WRITES.
 - Can you see why?



Some Limits of Raft (3/4)

- 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
 - You wouldn't build DNS with Raft
 - Can you see why?
- Blockchain (from Bitcoin) is a (maybe) similar logging protocol that works at large scale but (maybe) doesn't work well as small scale

Some Limits of Raft (4/4)

- 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

Replicating databases

