In Search of an Understandable Consensus Algorithm

Diego Ongaro & John Ousterhout

OSDI 2014

*Slides are a variant of slides by Nikolaos Karystinos-Avgerantonis

Clearly better title

Raft: Escaping the Paxos island

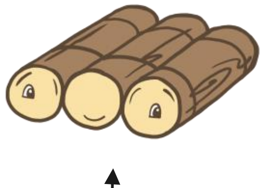
Clearly better title

Raft: Escaping the Paxos island

Okay, that's not the logo...

Raft: Escaping the Paxos island

What is Raft?



- Raft is a consensus algorithm (like Paxos)
- Raft manages a **replicated log**
- Raft's result is equivalent to multi-Paxos
  (single-decree Paxos can only reach consensus for one log entry)

Why do we like Raft?

- Infinitely more understandable than Paxos
  - Separates the key elements of consensus
    - Leader election
    - Log replication
    - Safety
  - Enforces stronger coherency to reduce the state space (a.k.a. reduced nondeterminism and less ways that servers can be inconsistent with each other)
    e.g. Logs in Raft are not allowed to have “holes”

Why do we like Raft?

Thus...

- Raft provides a better foundation for system builders

  Software Development Theorem (not really, but could be)

  ➢ If the programmer intuitively understands the algorithm he/she is implementing, the likelihood of him/her introducing a bug significantly drops.

- Raft is much easier for students to learn

What’s wrong with Paxos?

- It’s hard to understand…
    - Probably, the difficulty arises from the choice of the single-decree (one decision only) as a foundation for the multi-decision protocol (multi-Paxos which is equivalent to Raft)
    - Even single-decree Paxos is tough to grasp…
- It’s not a good foundation for implementations
    - Lamport only describes single-decree thoroughly
    - A multi-Paxos implementation is not widely agreed upon
    - Most systems that aim to implement Paxos usually end up implementing their own variation of the original opaque protocol, to satisfy real-world needs that Paxos doesn’t address

What’s wrong with Paxos?

“There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system… the final system will be based on an unproven protocol”

comment from the Chubby paper regarding Paxos

Replicated state machines

- Consensus algorithms are typically discussed in the context of replicated state machines
- Intuitively, each replica/server “runs” a state machine
- The aim is for all servers to perform identical steps in running their state machines so as to arrive to the same result even when some servers suffer stop-failures
- The replicated state machine abstraction allows solutions to various problems

Replicated state machines

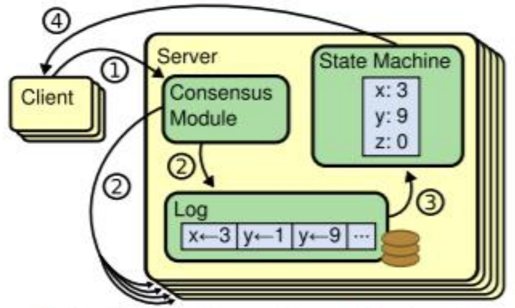


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

- Replicated state machines are usually implemented via a **replicated log**
- Each (local) log stores a sequence of commands that are issued in order to the state machine
- The state machines are deterministic => If the same sequence of commands is executed on all servers, the end result is guaranteed to be the same
- The job of a consensus algorithm is keeping the replicated log **consistent**!

Desirable Properties

We would like our consensus algorithm to:

1. Ensure safety, under a non-byzantine system model with lossy communication -- [With crash-faults, we need **2f+1** replicas where up to f are faulty]
2. Be available, as long as a majority of servers is up
3. Not depend on timing for correctness
4. Be fast in the common case. A request should be able to complete when a majority of the replicas has replied (some slow servers do not compromise overall performance)

The Raft algorithm

- Raft manages a **replicated log**
- At first, Raft requires the election of a **leader**
- The **leader** assumes the following responsibilities:
    - Accepts new log entries from clients (client requests)
    - Replicates entries to the cluster
    - Signals the application of log entries to the servers' state machines
- Having strong leadership simplifies information flow and the management of the log
- In case the leader dies, just elect a new one

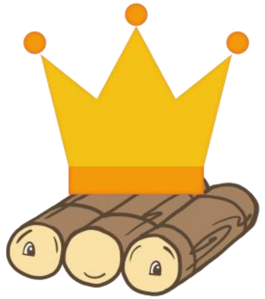
The Raft algorithm

- Raft decomposes the consensus problem to three relatively independent subproblems:
  - Leader election
  - Log replication
  - Safety*
- Safety is guaranteed by maintaining the following property:

  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

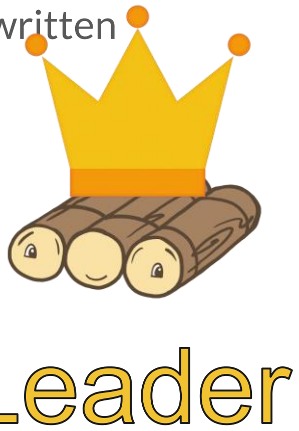
- A Raft cluster with **2n+1** servers can tolerate **n** failures (**n+1** is still a majority)
- There are three possible server states:



Leader Candidate Follower

Raft basics

- There is always at most **one** leader
- The leader handles all client requests (followers redirect clients to the leader)
- The leader replicates log entries to the rest of the cluster
- The leader's log is append-only. Committed entries are never overwritten or deleted



Raft basics

- The candidate state is an intermediary state used to elect a leader
- Candidates request votes from other servers to become leaders
- More on the election process soon!

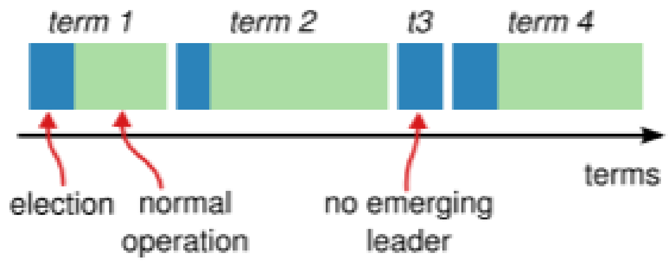
Candidate

Raft basics

- Followers are passive as their name suggests
- They only respond to requests from leaders or candidates
- If a candidate requests a follower's vote, the follower grants it provided a certain condition (we'll see later) holds

Follower

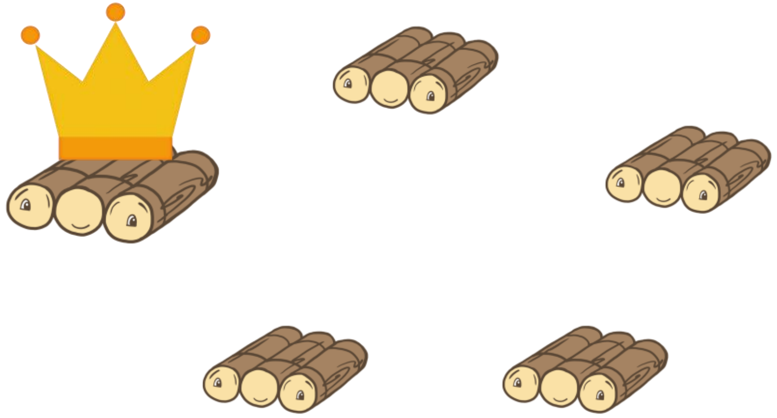
Raft basics



- Time is divided into **terms,** identified by positive, increasing integers
- At the beginning of each term there is an **election**
- During an **election** one or more candidates request votes from the rest of the servers to become leaders
- An **election** usually produces **one** leader for the current term, but elections can also result in a split vote (no candidate gathered a majority), in which case the term ends with no leader and a new term begins.
- Raft ensures **at most** one active leader per term
- Terms act as a kind of logical clock, allowing the detection of stale information stemming primarily from restarting after a crash

Raft basics

- A raft cluster with 5 servers would look like this during normal operation:



Note: *This cluster can tolerate 2 failures.*
3 nodes form a majority.

Raft basics

- Communication in Raft requires only two RPCs:
    - **RequestVote:** issued by candidates during election periods to acquire votes
    - **AppendEntries:** issued by the leader during normal operation to replicate log entries to the cluster. “Empty” AppendEntries calls are also used as a heartbeat so that the leader can assert its authority.

Leader election

- All servers start out as followers
- Servers remain in the follower state as long as they receive RPCs from leaders or candidates
- During idle periods (no client requests), the leader issues empty **AppendEntries** RPCs to indicate to followers that it is still up and running
- If a follower doesn’t hear from a leader or a candidate for a period of time called the **election timeout,** it sets in motion an election to choose a new leader

Leader election

- A follower begins an election by incrementing its current term and transitioning to the **candidate** state
- Following that, it votes for itself and issues **RequestVote** RPCs to all other servers in the cluster
- Three things can happen:
    a. It wins the election (i.e., collects f+1 votes)
    b. Another server wins the election
    c. No winner emerges from the election

(a) It wins the election

- The server received a vote from a majority of the cluster
- Each server votes for exactly one candidate
- The majority rule ensures the elected leader is unique
- The new leader sends out heartbeat messages to all servers to prevent new elections

(b) Another server wins the elections

- While the candidate still waits for votes from others to become leader, it might receive an **AppendEntries RPC** from another server claiming to be leader
- If the candidate’s current term is smaller or equal to the term contained in the AppendEntries RPC, the candidate recognizes the leader as legitimate and returns to the follower state
- Otherwise, the RPC is ignored

(c) No winner emerges from the election

- If many followers revert to candidate state at the same time, votes could be cast in such a way that no candidate receives a majority of votes
- If this happens, the term ends without a leader and a new term begins with another election

Election timeouts

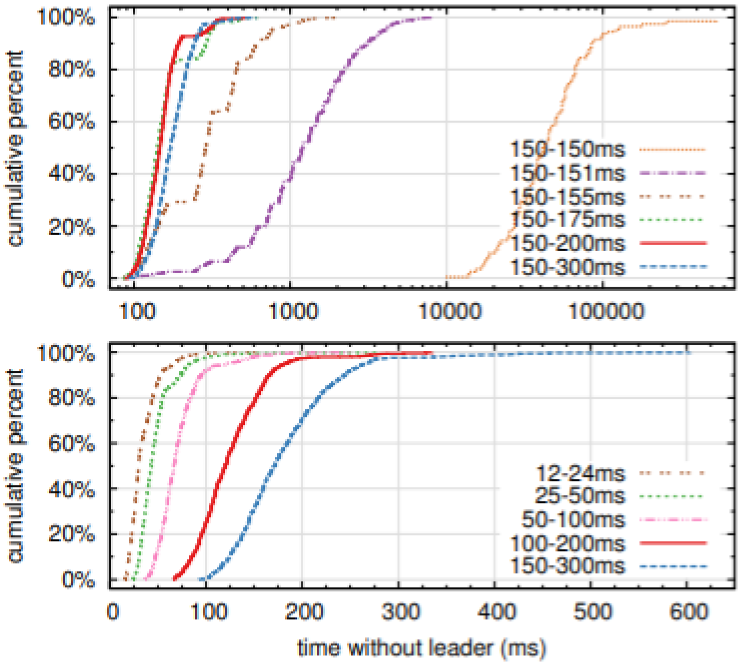
- It would be a bad idea to set a specific election timeout across all servers
- Why?
    - They would all timeout at the same time (after e.g., a leader crash), and revert to candidate state requesting votes from each other
    - This is bad because the probability of a split vote is pretty high
    - The election could still end with a leader since the network possibly introduces uneven delays to messages sent (e.g. a server might get lucky and push out his RequestVote RPCs much faster than the others)

Election timeouts

- **Idea:** “Choose the election timeout value randomly from a fixed interval (e.g. 150-200 ms)”
- This is great!
  - In the common case, a single server times out before all others and easily wins the election and becomes leader (it first sends out RequestVote RPCs to everyone and after receiving a majority of votes, it issues heartbeats [empty AppendEntries] to establish leadership. The rest of the servers usually have not even timed out at this point)
  - If a split vote happens, the servers reset their randomized timeouts for the next term thus the likelihood of a split vote in the next term drops significantly

Election timeouts

- The evaluation section shows that this idea works great in practice
- The top graph varies the length of the fixed interval
- The bottom graph demonstrates that lowering election timeouts minimizes downtime due to absence of leadership



Log replication

- After the election of a leader, its main job is to receive commands from clients and replicate them to the cluster
- After receiving a command, the leader:
    - Appends the command to its log
    - Sends out **AppendEntries RPCs** to every other server
- After replicating a command/log entry to a majority of the servers, the command is considered **committed** and the leader can now apply the command to its state machine and return the result to the client

Log replication

1. Why are the marked entries committed?
2. How do nodes that have fallen behind with the log get back on track? (e.g. node 4 here)

- Answer for question 2: With the help of the leader. We'll see how in a bit...

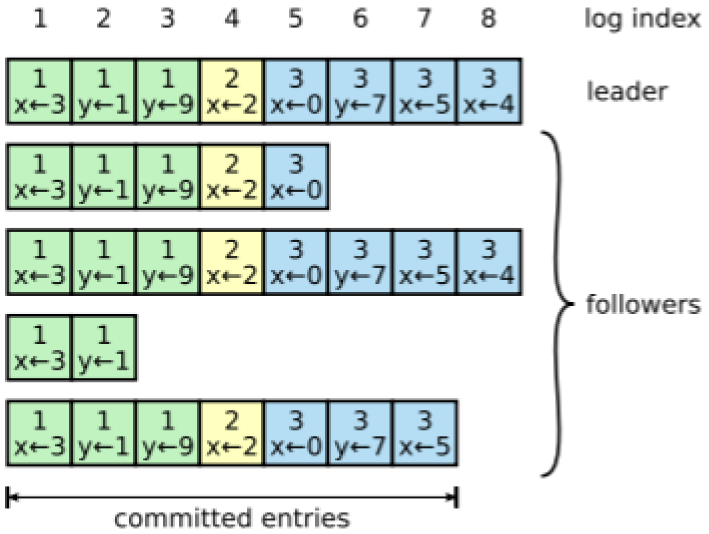


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

Log replication

- The leader is unique
- A leader creates at most one entry with a given index in a given term
- Log entries never change position

Property 1
If two entries in different logs have the same index and term, then they contain the same command.

- **AppendEntries** performs a consistency check : The leader includes the index and term of the previous log entry in the RPC. If the follower does not find a matching entry in its log the RPC fails.
- Inductively, with the empty log as a base case, this forces logs to be extended in a uniform fashion.

Property 2
If two entries in different logs have the same index and term, then the logs are identical in all previous entries.

Log convergence

- A follower’s log might:
  - Be missing entries that are present on the leader’s log
  - Have extra entries that are missing from the leader’s log
  - Be inconsistent in both of the above ways
- Solution: “The leader is always right”
  - The leader forces its followers to duplicate its log
  - How???
    - Leader keeps a **nextIndex** value for every follower marking which log entry is next to be sent to that follower
    - When **AppendEntries** fails, **nextIndex** is decremented for that follower
    - Eventually, the common prefix of the logs is established at which point the leader can start correctly extending the follower’s log (possibly overwriting other entries)

Possible log states for followers

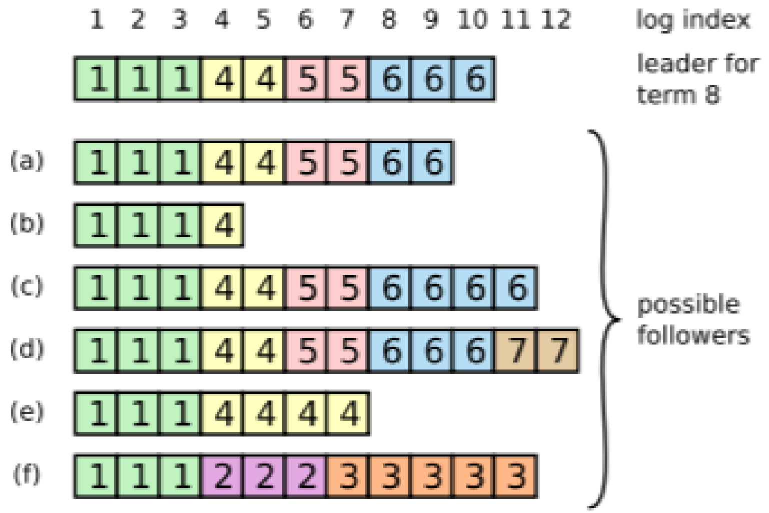


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

Safety

- We have already described leader election and log replication
- These mechanisms are not sufficient to guarantee safety (aka every state machine executes exactly the same commands in exactly the same order)
- Imagine this scenario for example:
    - A leader (**Machine A**) crashes before replicating some log entries to a majority of nodes
    - A new leader (**Machine B**) is elected and several new entries are committed
    - **Machine A** goes live and by chance is immediately elected leader (assume **Machine B** leader crashed)
    - **Machine A** will proceed to overwrite log entries that **Machine B** previously committed

Safety

- How can we prevent this?
- Intuition: “If a `stale` leader can ruin everyone’s logs, be careful about who you elect as leader!”
- Idea:
    - If someone requests my vote but my log is more “up-to-date”, I shouldn’t vote for them
    - “More up-to-date” semantics:
        - Check the index and term of the last entries in both logs
        - If terms differ, then the log with the larger term wins
        - Otherwise, the longer log wins

Safety

- What we just previously described coupled with the fact that a leader must receive a majority of votes means that a follower can only be elected to become leader **only** if its log contains all committed entries.
  (Remember: An entry is committed if it has been replicated to a majority)

Safety

- Small intricacy:
  A leader cannot know that an entry from a previous term is committed once it is stored on a majority of servers

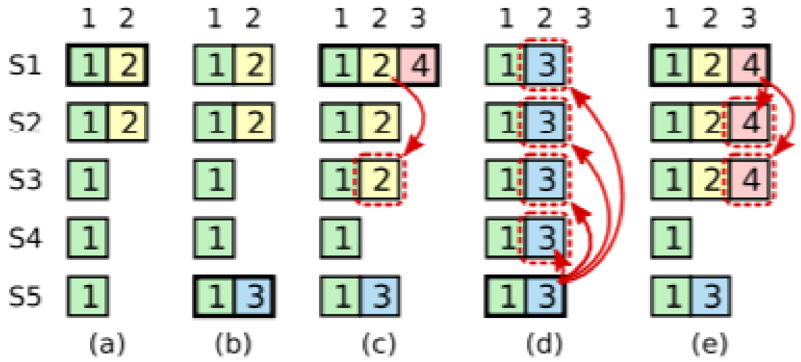


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

So much more...

- We could also more formally prove the safety argument and there's also a neat way to change the cluster's configuration (the machines that take part in the protocol) on the fly, but I'm already on slide 39, so let's not bother...

2 more slides

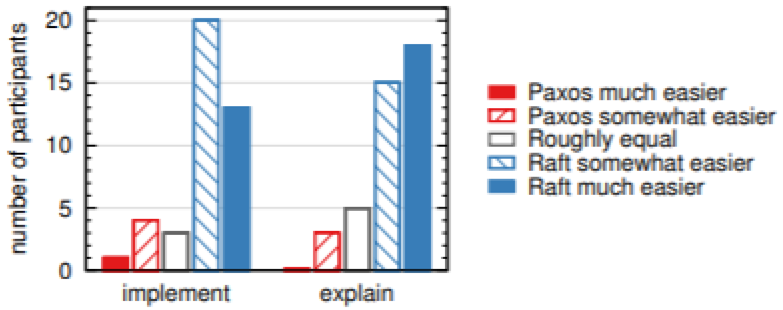
- Follower and candidate failures are simple to handle!
  - Just retry RPCs indefinitely if you get no answer
- For the protocol to function well we need this relationship to hold:
  - broadcastTime << electionTimeout << MTBF
    - broadcastTime: the time needed to issue parallel RPCs to everyone and get a response
    - electionTimeout: I hope you remember what this is 🙂
    - MTBF: Mean time between failures for a server

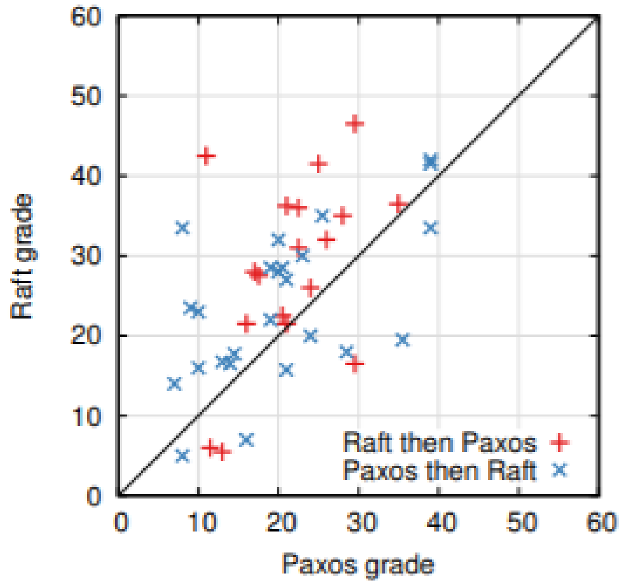
1 more slide

- Implementation
  - 2000 lines of C++ code
  - Source code freely available
  - About 25 independent third-party implementations
- Correctness
  - The consensus mechanism is mechanically proven correct
  - 400 lines of the TLA+ specification language

Last one I swear

- Understandability relative to Paxos
    - Conducted an experimental study on students of Advanced OS and Distributed Computing courses at Stanford and Berkeley
    - Results showed that Raft succeeds in being easier to understand





Let’s see Raft in action!

- Let’s take 5 more minutes to visually understand how Raft works with this amazing visualization: https://raft.github.io/
- I promise it’s fun
- Reminder for me! I need to show you:
    - Crash three of the five servers. Observe. Restart.
    - Initial leader election & stable system condition
    - Replication of one log entry
    - Leader crash
    - Two more log entries
    - Restart crashed node
    - Crash leader with uncommitted entries
    - Crash the new leader with one committed entry

Thanks for your time!

Any questions?...