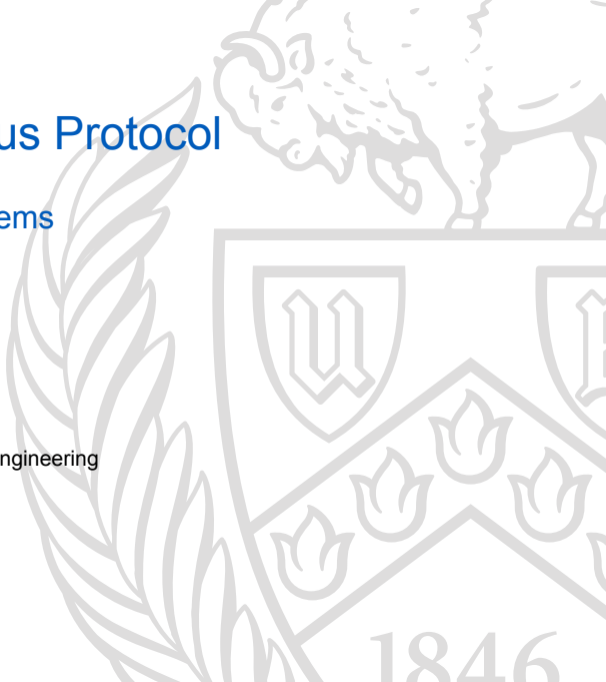


The Raft Consensus Protocol

CSE 486: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo



The Raft Consensus Protocol

Raft [5, 4] is a **relatively new** consensus protocol.

It uses a suite of inter-related protocols to provide:

- Membership
- Leader election
- Consensus on a **sequence of values**

Raft was designed specifically to be **understandable**.

Paxos

Prior to Raft, the most-used consensus algorithm was [Paxos](#) [2].

Paxos is [notoriously hard](#) to understand.

It was often implemented partially, or incorrectly, or just [badly](#).

This is partly due to complexity, and partly due to presentation.
(It's a Lamport [story paper](#).)

Raft was a specific reaction to this problem!

We'll probably see Paxos later.

Decomposition

Raft simplifies consensus by **decomposing it** into smaller problems.

Proving the safety of each part **individually** is sufficient.

Raft elects a **leader** to handle consensus at any given time.

If the leader is **correctly elected**, its decisions will be final.

The leader coordinates many of the details of consensus.

Goals

Raft provides:

- A **log of values** agreed upon by all processes
- Availability in the face of failures
- Robustness to asynchronous message delays and losses

Raft does **not** handle Byzantine failures!

All Raft participants must operate in good faith.

Servers

Raft participants are called **servers**.

Every server has the **same capabilities**.

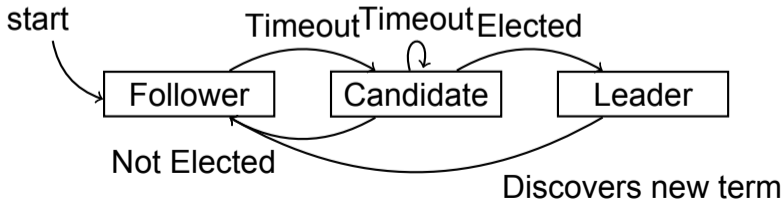
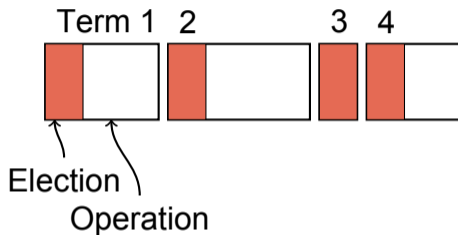
Not all servers perform all actions at all times.

Servers occupy three states:

- **Follower**: Followers replicate and store the agreed-upon log.
- **Candidate**: Candidates emerge to replace failed leaders.
- **Leader**: The (unique at any given time!) leader appends new entries to the log.

Leaders are elected for a **term**, typically after failure.

Server States



Quorum

Raft is based on [quorum](#).

Quorum is a [legal term](#); from Merriam-Webster:

the number (such as a majority) of officers or members of a body that when duly assembled is legally competent to transact business

Quorum systems appeared in the late 1970s [1].

Quorum provides [consensus](#) with [failures](#).

The Quorum Model

Quorum ensures that **no two incompatible changes** can be made.

It does this by requiring that **some subset** of processes agree.

Unlike other consensus we've seen, **not all processes** must agree.

A change that hasn't reached enough processes is **provisional**.

Any change that has reached enough processes is **committed**.

Achieving Quorum in Raft

Quorum is required in three places in Raft:

- Election
- Commitment
- Membership changes

For each of these actions, a quorum of servers must **approve**.

A quorum in Raft is 50% of servers + 1 server.

Servers can **refuse** only elections; others may be delayed.

Safety

50% + 1 server ensures that a change is permanent if:

- No server “forgets” what it has done
- No more than half of the servers fail

By contradiction:

- Assume that 50% + 1 servers agree on X
- Assume that 50% + 1 servers agree on $\neg X$
- **Contradiction:** At least one server agreed on $X \cup \neg X$!

The Raft State Machine

Raft emulates a [state machine](#).

(This is common; recall Lamport Clocks [3].)

Every server [replicates](#) the state machine.

Raft is indifferent to the properties of the state machine.

It merely assumes that:

- There is some starting state
- Deterministic changes are made to that state as log entries are committed

State Transitions

Notionally, **every log entry** is a state change.

Any server can **replay its log** to arrive at the current state.

If every server has the same log, **every server has the same state**.

We often think of state changes as **assignments**.

They can, however, be arbitrarily complex commands!

Commitment

A log entry is **committed** once a quorum of servers records it.

A committed entry **will always persist**, even with failure.

If no more than 50% of servers fail, **some server** will record it.

The rules of raft **ensure** that that server will propagate it.

The set of **committed states** defines the consensus state.

Uncommitted states **may be different** between servers.

Leaders and Terms

Time is divided into **terms** in Raft.

Each term is the tenure of a **leader**.

- It begins with election of the leader
- It ends with (perceived) failure of the leader

Only **one server** can be leader at a time.

Only the leader can append to the log.

If a leader sees a term higher than its own, **it becomes a follower**.

The Logical Clock

Terms form a **logical clock**.

Each term is **numbered**.

Higher numbers **replace lower numbers**, with restrictions.

Any decisions made by the leader of term T following S :

- Preserve every log entry committed as of the end of S
- Are superseded by any decisions made in U following T
subject to the same rules

Elections

After leader failure, **one or more** servers become candidates.

Servers may vote for **only one** server per term.

This means that **no more than one** server can win an election.

It is possible that **no server** wins an election!

In this case, the term will conclude with **a new election**.

Heartbeats

The current leader sends a **heartbeat** to all servers.

It contains:

- The current term
- The first log entry the leader believes this server needs
- The previous log entry's **index** and **term**

If a follower does not hear a heartbeat within a timeout interval, **it starts an election**.

The election is for the **next term**.

Starting an Election

A server that starts an election:

- Immediately votes for itself
- Sends a message to **every other server** asking for votes
- Starts an election timer

The election ends when either:

- The server receives a **quorum** of votes
- The timer expires

In the first case, it starts sending heartbeats.

In the second, it **starts another election**.

Voting

A server A will only vote for a server B if:

- A has not voted during this term
- B 's log is **at least as up-to-date** as A 's

A log is more **up-to-date** if:

- It contains a later term
- It ends with the same term but is **longer**

Election Safety

These rules guarantee that the elected leader:

- Knows about **every committed log entry**
 - More than half of the servers voted for it
 - It was **at least as up-to-date** as the servers that voted for it
- Is unique in a given term
 - More than half the servers voted for it
 - Servers vote only once per term

Log Management

Only the **elected leader** can append to the log.

Requests for state change are submitted by **clients**.

The leader confirms the request **only once the entry commits**.

An entry may not be on **all servers** when the leader confirms.

Appending an Entry

To append an entry:

1. A client request arrives at the leader
2. The leader sends the entry to **every up-to-date server**
3. When the entry commits, the leader confirms to the client
4. Updates are retried until **every server** eventually commits

The **highest committed index** is on every heartbeat.

Servers **apply committed entries** to their state machine.

Commitment

An entry commits when either:

- It propagates to a quorum **within the term** it is proposed
- It is in a leader's log when a later entry is committed

If an entry fails to commit **during its term**, it may be lost.

When the leader learns that an entry commits, it **updates its heartbeat**.

Safety **does not depend** on any server hearing that heartbeat!

Primacy of the Leader

When a leader is elected, it starts appending to **its own log**.

Other servers may:

- Have newer log entries
- Be missing log entries
- Have **both newer and missing** log entries

All newer log entries on other servers must be:

- Uncommitted
- From earlier terms

The new leader's log **becomes the canonical log**.

Processing Heartbeats

Recall that heartbeats contain:

- The current term
- The newest entry the leader believes a server needs
- The previous entry's term and index

A server applies the update if:

- Its current term is no larger than the heartbeat
- It has a log entry with the index and term of the heartbeat's predecessor

Newer Entries

If a server has **newer entries** than an accepted heartbeat

- They **must not have achieved quorum**
- They are **not committed**

Therefore they can be **discarded**.

The server will replace **all later entries** with the new entry.

Received: Index 2 Term 3 PrevTerm 1

Old:

1	1	2	2
---	---	---	---

New:

1	1	3
---	---	---

Missing Entries

If a server **does not know the predecessor** of a heartbeat:

- It rejects the heartbeat
- The leader **backs up one entry**

Received: Index 2 Term 3 PrevTerm 1

Old: **1**

New: **1**

Received: Index 1 Term 1 PrevTerm 1

Old: **1**

New: **1 1**

Safety

Election rules ensure that **the leader knows every committed entry**.

The leader always **sends the term of the previous entry**.

The leader **only proposes one entry** at any index.

The leader's history **replaces any conflicting entries**.

Configuration Changes

Raft maintains membership as a **configuration of servers**.

Changing configurations requires:

- A quorum of the **old configuration**
- A quorum of the **new configuration**

Configuration changes are **special log entries**.

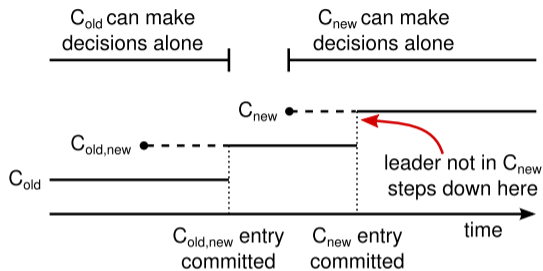
Transition

The configuration change requires **two phases**:

- A quorum of old servers acknowledge the new configuration
- A quorum of old + new servers adopt the new configuration

This ensures that there is **never a time with two leaders**.

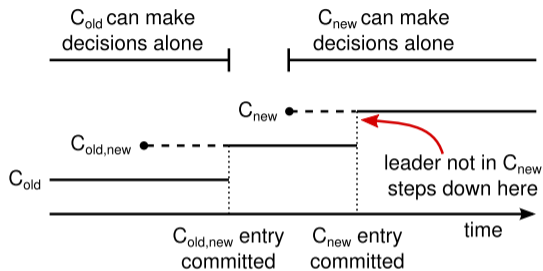
Phase 1



In the first phase, the leader is either:

- In the old configuration
- Elected by a majority of the **union** of the old and new members

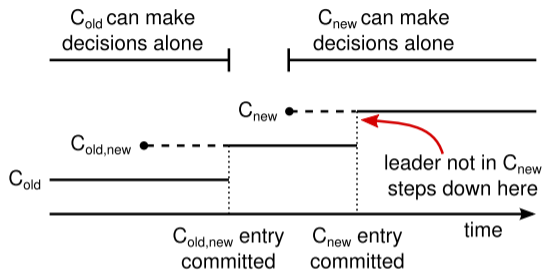
Phase 1



A leader in the old configuration **proposes a union configuration**.

When it commits, **all servers that commit it** use it for quorum.

Phase 2



After the **union configuration** commits, the leader **proposes the new configuration**

When it commits, a **leader from the new configuration** must be elected.

Safety

The union configuration provides safety.

A quorum of **old servers** must adopt the union configuration.

A quorum of **both old and new servers** must agree to remove outgoing servers.

This prevents **a minority from receiving quorum** at any point.

Avoiding Impossibility

Raft **doesn't contravene FLP**.

It is **technically possible** to have eternal elections.

In this case, **nothing commits**.

Raft uses **randomized timeouts** to reduce the window for this.

If server failures are **several orders of magnitude** less frequent than the timeout interval, **consensus is likely**.

Timeouts are $\ll 1$ s, failures are > 1 month!

More Information

Some really great resources on Raft are:

- The USENIX presentation by Diego Ongaro [4]
- The Raft web site at <https://raft.github.io/>
- The Secret Lives of Data at <http://thesecretlivesofdata.com/raft/>

The extended version [6] may help clear up details.

Summary

- Raft provides **consensus** through **quorum**.
- Almost **half of the participants** can fail without losing consensus.
- **Decomposing** elections, membership changes, and log manipulation makes Raft **easier to understand**.

References I

Required Readings

- [5] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *Proceedings of USENIX Annual Technical Conference*. USENIX, June 2014, pp. 305–319. URL: <https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf>.

Optional Readings

- [1] David K. Gifford. *Weighted Voting for Replicated Data*. Tech. rep. CSL-79-14. Xerox PARC, Sept. 1979. URL: http://bitsavers.org/pdf/xerox/parc/techReports/CSL-79-14_Weighted_Voting_for_Replicated_Data.pdf.

References II

- [2] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Transactions on Computing Systems* 16.2 (May 1998), pp. 133–169. DOI: 10.1145/279227.279229. URL: <https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-Part-Time-Parliament.pdf>.
- [3] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: 21.7 (July 1978). Ed. by R. Stockton Gaines, pp. 558–565. URL: <http://lamport.azurewebsites.net/pubs/time-clocks.pdf>.

References III

- [4] Diego Ongaro. *In Search of an Understandable Consensus Algorithm*. USENIX ATC '14 Presentation. 2014. URL: <https://youtube.com/watch?v=no5Im1daS-o>.
- [6] Diego Ongaro and John Ousterhout. *In Search of an Understandable Consensus Algorithm (Extended Version)*. Tech. rep. Stanford University, May 2014.

Copyright 2021, 2023, 2024 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.