Paxos vs Raft

Have we reached consensus on distributed consensus?

Heidi Howard
University of Cambridge

heidi.howard@cl.cam.ac.uk
heidihoward.co.uk
@heidiann360

Failures are inevitable

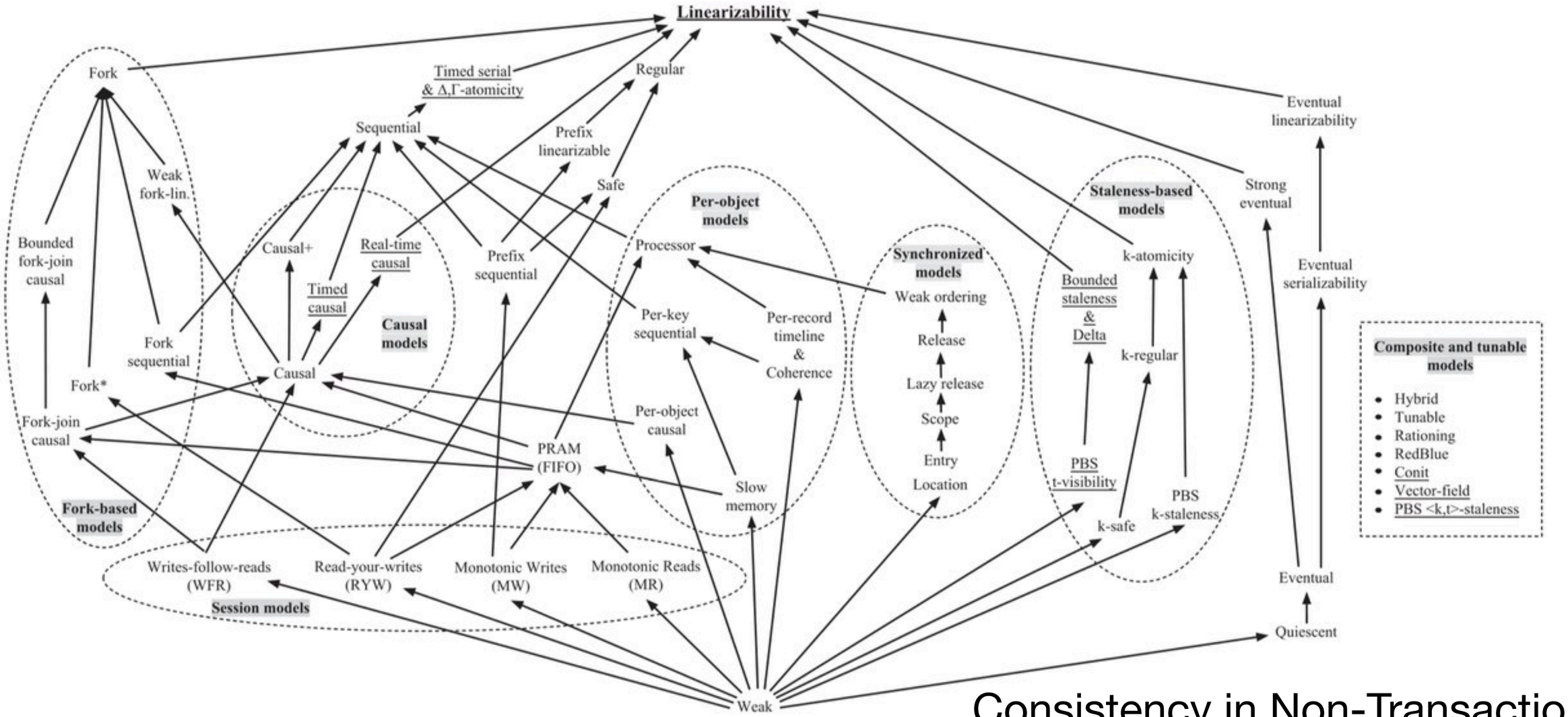
```
Received: by jumbo.dec.com (5.54.3/4.7.34)
        id AA09105; Thu, 28 May 87 12:23:29 PDT
Date: Thu, 28 May 87 12:23:29 PDT
From: lamport (Leslie Lamport)
Message-Id: <8705281923.AA09105@jumbo.dec.com>
To: src-t
Subject: distribution


There has been considerable debate over the years about what
constitutes a distributed system.  It would appear that the following
definition has been adopted at SRC:

  A distributed system is one in which the failure of a computer
  you didn't even know existed can render your own computer
  unusable.
```

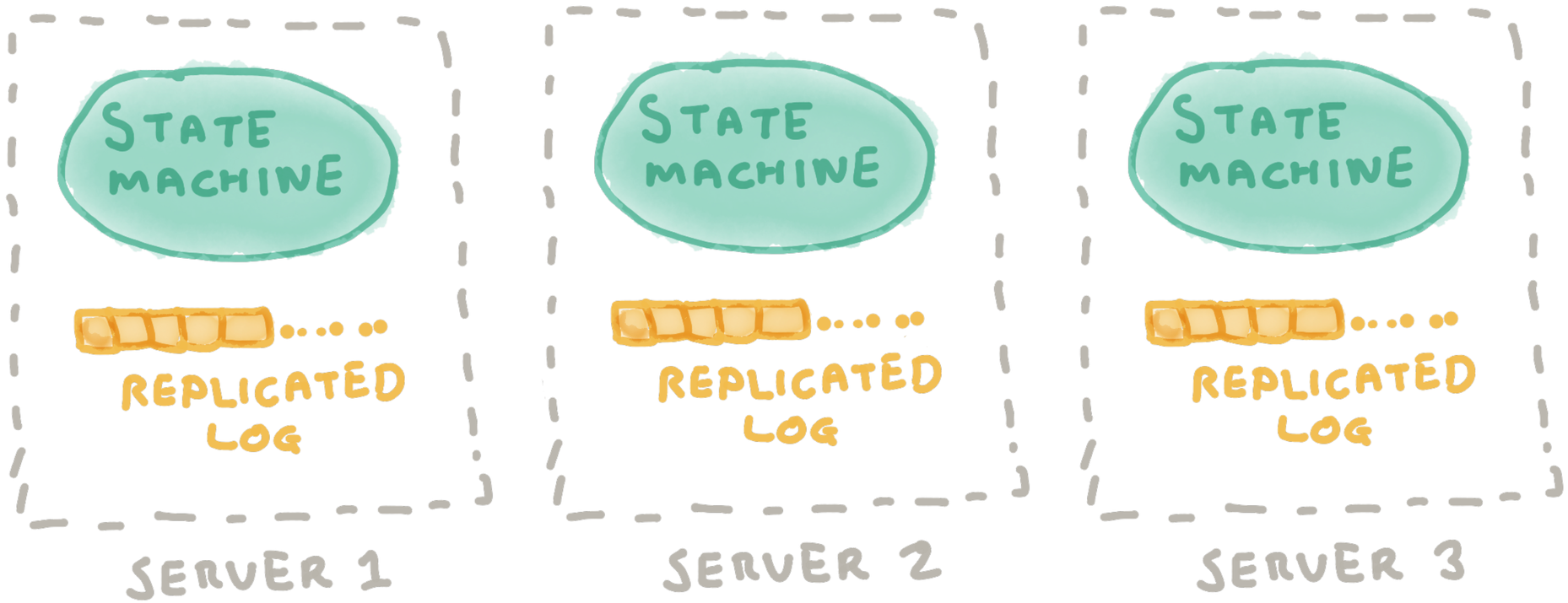
Email sent to a DEC SRC bulletin board

Many possible consistency guarantees



Consistency in Non-Transactional Distributed Storage Systems

State machine replication



This approach is wide spread in production

Cockroach DB

PaxosStore:
High-availability Storage Made Practical in WeChat

Jianjun Zheng† Qian Lin§†* Jiatao Xu† Cheng Wei†
Chuwei Zeng† Pingan Yang† Yunfan Zhang†

†Tencent Inc. §National University of Singapore
†{rockzheng, sunnyxu, dengoswei, eddyzeng, ypaapyyang, ifanzhang}@tencent.com
§linqian@comp.nus.edu.sg

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Clustrix

Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas

Microsoft

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.
{jasonbaker,chrisbond,jcorbett,jfurman,akhorlin,jimlarson,jm,yaweili,alloyd,vadimy}@google.com

Large-scale cluster management at Google with Borg

Abhishek Verma† Luis Pedrosa‡ Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes
Google Inc.

Dooz

HashiCorp
Consul

Team Paxos vs Team Raft

Amazon ECS

cassandra

PaxosStore:
High-availability Storage Made Practical in WeChat

Jianjun Zheng[†] Qian Lin[§†*] Jiatao Xu[†] Cheng Wei[†]
Chuwei Zeng[†] Pingan Yang[†] Yunfan Zhang[†]

[†]Tencent Inc. [§]National University of Singapore
[†]{rockzheng, sunnyxu, dengoswei, eddyzeng, ypaapyyang, ifanzhang}@tencent.com
[§]linqian@comp.nus.edu.sg

Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas

Microsoft

Clustrix

ceph

infinit

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.
{jasonbaker,chrisbond,jcorbett,jfurman,akhorlin,jimlarson,jm,yaweili,alloyd,vadimy}@google.com

Large-scale cluster management at Google with Borg

Abhishek Verma[†] Luis Pedrosa[‡] Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes

Google Inc.

Doozer

LogDevice

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

neo4j

Log Cabin

github.com/logcabin

TiKV

HashiCorp Consul

neo4j

RethinkDB

hazelcast

Atomix

etcd

Cockroach DB

In the blue corner, Paxos

- The classic solution
- Published in 1998 at ACM ToCS
- 3K citations
- 1K repos on GitHub

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

In the red corner, Raft

- The new solution
- Published in 2014 at ATC
- 1.2K citations
- 3K repos on GitHub

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.

Which algorithm is the best solution to distributed consensus?

This question is surprisingly hard to answer

$I3(p) \triangleq$ [Associated variables: $prevBal[p]$, $prevDec[p]$, $nextBal[p]$]

$\land\ prevBal[p] = MaxVote(\infty, p, \mathcal{B})_{bal}$
$\land\ prevDec[p] = MaxVote(\infty, p, \mathcal{B})_{dec}$
$\land\ nextBal[p] \geq prevBal[p]$

$I4(p) \triangleq$ [Associated variable: $prevVotes[p]$]

$(status[p] \neq idle) \Rightarrow$
$\forall v \in prevVotes[p] : \land\ v = MaxVote(lastTried[p], v_{pst}, \mathcal{B})$
$\land\ nextBal[v_{pst}] \geq lastTried[p]$

$I5(p) \triangleq$ [Associated variables: $quorum[p]$, $voters[p]$, $decree[p]$]

$(status[p] = polling) \Rightarrow$
$\land\ quorum[p] \subseteq \{v_{pst} : v \in prevVotes[p]\}$
$\land\ \exists B \in \mathcal{B} : \land\ quorum[p] = B_{qrm}$
$\land\ decree[p] = B_{dec}$
$\land\ voters[p] \subseteq B_{vot}$
$\land\ lastTried[p] = B_{bal}$

$I6 \triangleq$ [Associated variable: $\mathcal{B}$]

$\land\ B1(\mathcal{B}) \land B2(\mathcal{B}) \land B3(\mathcal{B})$
$\land\ \forall B \in \mathcal{B} : B_{qrm}$ is a majority set

$I7 \triangleq$ [Associated variable: $\mathcal{M}$]

$\land\ \forall NextBallot(b) \in \mathcal{M} : (b \leq lastTried[owner(b)])$
$\land\ \forall LastVote(b, v) \in \mathcal{M} : \land\ v = MaxVote(b, v_{pst}, \mathcal{B})$
$\land\ nextBal[v_{pst}] \geq b$
$\land\ \forall BeginBallot(b, d) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \land (B_{dec} = d)$
$\land\ \forall Voted(b, p) \in \mathcal{M} : \exists B \in \mathcal{B} : (B_{bal} = b) \land (p \in B_{vot})$
$\land\ \forall Success(d) \in \mathcal{M} : \exists p : outcome[p] = d \neq \text{BLANK}$

The Paxons had to prove that I satisfies the three conditions given above. The first condition, that I holds initially, requires checking that each conjunct is true for the initial values of all the variables. While not stated explicitly, these initial values can be inferred from the variables' descriptions, and checking the first condition is straightforward. The second condition, that I implies consistency, follows from $I1$, the first conjunct of $I6$, and Theorem 1. The hard part was proving the third condition, the invariance of I, which meant proving that I is left true by every action. This condition is proved by showing that, for each conjunct of I, executing any action when I is true leaves that conjunct true. The proofs are sketched below.

$I1(p)$ $\mathcal{B}$ is changed only by adding a new ballot or adding a new priest to B_{vot} for some $B \in \mathcal{B}$, neither of which can falsify $I1(p)$. The value of $outcome[p]$ is changed only by the **Succeed** and **Receive** *Success* **Message** actions. The enabling condition and $I5(p)$ imply that $I1(p)$ is left true by the **Succeed** action. The enabling condition, $I1(p)$, and the last conjunct of $I7$ imply that $I1(p)$ is left true by the **Receive** *Success* **Message** action.

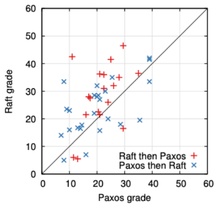


Figure 14: A scatter plot comparing 43 participants' performance on the Raft and Paxos quizzes. Points above the diagonal (33) represent participants who scored higher for Raft.

lecture covered enough material to create an equivalent replicated state machine, including single-decree Paxos, multi-decree Paxos, reconfiguration, and a few optimizations needed in practice (such as leader election). The quizzes tested basic understanding of the algorithms and also required students to reason about corner cases. Each student watched one video, took the corresponding quiz, watched the second video, and took the second quiz. About half of the participants did the Paxos portion first and the other half did the Raft portion first in order to account for both individual differences in performance and experience gained from the first portion of the study. We compared participants' scores on each quiz to determine whether participants showed a better understanding of Raft.

We tried to make the comparison between Paxos and Raft as fair as possible. The experiment favored Paxos in two ways: 15 of the 43 participants reported having some prior experience with Paxos, and the Paxos video is 14% longer than the Raft video. As summarized in Table 1, we have taken steps to mitigate potential sources of bias. All of our materials are available for review [28, 31].

On average, participants scored 4.9 points higher on the Raft quiz than on the Paxos quiz (out of a possible 60 points, the mean Raft score was 25.7 and the mean Paxos score was 20.8); Figure 14 shows their individual scores. A paired t-test states that, with 95% confidence, the true distribution of Raft scores has a mean at least 2.5 points larger than the true distribution of Paxos scores.

We also created a linear regression model that predicts a new student's quiz scores based on three factors: which quiz they took, their degree of prior Paxos experience, and

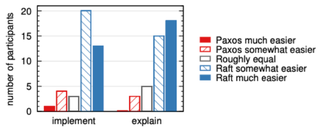


Figure 15: Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

the order in which they learned the algorithms. The model predicts that the choice of quiz produces a 12.5-point difference in favor of Raft. This is significantly higher than the observed difference of 4.9 points, because many of the actual students had prior Paxos experience, which helped Paxos considerably, whereas it helped Raft slightly less. Curiously, the model also predicts scores 6.3 points lower on Raft for people that have already taken the Paxos quiz; although we don't know why, this does appear to be statistically significant.

We also surveyed participants after their quizzes to see which algorithm they felt would be easier to implement or explain; these results are shown in Figure 15. An overwhelming majority of participants reported Raft would be easier to implement and explain (33 of 41 for each question). However, these self-reported feelings may be less reliable than participants' quiz scores, and participants may have been biased by knowledge of our hypothesis that Raft is easier to understand.

A detailed discussion of the Raft user study is available at [31].

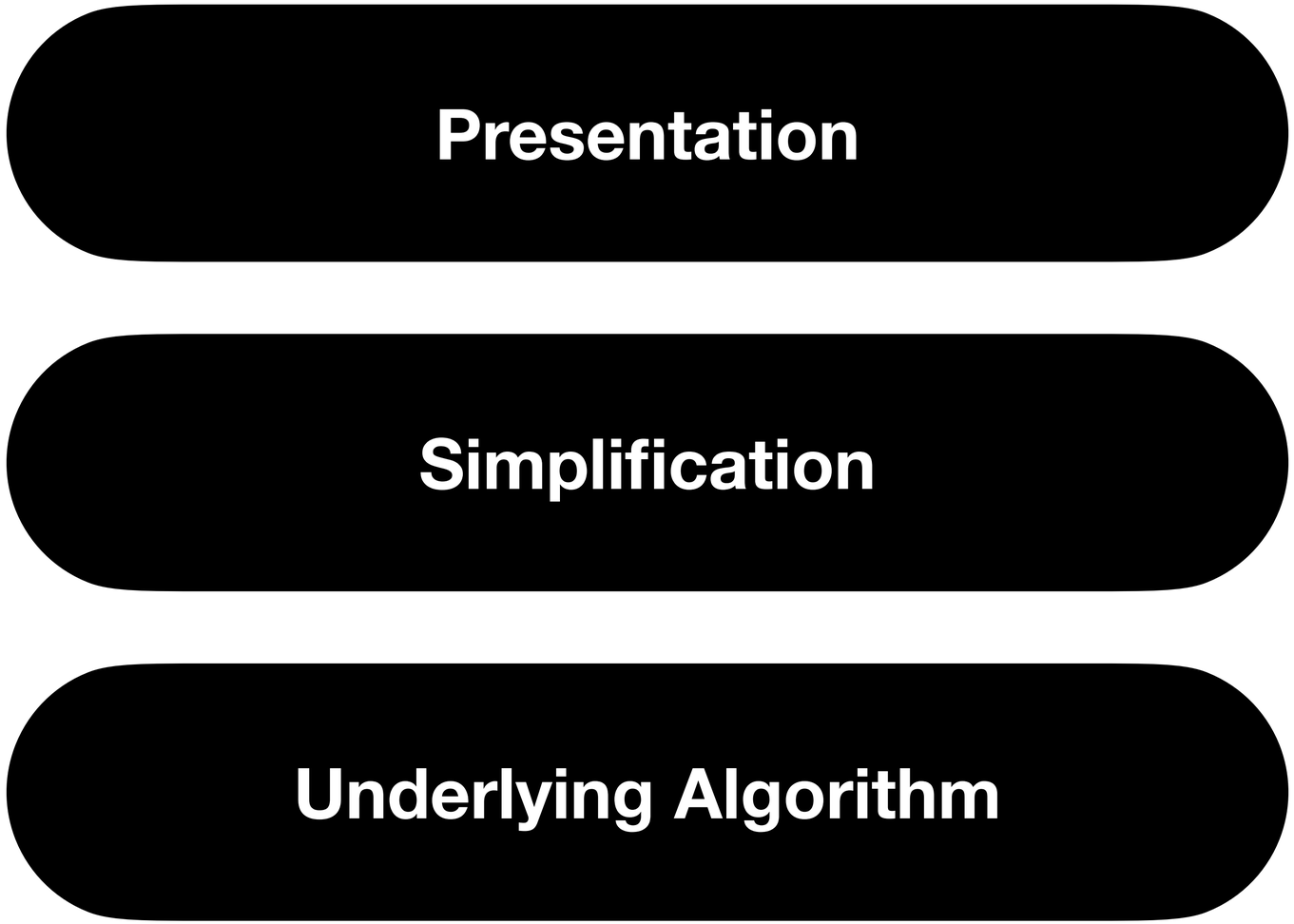
9.2 Correctness

We have developed a formal specification and a proof of safety for the consensus mechanism described in Section 5. The formal specification [31] makes the information summarized in Figure 2 completely precise using the TLA+ specification language [17]. It is about 400 lines long and serves as the subject of the proof. It is also useful on its own for anyone implementing Raft. We have mechanically proven the Log Completeness Property using the TLA proof system [7]. However, this proof relies on invariants that have not been mechanically checked (for example, we have not proven the type safety of the specification). Furthermore, we have written an informal proof [31] of the State Machine Safety property which is complete (it relies on the specification alone) and rela-

Concern	Steps taken to mitigate bias	Materials for review [28, 31]
Equal lecture quality	Same lecturer for both. Paxos lecture based on and improved from existing materials used in several universities. Paxos lecture is 14% longer.	videos
Equal quiz difficulty	Questions grouped in difficulty and paired across exams.	quizzes
Fair grading	Used rubric. Graded in random order, alternating between quizzes.	rubric

Table 1: Concerns of possible bias against Paxos in the study, steps taken to counter each, and additional materials available.

The Raft magic



Our approach is to “Raft-ify” Paxos

A Paxos Algorithm

This summarises our simplified, Raft-style Paxos algorithm. The text in red is unique to Paxos.

State

Persistent state on all servers: (Updated on stable storage before responding to RPCs)

currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)

log[ ] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)

lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on candidates: (Reinitialized after election)

entries[] Log entries received with votes

RequestVote RPC

Invoked by candidates to gather votes

Arguments:

term candidate’s term

leaderCommit candidate’s commit index

Results:

term currentTerm, for candidate to update itself

voteGranted true indicates candidate received vote

entries[] follower’s log entries after leaderCommit

Receiver implementation:

1. Reply false if term < currentTerm
2. Grant vote and send any log entries after leaderCommit

Rules for Servers

All Servers:

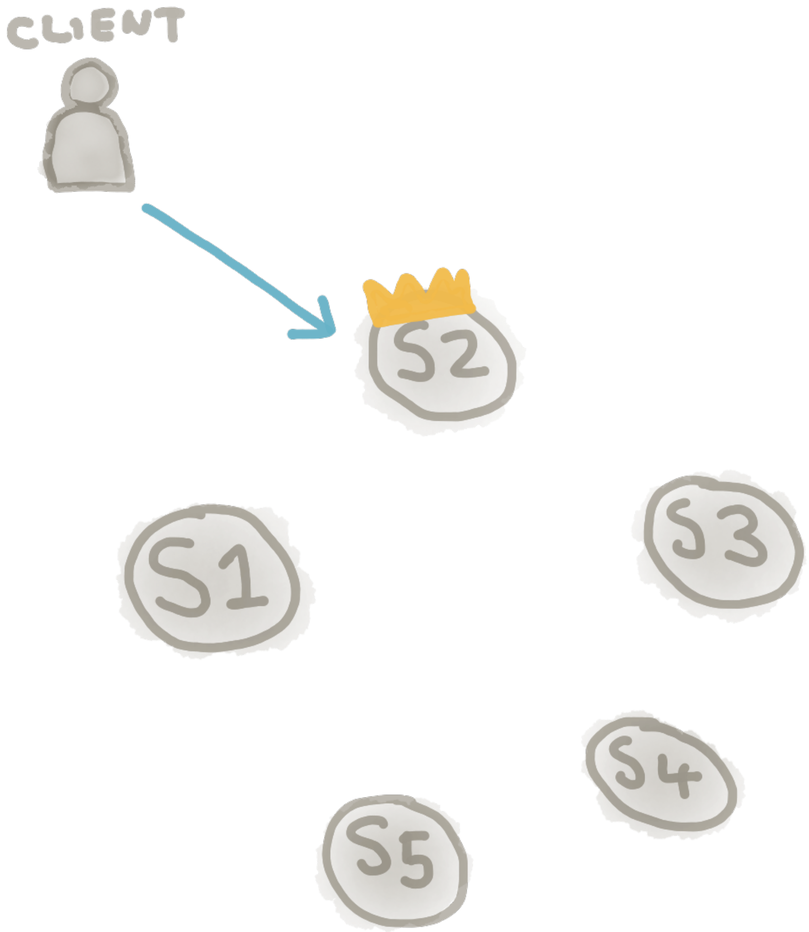
- If commitIndex > lastApplied: increment lastApplied and apply log[lastApplied] to state machine

How do Paxos & Raft differ in their approach to distributed consensus?

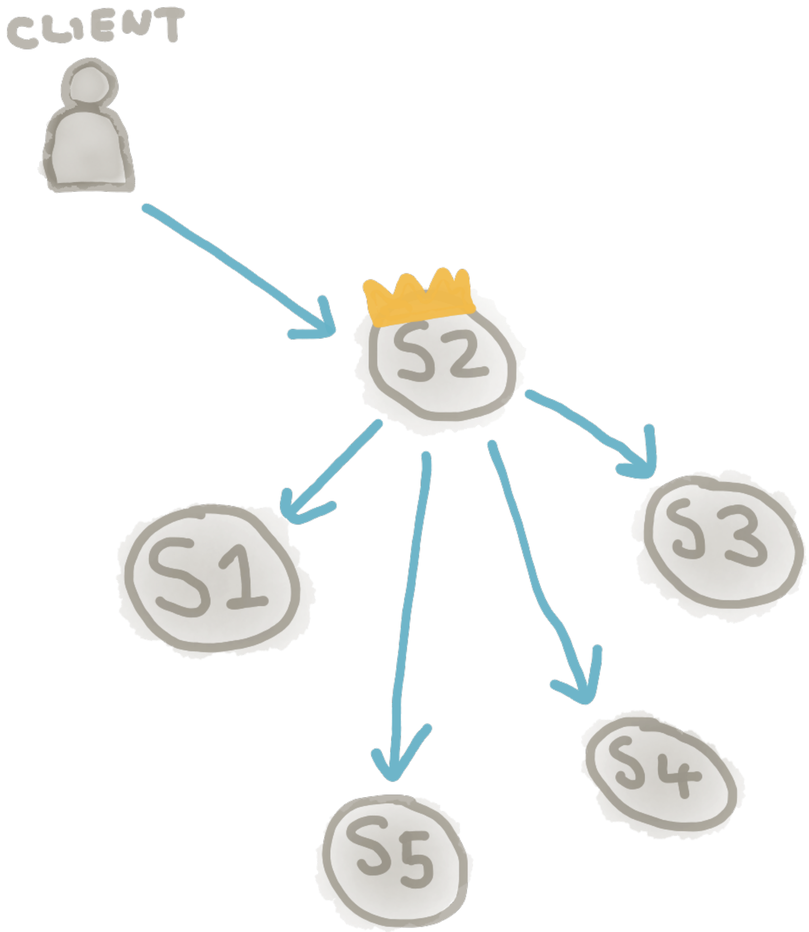
High-level approach to consensus

High-level approach to consensus

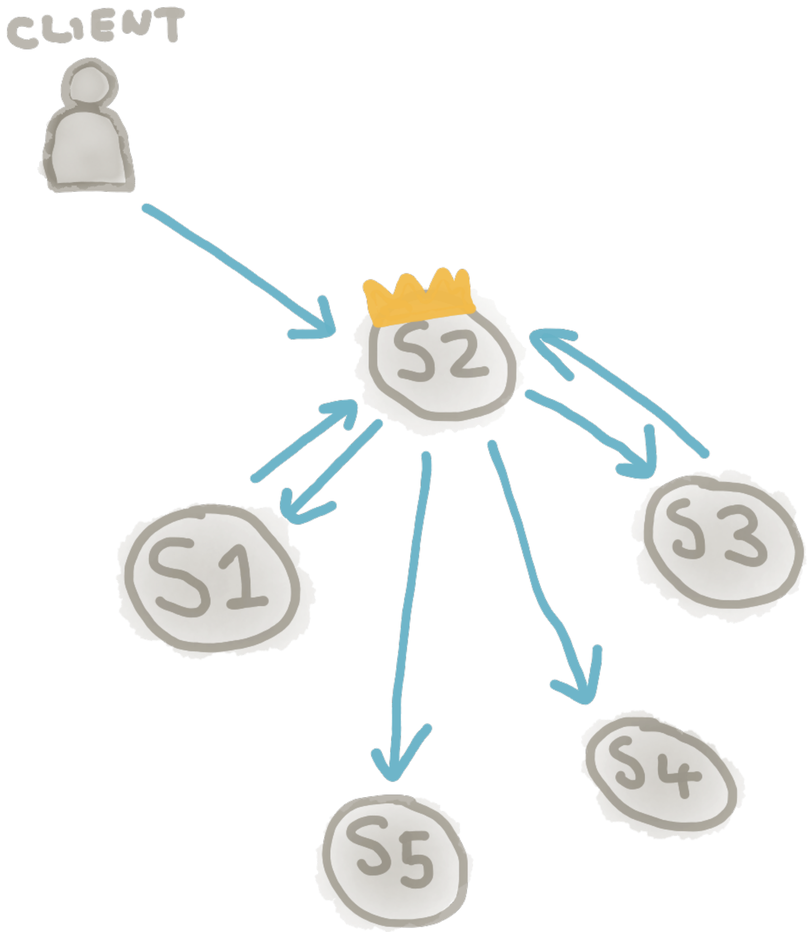
High-level approach to consensus



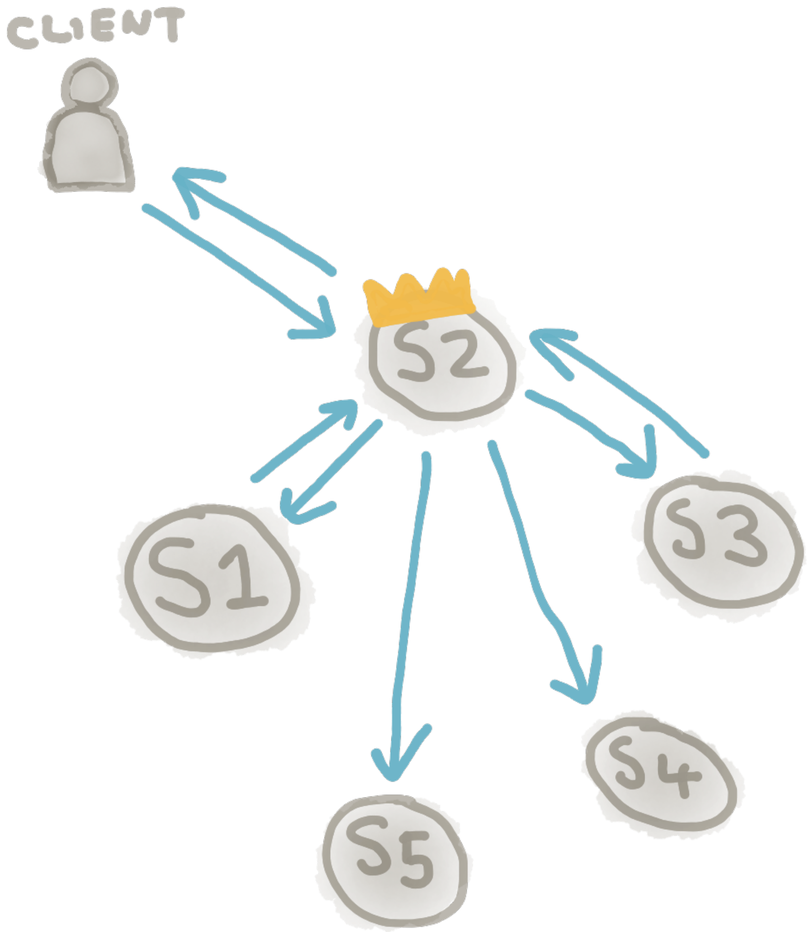
High-level approach to consensus



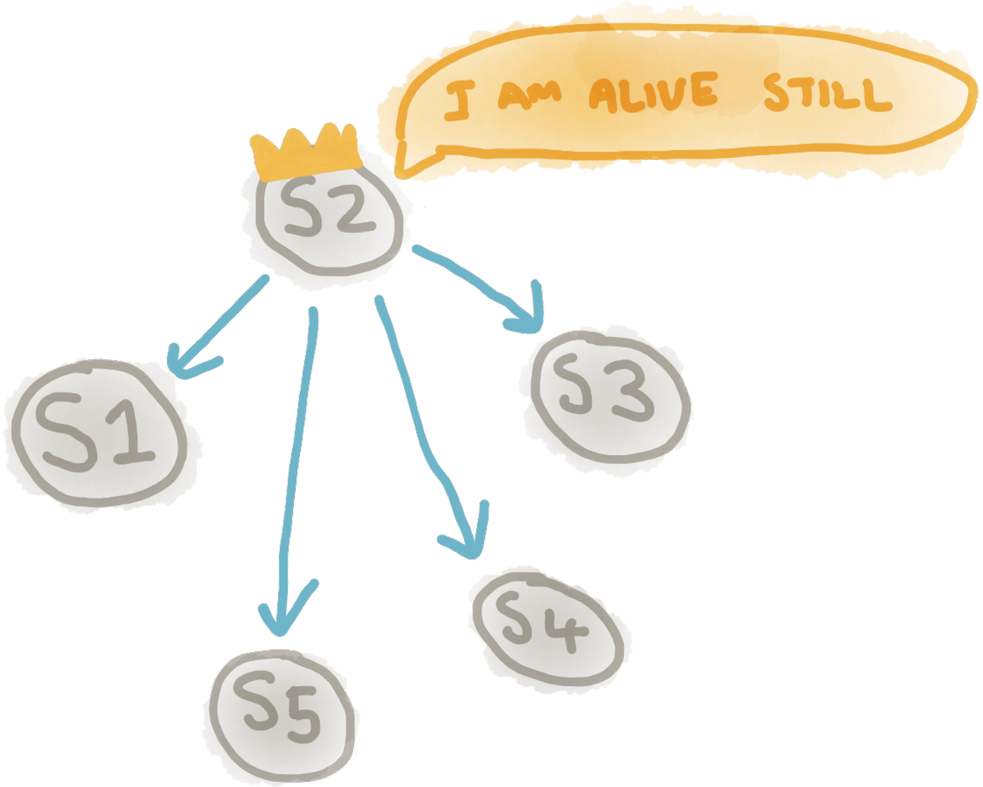
High-level approach to consensus



High-level approach to consensus

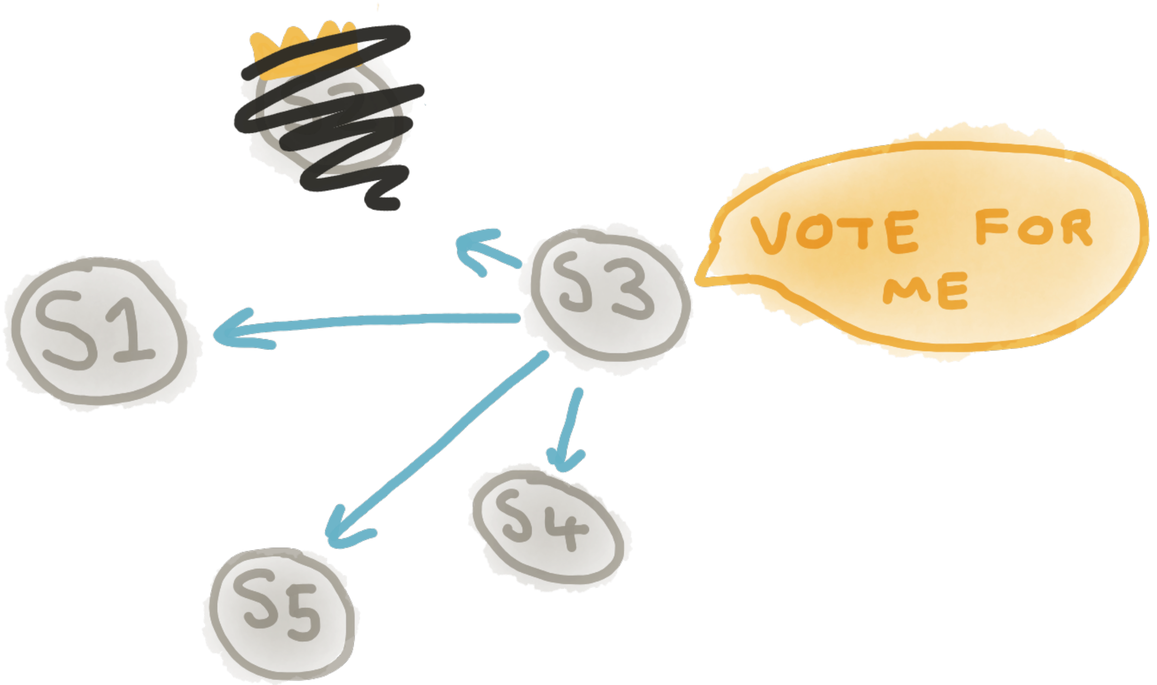


High-level approach to consensus

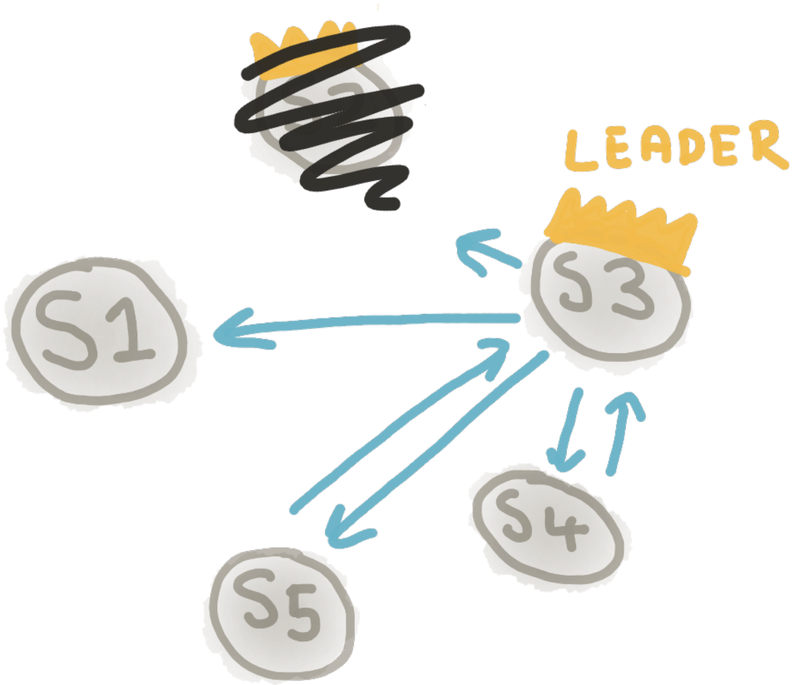


High-level approach to consensus

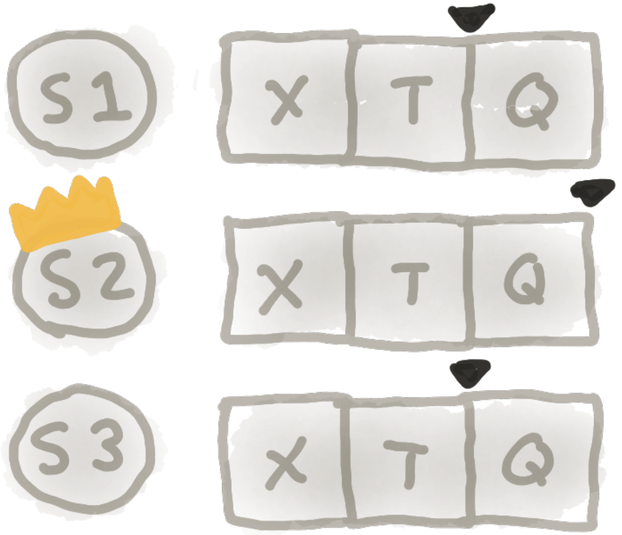
High-level approach to consensus

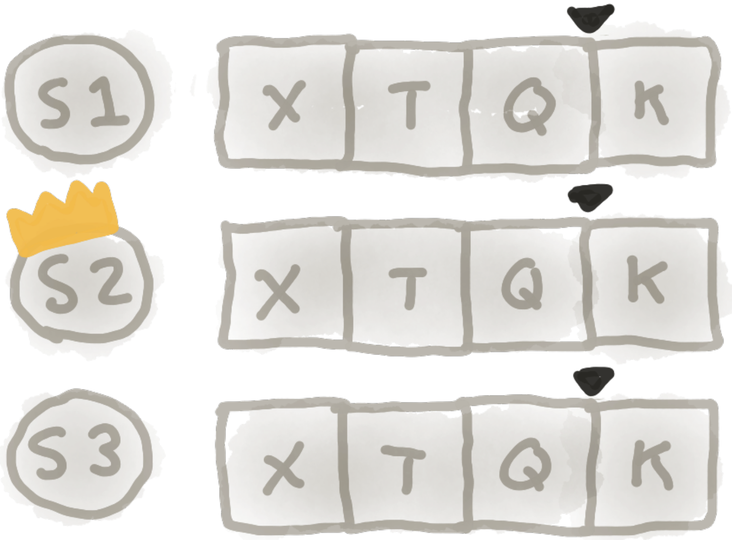


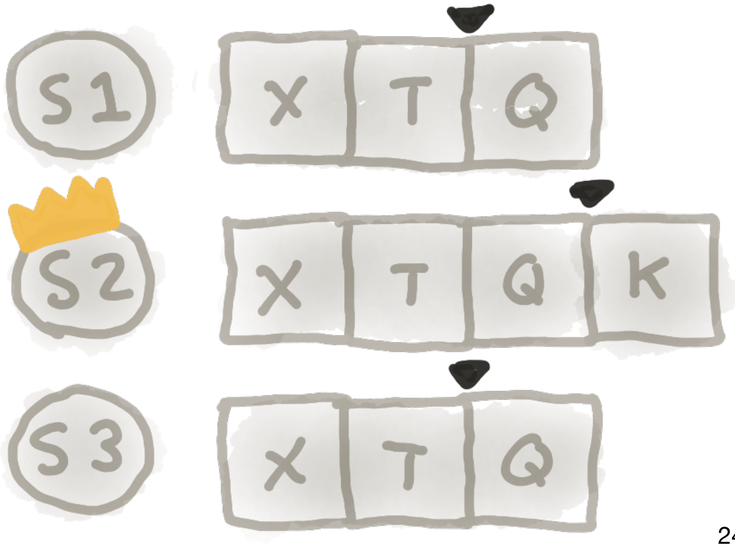
High-level approach to consensus

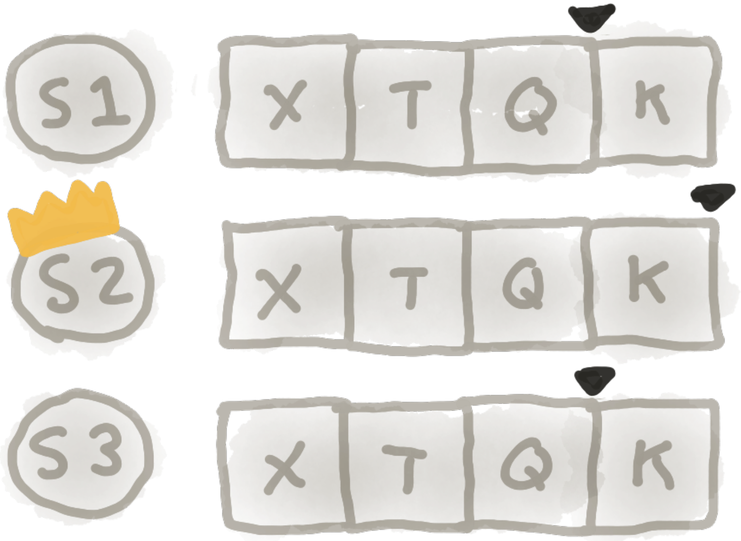


How does the leader copy operations?

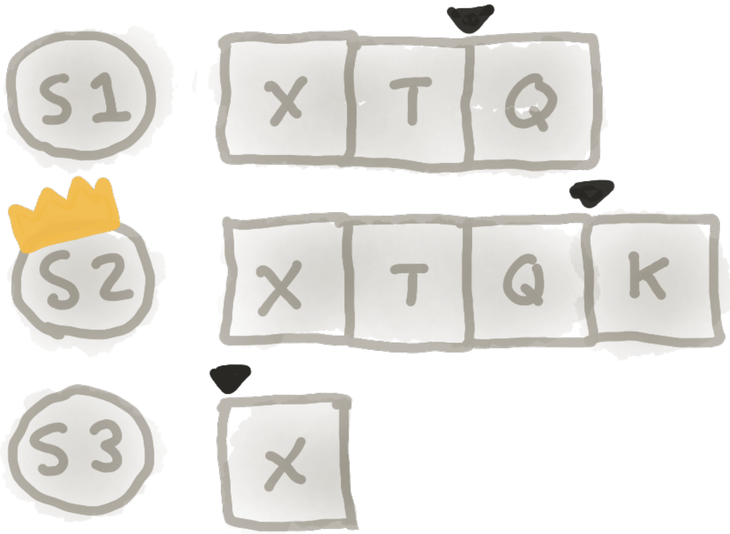


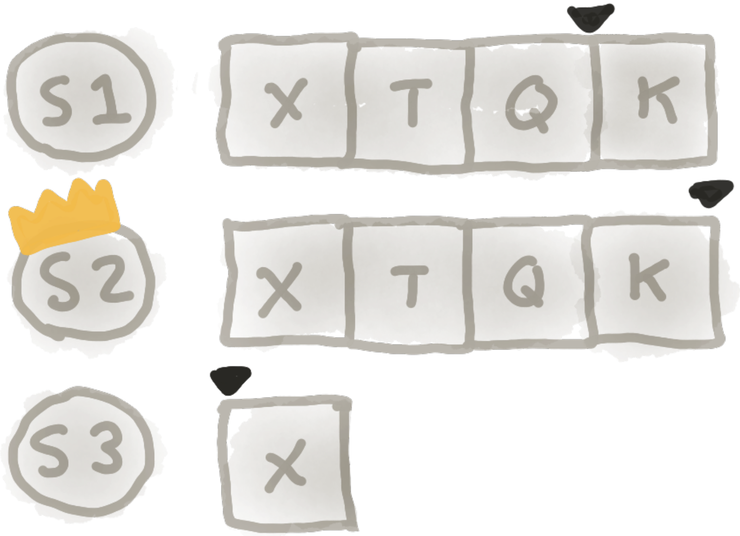


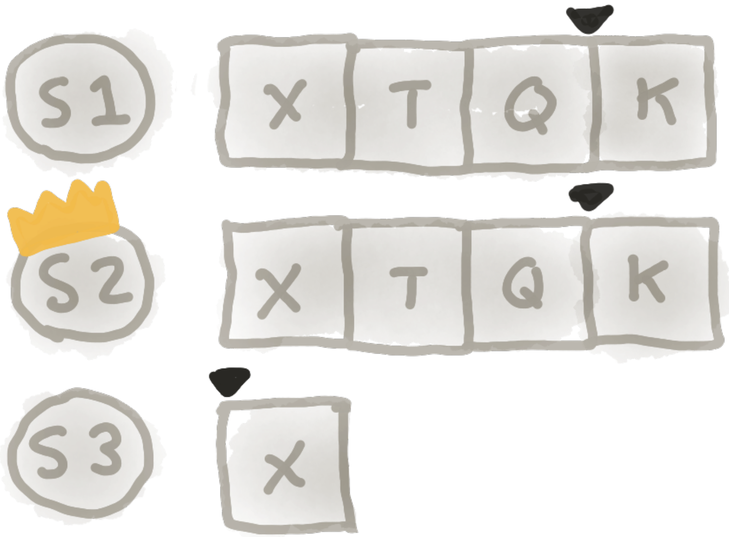


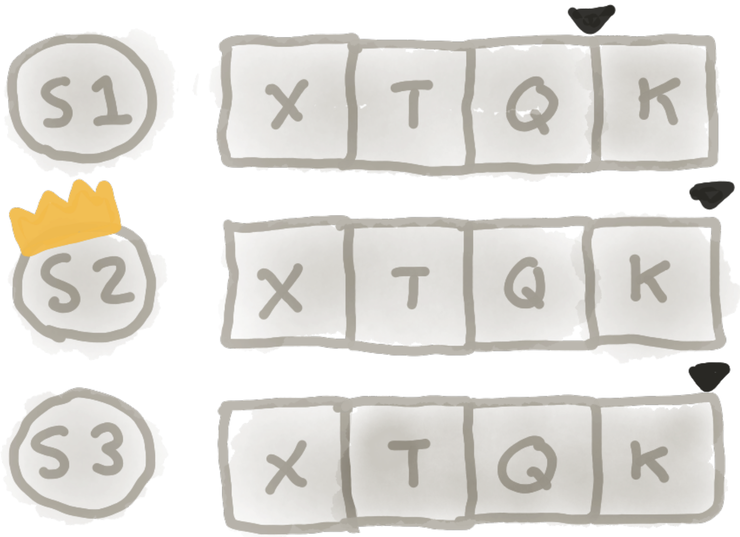


How do we handle follower failures?



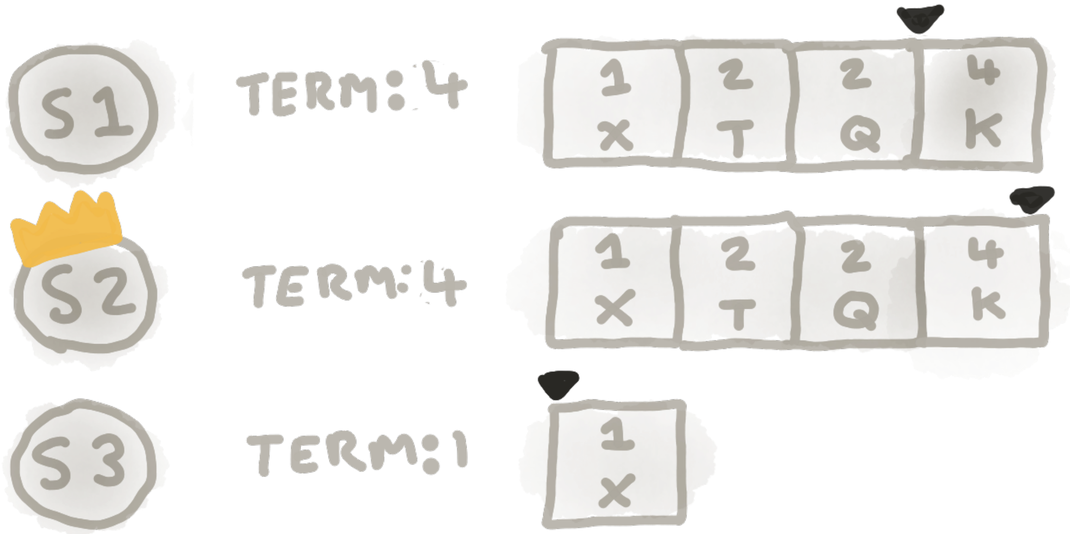






Introducing Terms

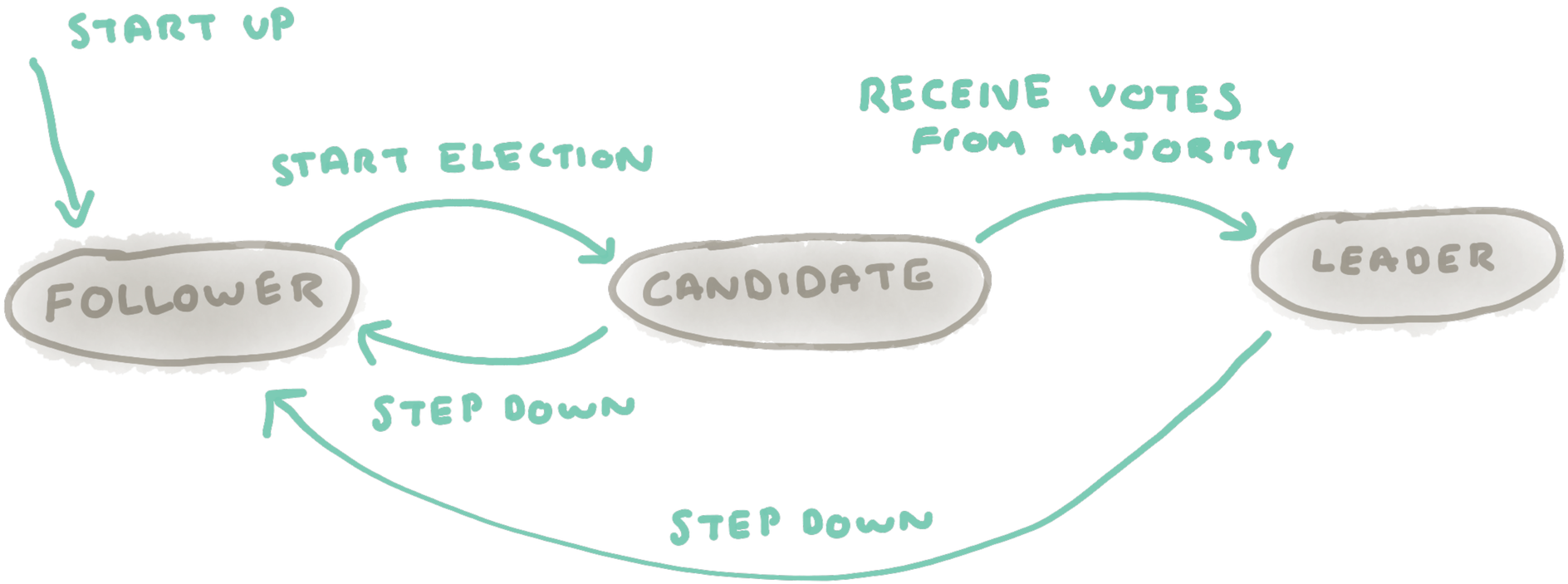
- Each leader serves for one term.
- There is at most one leader per term.
- Each server stores its current term.
- Every server starts in term 1 and the term increases over time.
- Every RPC includes the sender's term.



State transitions

State transitions

State transitions



How do we handle leader failures?

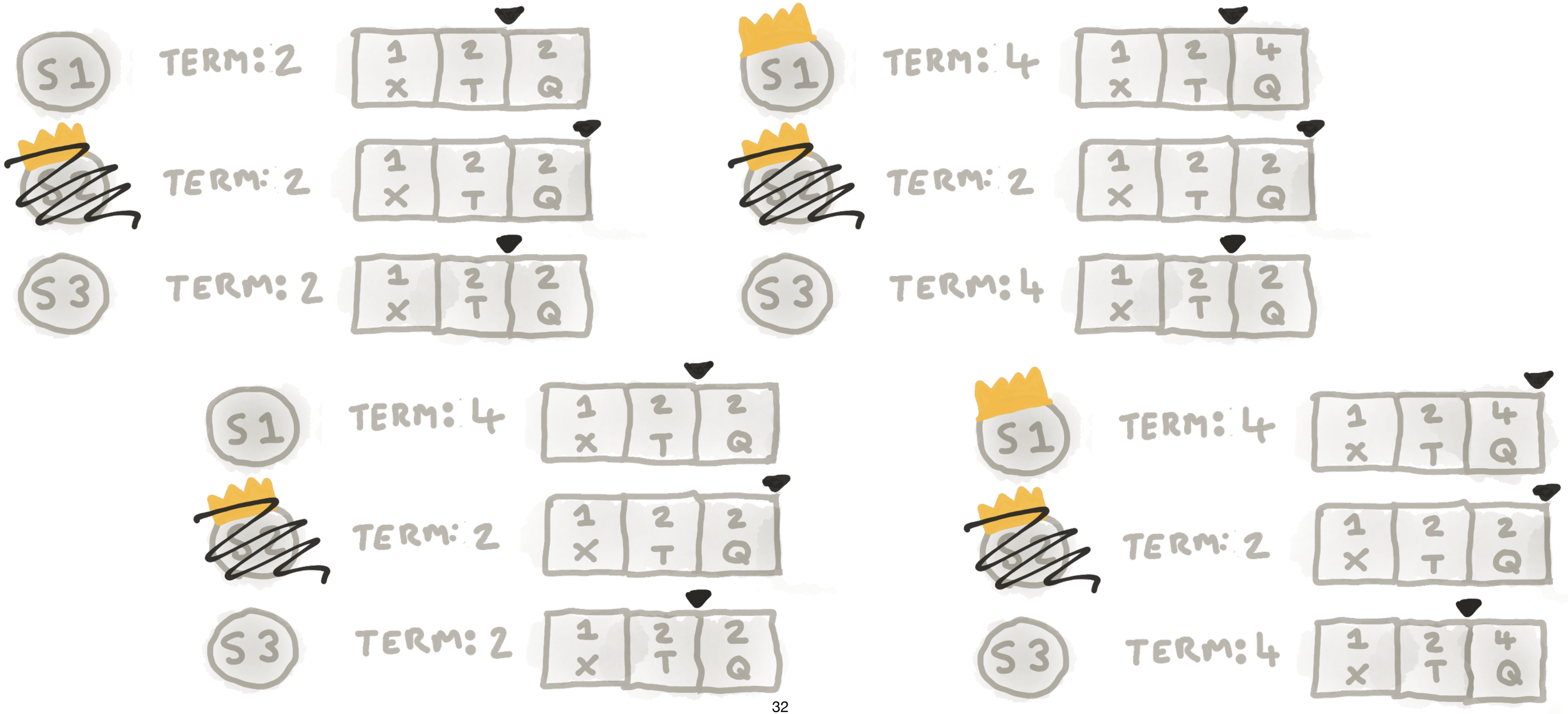
- The leader regularly sends AppendEntries RPCs to all servers.
- If a follower does not receive an AppendEntries RPC from the leader within a timeout then it becomes a candidate.
- The candidate updates its term and solicits votes from other servers using the RequestVotes RPC.
- If a majority of servers vote for it, then the candidate becomes the next leader.

Paxos & Raft must ensure that a newly elected leader has knowledge of all previously committed operations.

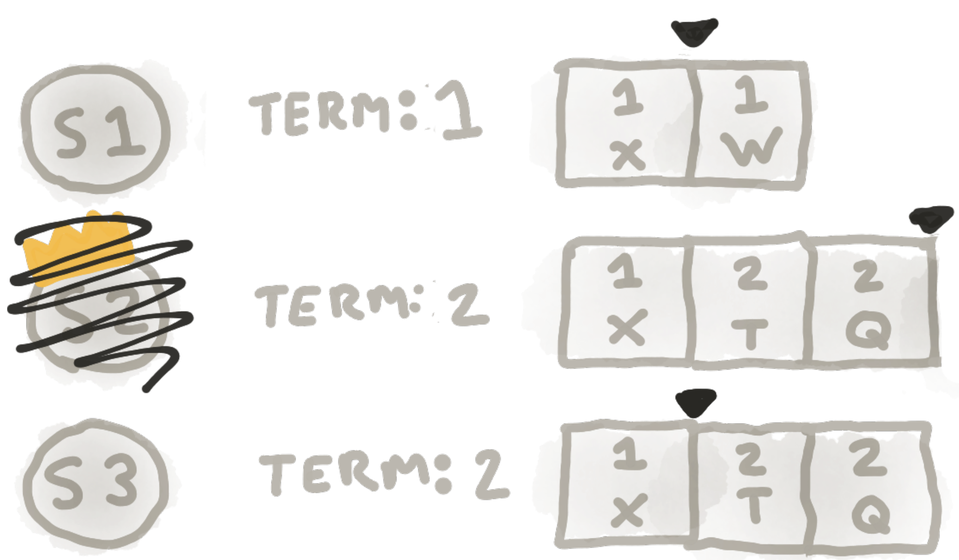
Paxos leader election algorithm

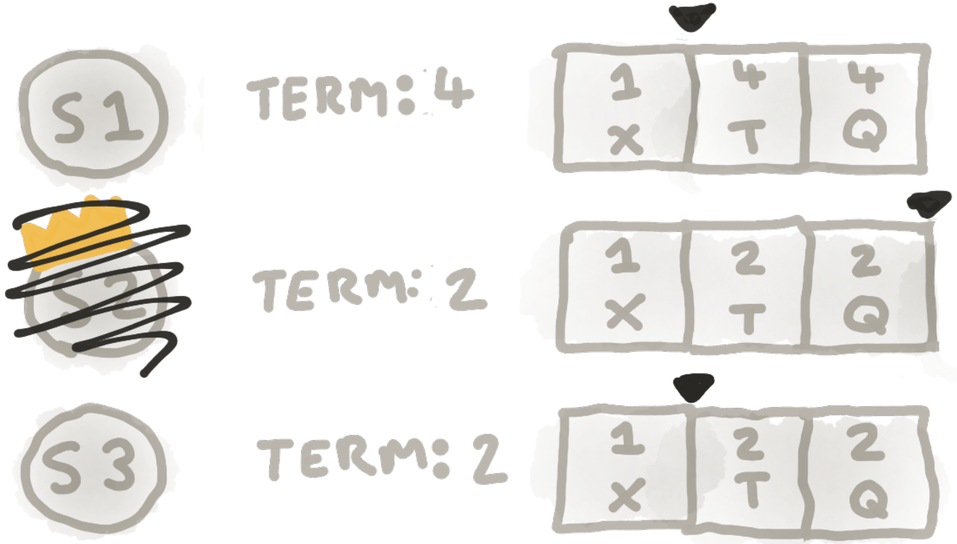
- The candidate begins by updating its term to the next assigned term. Terms are assigned round robin to servers.
- Next, the candidate sends RequestVotes RPCs to all servers. A server will vote for the candidate provided its term is less than the candidate's.
- The RequestVotes RPC includes the candidate's commit index. The RequestVotes reply includes all log entries after the commit index.
- If the candidate receives votes from a majority then it adds any commands received to its log with the new term. If multiple commands are received then it adds the one with a greater term. The candidate then becomes a leader.

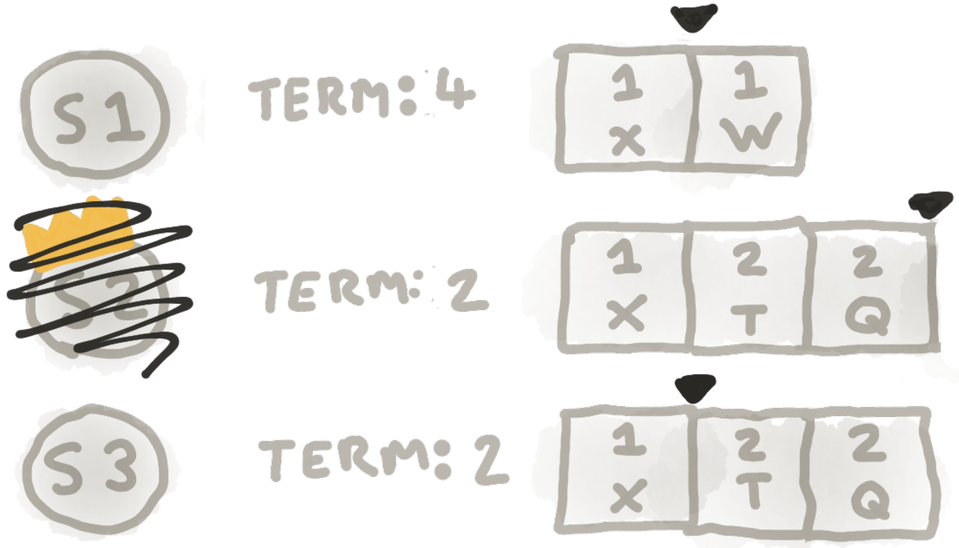
Paxos - Simple example

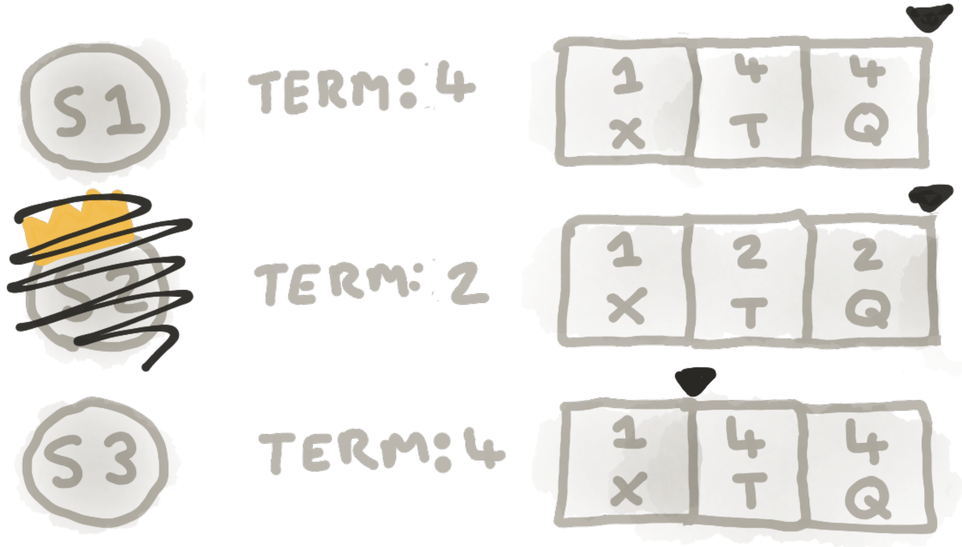


Paxos - A less straightforward example





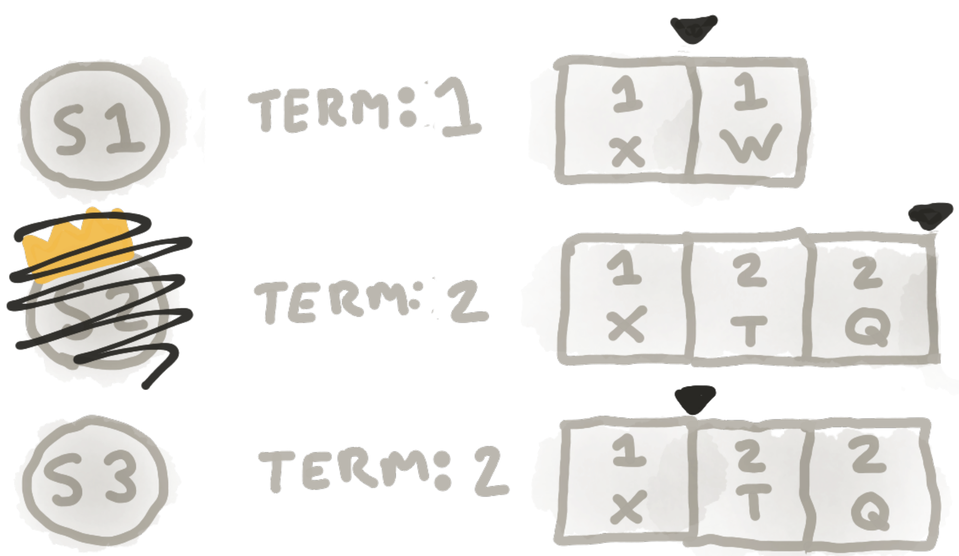


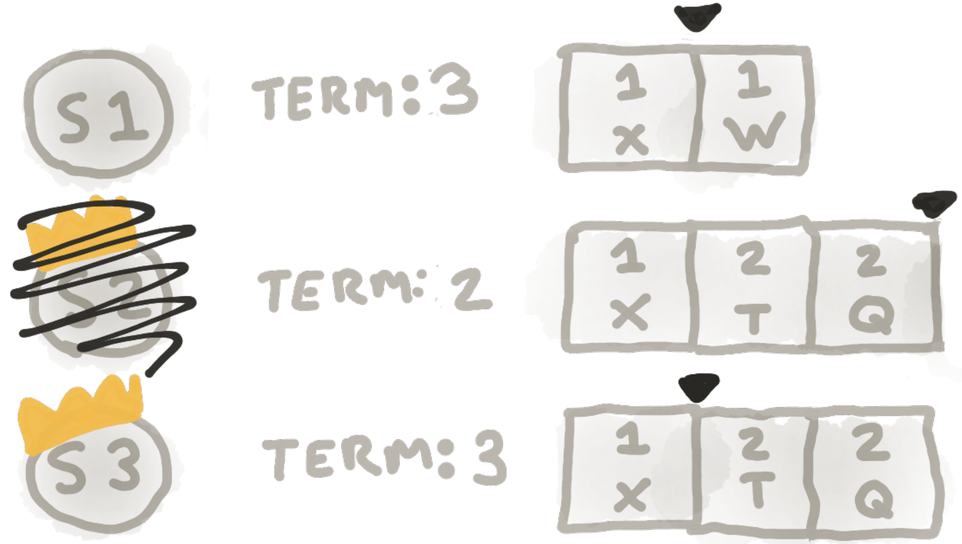


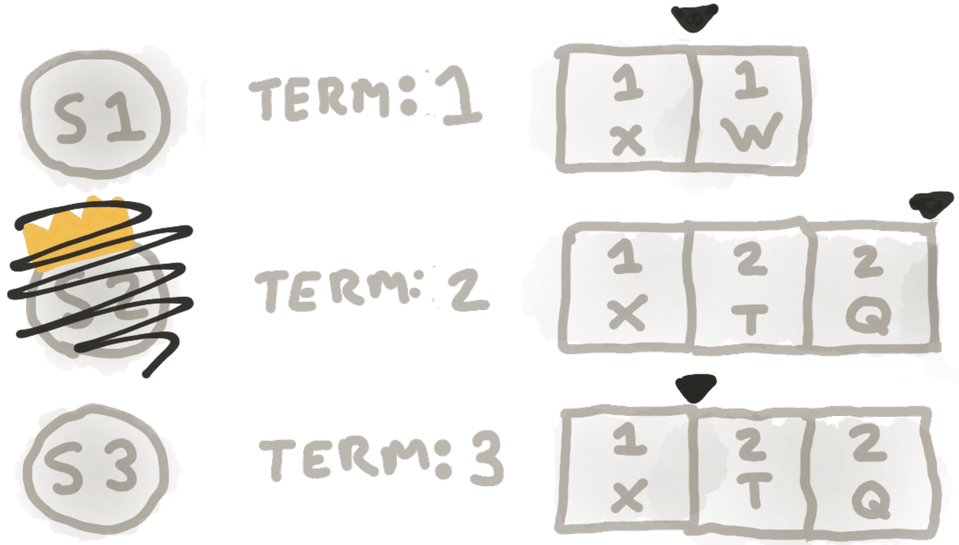
Raft leader election algorithm

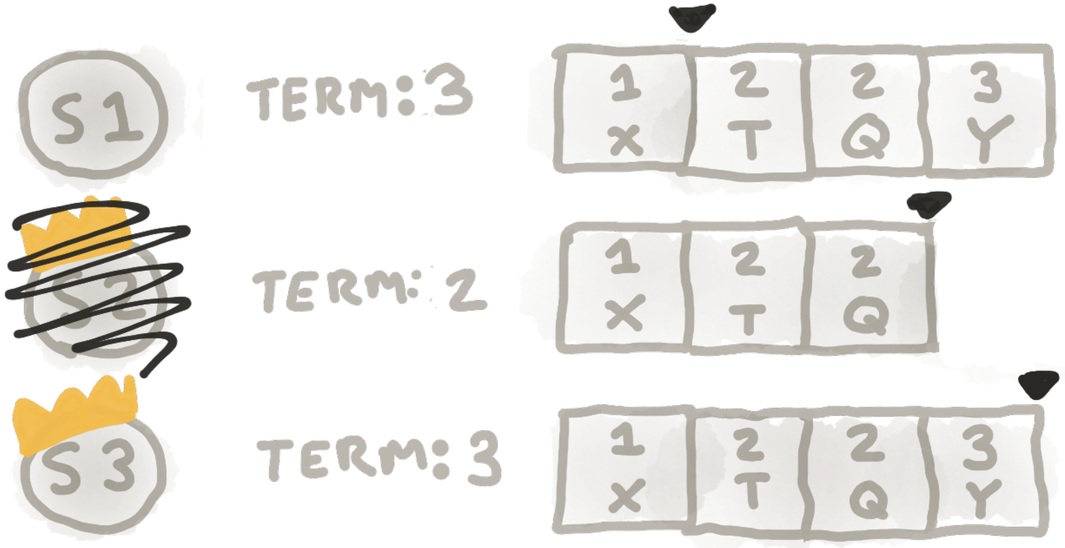
- The candidate begins by incrementing its term and sending RequestVotes RPCs to all servers.
- A follower will vote for the candidate provided the follower's terms is smaller than the candidate's term and the candidate's log is at least as up-to-date as the follower's log.
- If the candidate receives votes from the majority then it becomes a leader.
- The leader uses AppendEntries RPCs to append any uncommitted log entries. Log entries from previous terms are not committed until at least one log entry from the new term has been committed.

Raft - Example



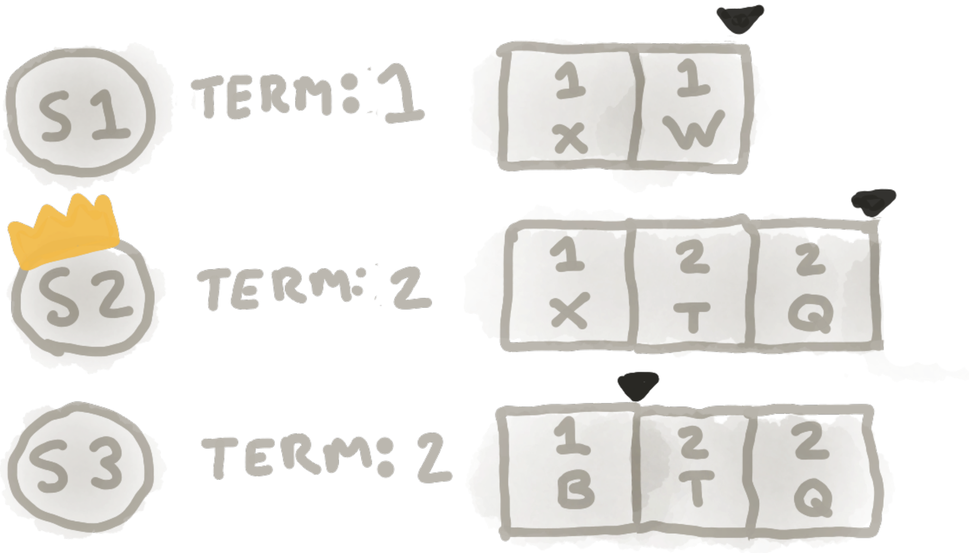






How do we know that Paxos & Raft are safe?

Both guarantee that at most one operation is committed at each index.



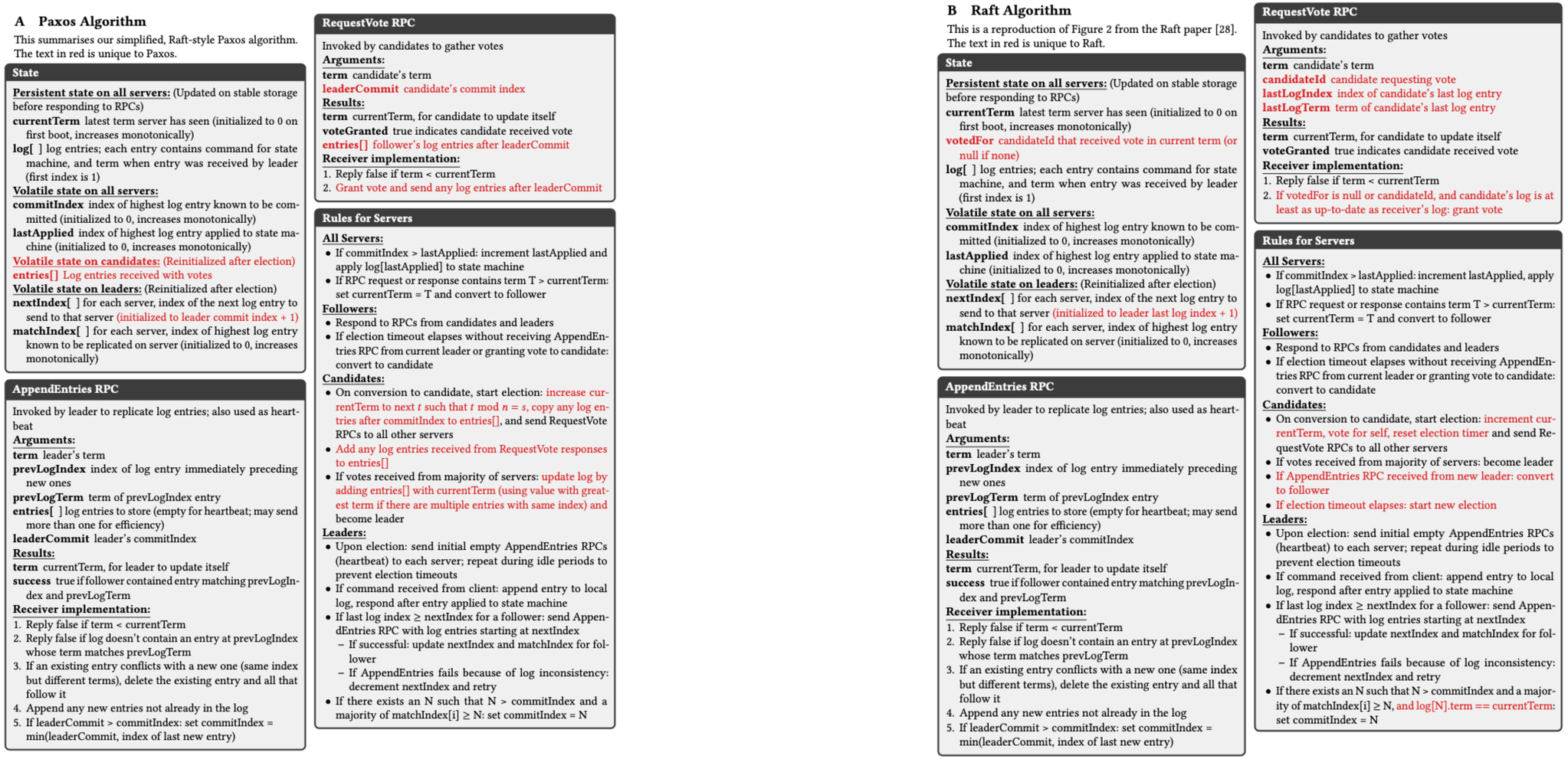
Ensuring safety within terms

- We want to prove that at most one operation is committed per term per index.
- We can prove a stronger statement: at most one operation is added to any log per term per index.
- Both Paxos & Raft ensure that each term has one leader and the leader will not overwrite its own log.

Ensuring safety across terms

- If an operation is committed then it must be present at the same index in the logs of all future leaders.
- We can prove that an if operation is committed in term *t* then it will be in the log of the next leader of a term greater than *t*. We can use a similar argument to prove that this applies to all future leaders.
- Paxos & Raft take different approaches:
  - In Paxos, any candidate can become a leader as the leader election phase ensures that the leader learns the latest log entries.
  - In Raft, only a candidate whose log is already up-to-date can become a leader.

How do Paxos & Raft differ?



A Paxos Algorithm

This summarises our simplified, Raft-style Paxos algorithm. The text in red is unique to Paxos.

State

Persistent state on all servers: (Updated on stable storage before responding to RPCs)
currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)
log[ ] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:
commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on candidates: (Reinitialized after election)
entries[] Log entries received with votes
Volatile state on leaders: (Reinitialized after election)
nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader commit index + 1)
matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat
Arguments:
term leader's term
prevLogIndex index of log entry immediately preceding new ones
prevLogTerm term of prevLogIndex entry
entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit leader's commitIndex
Results:
term currentTerm, for leader to update itself
success true if follower contained entry matching prevLogIndex and prevLogTerm
Receiver implementation:
1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes
Arguments:
term candidate's term
leaderCommit candidate's commit index
Results:
term currentTerm, for candidate to update itself
voteGranted true indicates candidate received vote
entries[] follower's log entries after leaderCommit
Receiver implementation:
1. Reply false if term < currentTerm
2. Grant vote and send any log entries after leaderCommit

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied and apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T and convert to follower

Followers:
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates:
- On conversion to candidate, start election: increase currentTerm to next t such that $t \bmod n = s$, copy any log entries after commitIndex to entries[], and send RequestVote RPCs to all other servers
- Add any log entries received from RequestVote responses to entries[]
- If votes received from majority of servers: update log by adding entries[] with currentTerm (using value with greatest term if there are multiple entries with same index) and become leader

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex and a majority of matchIndex[i] ≥ N: set commitIndex = N

B Raft Algorithm

This is a reproduction of Figure 2 from the Raft paper [28]. The text in red is unique to Raft.

State

Persistent state on all servers: (Updated on stable storage before responding to RPCs)
currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor candidateId that received vote in current term (or null if none)
log[ ] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:
commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on leaders: (Reinitialized after election)
nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat
Arguments:
term leader's term
prevLogIndex index of log entry immediately preceding new ones
prevLogTerm term of prevLogIndex entry
entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit leader's commitIndex
Results:
term currentTerm, for leader to update itself
success true if follower contained entry matching prevLogIndex and prevLogTerm
Receiver implementation:
1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes
Arguments:
term candidate's term
candidateId candidate requesting vote
lastLogIndex index of candidate's last log entry
lastLogTerm term of candidate's last log entry
Results:
term currentTerm, for candidate to update itself
voteGranted true indicates candidate received vote
Receiver implementation:
1. Reply false if term < currentTerm
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log: grant vote

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T and convert to follower

Followers:
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates:
- On conversion to candidate, start election: increment currentTerm, vote for self, reset election timer and send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex and a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N

Which algorithm is the best solution to distributed consensus?

How do we define best?

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the fail-

state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any

Which algorithm is more understandable?

- There is no significant difference in understandability.
- In Paxos, the leader only commits log entries from the current term. It is safe in Paxos to commit a log entry as soon as it has been appended to a majority of logs.
- In Raft, each command maintains the term it was assigned when it was first added to the leader’s log.

Which algorithm is more efficient?

- The algorithms differ only when recovering from a failure. They are equally as efficient in the steady state.
- If we have to choose a winner then I choose Raft.
- RequestVote RPCs in Paxos include all log entries after the leader's commit index. These messages could be quite large.
- Paxos sends commands unnecessarily when recovering log entries from previous terms.
- Raft can be slower to elect a leader due to contention.
- Both algorithms are naive and can be extensively optimised.

Which algorithm is the best?

- It does not matter. The differences are not significant for most use cases.
- Out-of-box Raft is generally more efficient at recovering from failures.
- With Paxos you can benefit from extensive existing literature.

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and

a stronger degree of coherency to reduc the
states that must be considered. Results fr m a
demonstrate that Raft is easier for student to
Paxos. Raft also includes a new mechanism fo
the cluster membership, which uses overlappin
ties to guarantee safety.

Read our paper for more details

Paxos vs Raft: Have we reached consensus on distributed consensus?

Heidi Howard
University of Cambridge
Cambridge, UK
first.last@cl.cam.ac.uk

Richard Mortier
University of Cambridge
Cambridge, UK
first.last@cl.cam.ac.uk

Abstract

Distributed consensus is a fundamental primitive for constructing fault-tolerant, strongly-consistent distributed systems. Though many distributed consensus algorithms have been proposed, just two dominate production systems: Paxos, the traditional, famously subtle, algorithm; and Raft, a more recent algorithm positioned as a more understandable alternative to Paxos.

In this paper, we consider the question of which algorithm, Paxos or Raft, is the better solution to distributed consensus? We analyse both to determine exactly how they differ by describing a simplified Paxos algorithm using Raft's terminology and pragmatic abstractions.

We find that both Paxos and Raft take a very similar approach to distributed consensus, differing only in their approach to leader election. Most notably, Raft only allows servers with up-to-date logs to become leaders, whereas Paxos allows any server to be leader provided it then updates its log to ensure it is up-to-date. Raft's approach is surprisingly efficient given its simplicity as, unlike Paxos, it does not require log entries to be exchanged during leader election. We surmise that much of the understandability of Raft comes from the paper's clear presentation rather than being fundamental to the underlying algorithm being presented.

1 Introduction

State machine replication [32] is widely used to compose a set of unreliable hosts into a single reliable service that can provide strong consistency guarantees including linearizability [13]. As a result, programmers can treat a service implemented using replicated state machines as a single system, making it easy to reason about expected behaviour. State machine replication requires that each state machine receives the same operations in the same order, which can be achieved by distributed consensus.

The Paxos algorithm [16] is synonymous with distributed consensus. Despite its success, Paxos is famously difficult to understand, making it hard to reason about, implement correctly, and safely optimise. This is evident in the numerous attempts to explain the algorithm in simpler terms [4, 17, 22, 23, 25, 29, 35], and was the motivation behind Raft [28].

Raft's authors' claim that Raft is as efficient as Paxos whilst being more understandable and thus provides a better foundation for building practical systems. Raft seeks to achieve this in three distinct ways:

Presentation Firstly, the Raft paper introduces a new abstraction for describing leader-based consensus in the context of state machine replication. This pragmatic presentation has proven incredibly popular with engineers.

Simplicity Secondly, the Raft paper prioritises simplicity over performance. For example, Raft decides log entries in-order whereas Paxos typically allows out-of-order decisions but requires an extra protocol for filling the log gaps which can occur as a result.

Underlying algorithm Finally, the Raft algorithm takes a novel approach to leader election which alters how a leader is elected and thus how safety is guaranteed.

Raft rapidly became popular [30] and production systems today are divided between those which use Paxos [3, 5, 31, 33, 36, 38] and those which use Raft [2, 8–10, 15, 24, 34].

To answer the question of which, Paxos or Raft, is the better solution to distributed consensus, we must first answer the question of how exactly the two algorithms differ in their approach to consensus? Not only will this help in evaluating these algorithms, it may also allow Raft to benefit from the decades of research optimising Paxos' performance [6, 12, 14, 18–20, 26, 27] and vice versa [1, 37].

However, answering this question is not a straightforward matter. Paxos is often regarded not as a single algorithm but as a family of algorithms for solving distributed consensus. Paxos' generality (or underspecification, depending on your point of view) means that descriptions of the algorithm vary, sometimes considerably, from paper to paper.

To overcome this problem, we present here a simplified version of Paxos that results from surveying the various published descriptions of Paxos. This algorithm, which we refer to simply as Paxos, corresponds more closely to how Paxos is used today than to how it was first described [16]. It has been referred to elsewhere as *multi-decree* Paxos, or just *MultiPaxos*, to distinguish it from *single-decree* Paxos, which decides a single value instead of a totally-ordered sequence of values. We also describe our simplified algorithm using the style and abstractions from the Raft paper, allowing a fair comparison between the two different algorithms.

A Paxos Algorithm

This summarises our simplified, Raft-style Paxos algorithm. The text in red is unique to Paxos.

State

Persistent state on all servers: (Updated on stable storage before responding to RPCs)
currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)
log[ ] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:
commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on candidates: (Reinitialized after election)
entries[] Log entries received with votes
Volatile state on leaders: (Reinitialized after election)
nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader commit index + 1)
matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat
Arguments:
term leader's term
prevLogIndex index of log entry immediately preceding new ones
prevLogTerm term of prevLogIndex entry
entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit leader's commitIndex
Results:
term currentTerm, for leader to update itself
success true if follower contained entry matching prevLogIndex and prevLogTerm
Receiver implementation:
1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes
Arguments:
term candidate's term
leaderCommit candidate's commit index
Results:
term currentTerm, for candidate to update itself
voteGranted true indicates candidate received vote
entries[] follower's log entries after leaderCommit
Receiver implementation:
1. Reply false if term < currentTerm
2. Grant vote and send any log entries after leaderCommit

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied and apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T and convert to follower

Followers:
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates:
- On conversion to candidate, start election: increase currentTerm to next t such that $t \bmod n = s$, copy any log entries after commitIndex to entries[], and send RequestVote RPCs to all other servers
- Add any log entries received from RequestVote responses to entries[]
- If votes received from majority of servers: update log by adding entries[] with currentTerm (using value with greatest term if there are multiple entries with same index) and become leader

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex and a majority of matchIndex[i] ≥ N: set commitIndex = N

B Raft Algorithm

This is a reproduction of Figure 2 from the Raft paper [28]. The text in red is unique to Raft.

State

Persistent state on all servers: (Updated on stable storage before responding to RPCs)
currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor candidateId that received vote in current term (or null if none)
log[ ] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)
Volatile state on all servers:
commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)
Volatile state on leaders: (Reinitialized after election)
nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat
Arguments:
term leader's term
prevLogIndex index of log entry immediately preceding new ones
prevLogTerm term of prevLogIndex entry
entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit leader's commitIndex
Results:
term currentTerm, for leader to update itself
success true if follower contained entry matching prevLogIndex and prevLogTerm
Receiver implementation:
1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes
Arguments:
term candidate's term
candidateId candidate requesting vote
lastLogIndex index of candidate's last log entry
lastLogTerm term of candidate's last log entry
Results:
term currentTerm, for candidate to update itself
voteGranted true indicates candidate received vote
Receiver implementation:
1. Reply false if term < currentTerm
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log: grant vote

Rules for Servers

All Servers:
- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine
- If RPC request or response contains term T > currentTerm: set currentTerm = T and convert to follower

Followers:
- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates:
- On conversion to candidate, start election: increment currentTerm, vote for self, reset election timer and send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:
- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts
- If command received from client: append entry to local log, respond after entry applied to state machine
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry
- If there exists an N such that N > commitIndex and a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N

Summary

- Paxos & Raft differ only in their approach to leader election.
- Raft is not significantly more understandable than Paxos. Much of the understandability of the Raft paper comes from the excellent presentation and focus on simplification, not from the underlying algorithm.
- Raft's leader election mechanism is surprisingly efficient for such a simple approach.
- **Q & A**

Heidi Howard
University of Cambridge

heidi.howard@cl.cam.ac.uk
heidihoward.co.uk
@heidiann360