Distributed Algorithms
Raft Consensus

Alberto Montresor

Università di Trento

2020/11/19

Acknowledgement: Diego Ongaro and John Ousterhout

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Table of contents

Atomic Broadcast

Definition (RB1 – Validity)

If a correct process broadcasts m, then it eventually delivers m

Definition (RB2 – Uniform Integrity)

m is delivered by a process at most once, and only if it was previously broadcast

Definition (RB3 – Agreement)

If a correct process delivers m, then all correct processes eventually deliver m

Definition (Total Order)

If correct processes p and q both deliver messages m,m', then p delivers m before m' if and only if q delivers m before m'

$$\mathit{deliver}_p(m) \rightarrow \mathit{deliver}_p(m') \Rightarrow \mathit{deliver}_q(m) \rightarrow \mathit{deliver}_q(m')$$

State machine replication

Definition (State machine)

A state machine consists of:

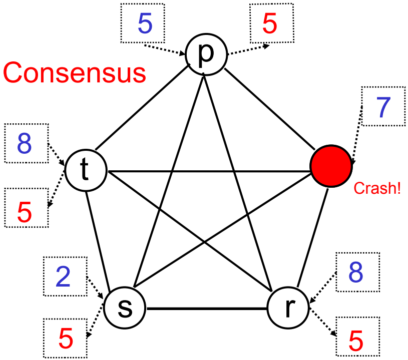
- State variables
- Commands which transforms its state
  - Implemented by deterministic programs
  - Atomic with respect to other commands

Specification

- Agreement: every correct replica receives the same set of commands
- Order: every non-faulty state machine processes the commands it receives in the same order

Consensus

In the (Uniform) Consensus problem, the processes propose values and need to decide (agree) on one of these values



Definition (Uniform Validity)

Any value decided is a value proposed

Definition ((Uniform) Agreement)

No two correct (any) processes decide differently

Definition (Termination)

Every correct process eventually decides

Definition (Uniform Integrity)

Every process decides at most once

Consensus, Atomic Broadcast, State Machine Replication

Equivalence between Consensus and Atomic Broadcast

1. There is an algorithm $T_{Consensus \rightarrow AtomicBroadcast}$
2. There is an algorithm $T_{AtomicBroadcast \rightarrow Consensus}$

From Atomic Broadcast to Consensus

Transformation executed by process p

upon initialization **do**
  boolean *decided* ← **false**

upon propose(v) **do**
  A-broadcast(v)

upon A-deliver(v) **do**
  if not *decided* **then**
    decided ← **true**
    decide(u)

From Consensus to Atomic Broadcast

Transformation executed by process p

upon initialization **do**

- SET *unordered* $\leftarrow \emptyset$ % Messages to be ordered
- SET *delivered* $\leftarrow \emptyset$ % Messages already delivered
- **boolean** *wait* $\leftarrow$ **false** % **true** when Consensus is running
- **integer** $s \leftarrow 1$ % Consensus protocol identifier

upon A-broadcast(m) **do**

- R-broadcast(m)

upon R-deliver(m) **do**

- **if not** $m \in \mathit{delivered}$ **then**
  - $\mathit{unordered} \leftarrow \mathit{unordered} \cup \{m\}$

From Consensus to Atomic Broadcast

Transformation executed by process p

upon decide$_s(S)$ **do**

- $\mathit{unordered} \leftarrow \mathit{unordered} - S$
- **foreach** $m \in S$ **do**
  - A-deliver(m) % In some deterministic order
- $\mathit{delivered} \leftarrow \mathit{delivered} \cup S$
- $s \leftarrow s + 1$
- $\mathit{wait} \leftarrow$ **false**

upon $\mathit{unordered} \neq \emptyset$ **and not** wait **do**

- $\mathit{wait} \leftarrow$ **true**
- propose$_s(\mathit{unordered})$

Discussion

Summary

Consensus and total order broadcast are equivalent problems in an asynchronous system with crashes and Perfect Channels

- Consensus can be obtained from total order broadcast
- Total order broadcast can be obtained from Consensus

Problem

This means that the impossibility results of Consensus apply to Atomic Broadcast as weel

Paxos History

1989 Leslie Lamport developed a new consensus protocol called Paxos; it was published as DEC SRC Technical Report 49. 42 pages!

Abstract

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 — *an approach that has received limited attention because it leads to designs of insufficient complexity.*

Paxos History

1989 Leslie Lamport developed a new consensus protocol called Paxos; it was published as DEC SRC Technical Report 49. 42 pages!

Abstract

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 — *an approach that has received limited attention because it leads to designs of insufficient complexity.*

From `http://the-paper-trail.org/blog/consensus-protocols-paxos/`

- Just to remember: the FLP result has been published in 1985. The first paper on failure detectors has been published in 1991.
- **The Part-time Parliament**. The original paper. Once you understand the protocol, you might well really enjoy this presentation of it. Contains proofs of correctness which the 'Paxos Made Simple' paper does not.

Paxos History

1990 Submitted to ACM Trans. on Comp. Sys. (TOCS). Rejected.

1996 "How to Build a Highly Available System Using Consensus", by B. Lampson was published in WDAG 1996, Bologna, Italy.

1997 "Revisiting the Paxos Algorithm", by R. De Prisco, B. Lampson, N. Lynch was published in WDAG 1997, Saarbrücken, Germany.

1998 The original paper is resubmitted and accepted by TOCS.

2001 Lamport publishes "Paxos made simple" in ACM SIGACT News

- Because Lamport "*got tired of everyone saying how difficult it was to understand the Paxos algorithm*"
- Abstract: "*The Paxos algorithm, when presented in plain English, is very simple*"
- Introduces the concept of Multi-Paxos

Paxos History

- 1990 Submitted to ACM Trans. on Comp. Sys. (TOCS). Rejected.
- 1996 "How to Build a Highly Available System Using Consensus", by B. Lampson was published in WDAG 1996, Bologna, Italy.
- 1997 "Revisiting the Paxos Algorithm", by R. De Prisco, B. Lampson, N. Lynch was published in WDAG 1997, Saarbrücken, Germany.
- 1998 The original paper is resubmitted and accepted by TOCS.
- 2001 Lamport publishes "Paxos made simple" in ACM SIGACT News
  - Because Lamport "*got tired of everyone saying how difficult it was to understand the Paxos algorithm*"
  - Abstract: "*The Paxos algorithm, when presented in plain English, is very simple*"
  - Introduces the concept of Multi-Paxos

From `http://the-paper-trail.org/blog/consensus-protocols-paxos/`

- **How To Build a Highly Available System Using Consensus**. Butler Lampson demonstrates how to employ Paxon consensus as part of a larger system. This paper was partly responsible for ensuring the success of Paxos by popularizing it within the distributed systems community.
- **Paxos Made Simple**. Presents Paxos in a ground-up fashion as a consequence of the requirements and constraints that the protocol must operate within. Short and very readable, it should probably be your first visit after this article.

  If each command is the result of a single instance of the Basic Paxos protocol a significant amount of overhead would result. This paper defines Paxos to be what is commonly called "Multi-Paxos" which in steady state uses a distinguished leader to coordinate an infinite stream of commands. A typical deployment of Paxos uses a continuous stream of agreed values acting as commands to update a distributed state machine.

Paxos History

Paxos optimizations and extensions

2004 Leslie Lamport and Mike Massa. "Cheap Paxos". DSN'04, Florence, Italy

2005 Leslie Lamport. "Generalized Consensus and Paxos". Technical Report MSR-TR-2005-33, Microsoft Research

2006 Leslie Lamport. "Fast Paxos". *Distributed Computing* 19(2):79-103

An important milestone

2007 T. D. Chandra, R. Griesemer, J. Redstone. Paxos made live: an engineering perspective. PODC 2007, Portland, Oregon.

Paxos History

Paxos optimizations and extensions

2004 Leslie Lamport and Mike Massa. "Cheap Paxos". DSN'04, Florence, Italy

2005 Leslie Lamport. "Generalized Consensus and Paxos". Technical Report MSR-TR-2005-33, Microsoft Research

2006 Leslie Lamport. "Fast Paxos". *Distributed Computing* 19(2):79-103

An important milestone

2007 T. D. Chandra, R. Griesemer, J. Redstone. Paxos made live: an engineering perspective. PODC 2007, Portland, Oregon.

From `http://the-paper-trail.org/blog/consensus-protocols-paxos/`

- **Cheap Paxos and Fast Paxos**. Two papers that present some optimizations on the original protocol.
- **Paxos Made Live**.
  - This paper from Google bridges the gap between theoretical algorithm and working system. There are a number of practical issues to consider when implementing Paxos that you might well not have imagined. If you want to build a system using Paxos, you should read this paper beforehand.
  - It describes how Paxos is used in Chubby - the Google lock manager.

Paxos implementations

- Google uses the Paxos algorithm in their Chubby distributed lock service in order to keep replicas consistent in case of failure. Chubby is used by Bigtable which is now in production in Google Analytics and other products.
- Google Spanner and Megastore use the Paxos algorithm internally.
- The OpenReplica replication service uses Paxos to maintain replicas for an open access system that enables users to create fault-tolerant objects. It provides high performance through concurrent rounds and flexibility through dynamic membership changes.
- IBM supposedly uses the Paxos algorithm in their IBM SAN Volume Controller product to implement a general purpose fault-tolerant virtual machine used to run the configuration and control components of the storage virtualization services offered by the cluster. (Original MIT & IBM research paper)
- Microsoft uses Paxos in the Autopilot cluster management service from Bing, and in Windows Server Failover Clustering.
- WANdisco have implemented Paxos within their DConE active-active replication technology.[26]
- XtreemFS uses a Paxos-based lease negotiation algorithm for fault-tolerant and consistent replication of file data and metadata.[27]
- Heroku uses Doozerd which implements Paxos for its consistent distributed data store.
- Ceph uses Paxos as part of the monitor processes to agree which OSDs are up and in the cluster.
- The Clustrix distributed SQL database uses Paxos for distributed transaction resolution.
- Neo4j HA graph database implements Paxos, replacing Apache ZooKeeper from v1.9
- Apache Cassandra NoSQL database uses Paxos for Light Weight Transaction feature only
- Amazon Elastic Container Services uses Paxos to maintain a consistent view of cluster state

The sad state of Paxos

About publications...

"The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-)." – NSDI reviewer

About implementations...

"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." – Chubby authors

Raft Consensus Protocol

An algorithm to build real systems

- Must be correct, complete, and perform well
- Must be understandable

Key design ideas

- What would be easier to understand or explain?
- Less complexity in state space
- Less mechanisms

Bibliography

- D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm.
  In *2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.
  http://www.disi.unitn.it/~montreso/ds/papers/raft.pdf

Raft implementations

Actual deployments

- HydraBase by Facebook (replacement for Apache HBase)
- Consul by HashiCorp (datacenter management)
- Rafter by Basho (NOSQL key-value store called Riak)
- Apache Kudu (distributed database)
- Kubernetes and Docker Swarm (container management)

Raft implementations

Open-source projects: 80+ total (May 2017)

Language	**Numbers**	**Language**	**Numbers**
Java	18	Javascript	6
Go	8	Clojure	4
Ruby	8	Erlang	4
C/C++	8	Rust	3
Scala	7	Bloom	3
Python	6	Others	9

Introduction

Two approaches to consensus / atomic broadcast / state replication:

- Symmetric, leader-less, active replication:
  - All servers have equal roles
  - Clients can contact any server
- Asymmetric, leader-based, passive replication:
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader

Raft is leader-based

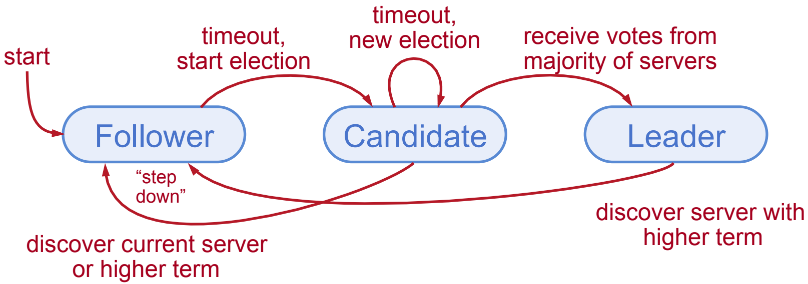
- Decomposes the problem (normal operation, leader changes)
- Simplifies normal operation (no conflicts)
- More efficient than leader-less approaches

Raft overview

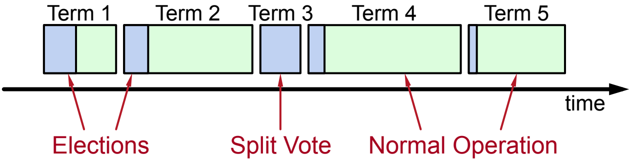
1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Normal operation
   - Basic log replication
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
   - Implementing linearizeable semantics
6. Configuration changes
   - Adding and removing servers

Server states

LEADER	Handles all client interactions, log replication At most 1 viable leader at a time
FOLLOWER	Completely passive (issues no RPCs, responds to incoming RPCs)
CANDIDATE	Used to elect a new leader Normal operation: 1 leader, N-1 followers



Terms



- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most one leader per term
- Some terms have no leader (failed election)
- Each server maintains current term value
- Key role of terms: identify obsolete information

Server state

Persistent state

Each server persists the following variables to stable storage synchronously before responding to RPCs:

currentTerm	Latest term server has seen (initialized to 0 on first boot)
votedFor	ID of the candidate that received vote in current term (or null if none)
log[ ]	Log entries: *term*: term when entry was received by leader *command*: command for state machine

Server state

Non-persistent state

state	Current state; could be LEADER, CANDIDATE, FOLLOWER
leader	ID of the leader
commitIndex	index of highest log entry known to be committed
nextIndex[ ]	index of next log entry to send to peer
matchIndex[ ]	index of highest log entry known to be replicated

Initialization

$currentTerm \leftarrow 1$
$votedFor \leftarrow \textbf{nil}$
$log \leftarrow \{\}$
$state \leftarrow \text{FOLLOWER}$

$leader \leftarrow \textbf{nil}$
$commitIndex \leftarrow 0$
$nextIndex = \{1, 1, \ldots, 1\}$
$matchIndex = \{0, 0, \ldots, 0\}$

RPCs

Communication between leader and followers happen through two RPCs:

- AppendEntries
  - Add an entry to the log, *or*
  - Empty messages used as heartbeats
  - Message tags: AppendReq, AppendRep
- Vote
  - Message used by candidates to ask votes and win elections
  - Message tags: VoteReq, VoteRep

Hearthbeats and timeouts

- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send empty AppendEntries RPCs to maintain authority
- If $\Delta_{election}$ time units elapse with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

Election basics - Election start

1. Set new timeout in range $[\Delta_{election}, 2 \cdot \Delta_{election}]$
2. Increment current term
3. Change to Candidate state
4. Vote for self
5. Send Vote RPCs to all other servers, retry until either:
    - Receive votes from majority of servers:
        - Become Leader
        - Send AppendEntries heartbeats to all other servers
    - Receive AppendEntries from valid leader:
        - Return to Follower state
    - No one wins election (election timeout elapses):
        - Start new election

Election - Pseudocode

Election code - executed by process p

upon timeout $\langle \textsc{ElectionTimeout} \rangle$ **do**
- **if** $state \in \{\textsc{follower}, \textsc{candidate}\}$ **then**
  - $t \leftarrow \mathsf{random}(1.0, 2.0) \cdot \Delta_{election}$
  - **set timeout** $\langle \textsc{ElectionTimeout} \rangle$ **at** $\mathsf{now}() + t$
  - $currentTerm \leftarrow currentTerm + 1$
  - $state \leftarrow \textsc{candidate}$
  - $votedFor \leftarrow p$
  - $votes \leftarrow \{p\}$
  - **foreach** $q \in \Pi$ **do**
    - **cancel timeout** $\langle \textsc{RpcTimeout}, q \rangle$
    - **set timeout** $\langle \textsc{RpcTimeout}, q \rangle$ **at** $\mathsf{now}()$

Election - Pseudocode

RPC timeout code - executed by process p

upon timeout $\langle \textsc{RpcTimeout}, q \rangle$ **do**
  if $\mathit{state} = \textsc{candidate}$ **then**
    set timeout $\langle \textsc{RpcTimeout}, q \rangle$ **at** $\mathsf{now}() + \Delta_{vote}$
    send $\langle \textsc{VoteReq}, \mathit{currentTerm} \rangle$ **to** q

Election - Pseudocode

Election code - executed by process p

on receive $\langle \textsc{VoteReq}, \mathit{term} \rangle$ **from** q **do**

  if $\mathit{term} > \mathit{currentTerm}$ **then**

    $\mathsf{stepdown}(\mathit{term})$

  if $\mathit{term} = \mathit{currentTerm}$ **and** $\mathit{votedFor} \in \{q, \mathbf{nil}\}$ **then**

    $\mathit{votedFor} \leftarrow q$

    $t \leftarrow \mathsf{random}(1.0, 2.0) \cdot \Delta_{\mathit{election}}$

    set timeout $\langle \textsc{ElectionTimeout} \rangle$ **at** $\mathsf{now}() + t$

    send $\langle \textsc{VoteRep}, \mathit{term}, \mathit{votedFor} \rangle$ **to** q

Election - Pseudocode

Election code - executed by process p

on receive $\langle \textsc{VoteRep}, \mathit{term}, \mathit{vote} \rangle$ **from** q **do**

- **if** $\mathit{term} > \mathit{currentTerm}$ **then**
  - $\mathsf{stepdown}(\mathit{term})$
- **if** $\mathit{term} = \mathit{currentTerm}$ **and** $\mathit{state} = \textsc{candidate}$ **then**
  - **if** $\mathit{vote} = p$ **then**
    - $\mathit{votes} \leftarrow \mathit{votes} \cup \{q\}$
  - **cancel timeout** $\langle \textsc{RpcTimeout}, q \rangle$
  - **if** $|\mathit{votes}| > |\Pi|/2$ **then**
    - $\mathit{state} \leftarrow \textsc{leader}$
    - $\mathit{leader} \leftarrow p$
    - **foreach** $q \in P - \{p\}$ **do**
      - $\mathsf{sendAppendEntries}(q)$

Election - Pseudocode

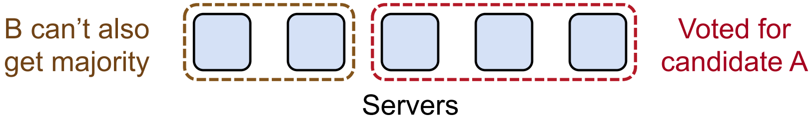
procedure stepdown($term$)

- $currentTerm \leftarrow term$
- $state \leftarrow$ FOLLOWER
- $votedFor \leftarrow$ **nil**
- $t \leftarrow \mathsf{random}(1.0, 2.0) \cdot \Delta_{election}$
- **set timeout** $\langle$ELECTIONTIMEOUT$\rangle$ **at** $\mathsf{now}() + t$

Election - Correctness

Safety: allow at most one winner per term

- Each server gives out only one vote per term (persist on disk)
- Two different candidates can't accumulate majorities in same term

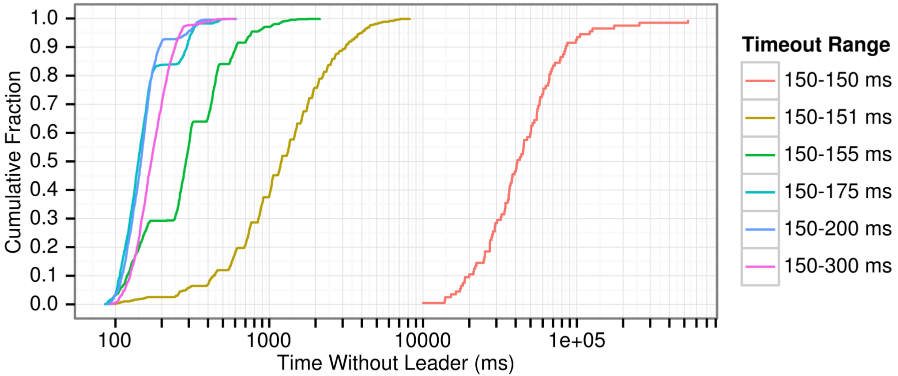


Liveness: some candidate must eventually win

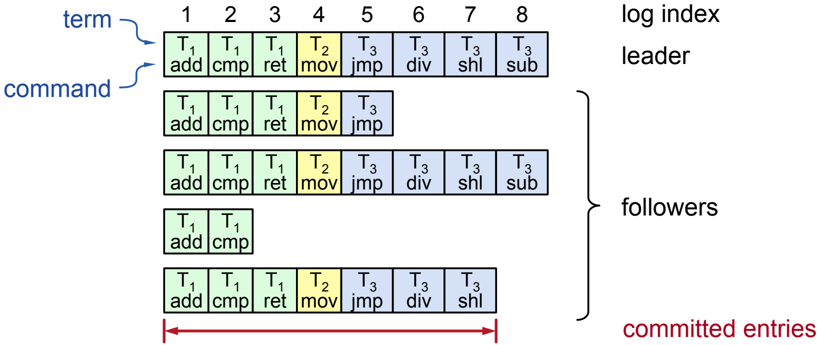
- Choose election timeouts randomly in $[\Delta_{election}, 2 \cdot \Delta_{election}]$
- One server usually times out and wins election before others wake up
- Works well if $\Delta_{election} >> $ broadcast time

Randomize timeouts

- How much randomization is needed to avoid split votes?
- Conservatively, use random range ≈ 10× network latency



Log structure



- 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

Normal operation

- Client sends command to leader
- Leader appends command to its log

Normal operation code executed by process p

upon receive $\langle$REQUEST, *command*$\rangle$ **from** client **do**
    if $state = $ LEADER **then**
        log.append($currentTerm$, $command$)
        foreach $q \in P - \{p\}$ **do**
            sendAppendEntries(q)

Normal operation

- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- Crashed/slow followers?
  - Leader retries RPCs until they succeed
  - Performance is optimal in common case: one successful RPC to any majority of servers

Normal operation

RPC timeout code executed by process p

upon timeout $\langle \textsc{RpcTimeout}, q \rangle$ **do**
- **if** $\mathit{state} = \textsc{candidate}$ **then**
  - **set timeout** $\langle \textsc{RpcTimeout}, q \rangle$ **at** $\mathsf{now}() + \Delta_{vote}$
  - **send** $\langle \textsc{VoteReq}, \mathit{currentTerm} \rangle$ **to** q
- **if** $\mathit{state} = \textsc{leader}$ **then**
  - $\mathsf{sendAppendEntries}(q)$

How to send append entries

procedure $\mathsf{sendAppendEntries}(q)$

- **set timeout** $\langle \textsc{RpcTimeout}, q\rangle$ **at** $\mathsf{now}() + \Delta_{election}/2$
- $lastLogIndex \leftarrow \text{choose in}[nextIndex[q], log.\mathsf{len}()]$
- $nextIndex[q] = lastLogIndex$
- **send** $\langle \textsc{AppendReq}, term, lastLogIndex - 1, log[lastLogIndex[q] - 1].term$
  $log[lastLogIndex \ldots log.\mathsf{len}()], commitIndex\rangle$ **to** q

Log consistency

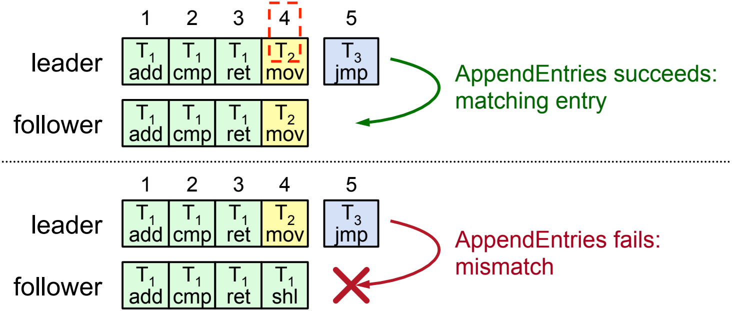
Consistency in logs

- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries
- If a given entry is committed, all preceding entries are also committed

1	2	3	4	5	6
T_1 add	T_1 cmp	T_1 ret	T_2 mov	T_3 jmp	T_3 div
T_1 add	T_1 cmp	T_1 ret	T_2 mov	T_3 jmp	T_4 sub

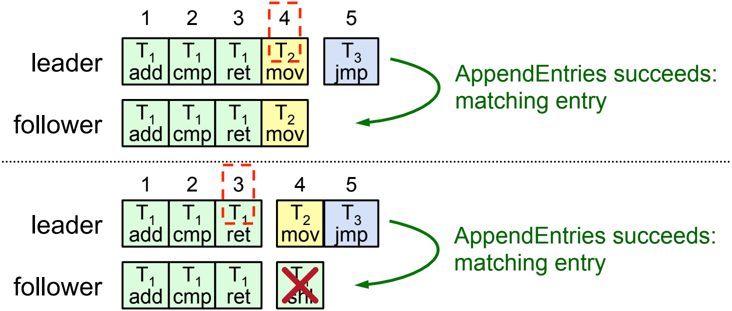
AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an induction step, ensures coherency



AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an induction step, ensures coherency



Normal operation - Pseudocode

Normal operation code - executed by process p

```
on receive
  ⟨APPENDREQ, term, prevIndex, prevTerm, entries, commitIndex⟩ from q
do
    if term > currentTerm then
        stepdown(term)
    if term < currentTerm then
        send ⟨APPENDREP, currentTerm, false⟩ to q
    else
        index ← 0
        success ← prevIndex = 0 or (prevIndex ≤ log.len() and
          log[prevIndex].term = prevTerm)
        if success then
            index ← storeEntries(prevIndex, entries, commitIndex)
        send ⟨APPENDREP, currentTerm, success, index⟩
```

At beginning of new leader's term

- Old leader may have left entries partially replicated
- No special steps by new leader: just start normal operation
- Leader's log is "the truth"
- Will eventually make follower's logs identical to leader's
- Multiple crashes can leave many extraneous log entries

	1	2	3	4	5	6	7	8
s_1	T_1	T_1	T_5	T_6	T_6	T_6		
s_2	T_1	T_1	T_5	T_6	T_7	T_7	T_7	
s_3	T_1	T_1	T_5	T_5				
s_4	T_1	T_1	T_2	T_4				
s_5	T_1	T_1	T_2	T_2	T_3	T_3	T_3	

Safety Requirement

Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

- Raft safety property:
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
  - This guarantees the safety requirement
- Leaders never overwrite entries in their logs
  - Only entries in the leader's log can be committed
  - Entries must be committed before applying to state machine

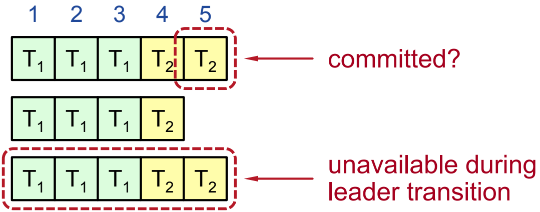
Committed → Present in future leaders' logs

Restrictions on commitment

Restrictions on leader election

Picking the Best Leader

- Can't tell which entries are committed!



- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include index & term of last log entry in VOTEREQ
  - Voting server V denies vote if its log is "more complete":
    $(lastLogTerm_C < lastLogTerm_V)$ **or**
    $(lastLogTerm_C = lastLogTerm_V$ **and** $lastLogIndex_C < lastLogIndex_V)$
  - Leader will have "most complete" log among electing majority

Election - Modified pseudocode

RPC timeout code - executed by process p

upon timeout $\langle \textsc{RpcTimeout}, q \rangle$ **do**
  if $state = \textsc{candidate}$ **then**
    set timeout $\langle \textsc{RpcTimeout}, q \rangle$ **at** $\mathsf{now}() + \Delta_{vote}$
    $lastLogTerm \leftarrow log[log.\mathsf{len}()].term$
    $lastLogIndex \leftarrow log.\mathsf{len}()$
    send $\langle \textsc{VoteReq}, currentTerm, lastLogTerm, lastLogIndex \rangle$ **to** q
  if $state = \textsc{leader}$ **then**
    set timeout $\langle \textsc{RpcTimeout}, q \rangle$ **at** $\mathsf{now}() + \Delta_{election}/2$
    $\mathsf{sendAppendEntries}(q)$

Election - Modified pseudocode

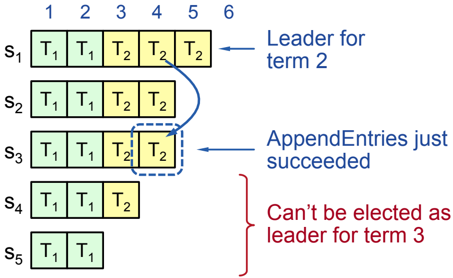
Election code - executed by process p

on receive $\langle \textsc{VoteReq}, \mathit{term}, \mathit{lastLogTerm}, \mathit{lastLogIndex} \rangle$ **from** q **do**

- **if** $\mathit{term} > \mathit{currentTerm}$ **then**
  - $\mathsf{stepdown}(\mathit{term})$
- **if** $\mathit{term} = \mathit{currentTerm}$ **and** $\mathit{votedFor} \in \{q, \mathbf{nil}\}$ **and** $(\mathit{lastLogTerm} > \mathit{log}[\mathit{log}.\mathsf{len}()].\mathit{term}$ **or** $(\mathit{lastLogTerm} = \mathit{log}[\mathit{log}.\mathsf{len}()].\mathit{term}$ **and** $\mathit{lastLogIndex} \geq \mathit{log}.\mathsf{len}())\ )$ **then**
  - $\mathit{votedFor} \leftarrow q$
  - $t \leftarrow \mathsf{random}(1.0, 2.0) \cdot \Delta_{\mathit{election}}$
  - **set timeout** $\langle \textsc{ElectionTimeout} \rangle$ **at** $\mathsf{now}() + t$
  - **send** $\langle \textsc{VoteRep}, \mathit{term}, \mathit{votedFor} \rangle$

Committing Entry from Current Term

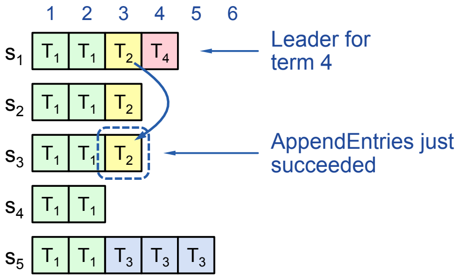
Case 1/2: Leader decides entry in current term is committed



Safe: leader for term T_3 must contain entry 4

Committing Entry from Earlier Terms

Case 2/2: Leader is trying to commit entry from an earlier term

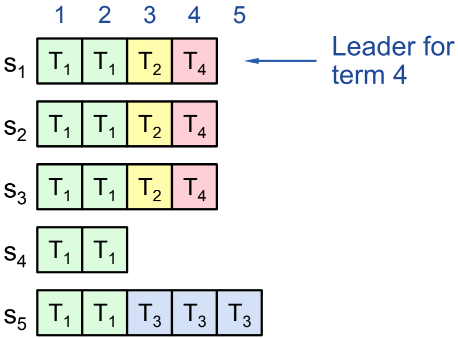


Unsafe: Entry 3 not safely committed

- s_5 can be elected as leader for term T_5
- If elected, it will overwrite entry 3 on s_1, s_2, and s_3!

New commitment rule

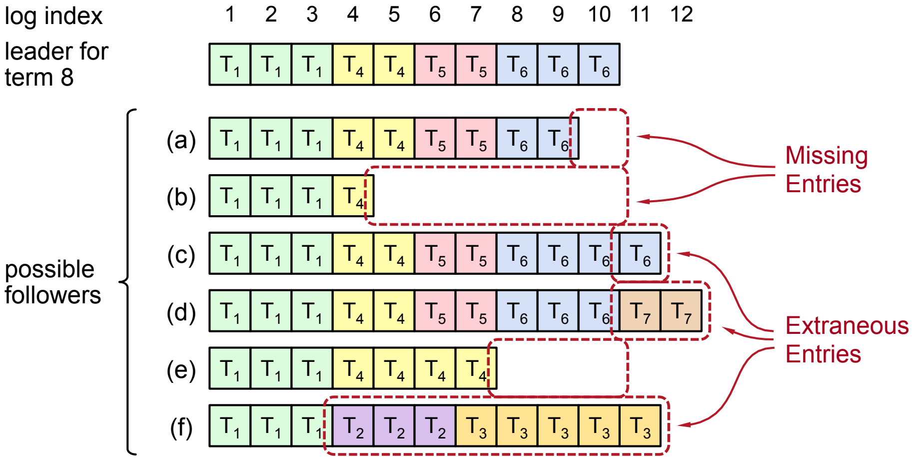
- For a leader to decide that an entry is committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers
- Once entry 4 committed:
  - s_5 cannot be elected leader for term T_5
  - Entries 3 and 4 both safe



Combination of election and commitment rules makes Raft safe

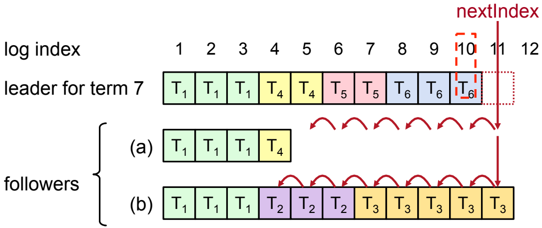
Log inconsistencies

Leader changes can result in log inconsistencies:



Repairing follower log

- 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



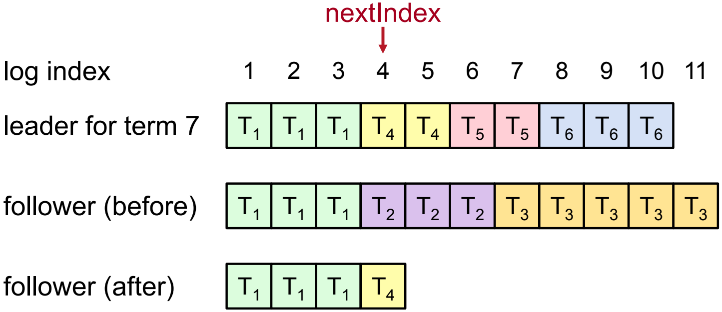
Repairing follower log – Pseudocode

Normal operation code - executed by process p

```
upon receive⟨AppendRep, term, success, index⟩ from q do
    if term > currentTerm then
        stepdown(term)
    else if state = LEADER and term = currentTerm then
        if success then
            nextIndex[q] ← index + 1
        else
            nextIndex[q] ← max(1, nextIndex[q] − 1)
        if nextIndex[q] ≤ log.len() then
            sendAppendEntries(q)
```

Repairing follower log

When follower overwrites inconsistent entry, it deletes all subsequent entries



Repairing follower log

procedure storeEntries $(prevIndex, entries, c)$

- $index \leftarrow prevIndex$
- **for** $j \leftarrow 1$ **to** $entries.\mathsf{len}()$ **do**
  - $index \leftarrow index + 1$
  - **if** $log[index].term \neq entries[j].term$ **then**
    - $log = log[1 \ldots index - 1] + entries[j]$
- $commitIndex \leftarrow \mathsf{min}(c, index)$
- **return** $index$

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

Neutralizing Old Leaders

Normal operation code - executed by process p

on receive $\langle$APPENDREQ, *term*, *prevIndex*, *prevTerm*, ...$\rangle$ **from** q
do
  if $term > currentTerm$ **then**
    stepdown($term$)
  if $term < currentTerm$ **then**
    send $\langle$APPENDREP, *currentTerm*, **false**$\rangle$ **to** q
  else
    [...]

Client protocol

Clients sends commands to leader:

- If leader unknown, contact any server
- If contacted server not leader, it will redirect to leader

Leader responds when:

- command has been logged
- command has been committed
- command has been executed by leader's state machine

If request times out (e.g., leader crash):

- Client re-issues command to some other server
- Eventually redirected to new leader
- Retry request with new leader

Client protocol

What if leader crashes after executing command, but before responding?

- Must not execute command twice

Solution: client embeds a unique id in each command

- Server includes id and response in log entry
- Before accepting command, leader checks its log for entry with that id
- If id found in log, ignore new command, return response from old command

Result: exactly-once semantics as long as client doesn't crash

Configuration

System configuration

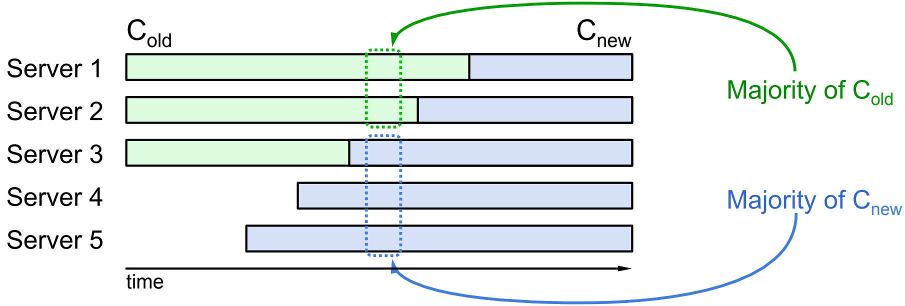
- ID, address for each server
- Determines what constitutes a majority

Consensus mechanism must support changes in the configuration

- Replace failed machine
- Change degree of replication

Configuration changes

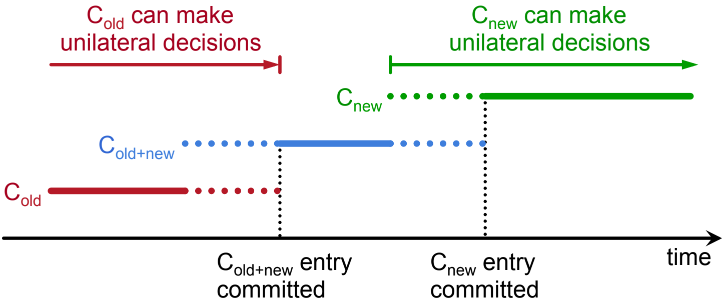
Cannot switch directly from one configuration to another: conflicting majorities could arise



Joint consensus

Raft uses a 2-phase approach

- Intermediate phase uses joint consensus (need majority of both old and new configurations for elections, commitment)
- Once joint consensus is committed, begin replicating log entry for final configuration



Reading Material

- D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm.

  In *2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

  http://www.disi.unitn.it/~montreso/ds/papers/raft.pdf