Utah Distributed Systems Meetup and Reading Group - Raft

JT Olds

Space Monkey
Vivint R&D

November 18, 2014

Outline

Outline

Introduction

Nearly all content and images from Diego Ongaro and John Ousterhout's 2014 paper, In Search of an Understandable Consensus Algorithm (Extended Version).

Introduction

- Raft is a consensus algorithm for managing a replicated log.
- Equivalent to Paxos in operation, except more understandable.
- Separates leader election, log replication, safety, and reduces possible states.
- Easier to learn.
- Supports cluster membership changes.

- Raft is a consensus algorithm for managing a replicated log.
- Equivalent to Paxos in operation, except more understandable.
- Separates leader election, log replication, safety, and reduces possible states.
- Easier to learn.
- Supports cluster membership changes.

- Raft is a consensus algorithm for managing a replicated log.
- Equivalent to Paxos in operation, except more understandable.
- Separates leader election, log replication, safety, and reduces possible states.
- Easier to learn.
- Supports cluster membership changes.

- Raft is a consensus algorithm for managing a replicated log.
- Equivalent to Paxos in operation, except more understandable.
- Separates leader election, log replication, safety, and reduces possible states.
- Easier to learn.
- Supports cluster membership changes.

- Raft is a consensus algorithm for managing a replicated log.
- Equivalent to Paxos in operation, except more understandable.
- Separates leader election, log replication, safety, and reduces possible states.
- Easier to learn.
- Supports cluster membership changes.

Introduction

- Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failure of some members.
- Paxos has been the primary consensus algorithm for too long.
- Paxos is difficult to understand and implement.
- Raft's key goal is understandability.

- Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failure of some members.
- Paxos has been the primary consensus algorithm for too long.
- Paxos is difficult to understand and implement.
- Raft’s key goal is understandability.

- Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failure of some members.
- Paxos has been the primary consensus algorithm for too long.
- Paxos is difficult to understand and implement.
- Raft's key goal is understandability.

- Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failure of some members.
- Paxos has been the primary consensus algorithm for too long.
- Paxos is difficult to understand and implement.
- Raft's key goal is understandability.

Notable raft features

- Strong leader
- Randomized timeouts for leader election
- Membership changes

Notable raft features

- Strong leader
- Randomized timeouts for leader election
- Membership changes

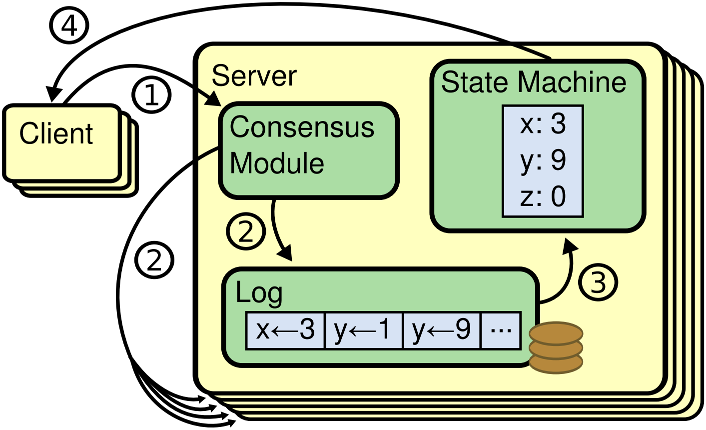
Notable raft features

- Strong leader
- Randomized timeouts for leader election
- Membership changes

Notable raft features

- Strong leader
- Randomized timeouts for leader election
- Membership changes

Introduction


④
Server
State Machine
x: 3
y: 9
z: 0
Client
①
Consensus Module
②
②
Log
x←3
y←1
y←9
...
③

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

- State machines must be completely deterministic.
- State machines operate on events popped from a log.
- Logs are managed by the consensus algorithm.

Consensus algorithms

- should provide safety - never return an incorrect result.
- should provide availability - must work when a majority of servers are up.
- should not depend on timing.
- can mitigate poor performance. A slow minority shouldn't be waited for.

Introduction

Brief aside

- 1970s - Jim Gray and others propose two-phase commit. Vulnerable to partitions.
- 1980s - Three-phase commit. Not provably correct, has problems.
- 1990s - Leslie Lamport tries to prove the properties of what became Paxos is impossible, comes up with Paxos.

Brief aside

- 1970s - Jim Gray and others propose two-phase commit. Vulnerable to partitions.
- 1980s - Three-phase commit. Not provably correct, has problems.
- 1990s - Leslie Lamport tries to prove the properties of what became Paxos is impossible, comes up with Paxos.

Brief aside

- 1970s - Jim Gray and others propose two-phase commit. Vulnerable to partitions.
- 1980s - Three-phase commit. Not provably correct, has problems.
- 1990s - Leslie Lamport tries to prove the properties of what became Paxos is impossible, comes up with Paxos.

Brief aside

- 1970s - Jim Gray and others propose two-phase commit. Vulnerable to partitions.
- 1980s - Three-phase commit. Not provably correct, has problems.
- 1990s - Leslie Lamport tries to prove the properties of what became Paxos is impossible, comes up with Paxos.

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 2

- The part-time parliament - 1998
- Paxos made simple - 2001
- The ABCD's of Paxos - 2001
- Generalized consensus and Paxos - 2005
- Fast paxos - 2006
- Paxos made live: an engineering perspective - 2007
- Paxos made practical - 2007
- Paxos for system builders - 2008
- Paxos made moderately complex - 2012
- In search of an understandable consensus algorithm - 2013

Brief aside - part 3

- Go To Statement Considered Harmful - 1968
- ‘GOTO Considered Harmful’ Considered Harmful - 1987
- ‘"GOTO Considered Harmful" Considered Harmful’ Considered Harmful? - 1987
- On a Somewhat Disappointing Correspondence - 1987

Brief aside - part 3

- Go To Statement Considered Harmful - 1968
- ‘GOTO Considered Harmful’ Considered Harmful - 1987
- ‘"GOTO Considered Harmful" Considered Harmful’ Considered Harmful? - 1987
- On a Somewhat Disappointing Correspondence - 1987

Brief aside - part 3

- Go To Statement Considered Harmful - 1968
- ‘GOTO Considered Harmful’ Considered Harmful - 1987
- ‘"GOTO Considered Harmful" Considered Harmful’ Considered Harmful? - 1987
- On a Somewhat Disappointing Correspondence - 1987

Brief aside - part 3

- Go To Statement Considered Harmful - 1968
- ‘GOTO Considered Harmful’ Considered Harmful - 1987
- ‘"GOTO Considered Harmful" Considered Harmful’ Considered Harmful? - 1987
- On a Somewhat Disappointing Correspondence - 1987

Brief aside - part 3

- Go To Statement Considered Harmful - 1968
- ‘GOTO Considered Harmful’ Considered Harmful - 1987
- ‘"GOTO Considered Harmful" Considered Harmful’ Considered Harmful? - 1987
- On a Somewhat Disappointing Correspondence - 1987

Paxos is broken into

- single-decree Paxos - goal is to replicate one log entry
- multi-Paxos - combines single-decree Paxos to decide a full log.

Paxos is broken into

- single-decree Paxos - goal is to replicate one log entry
- multi-Paxos - combines single-decree Paxos to decide a full log.

Paxos is broken into

- single-decree Paxos - goal is to replicate one log entry
- multi-Paxos - combines single-decree Paxos to decide a full log.

Problems?

- super opaque and subtle - probably due to weird decomposition.
- multi-Paxos only has possible approach sketches! Attempts to flesh out missing details differ from Lamport's sketch and each other, and some have not been published.
- Paxos is symmetric peer-to-peer at its core (no leaders) which is inefficient when a bunch of decisions need to be made.
- Paxos is good for proving theorems about Paxos, but said proofs matter little when real implementations can differ so drastically.

Problems?

- super opaque and subtle - probably due to weird decomposition.
- multi-Paxos only has possible approach sketches! Attempts to flesh out missing details differ from Lamport's sketch and each other, and some have not been published.
- Paxos is symmetric peer-to-peer at its core (no leaders) which is inefficient when a bunch of decisions need to be made.
- Paxos is good for proving theorems about Paxos, but said proofs matter little when real implementations can differ so drastically.

Problems?

- super opaque and subtle - probably due to weird decomposition.
- multi-Paxos only has possible approach sketches! Attempts to flesh out missing details differ from Lamport's sketch and each other, and some have not been published.
- Paxos is symmetric peer-to-peer at its core (no leaders) which is inefficient when a bunch of decisions need to be made.
- Paxos is good for proving theorems about Paxos, but said proofs matter little when real implementations can differ so drastically.

Problems?

- super opaque and subtle - probably due to weird decomposition.
- multi-Paxos only has possible approach sketches! Attempts to flesh out missing details differ from Lamport's sketch and each other, and some have not been published.
- Paxos is symmetric peer-to-peer at its core (no leaders) which is inefficient when a bunch of decisions need to be made.
- Paxos is good for proving theorems about Paxos, but said proofs matter little when real implementations can differ so drastically.

Problems?

- super opaque and subtle - probably due to weird decomposition.
- multi-Paxos only has possible approach sketches! Attempts to flesh out missing details differ from Lamport's sketch and each other, and some have not been published.
- Paxos is symmetric peer-to-peer at its core (no leaders) which is inefficient when a bunch of decisions need to be made.
  Is this actually a problem? Byzantine empires might say no.
- Paxos is good for proving theorems about Paxos, but said proofs matter little when real implementations can differ so drastically.

Problems?

- super opaque and subtle - probably due to weird decomposition.
- multi-Paxos only has possible approach sketches! Attempts to flesh out missing details differ from Lamport's sketch and each other, and some have not been published.
- Paxos is symmetric peer-to-peer at its core (no leaders) which is inefficient when a bunch of decisions need to be made.
  Is this actually a problem? Byzantine empires might say no.
- Paxos is good for proving theorems about Paxos, but said proofs matter little when real implementations can differ so drastically.

Introduction

1 Introduction

- Abstract
- 1. Introduction
- 2. Replicated state machines
- 3. What's wrong with Paxos?
- 4. Designing for understandability

Goals

- Reduce developer design work (no unproven protocols)
- Safe under all conditions
- Available under typical conditions
- Efficient for common operations
- Understandable

Goals

- Reduce developer design work (no unproven protocols)
- Safe under all conditions
- Available under typical conditions
- Efficient for common operations
- Understandable

Goals

- Reduce developer design work (no unproven protocols)
- Safe under all conditions
- Available under typical conditions
- Efficient for common operations
- Understandable

Goals

- Reduce developer design work (no unproven protocols)
- Safe under all conditions
- Available under typical conditions
- Efficient for common operations
- Understandable

Goals

- Reduce developer design work (no unproven protocols)
- Safe under all conditions
- Available under typical conditions
- Efficient for common operations
- Understandable

Goals

- Reduce developer design work (no unproven protocols)
- Safe under all conditions
- Available under typical conditions
- Efficient for common operations
- Understandable

Understandability

- When faced with a choice, choose the easiest to explain.
- Subdivide problems
- Shrink state space

Nondeterminism

- Nondeterminism usually eliminated
- except where it makes the system simpler! (randomized approaches)

Understandability

- When faced with a choice, choose the easiest to explain.
- Subdivide problems
- Shrink state space

Nondeterminism

- Nondeterminism usually eliminated
- except where it makes the system simpler! (randomized approaches)

Understandability

- When faced with a choice, choose the easiest to explain.
- Subdivide problems
- Shrink state space

Nondeterminism

- Nondeterminism usually eliminated
- except where it makes the system simpler! (randomized approaches)

Understandability

- When faced with a choice, choose the easiest to explain.
- Subdivide problems
- Shrink state space

Nondeterminism

- Nondeterminism usually eliminated
- except where it makes the system simpler! (randomized approaches)

Understandability

- When faced with a choice, choose the easiest to explain.
- Subdivide problems
- Shrink state space

Nondeterminism

- Nondeterminism usually eliminated
- except where it makes the system simpler! (randomized approaches)

Understandability

- When faced with a choice, choose the easiest to explain.
- Subdivide problems
- Shrink state space

Nondeterminism

- Nondeterminism usually eliminated
- except where it makes the system simpler! (randomized approaches)

Understandability

- When faced with a choice, choose the easiest to explain.
- Subdivide problems
- Shrink state space

Nondeterminism

- Nondeterminism usually eliminated
- except where it makes the system simpler! (randomized approaches)

Outline

Algorithm

2 Algorithm

- 5. The Raft consensus algorithm
- 5.1. Raft basics
- 5.2. Leader election
- 5.3. Log replication
- 5.4. Safety
- 5.5. Follower and candidate crashes
- 5.6. Timing and availability

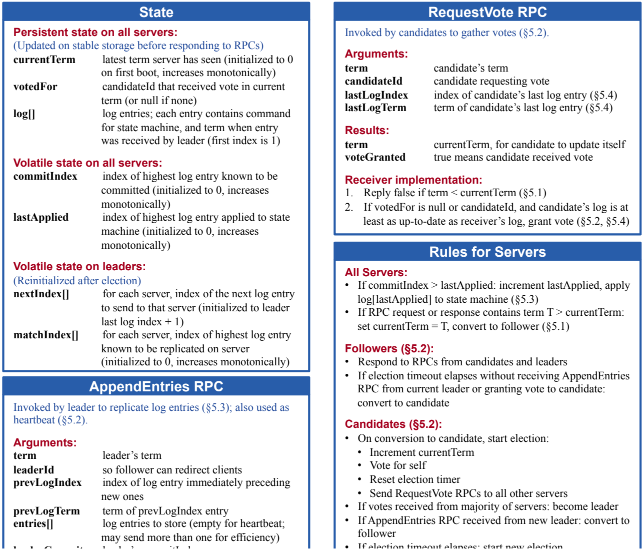
Simulation

`raftconsensus.github.io`

Algorithm

2 Algorithm

- 5. The Raft consensus algorithm
- 5.1. Raft basics
- 5.2. Leader election
- 5.3. Log replication
- 5.4. Safety
- 5.5. Follower and candidate crashes
- 5.6. Timing and availability



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 (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
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)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

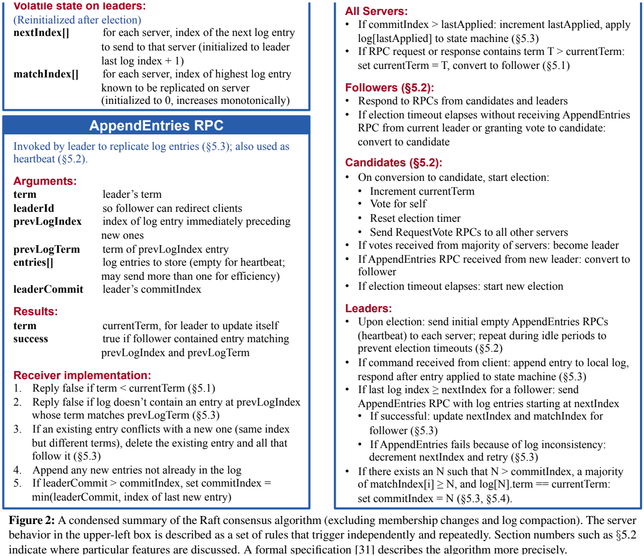
- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- 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 (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - 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



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 (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
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 (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- 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 (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - 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 (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- 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 (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.

Leaders

- Leaders get complete responsibility for managing the replicated log.
- All changes flow from the leader to others.

Subproblems

- Leader election (5.2)
- Log replication (5.3)
- Safety (5.4)

Leaders

- Leaders get complete responsibility for managing the replicated log.
- All changes flow from the leader to others.

Subproblems

- Leader election (5.2)
- Log replication (5.3)
- Safety (5.4)

Leaders

- Leaders get complete responsibility for managing the replicated log.
- All changes flow from the leader to others.

Subproblems

- Leader election (5.2)
- Log replication (5.3)
- Safety (5.4)

Leaders

- Leaders get complete responsibility for managing the replicated log.
- All changes flow from the leader to others.

Subproblems

- Leader election (5.2)
- Log replication (5.3)
- Safety (5.4)

Leaders

- Leaders get complete responsibility for managing the replicated log.
- All changes flow from the leader to others.

Subproblems

- Leader election (5.2)
- Log replication (5.3)
- Safety (5.4)

Leaders

- Leaders get complete responsibility for managing the replicated log.
- All changes flow from the leader to others.

Subproblems

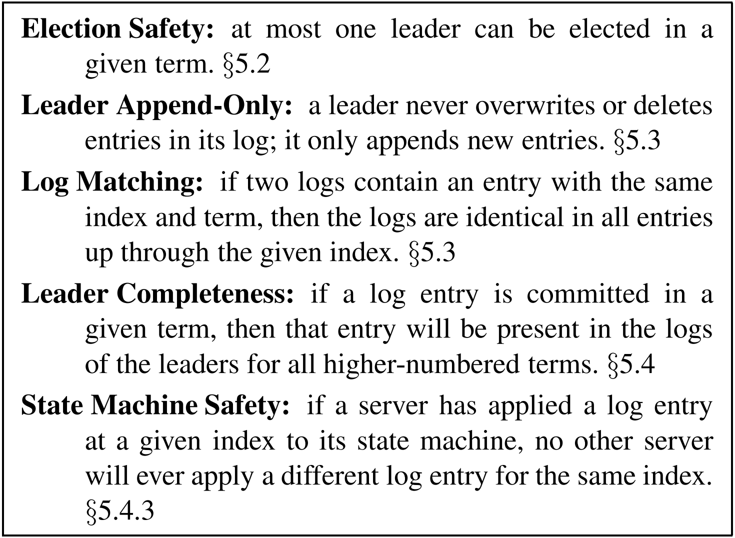
- Leader election (5.2)
- Log replication (5.3)
- Safety (5.4)

Leaders

- Leaders get complete responsibility for managing the replicated log.
- All changes flow from the leader to others.

Subproblems

- Leader election (5.2)
- Log replication (5.3)
- Safety (5.4)



Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Algorithm

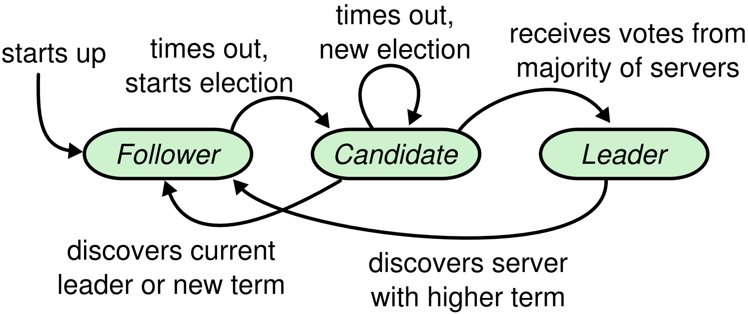
- Raft cluster contains several servers - e.g. five allows for two failures.
- Servers are in one of only three states - leader, follower, or candidate.
- There should only be one leader. Leader handles all client requests.
- Leaders typically operate until they fail.
- Followers are passive - all client requests are forwarded to the leader.

- Raft cluster contains several servers - e.g. five allows for two failures.
- Servers are in one of only three states - leader, follower, or candidate.
- There should only be one leader. Leader handles all client requests.
- Leaders typically operate until they fail.
- Followers are passive - all client requests are forwarded to the leader.

- Raft cluster contains several servers - e.g. five allows for two failures.
- Servers are in one of only three states - leader, follower, or candidate.
- There should only be one leader. Leader handles all client requests.
- Leaders typically operate until they fail.
- Followers are passive - all client requests are forwarded to the leader.

- Raft cluster contains several servers - e.g. five allows for two failures.
- Servers are in one of only three states - leader, follower, or candidate.
- There should only be one leader. Leader handles all client requests.
- Leaders typically operate until they fail.
- Followers are passive - all client requests are forwarded to the leader.

- Raft cluster contains several servers - e.g. five allows for two failures.
- Servers are in one of only three states - leader, follower, or candidate.
- There should only be one leader. Leader handles all client requests.
- Leaders typically operate until they fail.
- Followers are passive - all client requests are forwarded to the leader.


starts up
times out,
starts election
times out,
new election
receives votes from
majority of servers
Follower
Candidate
Leader
discovers current
leader or new term
discovers server
with higher term

Terms

- A term is arbitrary length.
- Terms are numbered with consecutive integers.
- Terms begin with an election.
- Terms with split-vote elections end with no leader, and a new term starts.
- Terms form a logical clock and the current term is exchanged during all communications.
- Stale terms are rejected, new terms are immediately accepted (reverting to follower state).

Terms

- A term is arbitrary length.
- Terms are numbered with consecutive integers.
- Terms begin with an election.
- Terms with split-vote elections end with no leader, and a new term starts.
- Terms form a logical clock and the current term is exchanged during all communications.
- Stale terms are rejected, new terms are immediately accepted (reverting to follower state).

Terms

- A term is arbitrary length.
- Terms are numbered with consecutive integers.
- Terms begin with an election.
- Terms with split-vote elections end with no leader, and a new term starts.
- Terms form a logical clock and the current term is exchanged during all communications.
- Stale terms are rejected, new terms are immediately accepted (reverting to follower state).

Terms

- A term is arbitrary length.
- Terms are numbered with consecutive integers.
- Terms begin with an election.
- Terms with split-vote elections end with no leader, and a new term starts.
- Terms form a logical clock and the current term is exchanged during all communications.
- Stale terms are rejected, new terms are immediately accepted (reverting to follower state).

Terms

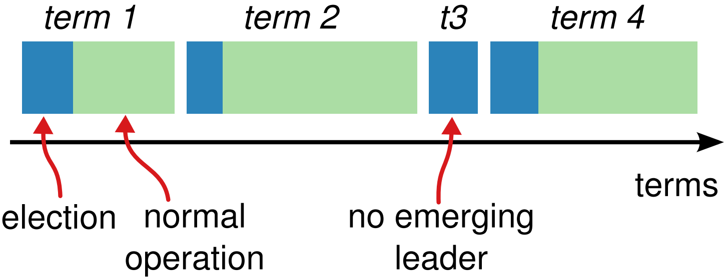
- A term is arbitrary length.
- Terms are numbered with consecutive integers.
- Terms begin with an election.
- Terms with split-vote elections end with no leader, and a new term starts.
- Terms form a logical clock and the current term is exchanged during all communications.
- Stale terms are rejected, new terms are immediately accepted (reverting to follower state).

Terms

- A term is arbitrary length.
- Terms are numbered with consecutive integers.
- Terms begin with an election.
- Terms with split-vote elections end with no leader, and a new term starts.
- Terms form a logical clock and the current term is exchanged during all communications.
- Stale terms are rejected, new terms are immediately accepted (reverting to follower state).

Terms

- A term is arbitrary length.
- Terms are numbered with consecutive integers.
- Terms begin with an election.
- Terms with split-vote elections end with no leader, and a new term starts.
- Terms form a logical clock and the current term is exchanged during all communications.
- Stale terms are rejected, new terms are immediately accepted (reverting to follower state).


term 1
term 2
t3
term 4
terms
election
normal operation
no emerging leader

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Only two main RPCs, three if you count log compaction

- RequestVote - initiated by candidates, used during elections.
- AppendEntries - initiated by leaders for heartbeats and log replication.
- InstallSnapshot - used for log compaction extension

RPC properties

- RPCs are retried until responses are received.
- RPCs are idempotent.
- RPCs are issued in parallel wherever possible.

Algorithm

2 Algorithm
- 5. The Raft consensus algorithm
- 5.1. Raft basics
- 5.2. Leader election
- 5.3. Log replication
- 5.4. Safety
- 5.5. Follower and candidate crashes
- 5.6. Timing and availability

- All servers begin as followers.
- Servers stay followers as long as they receive AppendEntries RPCs heartbeats (whether or not there are any log entries).
- If a server hears no AppendEntries call before an election timeout, it begins an election.

- All servers begin as followers.
- Servers stay followers as long as they receive AppendEntries RPCs heartbeats (whether or not there are any log entries).
- If a server hears no AppendEntries call before an election timeout, it begins an election.

- All servers begin as followers.
- Servers stay followers as long as they receive AppendEntries RPCs heartbeats (whether or not there are any log entries).
- If a server hears no AppendEntries call before an election timeout, it begins an election.

Elections

- Follower increments its current term and transitions to candidate state.
- Votes for itself and requests votes from the other servers.

Election termination

One of three things:

- it wins the election; now it's the leader
- it finds out about another leader; now it's a follower
- neither previous case happens before another election timeout; the election starts over.

Elections

- Follower increments its current term and transitions to candidate state.
- Votes for itself and requests votes from the other servers.

Election termination

One of three things:

- it wins the election; now it's the leader
- it finds out about another leader; now it's a follower
- neither previous case happens before another election timeout; the election starts over.

Elections

- Follower increments its current term and transitions to candidate state.
- Votes for itself and requests votes from the other servers.

Election termination

One of three things:

- it wins the election; now it's the leader
- it finds out about another leader; now it's a follower
- neither previous case happens before another election timeout; the election starts over.

Elections

- Follower increments its current term and transitions to candidate state.
- Votes for itself and requests votes from the other servers.

Election termination

One of three things:

- it wins the election; now it's the leader
- it finds out about another leader; now it's a follower
- neither previous case happens before another election timeout; the election starts over.

Elections

- Follower increments its current term and transitions to candidate state.
- Votes for itself and requests votes from the other servers.

Election termination

One of three things:

- it wins the election; now it's the leader
- it finds out about another leader; now it's a follower
- neither previous case happens before another election timeout; the election starts over.

Elections

- Follower increments its current term and transitions to candidate state.
- Votes for itself and requests votes from the other servers.

Election termination

One of three things:

- it wins the election; now it's the leader
- it finds out about another leader; now it's a follower
- neither previous case happens before another election timeout; the election starts over.

Elections

- Follower increments its current term and transitions to candidate state.
- Votes for itself and requests votes from the other servers.

Election termination

One of three things:

- it wins the election; now it's the leader
- it finds out about another leader; now it's a follower
- neither previous case happens before another election timeout; the election starts over.

Voting

- Winning is assumed if you receive a majority of votes.
- Each follower will vote for at most one candidate per term, first-come-first-served.
- At any time if any server hears a heartbeat message with a leader in the current term or newer, it assumes the source is the leader.

Split votes

Randomized election timeouts!

Voting

- Winning is assumed if you receive a majority of votes.
- Each follower will vote for at most one candidate per term, first-come-first-served.
- At any time if any server hears a heartbeat message with a leader in the current term or newer, it assumes the source is the leader.

Split votes

Randomized election timeouts!

Voting

- Winning is assumed if you receive a majority of votes.
- Each follower will vote for at most one candidate per term, first-come-first-served.
- At any time if any server hears a heartbeat message with a leader in the current term or newer, it assumes the source is the leader.

Split votes

Randomized election timeouts!

Voting

- Winning is assumed if you receive a majority of votes.
- Each follower will vote for at most one candidate per term, first-come-first-served.
- At any time if any server hears a heartbeat message with a leader in the current term or newer, it assumes the source is the leader.

Split votes

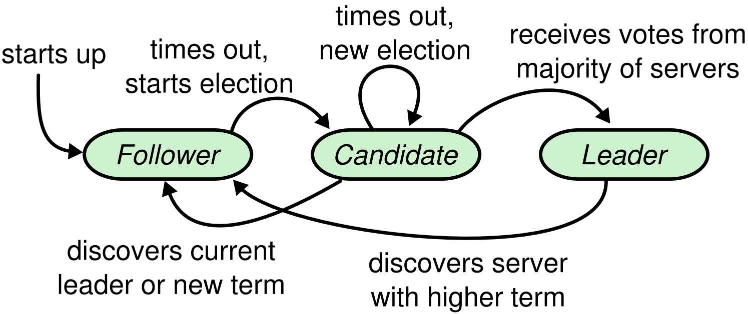
Randomized election timeouts!

Voting

- Winning is assumed if you receive a majority of votes.
- Each follower will vote for at most one candidate per term, first-come-first-served.
- At any time if any server hears a heartbeat message with a leader in the current term or newer, it assumes the source is the leader.

Split votes

Randomized election timeouts!



starts up

times out, starts election

times out, new election

receives votes from majority of servers

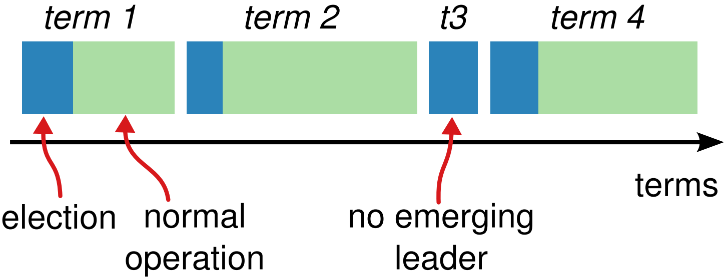
Follower

Candidate

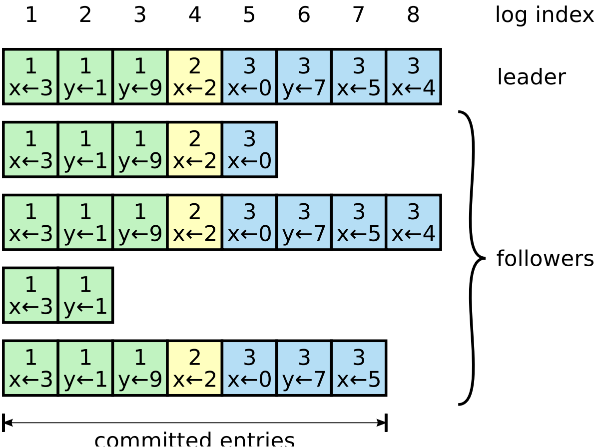
Leader

discovers current leader or new term

discovers server with higher term


term 1
term 2
t3
term 4
terms
election
normal operation
no emerging leader

Algorithm



log index	1	2	3	4	5	6	7	8
leader	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
followers	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0			
	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	3 x←4
	1 x←3	1 y←1						
	1 x←3	1 y←1	1 y←9	2 x←2	3 x←0	3 y←7	3 x←5	

- Leaders service client requests.
- Client request commands are added to the leader's log.
- Leaders then pester followers to add the command to their logs via AppendEntries.
- Entries are identified by their term number and log index.
- Entries are uncommitted until the leader has determined that a majority of servers have the entry.
- AppendEntries calls (including heartbeats) indicate the highest committed index.
- Committed entries are passed off to each server's state machine in order.

- Leaders service client requests.
- Client request commands are added to the leader’s log.
- Leaders then pester followers to add the command to their logs via AppendEntries.
- Entries are identified by their term number and log index.
- Entries are uncommitted until the leader has determined that a majority of servers have the entry.
- AppendEntries calls (including heartbeats) indicate the highest committed index.
- Committed entries are passed off to each server’s state machine in order.

- Leaders service client requests.
- Client request commands are added to the leader's log.
- Leaders then pester followers to add the command to their logs via AppendEntries.
- Entries are identified by their term number and log index.
- Entries are uncommitted until the leader has determined that a majority of servers have the entry.
- AppendEntries calls (including heartbeats) indicate the highest committed index.
- Committed entries are passed off to each server's state machine in order.

- Leaders service client requests.
- Client request commands are added to the leader's log.
- Leaders then pester followers to add the command to their logs via AppendEntries.
- Entries are identified by their term number and log index.
- Entries are uncommitted until the leader has determined that a majority of servers have the entry.
- AppendEntries calls (including heartbeats) indicate the highest committed index.
- Committed entries are passed off to each server's state machine in order.

- Leaders service client requests.
- Client request commands are added to the leader's log.
- Leaders then pester followers to add the command to their logs via AppendEntries.
- Entries are identified by their term number and log index.
- Entries are <u>uncommitted</u> until the leader has determined that a majority of servers have the entry.
- AppendEntries calls (including heartbeats) indicate the highest committed index.
- Committed entries are passed off to each server's state machine in order.

- Leaders service client requests.
- Client request commands are added to the leader’s log.
- Leaders then pester followers to add the command to their logs via AppendEntries.
- Entries are identified by their term number and log index.
- Entries are uncommitted until the leader has determined that a majority of servers have the entry.
- AppendEntries calls (including heartbeats) indicate the highest committed index.
- Committed entries are passed off to each server’s state machine in order.

- Leaders service client requests.
- Client request commands are added to the leader's log.
- Leaders then pester followers to add the command to their logs via AppendEntries.
- Entries are identified by their term number and log index.
- Entries are uncommitted until the leader has determined that a majority of servers have the entry.
- AppendEntries calls (including heartbeats) indicate the highest committed index.
- Committed entries are passed off to each server's state machine in order.

Logs match

- Every log entry is given a term id.
- There is only one leader per term, and leaders never change log entry indices.
- So, given a term id, the log index is unique.
- AppendEntries includes the previous term id and log index, so if that log entry is missing, the follower will reject the call.
- The leader will back up and replay the log up to the offending entry.

Logs match

- Every log entry is given a term id.
- There is only one leader per term, and leaders never change log entry indices.
- So, given a term id, the log index is unique.
- AppendEntries includes the previous term id and log index, so if that log entry is missing, the follower will reject the call.
- The leader will back up and replay the log up to the offending entry.

Logs match

- Every log entry is given a term id.
- There is only one leader per term, and leaders never change log entry indices.
- So, given a term id, the log index is unique.
- AppendEntries includes the previous term id and log index, so if that log entry is missing, the follower will reject the call.
- The leader will back up and replay the log up to the offending entry.

Logs match

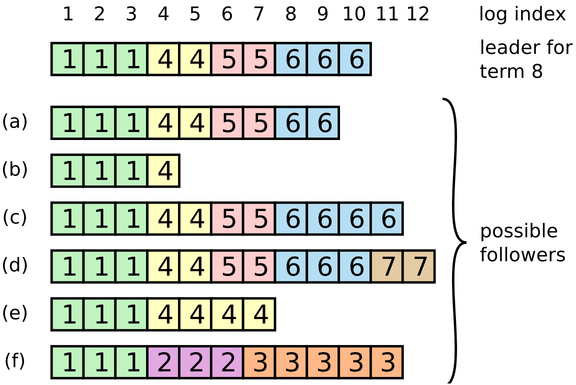
- Every log entry is given a term id.
- There is only one leader per term, and leaders never change log entry indices.
- So, given a term id, the log index is unique.
- AppendEntries includes the previous term id and log index, so if that log entry is missing, the follower will reject the call.
- The leader will back up and replay the log up to the offending entry.

Logs match

- Every log entry is given a term id.
- There is only one leader per term, and leaders never change log entry indices.
- So, given a term id, the log index is unique.
- AppendEntries includes the previous term id and log index, so if that log entry is missing, the follower will reject the call.
- The leader will back up and replay the log up to the offending entry.

Logs match

- Every log entry is given a term id.
- There is only one leader per term, and leaders never change log entry indices.
- So, given a term id, the log index is unique.
- AppendEntries includes the previous term id and log index, so if that log entry is missing, the follower will reject the call.
- The leader will back up and replay the log up to the offending entry.



log index	1	2	3	4	5	6	7	8	9	10	11	12
leader for term 8	1	1	1	4	4	5	5	6	6	6		
(a)	1	1	1	4	4	5	5	6	6			
(b)	1	1	1	4								
(c)	1	1	1	4	4	5	5	6	6	6	6	
(d)	1	1	1	4	4	5	5	6	6	6	7	7
(e)	1	1	1	4	4	4	4					
(f)	1	1	1	2	2	2	3	3	3	3	3	

possible followers

Conflict handling

- Leaders force followers logs to duplicate their own.
- Conflicting entries will get overwritten.
- Leaders never overwrite or delete their own entries.

Conflict handling

- Leaders force followers logs to duplicate their own.
- Conflicting entries will get overwritten.
- Leaders never overwrite or delete their own entries.

Conflict handling

- Leaders force followers logs to duplicate their own.
- Conflicting entries will get overwritten.
- Leaders never overwrite or delete their own entries.

Conflict handling

- Leaders force followers logs to duplicate their own.
- Conflicting entries will get overwritten.
- Leaders never overwrite or delete their own entries.

Whoa?!

Algorithm

2 Algorithm

5.4.1 Election restriction

- A leader will not get voted for if it’s missing entries the voter has.
- Logs are efficiently compared by sorting the 2-tuple (term id, log index)

5.4.1 Election restriction

- A leader will not get voted for if it’s missing entries the voter has.
- Logs are efficiently compared by sorting the 2-tuple (term id, log index)

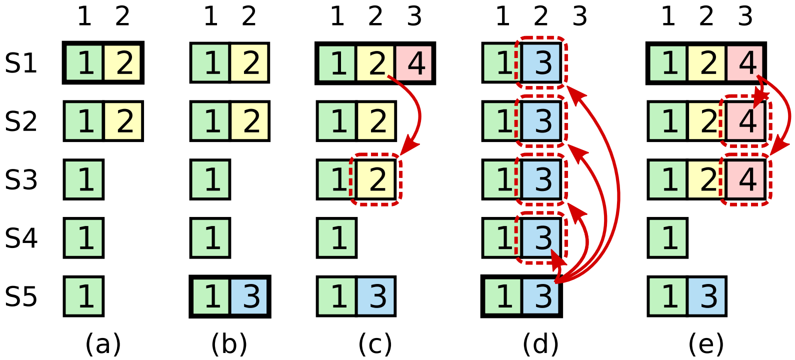
5.4.1 Election restriction

- A leader will not get voted for if it’s missing entries the voter has.
- Logs are efficiently compared by sorting the 2-tuple (term id, log index)

5.4.2 Committing entries from previous terms

A leader cannot assume an entry that exists on a majority of servers from a previous term is committed.

5.4.2 Committing entries from previous terms



5.4.3 Safety argument

Proof

Algorithm

2 Algorithm

- 5. The Raft consensus algorithm
- 5.1. Raft basics
- 5.2. Leader election
- 5.3. Log replication
- 5.4. Safety
- **5.5. Follower and candidate crashes**
- 5.6. Timing and availability

- Raft RPCs are idempotent
- Raft retries failed requests indefinitely
- Follower and candidate crashes are trivially handled for free.

- Raft RPCs are idempotent
- Raft retries failed requests indefinitely
- Follower and candidate crashes are trivially handled for free.

- Raft RPCs are idempotent
- Raft retries failed requests indefinitely
- Follower and candidate crashes are trivially handled for free.

Algorithm

2 Algorithm
- 5. The Raft consensus algorithm
- 5.1. Raft basics
- 5.2. Leader election
- 5.3. Log replication
- 5.4. Safety
- 5.5. Follower and candidate crashes
- 5.6. Timing and availability

Timing requirement

- $broadcastTime << electionTimeout << MTBF$
- *broadcastTime* and *MTBF* are usually fixed.
- *broadcastTime* is usually dominated by disk write time, since logs are persisted to stable storage.
- So, *electionTimeout* is typically between 10ms and 500ms.

Timing requirement

- $broadcastTime << electionTimeout << MTBF$
- *broadcastTime* and *MTBF* are usually fixed.
- *broadcastTime* is usually dominated by disk write time, since logs are persisted to stable storage.
- So, *electionTimeout* is typically between 10ms and 500ms.

Timing requirement

- $broadcastTime << electionTimeout << MTBF$
- *broadcastTime* and *MTBF* are usually fixed.
- *broadcastTime* is usually dominated by disk write time, since logs are persisted to stable storage.
- So, *electionTimeout* is typically between 10ms and 500ms.

Timing requirement

- $broadcastTime << electionTimeout << MTBF$
- *broadcastTime* and *MTBF* are usually fixed.
- *broadcastTime* is usually dominated by disk write time, since logs are persisted to stable storage.
- So, *electionTimeout* is typically between 10ms and 500ms.

Timing requirement

- $broadcastTime << electionTimeout << MTBF$
- *broadcastTime* and *MTBF* are usually fixed.
- *broadcastTime* is usually dominated by disk write time, since logs are persisted to stable storage.
- So, *electionTimeout* is typically between 10ms and 500ms.

Simulation

`raftconsensus.github.io`

Outline

Other practical concerns

Other practical concerns

3 Other practical concerns
- 6. Cluster membership changes
- 7. Log compaction
- 8. Client interaction

Overall idea

- Must adhere to one-leader-per-term rule during switch.
- Rules out any direct or atomic configuration switches.
- Two-phase approach uses joint-consensus: for a term the system uses the union of the two configurations.

Overall idea

- Must adhere to one-leader-per-term rule during switch.
- Rules out any direct or atomic configuration switches.
- Two-phase approach uses joint-consensus: for a term the system uses the union of the two configurations.

Overall idea

- Must adhere to one-leader-per-term rule during switch.
- Rules out any direct or atomic configuration switches.
- Two-phase approach uses joint-consensus: for a term the system uses the union of the two configurations.

Overall idea

- Must adhere to one-leader-per-term rule during switch.
- Rules out any direct or atomic configuration switches.
- Two-phase approach uses joint-consensus: for a term the system uses the union of the two configurations.

Implementation

- Uses a special configuration log entry. The latest configuration log entry applies regardless of committedness.
- Once a configuration is committed it is safe to move to the next configuration (from joint to new).

Implementation

- Uses a special configuration log entry. The latest configuration log entry applies regardless of committedness.
- Once a configuration is committed it is safe to move to the next configuration (from joint to new).

Implementation

- Uses a special configuration log entry. The latest configuration log entry applies regardless of committedness.
- Once a configuration is committed it is safe to move to the next configuration (from joint to new).

Issues

- New servers might be incredibly behind - can join as non-voting members before new configuration is applied
- Current leader might not be part of new configuration - leaders step down after committing configuration and possibly shouldn't count themselves as part of the majority.
- Cluster can be disrupted by old nodes interferring and becoming candidates - servers can disregard RequestVote when they believe a leader exists, but it's best to get old nodes out.

Issues

- New servers might be incredibly behind - can join as non-voting members before new configuration is applied
- Current leader might not be part of new configuration - leaders step down after committing configuration and possibly shouldn't count themselves as part of the majority.
- Cluster can be disrupted by old nodes interferring and becoming candidates - servers can disregard RequestVote when they believe a leader exists, but it's best to get old nodes out.

Issues

- New servers might be incredibly behind - can join as non-voting members before new configuration is applied
- Current leader might not be part of new configuration - leaders step down after committing configuration and possibly shouldn't count themselves as part of the majority.
- Cluster can be disrupted by old nodes interferring and becoming candidates - servers can disregard RequestVote when they believe a leader exists, but it's best to get old nodes out.

Issues

- New servers might be incredibly behind - can join as non-voting members before new configuration is applied
- Current leader might not be part of new configuration - leaders step down after committing configuration and possibly shouldn't count themselves as part of the majority.
- Cluster can be disrupted by old nodes interferring and becoming candidates - servers can disregard RequestVote when they believe a leader exists, but it's best to get old nodes out.

Other practical concerns

3 Other practical concerns
- 6. Cluster membership changes
- 7. Log compaction
- 8. Client interaction

Snapshotting

- Requires interaction with state machine and state machine serialization.
- Snapshot should indicate last included log index.
- InstallSnapshot RPC applies a snapshot to a follower when the follower is farther behind what the log has.

Snapshotting

- Requires interaction with state machine and state machine serialization.
- Snapshot should indicate last included log index.
- InstallSnapshot RPC applies a snapshot to a follower when the follower is farther behind what the log has.

Snapshotting

- Requires interaction with state machine and state machine serialization.
- Snapshot should indicate last included log index.
- InstallSnapshot RPC applies a snapshot to a follower when the follower is farther behind what the log has.

Snapshotting

- Requires interaction with state machine and state machine serialization.
- Snapshot should indicate last included log index.
- InstallSnapshot RPC applies a snapshot to a follower when the follower is farther behind what the log has.

Other practical concerns

3 Other practical concerns
- 6. Cluster membership changes
- 7. Log compaction
- 8. Client interaction

Linearizability

Clients must make all operations idempotent, or attach unique serial numbers to all commands, in case the request is received but the response is lost.

Read-only ops

- Leaders should know the latest information on what entries are committed, so at least one heartbeat or operation needs to have happened when the leader starts.
- Leaders may have gotten deposed, so they need to check with the cluster before responding to read-only requests.

Linearizability

Clients must make all operations idempotent, or attach unique serial numbers to all commands, in case the request is received but the response is lost.

Read-only ops

- Leaders should know the latest information on what entries are committed, so at least one heartbeat or operation needs to have happened when the leader starts.
- Leaders may have gotten deposed, so they need to check with the cluster before responding to read-only requests.

Linearizability

Clients must make all operations idempotent, or attach unique serial numbers to all commands, in case the request is received but the response is lost.

Read-only ops

- Leaders should know the latest information on what entries are committed, so at least one heartbeat or operation needs to have happened when the leader starts.
- Leaders may have gotten deposed, so they need to check with the cluster before responding to read-only requests.

Linearizability

Clients must make all operations idempotent, or attach unique serial numbers to all commands, in case the request is received but the response is lost.

Read-only ops

- Leaders should know the latest information on what entries are committed, so at least one heartbeat or operation needs to have happened when the leader starts.
- Leaders may have gotten deposed, so they need to check with the cluster before responding to read-only requests.

Outline

Paper Conclusion

4 Paper Conclusion
- 9. Implementation and evaluation
- 10. Related work
- 11, 12. Conclusion & Acknowledgements

Paper Conclusion

4 Paper Conclusion
- 9. Implementation and evaluation
- 10. Related work
- 11, 12. Conclusion & Acknowledgements

9.1. Understandability

- How do you measure understandability?
- You teach two otherwise-identical classes on Paxos and Raft, and make kids take tests!
- Paxos lecture: `http://youtu.be/JEpsBg0AO6o`
- Raft lecture: `http://youtu.be/YbZ3zDzDnrw`
- Exams: `https://ramcloud.stanford.edu/~ongaro/userstudy/quizzes.html`

9.1. Understandability

- How do you measure understandability?
- You teach two otherwise-identical classes on Paxos and Raft, and make kids take tests!
- Paxos lecture: `http://youtu.be/JEpsBg0AO6o`
- Raft lecture: `http://youtu.be/YbZ3zDzDnrw`
- Exams: `https://ramcloud.stanford.edu/~ongaro/userstudy/quizzes.html`

9.1. Understandability

- How do you measure understandability?
- You teach two otherwise-identical classes on Paxos and Raft, and make kids take tests!
- Paxos lecture: `http://youtu.be/JEpsBg0AO6o`
- Raft lecture: `http://youtu.be/YbZ3zDzDnrw`
- Exams: `https://ramcloud.stanford.edu/~ongaro/userstudy/quizzes.html`

9.1. Understandability

- How do you measure understandability?
- You teach two otherwise-identical classes on Paxos and Raft, and make kids take tests!
- Paxos lecture: `http://youtu.be/JEpsBg0AO6o`
- Raft lecture: `http://youtu.be/YbZ3zDzDnrw`
- Exams: `https://ramcloud.stanford.edu/~ongaro/userstudy/quizzes.html`

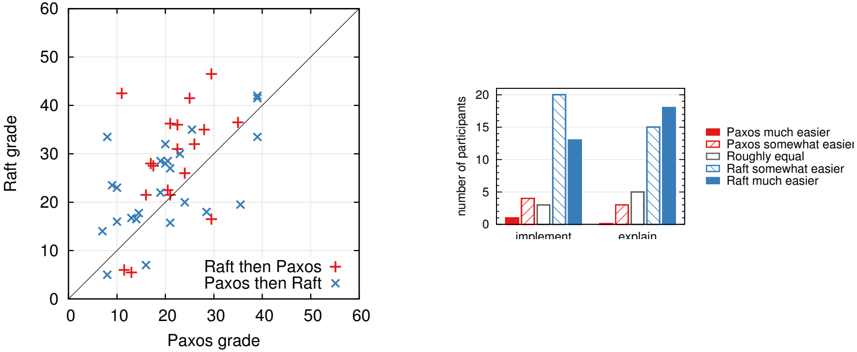
9.1. Understandability

- How do you measure understandability?
- You teach two otherwise-identical classes on Paxos and Raft, and make kids take tests!
- Paxos lecture: `http://youtu.be/JEpsBg0AO6o`
- Raft lecture: `http://youtu.be/YbZ3zDzDnrw`
- Exams: `https://ramcloud.stanford.edu/~ongaro/userstudy/quizzes.html`

9.1. Understandability

- How do you measure understandability?
- You teach two otherwise-identical classes on Paxos and Raft, and make kids take tests!
- Paxos lecture: `http://youtu.be/JEpsBg0AO6o`
- Raft lecture: `http://youtu.be/YbZ3zDzDnrw`
- Exams: `https://ramcloud.stanford.edu/~ongaro/userstudy/quizzes.html`

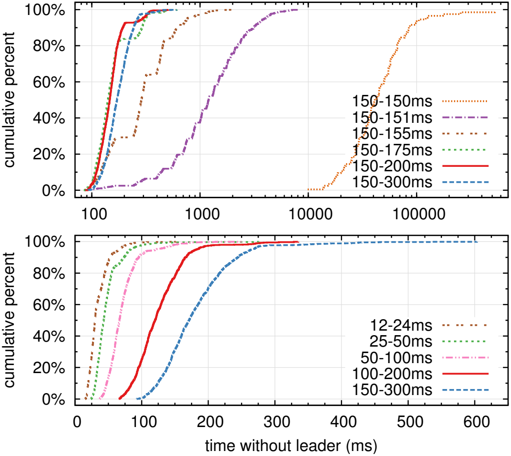
How’d they do?


Raft grade
60
50
40
30
20
10
0
0
10
20
30
40
50
60
Paxos grade
Raft then Paxos +
Paxos then Raft ×
number of participants
20
15
10
5
0
implement
explain
Paxos much easier
Paxos somewhat easier
Roughly equal
Raft somewhat easier
Raft much easier

9.2. Correctness

They wrote proofs! See citations.

9.3. Performance



Paper Conclusion

Categories of related work

- Paxos
- Implementations of Paxos
- Implementations of consensus systems (Chubby, ZooKeeper, Spanner, etc)
- Performance improvements for Paxos
- Viewstamped Replication - similar to Raft, about as old as Paxos.

Categories of related work

- Paxos
- Implementations of Paxos
- Implementations of consensus systems (Chubby, ZooKeeper, Spanner, etc)
- Performance improvements for Paxos
- Viewstamped Replication - similar to Raft, about as old as Paxos.

Categories of related work

- Paxos
- Implementations of Paxos
- Implementations of consensus systems (Chubby, ZooKeeper, Spanner, etc)
- Performance improvements for Paxos
- Viewstamped Replication - similar to Raft, about as old as Paxos.

Categories of related work

- Paxos
- Implementations of Paxos
- Implementations of consensus systems (Chubby, ZooKeeper, Spanner, etc)
- Performance improvements for Paxos
- Viewstamped Replication - similar to Raft, about as old as Paxos.

Categories of related work

- Paxos
- Implementations of Paxos
- Implementations of consensus systems (Chubby, ZooKeeper, Spanner, etc)
- Performance improvements for Paxos
- Viewstamped Replication - similar to Raft, about as old as Paxos.

Categories of related work

- Paxos
- Implementations of Paxos
- Implementations of consensus systems (Chubby, ZooKeeper, Spanner, etc)
- Performance improvements for Paxos
- Viewstamped Replication - similar to Raft, about as old as Paxos.

Comparisons

- Raft's leader-based approach is touted as better than Paxos (leadership in Paxos is only a performance optimization)
- VR and ZooKeeper are also leaderbased, but are more complicated (you can add log entries during elections, etc)
- Raft has less message-types in general.
- Egalitarian Paxos can be faster under certain conditions due to lack of leader.
- Cluster membership changes have a variety of approaches, but Raft's played to its own strengths.

Comparisons

- Raft’s leader-based approach is touted as better than Paxos (leadership in Paxos is only a performance optimization)
- VR and ZooKeeper are also leaderbased, but are more complicated (you can add log entries during elections, etc)
- Raft has less message-types in general.
- Egalitarian Paxos can be faster under certain conditions due to lack of leader.
- Cluster membership changes have a variety of approaches, but Raft’s played to its own strengths.

Comparisons

- Raft's leader-based approach is touted as better than Paxos (leadership in Paxos is only a performance optimization)
- VR and ZooKeeper are also leaderbased, but are more complicated (you can add log entries during elections, etc)
- Raft has less message-types in general.
- Egalitarian Paxos can be faster under certain conditions due to lack of leader.
- Cluster membership changes have a variety of approaches, but Raft's played to its own strengths.

Comparisons

- Raft's leader-based approach is touted as better than Paxos (leadership in Paxos is only a performance optimization)
- VR and ZooKeeper are also leaderbased, but are more complicated (you can add log entries during elections, etc)
- Raft has less message-types in general.
- Egalitarian Paxos can be faster under certain conditions due to lack of leader.
- Cluster membership changes have a variety of approaches, but Raft's played to its own strengths.

Comparisons

- Raft’s leader-based approach is touted as better than Paxos (leadership in Paxos is only a performance optimization)
- VR and ZooKeeper are also leaderbased, but are more complicated (you can add log entries during elections, etc)
- Raft has less message-types in general.
- Egalitarian Paxos can be faster under certain conditions due to lack of leader.
- Cluster membership changes have a variety of approaches, but Raft’s played to its own strengths.

Comparisons

- Raft's leader-based approach is touted as better than Paxos (leadership in Paxos is only a performance optimization)
- VR and ZooKeeper are also leaderbased, but are more complicated (you can add log entries during elections, etc)
- Raft has less message-types in general.
- Egalitarian Paxos can be faster under certain conditions due to lack of leader.
- Cluster membership changes have a variety of approaches, but Raft's played to its own strengths.

Paper Conclusion

4 Paper Conclusion

- 9. Implementation and evaluation
- 10. Related work
- 11, 12. Conclusion & Acknowledgements

Outline

Raft issues

Load balancing

- Every server must completely manage its own state machine.
- Every request must go through the leader.
- Doesn't horizontally scale well.

Load balancing

- Every server must completely manage its own state machine.
- Every request must go through the leader.
- Doesn't horizontally scale well.

Load balancing

- Every server must completely manage its own state machine.
- Every request must go through the leader.
- Doesn't horizontally scale well.

Load balancing

- Every server must completely manage its own state machine.
- Every request must go through the leader.
- Doesn’t horizontally scale well.

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they're on an insecure network.
- You don't trust some of your servers.
- You don't trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they’re on an insecure network.
- You don’t trust some of your servers.
- You don’t trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they're on an insecure network.
- You don't trust some of your servers.
- You don't trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they're on an insecure network.
- You don't trust some of your servers.
- You don't trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they’re on an insecure network.
- You don’t trust some of your servers.
- You don’t trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they’re on an insecure network.
- You don’t trust some of your servers.
- You don’t trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they’re on an insecure network.
- You don’t trust some of your servers.
- You don’t trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Byzantine failures

- You trust all your servers, but one gets hacked.
- You trust all your servers, but they’re on an insecure network.
- You don’t trust some of your servers.
- You don’t trust all of your servers.
- Bitcoin comparison?

Discussion

- How can you hack a Raft cluster?

Other problems?

- Can Raft be made to work in a distributed system when peers are constantly leaving and joining?
- Anything else?

Other problems?

- Can Raft be made to work in a distributed system when peers are constantly leaving and joining?
- Anything else?

Other problems?

- Can Raft be made to work in a distributed system when peers are constantly leaving and joining?
- Anything else?

Space
MONKEY

Space Monkey!

- Distributed Hash Tables
- Consensus algorithms
- Reed Solomon
- Monitoring and sooo much data
- Security and cryptography engineering

Space Monkey!

Come work with us!