In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University
(Draft of July 22, 2013)

Abstract

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

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [10, 11] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture is unsuitable for building practical systems, requiring complex changes to create an efficient and complete solution. As a result, both system builders and students struggle with Paxos.

After struggling with Paxos ourselves, we set out to find a new consensus algorithm that could provide a better foundation for system building and education. Our approach was unusual in that our primary goal was *understandability*: could we define a consensus algorithm and describe it in a way that is significantly easier to learn than Paxos, and that facilitates the development of intuitions that are essential for system builders? It was important not just for the algorithm to work, but for it to be obvious why it works. In addition, the algorithm needed to be complete enough to cover all the major issues required for an implementation.

The result of our effort is a consensus algorithm called Raft. Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [18]), but it has several novel aspects:

- **Design for understandability:** understandability was our most important criterion in evaluating design alternatives. We applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety so that they can be understood relatively independently) and state space reduction (Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other, in order to make it easier to reason about the system).
- **Strong leader:** Raft differs from other consensus algorithms in that it employs a strong form of leadership where only leaders (or would-be leaders) issue requests; other servers are completely passive. This makes Raft easier to understand and also simplifies the implementation.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a simple *joint consensus* approach where the majorities of two different configurations overlap during transitions.

We performed a user study with 43 graduate students at two universities to test our hypothesis that Raft is more understandable than Paxos. After learning both algorithms, students were able to answer questions about Raft 23% better than questions about Paxos.

We have implemented Raft in about 1500 lines of C++ code, and the implementation is used as part of RAMCloud [19]. We have also proven the correctness of the Raft algorithm.

The remainder of the paper introduces the replicated state machine problem (Section 2), discusses the strengths and weaknesses of Paxos (Section 3), describes our general approach to understandability (Section 4), presents the Raft consensus algorithm (Sections 5-7), evaluates Raft (Section 8), and discusses related work (Section 9).

2 Achieving fault-tolerance with replicated state machines

Consensus algorithms typically arise in the context of *replicated state machines* [21]. In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines are used to solve a variety of fault-tolerance problems in distributed systems. For example, large-scale systems that have a single cluster leader, such as GFS [4], HDFS [22], and RAMCloud [19], typically use a separate replicated state machine to manage leader election and store config-

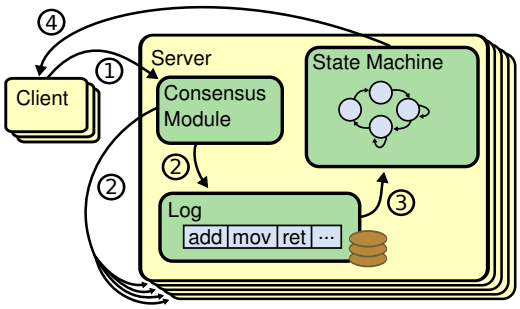

Figure 1: Replicated state machine architecture. A consensus module manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

uration information that must survive leader crashes. Examples of replicated state machines include Chubby [1] and ZooKeeper [7].

Replicated state machines are typically implemented using a replicated log, as shown in Figure 1. Each server stores a log containing a series of commands, which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Then, because the state machines are deterministic, each computes the same state and the same sequence of outputs.

Keeping the replicated log consistent is the job of the consensus algorithm. As shown in Figure 1, the consensus module on a server receives commands from clients and adds them to its log. By communicating with the consensus modules on other servers, it guarantees that every log will eventually have the same requests in the same order, even if some servers fail. Once commands are replicated safely, each server's state machine processes them in log order, and the outputs are returned to clients. For the system to be *safe* (meaning that it never behaves incorrectly), the consensus algorithm must ensure that each state machine executes exactly the same commands in the same order. This maintains the illusion that the servers form a single, highly-reliable state machine.

This paper is concerned with consensus algorithms for building practical systems. These algorithms typically have the following properties:

- They are fully functional (*available*) as long as any majority of the servers are operational and can communicate with each other and with clients. Thus, a typical cluster of five servers can tolerate the failure of any two servers. Servers are assumed to fail by stopping; they may later recover from state on stable storage and rejoin the cluster. (Other algorithms also handle Byzantine failures, but these algorithms are more complex and less efficient.)
- They are relatively efficient: in the common case, a command can complete as soon as any majority of the cluster has responded to a single round of remote procedure calls; a minority of slow servers need not impact overall system performance.
- Their safety is not affected by timing: faulty clocks and extreme message delays can, at worst, cause availability problems.

3 What's wrong with Paxos?

Over the last ten years, Leslie Lamport's Paxos protocol [10] has become almost synonymous with consensus: it is the protocol most commonly taught in courses, and most implementations of consensus use it as a starting point. Paxos first defines a protocol capable of reaching agreement on a single decision, such as a single replicated log entry. We refer to this subset as *single-decree Paxos*. Paxos then combines multiple instances of this protocol to facilitate a series of decisions such as a log (*multi-Paxos*). Paxos ensures both safety and liveness, and it supports changes in cluster membership. Its correctness has been proven, and it is efficient in the normal case.

Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [10] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [11, 13, 14]. These explanations focus on single-decree Paxos; they are still challenging, yet they leave the reader without enough information to build practical systems. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the original paper [10] until after reading several simplified explanations and designing our own alternative protocol, a process that took several months.

We hypothesize that Paxos' opaqueness derives from its choice of the single-decree subset as its foundation. Single-decree Paxos is dense and subtle: it is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. Because of this, it is difficult to develop intuitions about why the single-decree protocol works. The composition rules for multi-Paxos add significant additional complexity and subtlety. We believe that the overall problem of reaching consensus on multiple decisions (i.e., a log instead of a single entry) can be decomposed in other ways that are more direct and obvious.

The second problem with Paxos is that its architecture is a poor one for building practical systems; this too is a consequence of the single-decree decomposition. For example, there is no benefit to choosing a collection of log entries independently and then melding them into a sequential log; this just adds complexity. It is simpler and more efficient to design a system around a log, where new entries are appended sequentially in a constrained order.

Paxos also uses a symmetric peer-to-peer approach at its core (though it eventually suggests a weak form of leadership as a performance optimization). This makes sense in a simplified world where only one decision will be made, but few practical systems use this approach. If a series of decisions must be made, it is simpler and faster to first elect a leader, then have the leader coordinate the decisions.

As a result, practical systems bear little resemblance to Paxos. Each implementation begins with Paxos, discovers the difficulties in implementing it, and then develops a significantly different architecture. This is time-consuming and error-prone. The difficulties of understanding Paxos exacerbate the problem: system builders must modify the Paxos algorithm in major ways, yet Paxos does not provide them with the intuitions needed for this. Paxos' formulation may be a good one for proving theorems about its correctness, but real implementations are so different from Paxos that the proofs have little value. The following comment from the Chubby implementors is typical:

> 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 [2].

Because of these problems, we have concluded that Paxos does not provide a good foundation either for system building or for education. Given the importance of consensus in large-scale software systems, we decided to see if we could design an alternative consensus algorithm with better properties than Paxos. Raft is the result of that experiment.

4 Designing for understandability

We had several goals in designing Raft: it must provide a complete and appropriate foundation for system building, so that it reduces significantly the amount of design work required of developers; it must be safe under all conditions and available under typical operating conditions; and it must be efficient for common operations. But our most important goal—and most difficult challenge—was *understandability*. It must be possible for a large audience to understand the algorithm comfortably. In addition, it must be possible to develop intuitions about the algorithm, so that system builders can make the extensions that are inevitable in real-world implementations.

There were numerous points in the design of Raft where we had to choose among alternative approaches. In these situations we evaluated the alternatives based on understandability: how hard is it to explain each alternative (for example, how complex is its state space, and does it have subtle implications?), and how easy will it be for a reader to completely understand the approach and its implications? Given a choice between an alternative that was concise but subtle and one that was longer (either in lines of code or explanation) but more obvious, we chose the more obvious approach. Fortunately, in most cases the more obvious approach was also more concise.

We recognize that there is a high degree of subjectivity in such analysis; nonetheless, we used two techniques that we believe are generally applicable. The first technique is the well-known approach of problem decomposition: wherever possible, we divided problems into separate pieces that could be solved, explained, and understood relatively independently. For example, in Raft we separated leader election, log replication, and ensuring safety.

Our second approach was to simplify the state space by reducing the number of states to consider, making the system more coherent and eliminating nondeterminism where possible. For example, logs are not allowed to have holes, and Raft limits the ways in which logs can become inconsistent with each other. This approach conflicts with advice given by Lampson: "More nondeterminism is better, because it allows more implementations [13]." In our situation we needed only a single implementation, but it needed to be understandable; we found that reducing nondeterminism usually improved understandability. We suspect that trading off implementation flexibility for understandability makes sense for most system designs.

5 The Raft consensus algorithm

Raft uses a collection of servers communicating with remote procedure calls (RPCs) to implement a replicated log of the form described in Section 2. Figure 2 summarizes the algorithm in condensed form for reference; the components of Figure 2 are discussed piecewise over the rest of this section.

Raft implements consensus by first electing a distinguished *leader*, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. Having a leader simplifies the algorithm in several ways. For example, a leader can make decisions unilaterally without fear of conflicting decisions made elsewhere. In addition, Raft ensures that the leader's log is always "the truth;" logs can occasionally become inconsistent after crashes, but the leader resolves these situations by forcing the other servers' logs into agreement with its own. A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

Ensuring safety is critical to any consensus algorithm. In Raft clients only interact with the leader, so the only behavior they see is that of the leader; as a result, safety can be defined in terms of leaders. The Raft safety property is this: if a leader has applied a particular log entry to its state machine (in which case the results of that command could be visible to clients), then no other server may

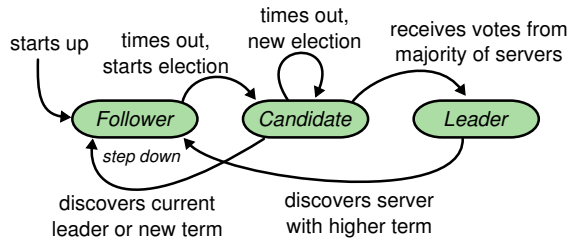

Figure 3: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

apply a different command for the same log entry. Raft has two interlocking policies that ensure safety. First, the leader decides when it is safe to apply a log entry to its state machine; such an entry is called *committed*. Second, the leader election process ensures that no server can be elected as leader unless its log contains all committed entries; this preserves the property that the leader's log is "the truth."

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems, which are discussed in the subsections that follow:

- **Leader election**: a new leader must be chosen when an existing leader fails, and Raft must guarantee that exactly one leader is chosen (Section 5.2).
- **Log replication**: the leader must accept log entries from clients and replicate them faithfully across the cluster, forcing all other logs to agree with its own (Section 5.3).
- **Safety**: Section 5.4 describes how Raft decides when a log entry has been committed, and how it ensures that leaders always hold all committed entries in their logs.

After presenting the consensus algorithm, this section discusses the issue of availability and the role of timing in the system.

5.1 Raft basics

A Raft cluster contains several servers (five is a typical number, which allows the system to tolerate two failures). At any given time each server is in one of three states: *leader*, *follower*, or *candidate*. In normal operation there is exactly one leader and all of the other servers are followers. Followers are passive: they issue no RPCs on their own but simply respond to RPCs from leaders and candidates. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). The third state, candidate, is used to elect a new leader as described in Section 5.2. Figure 3 shows the states and their transitions; the transitions are discussed in the subsections below.

Raft divides time into *terms* of arbitrary length, as shown in Figure 4. Terms are numbered with consecutive integers. Each term begins with an *election*, in which

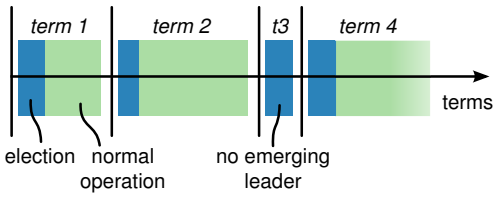

Figure 4: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader.

one or more candidates attempt to become leader as described in Section 5.2. If a candidate wins the election, then it serves as leader for the rest of the term. In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election) will begin shortly. Raft ensures that there is at most one leader in a given term.

Terms act as a logical clock [9] in Raft, and they allow Raft servers to detect obsolete information such as stale leaders. Each server stores a *current term* number, which increases monotonically over time. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other, then it updates its current term to the larger value. If a server receives a request with a stale term number, it rejects the request.

Raft uses only two remote procedure calls (RPCs) for communication between servers. RequestVote RPCs are initiated by candidates during elections (Section 5.2), and AppendEntries RPCs are initiated by leaders to replicate log entries and to provide a form of heartbeat (Section 5.3).

5.2 Leader election

Raft uses a heartbeat mechanism to trigger leader election. Almost all of the servers in Raft are in follower state at any given time, and when servers start up they begin as followers. A server remains in follower state as long as it receives valid RPCs from a leader or candidate. Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the *election timeout*, then it assumes there is no viable leader and begins an election to choose a new leader.

To begin an election, a follower increments its current term and transitions to candidate state. It then issues RequestVote RPCs in parallel to each of the other servers in the cluster. If the candidate receives no response for an RPC, it reissues the RPC repeatedly until a response arrives or the election concludes. A candidate continues in this state until one of three things happens: (a) it wins the election, (b) another server establishes itself as leader, or (c) a period of time goes by with no winner. These outcomes are discussed separately in the paragraphs below.

A candidate wins an election if it receives votes from a

Followers (§5.2)

- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates (§5.2)

- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
  - Election timeout elapses without election resolution: increment term, start new election
  - Discover higher term: step down (§5.1)

Leaders

- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts (§5.2)
- Accept commands from clients, append new entries to local log (§5.3)
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful (§5.3)
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry (§5.3)
- Mark entries committed if stored on a majority of servers and some entry from current term is stored on a majority of servers. Apply newly committed entries to state machine.
- Step down if currentTerm changes (§5.5)

Persistent State

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

currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries

Log Entry

term	term when entry was received by leader
index	position of entry in the log
command	command for state machine

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

Implementation:

1. Return if term < currentTerm (§5.1)
2. If term > currentTerm, currentTerm ← term (§5.1) (step down if candidate or leader)
3. If votedFor is null or candidateId, and candidate's log is at least as complete as local log (§5.2, §5.4), grant vote and reset election timeout

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)
commitIndex	last entry known to be committed

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return if term < currentTerm (§5.1)
2. If term > currentTerm, currentTerm ← term (§5.1)
3. If candidate (§5.2) or leader (§5.5), step down
4. Reset election timeout (§5.2)
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry(§5.3)
7. Append any new entries not already in the log
8. Apply newly committed entries to state machine (§5.3)

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes). Section numbers such as §5.2 indicate where particular features are discussed. The formal specification [20] describes the algorithm more precisely. Initialize nextIndex for each WHAT?

majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis (note: Section 5.4 adds an additional restriction on votes). The majority rule ensures that only one candidate can win. Once a candidate wins an election, it becomes leader. It then sends heartbeat messages to every other server to establish its authority and prevent new elections.

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader's term (included in its RPC) is at least as large as the candidate's current term, then the candidate recognizes the leader as legitimate and *steps down*, meaning that it returns to follower state. If the term in the RPC is older than the candidate's current term, then the candidate rejects the RPC and continues in candidate

state.

The third possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will start a new election by incrementing its term and initiating another round of RequestVote RPCs. However, without extra measures this process could repeat indefinitely without any candidate ever receiving a majority of the votes.

Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from an interval between a fixed minimum value and twice that value (currently 150-300ms in our implementation). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out. The same mechanism is used to handle split votes. Each candidate restarts its (randomized) election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election. Section 8 shows that this approach converges rapidly.

This election process is safe, meaning that there can be at most one leader in any given term. A candidate requires a majority of the votes from the same term in order to become leader, and each server votes for at most one candidate per term; therefore at most one candidate can acquire enough votes to become leader in a given term.

However, it is possible for multiple servers to believe they are the leader at the same time, if they were elected in different terms. Section 5.5 describes how Raft neutralizes all but the most recent leader.

Elections are an example of how understandability guided our choice between design alternatives. Initially we planned to use a ranking system: each candidate was assigned a unique rank, which was used to select between competing candidates. If a candidate discovered another candidate with higher rank, it would return to follower state so that the higher ranking candidate could more easily win the next election. We found that this approach created subtle issues around availability, particularly when combined with the safety issues discussed in Section 5.4. We made adjustments to the algorithm several times, but after each adjustment new corner cases appeared. Eventually we concluded that the randomized retry approach is more obvious and understandable.

5.3 Log replication

Once a leader has been elected, it begins servicing client requests. Each client request contains a command that must eventually be executed by the replicated state machines. The leader appends the command to its log as a new entry, then issues AppendEntries RPCs in parallel to each of the other servers to replicate the entry. When the leader decides that a log entry is *committed*, it applies the entry to its state machine and returns the result of that execution to the client. If followers crash or run slowly, or if network packets are lost, the leader retries AppendEntries RPCs indefinitely (even after it has responded to the client) until all followers eventually store all log entries.

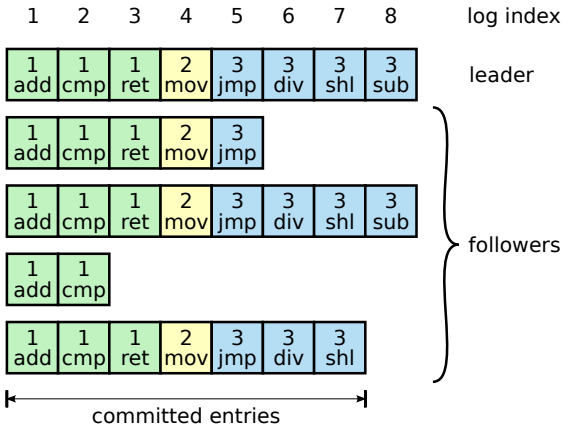

Figure 5: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

Logs are organized as shown in Figure 5. Each log entry stores a state machine command along with the term number when the entry was received by the leader. The term numbers in log entries are used to detect inconsistencies between logs and to ensure the Raft safety property as described in Section 5.4. Each log entry also has an integer index identifying its position in the log.

In the simple case of a leader replicating entries from its current term, a log entry is committed once it is stored on a majority of servers (e.g., entries 1-7 in Figure 5). Section 5.4 will extend this rule to handle other situations. The leader keeps track of the highest index known to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out. Once a follower learns that a log entry is committed, it applies the entry to its local state machine. If an entry is committed, Raft guarantees that it is durable and will eventually be executed by all of the replicated state machines.

We designed the Raft log mechanism to maintain a high level of coherency between the logs on different servers. Not only does this simplify the system's behavior and make it more predictable, but it is an important component of ensuring safety. Raft maintains the following properties at all times:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index

and term, then the logs will be identical in all preceding entries.

The first property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log.

The second property is guaranteed by a simple consistency check performed by AppendEntries. When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries. The follower will not add the entries to its log unless its log contains an entry with the same index and term. The consistency check acts as an induction step: the initial empty state of the logs satisfies the preceding properties, and the consistency check preserves the properties whenever logs are extended. As a result, whenever AppendEntries returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries.

During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails. However, leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all of the entries in its log). These inconsistencies can compound over a series of leader and follower crashes. Figure 6 illustrates the ways in which followers' logs may differ from that of a new leader. A follower may be missing entries that are present on the leader (a-b), it may have extra entries that are not present on the leader (c-d), or both (e-f). Missing and extraneous entries in a log may span multiple terms, but extraneous entries will always be the last entries in the log, and missing entries will always be after all other entries in the log (these properties are guaranteed by the AppendEntries consistency check).

In Raft, the leader's log is always "the truth," so the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten by values from the leader's log. Section 5.4 will show that this is safe.

To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point. All of these actions happen in response to the consistency check performed by AppendEntries RPCs. The leader maintains a *nextIndex* for each follower, which is the index of the next log entry the leader will send to that follower. When a leader first comes to power it initializes all nextIndex values to the index just after the last one in its log (11 in Figure 6). If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC. After a rejection, the leader decrements nextIndex and retries the

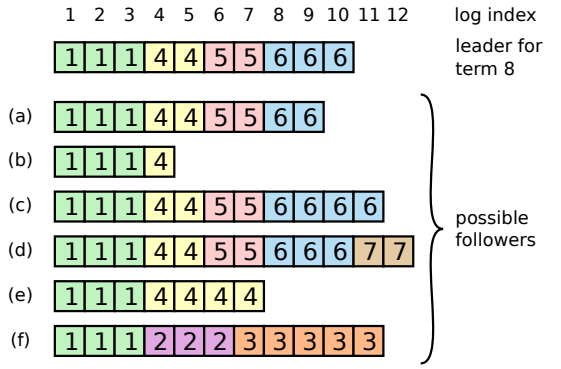

Figure 6: When the leader at the top comes to power, it is possible that any of scenarios (a-f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a-b), may have extra uncommitted entries (c-d), or both (e-f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log and crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms after that.

AppendEntries RPC. Eventually nextIndex will reach a point where the leader and follower logs match. When this happens, AppendEntries will succeed; it will remove any conflicting entries in the follower's log and append entries from the leader's log (if any). Once AppendEntries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term.

If desired, the protocol can be optimized to reduce the number of rejected AppendEntries RPCs. For example, when rejecting an AppendEntries request, the follower can include information about the term that contains the conflicting entry (term identifier and indexes of the first and last log entries for this term). With this information, the leader can decrement nextIndex to bypass all of the conflicting entries in that term; one AppendEntries RPC will be required for each term with conflicting entries, rather than one RPC per entry. In practice, we doubt this optimization is necessary, since failures happen infrequently and there are unlikely to be many inconsistent entries.

With this mechanism, a leader does not need to take any special actions when it comes to power to restore log consistency. It just begins normal operation and the logs automatically converge in response to failures of the AppendEntries consistency check.

This log replication mechanism exhibits the desirable consensus properties described in Section 2: Raft can accept, replicate, and apply new log entries as long as a majority of the servers are up; in the normal case a new entry can be replicated with a single round of RPCs to a majority of the cluster; and a single slow follower will not

impact performance.

5.4 Safety

For Raft to be safe, it must guarantee that if a leader has applied a particular log entry to its state machine, then no other server will apply a different command for the same log entry. Raft achieves this by ensuring a narrower property, which we call the *Leader Log Property*: once a log entry has been committed, that entry will be present in the logs of all future leaders. The overall safety property follows from the Leader Log Property because a server cannot apply a command to its state machine unless its log is identical to the leader's log up through that entry (the AppendEntries consistency check guarantees this).

Raft has two related policies that enforce the Leader Log Property: how to choose a leader during elections, and how to decide that a log entry is committed. This section describes the policies and how they work together to ensure that leaders always hold all committed entries.

Raft uses the election voting mechanism to select a new leader whose log is as "up-to-date" as possible. When requesting votes, a candidate includes information about its log in the RequestVote RPC; if the voter's log is more up-to-date than the candidate's, then the voter denies its vote. This guarantees that the winning candidate's log is at least as up-to-date as any log in the voting majority.

Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If two logs have last entries with different terms, then the log with the later term is more up-to-date. If two logs end with the same term, then whichever log is longer is more up-to-date (it is a strict superset of the other log).

Given these rules for choosing leaders, Raft uses a compatible policy for committing log entries: a log entry may only be considered committed if it is impossible for a server that does not store the entry to be elected leader. There are two interesting cases to consider, which are diagrammed in Figure 7. If a leader is replicating an entry from the current term, as in Figure 7(a), the entry is committed as soon as the leader can confirm that it is stored on a majority of the full cluster: at this point only the servers storing the entry can be elected as leader.

However, things are more complex if a leader crashes before committing a log entry, so that a future leader must finish the commitment process. For example, in Figure 7(b) the leader for term 4 is replicating log index 2, which was originally created, but not fully replicated, in term 2. In this case it is possible for another server to overwrite this entry: in Figure 7(b), S5 was elected leader for term 3 (with votes from S3 and S4). It created a new entry in its own log, but crashed before replicating that entry. In this situation, the leader for term 4 cannot consider log index 2 committed even if it is stored on a majority of the servers: S5 could still be elected leader and propagate its own value for index 2.

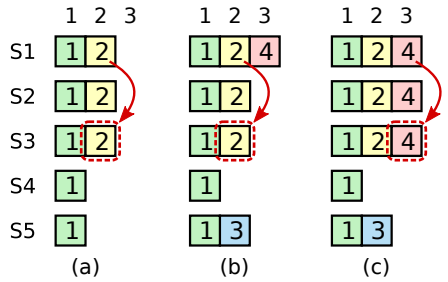

Figure 7: Scenarios for commitment. In each scenario S1 is leader and has just finished replicating a log entry to S3. In (a) the entry is from the leader's current term (2), so it is now committed. In (b) the leader for term 4 is replicating an entry from term 2; index 2 is not safely committed because S5 could become leader of term 5 (with votes from S2, S3, and S4) and overwrite the entry. Once the leader for term 4 has replicated an entry from term 4 in scenario (c), S5 cannot win an election so both indexes 2 and 3 are now committed.

To handle this situation, the full rule for commitment as as follows: **a log entry may only be considered committed if the entry is stored on a majority of the servers; in addition, at least one entry from the leader's current term must also be stored on a majority of the servers**. Figure 7(c) shows how this preserves the Leader Log Property: once the leader has replicated an entry from term 4 on a majority of the cluster, it is impossible for S5 to be elected leader.

Taken together, the rules for elections and commitment ensure that a newly elected leader stores all committed entries in its log. Leaders never overwrite entries in their logs, so the committed entries will be preserved throughout the leader's term. This ensures that the leader's log really is "the truth," which has been assumed by the other parts of the Raft algorithm.

5.5 Neutralizing deposed leaders

In Raft, it is possible for more than one server to act as leader at the same time. This can happen if a leader becomes temporarily disconnected from the rest of the cluster, triggering election of a new leader, then becomes reconnected after the election has completed. The deposed leader may continue to act as leader, accepting client requests and attempting to replicate them. These log entries could conflict with entries being replicated by the new leader.

Fortunately, Raft's term mechanism prevents deposed leaders from taking any actions that affect the safety of the consensus protocol. As described in Section 5.1, servers exchange their current terms in every RPC. Any server contacted by the new leader will store the leader's higher term number; this will include a majority of the cluster before the new leader wins an election. If a deposed leader sends an RPC to any of these servers, it will discover that its term is out of date, which will cause it to step down. In order for a deposed leader to commit a new log entry, it must communicate with a majority of the cluster, which

will include at least one server with a higher term. Thus, the deposed leader cannot commit new log entries.

However, it is possible for a deposed leader to finish committing an existing entry. For example, if one of the servers voting for the new leader received a log entry from the old leader before casting its vote, the deposed leader could finish committing that entry using servers that are not part of the electing majority. Fortunately, this situation is still safe. In order for the deposed leader to commit a log entry after the next election completes, the entry must be stored on at least one server that voted for the new leader. The election rules described in Section 5.4 guarantee that the new leader must be one of the servers that stores the entry, and it will commit the entry itself, so the old leader is simply assisting the new leader. Combining this paragraph with the previous one produces the following rule: once a new leader has been elected, no previous leader can commit log entries that conflict with those on the new leader.

5.6 Follower and candidate crashes

Until this point we have focused on leader failures. Follower and candidate crashes are much simpler to handle than leader crashes, and they are both handled in the same way. If a follower or candidate crashes, then future RequestVote and AppendEntries RPCs sent to it will fail. Raft handles these failures by retrying indefinitely; the server will eventually restart (as a follower) and the RPC will complete successfully. If a server crashes after completing an RPC but before responding, then it will receive the same RPC again after it restarts. Fortunately, Raft RPCs are idempotent so this causes no harm. For example, if a follower receives an AppendEntries request that includes log entries already present in its log, it ignores those entries in the new request.

5.7 Timing and availability

One of our requirements for Raft is that safety must not depend on timing: the system must not produce incorrect results just because some event happens more quickly or slowly than expected. However, availability (the ability of the system to respond to clients in a timely manner) is a different story: it must inevitably depend on timing. For example, if message exchanges take longer than the typical time between server crashes, candidates will not stay up long enough to win an election; without a steady leader, Raft cannot make progress.

Leader election is the aspect of Raft where timing is most critical. Raft will be able to elect and maintain a steady leader as long as the system satisfies the following *timing requirement*:

$$broadcastTime \ll electionTimeout \ll MTBF$$

In this inequality *broadcastTime* is the average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses; *electionTimeout* is the election timeout described in Section 5.2; and *MTBF* is the average time between failures for a single server. The broadcast time must be an order of magnitude less than the election timeout so that leaders can reliably send the heartbeat messages required to keep followers from starting elections; given the randomized approach used for election timeouts, this inequality also makes split votes unlikely. The election timeout must be a few orders of magnitude less than MTBF so that the system makes steady progress. When the leader crashes, the system will be unavailable for roughly the election timeout; we would like this to represent only a small fraction of overall time.

The broadcast time and MTBF are properties of the underlying system, while the election timeout is something we must choose. Raft's RPCs typically require the recipient to commit information to stable storage, so the broadcast time may range from 0.5ms to 20ms, depending on storage technology. As a result, the election timeout is likely to be somewhere between 10ms and 500ms. Typical server MTBFs are several months or more, which easily satisfies the timing requirement.

Raft will continue to function correctly even if the timing requirement is occasionally violated. For example, the system can tolerate short-lived networking glitches that make the broadcast time larger than the election timeout. If the timing requirement is violated over a significant period of time, then the cluster may become unavailable. Once the timing requirement is restored, the system will become available again.

6 Cluster membership changes

Up until now we have assumed that the cluster *configuration* (the set of servers participating in the consensus algorithm) is fixed. In practice, it will occasionally be necessary to change the configuration, for example to replace servers when they fail or to change the degree of replication. Although this can be done by taking the entire cluster off-line, updating configuration files, and then restarting the cluster, this will leave the cluster unavailable during the changeover. In addition, if there are any manual steps, they risk operator error. In order to avoid these issues, we decided to automate configuration changes and incorporate them into the Raft consensus algorithm.

The biggest challenge for configuration changes is to ensure safety: there must be no point during the transition where it is possible for two leaders to be elected simultaneously. Unfortunately, any approach where servers switch directly from the old configuration to the new configuration is unsafe. It isn't possible to atomically switch all of the servers at once, so there will be a period of time when some of the servers are using the old configuration while others have switched to the new configuration. For some configuration changes, such as the one shown in Figure 8, this can result in two independent majorities.

In order to ensure safety, configuration changes must use a two-phase approach. There are a variety of ways to

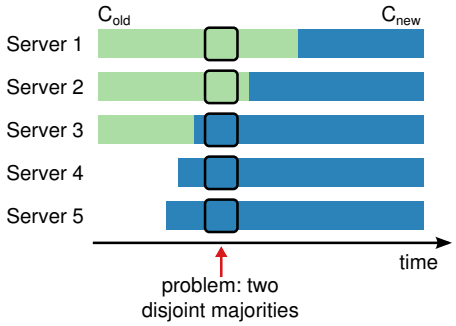

Figure 8: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected simultaneously, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

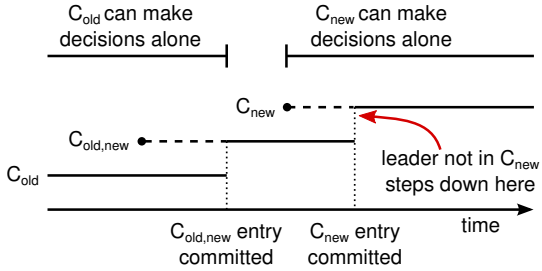

Figure 9: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a majority of C_{old} and a majority of C_{new}). Then it creates the C_{new} entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.

implement the two phases. For example, some systems (e.g. [15]) use the first phase to disable the old configuration so it cannot process client requests; then the second phase enables the new configuration. In Raft the cluster first switches to a transitional configuration we call *joint consensus*; once the joint consensus has been committed, the system then transitions to the new configuration. The joint consensus combines both the old and new configurations:

- Log entries are replicated to all servers in both configurations.
- Any server from either configuration may serve as leader.
- Agreement (for elections and entry commitment) requires majorities from *both* the old and new configurations.

As will be shown below, the joint consensus allows individual servers to transition between configurations at different times without compromising safety. Furthermore, joint consensus allows the cluster to continue servicing client requests throughout the configuration change.

Cluster configurations are stored and communicated using special entries in the replicated log; Figure 9 illustrates the configuration change process. When the leader receives a request to change the configuration from C_{old} to C_{new}, it stores the configuration for joint consensus ($C_{old,new}$ in the figure) as a log entry and replicates that entry using the mechanisms described previously. Once a given server adds the new configuration entry to its log, it uses that configuration for all future decisions (it does not wait for the entry to become committed). This means that the leader will use the rules of $C_{old,new}$ to determine when the log entry for $C_{old,new}$ is committed. If the leader crashes, a new leader may be chosen under either C_{old} or $C_{old,new}$, depending on whether the winning candidate has received $C_{old,new}$. In any case, C_{new} cannot make unilateral decisions during this period.

Once $C_{old,new}$ has been committed, neither C_{old} nor C_{new} can make decisions without approval of the other, and the Leader Log Property ensures that only servers with the $C_{old,new}$ log entry can be elected as leader. It is now safe for the leader to create a log entry describing C_{new} and replicate it to the cluster. Again, this configuration will take effect on each server as soon as it is seen. When the new configuration has been committed under the rules of C_{new}, the old configuration is irrelevant and servers not in the new configuration can be shut down. As shown in Figure 9, there is no time when C_{old} and C_{new} can both make unilateral decisions; this guarantees safety.

There are two more issues to address for reconfiguration. First, if the leader is part of C_{old} but not part of C_{new}, it must eventually step down. In Raft the leader steps down immediately after committing a configuration entry that does not include itself. This means that there will be a period of time (while it is committing C_{new}) where the leader is managing a cluster that does not include itself; it replicates log entries but does not count itself in majorities. The leader should not step down earlier, because members not in C_{new} could still be elected, resulting in unnecessary elections.

The second issue is that new servers may not initially store any log entries. If they are added to the cluster in this state, it could take quite a while for them to catch up, during which time it might not be possible to commit new log entries. In order to avoid availability gaps, Raft introduces an additional phase before the configuration change, in which the new servers join the cluster as non-voting members (the leader will replicate log entries to them, but they are not considered for majorities). Once the new servers' logs have caught up with the rest of the cluster, the reconfiguration can proceed as described above.

7 Clients and log compaction

This section describes how clients interact with Raft, including finding the cluster leader and supporting lin-

earizable semantics [6]. It also discusses the issue of log compaction (responsibility for which falls primarily with the state machine).

7.1 Client interaction

This text (and more) was commented out. Clients of Raft send all of their requests to the leader. When a client first starts up, it connects to a randomly-chosen server. If the client's first choice is not the leader, that server will reject the client's request and supply information about the most recent leader it has heard from (AppendEntries requests include the network address of the leader). If the leader crashes, client requests will time out; clients then try again with randomly-chosen servers.

Raft provides at-least-once semantics for clients, but the state machine can filter duplicate commands for exactly-once semantics. The problem arises when a client fails to get a response to a request, and it retries the request repeatedly until it eventually completes. If the first attempt was committed but the leader crashed before responding to the client, the retry will cause the state machine command to be duplicated in the log. So that the state machine can filter these duplicated commands, clients assign unique serial numbers to every command. Then, the state machine tracks the latest serial number processed for each client, along with its associated output. If it receives a command whose serial number has already been executed, it responds immediately without re-executing the request.

Read-only operations in Raft require extra precautions so that they do not return stale information. These operations are not serialized into the log, so if the leader has been deposed but does not yet know it, it could return information that was already stale when the request was sent (this violates linearizability [6]). Raft's solution is for the leader to reconfirm its leadership (for example, by sending heartbeat messages to a majority of the cluster) before responding to read-only requests. Alternatively, the leader could rely on the heartbeat mechanism to provide a form of lease [5], but this would rely on timing for safety (it assumes bounded clock skew).

7.2 Log compaction

To keep the log from growing without bound, the necessary information from the log must be extracted and those log entries discarded periodically. In Raft, the necessary information is just the current state machine state and the current cluster configuration (for cluster membership changes; see Section 6). Snapshotting is a common and simple approach used in Chubby and ZooKeeper: all current information is rewritten in a more compact *snapshot*, then the entire log up to that point is discarded. This is easily done in parallel with normal operation to avoid affecting availability.

Raft supports snapshotting as follows: at any time, a server may snapshot the committed prefix of its log, then discard this log prefix. Servers do this independently without coordination. Discarding log entries opens up the possibility that a leader may not be able to replicate a log entry to a follower that has fallen too far behind (if it has already discarded the log entry that follower needs next). Instead, the leader sends that follower its latest snapshot, along with the index and term of last log entry covered by the snapshot. Followers discard their logs and load in the snapshot upon receiving a current snapshot from a current leader. They use the last index and term of the snapshot for subsequent AppendEntries consistency checks.

8 Implementation and evaluation

We have implemented Raft as part of a replicated state machine that stores configuration information for RAMCloud [19] and assists in failover of the RAMCloud coordinator. This implementation includes the consensus mechanism described in Section 5 and the configuration change mechanism of Section 6, but not all of the features described in Section 7 (log compaction is not yet implemented). The Raft implementation contains roughly 1500 lines of C++ code, not including tests, comments, or blank lines. The source code is freely available [16].

The remainder of this section evaluates Raft using three criteria: understandability, correctness, and performance.

8.1 Understandability

To measure Raft's understandability, we conducted an experimental study using CS students at two universities. We recorded a video lecture of Raft and another of Paxos, and created corresponding quizzes. Upper-level undergraduate and graduate students from an Advanced Operating Systems course and a Distributed Computing course each watched one video, took the corresponding quiz, watched the second video, and took the second quiz. About half of the participants did the Paxos portion first and the other half did the Raft portion first in order to account for both individual differences in performance and experience gained from the first portion of the study. We compared participants' scores on each quiz to determine whether participants showed a better understanding of Raft.

We tried to make the comparison between Paxos and Raft as fair as possible. The experiment favored Paxos in

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

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

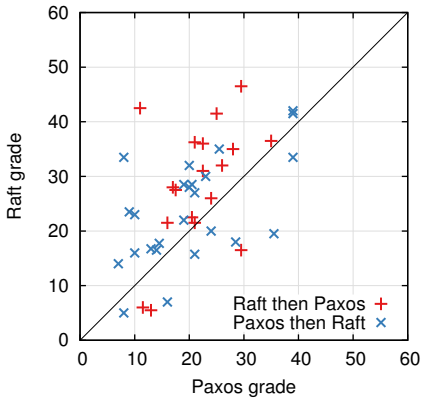

Figure 10: A scatter plot of 43 participants' grades comparing their performance on each exam. Points above the diagonal (33) represent participants who scored higher on the Raft exam.

two cases: 15 of the 43 participants reported having some prior experience with Paxos, and the Paxos video is 14% longer than the Raft video. As summarized in Table 1, we have taken steps to mitigate potential sources of bias and have made all of our materials available for review [17].

On average, participants scored 4.9 points higher on the Raft quiz than on the Paxos quiz (out of a possible 60 points, the mean Raft score was 25.7 and the mean Paxos score was 20.8); Figure 10 shows their individual scores. A paired t-test states that, with 95% confidence, the true distribution of Raft scores has a mean at least 2.5 points larger than the true distribution of Paxos scores. Accounting for whether people learn Paxos or Raft first and prior experience with Paxos, a linear regression model predicts scores 11.0 points higher on the Raft exam than on the Paxos exam (prior Paxos experience helps Paxos significantly and helps Raft slightly less). Curiously, the model also predicts scores 6.3 points lower on Raft for people that have already taken the Paxos quiz; although we don't know why, this does appear to be statistically significant.

We also surveyed participants after their quizzes to see which algorithm they felt would be easier to implement or explain; these results are shown in Figure 11. An overwhelming majority of participants reported Raft would be easier to implement and explain (33 of 41 for each question). However, these self-reported feelings may be less reliable than participants' quiz scores, and participants may have been biased by knowledge of our hypothesis that Raft is easier to understand.

8.2 Correctness

We have developed a formal specification and a proof of safety for the consensus mechanism described in Section 5. The formal specification [20] makes the information summarized in Figure 2 completely precise using the TLA+ specification language [12]. It is about 400 lines long. It specifies the actions each server may take, and the conditions which enable these actions. This is useful

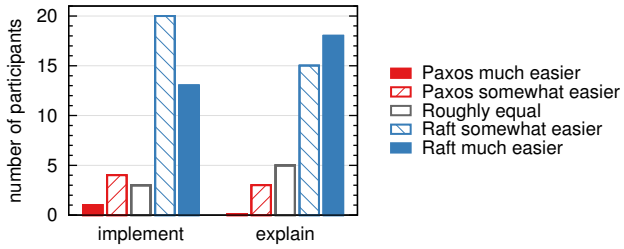

Figure 11: Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

on its own for anyone implementing Raft and also serves as the subject of the proof.

The safety proof shows invariants that hold in every state of every execution that the specification allows. Its main lemma states that a leader only marks a log entry committed if every subsequent leader must also have this entry. (Safety follows easily from this lemma.) We have an informal proof [20] for Raft which is relatively precise and complete (about 9 pages or 3500 words long). We also have a mechanically-checked version of the main lemma and other portions of the proof using the TLA proof system [3]. However, this version relies on invariants whose proofs have not been mechanically checked (for example, we have not proven the type safety of the specification).

8.3 Performance

The performance of Raft is equivalent to other consensus algorithms such as Paxos. The most important case for performance is where an established leader is replicating new log entries. Raft is optimal in this situation. The leader must complete a single round-trip RPC to any half of the followers in the cluster before responding to the client and passing the command to the local state machine (along with the leader, this constitutes a majority). No algorithm can ensure the basic properties of consensus without at least this many messages. If multiple clients are making requests simultaneously, the leader can optimize performance by batching several log entries in a single AppendEntries RPC or by issuing overlapping AppendEntries RPCs to the same follower. Importantly, these optimizations do not affect the basic properties of Raft.

We used the Raft implementation to measure the performance of Raft's leader election algorithm and answer two questions. First, does the election process converge quickly? Second, what is the minimum downtime that can be achieved after leader crashes?

To measure leader election, we repeatedly crashed the leader of a cluster of 5 servers and timed how long it took to detect the crash and elect a new leader (see Figure 12). To generate a worst-case scenario, the servers in each trial had different log lengths, so some candidates were not eligible to become leader. Furthermore, to encourage split

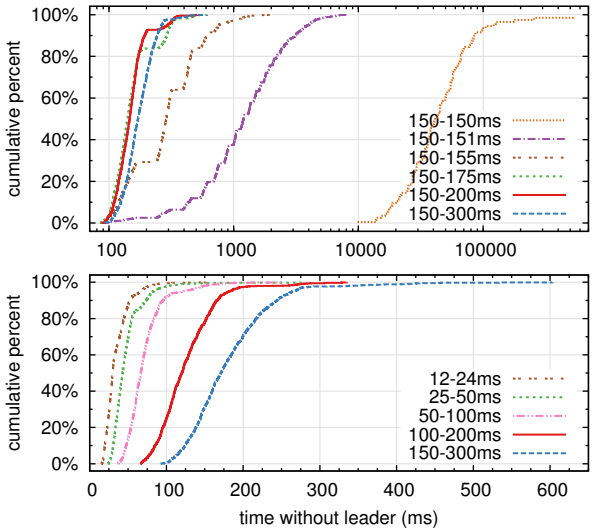

Figure 12: The time to detect and replace a crashed leader. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials (except for 100 trials for "150-150ms") and corresponds to a particular choice of election timeouts; for example, "150-155ms" means that election timeouts were chosen randomly and uniformly between 150ms and 155ms. The measurements were taken on a cluster of 5 servers with a broadcast time of roughly 15ms (results for a cluster of 9 servers are similar).

votes, our test script triggered a synchronized broadcast of heartbeat RPCs from the leader before terminating its process (this approximates the behavior of the leader replicating a new log entry prior to crashing). The leader was crashed uniformly randomly within its heartbeat interval, which was half of the minimum election timeout for all tests. Thus, the smallest possible downtime was about half of the minimum election timeout.

The top graph in Figure 12 shows that a small amount of randomization in the election timeout is enough to avoid split votes in elections. In the absence of randomness, leader election consistently took longer than 10 seconds in our tests, implying many split votes. Adding just 5ms of randomness helps significantly, resulting in a median downtime of 287ms. Using more randomness improves worst-case behavior: with 50ms of randomness the worst-case completion time (over 1000 trials) was 513ms.

The bottom graph in Figure 12 shows that downtime can be reduced by reducing the election timeout. With an election timeout of 12-24ms, it takes only 35ms on average to elect a leader (the longest trial took 152ms). However, lowering the timeouts further violates Raft's timing requirement: leaders have difficulty broadcasting heartbeats before other servers start new elections. This can cause unnecessary leader changes and lower overall system availability. We recommend using a conservative election timeout such as 150-300ms; such timeouts are unlikely to cause unnecessary leader changes and will still provide good availability.

9 Related Work

Raft is quite different from Paxos. As already discussed, Raft is based on a coherent log rather than a collection of relatively independent decisions, and it uses a strong form of leadership whereas Paxos is mostly peer-to-peer with a weak notion of leaders. These differences make Raft simpler than Paxos; for example, Raft can record the entire state of a log with two numbers (last log index and last log term), whereas Paxos must maintain independent state for each incomplete decision. Raft also uses a different mechanism than Paxos for changes in cluster membership; the Paxos mechanism will not work in Raft because it could result in deadlocks where no leader can be elected because the cluster is not up-to-date, and the cluster cannot be updated without a leader.

Although Raft is easier to understand than Paxos, it provides all the same benefits. Raft's replicated log is equivalent to multi-decree Paxos. Raft survives the same set of failures as Paxos (fail-stop crashes of any minority of the servers). And Raft has the same performance as Paxos: both systems require one round of RPCs to commit a new entry in the common case and both require three rounds of RPCs, starting from a cold start, to commit the first entry and notify the entire cluster of its commitment. Furthermore, an efficient implementation of Raft followed naturally from the algorithm described here; in contrast, Paxos implementations require numerous optimizations beyond the base algorithm to achieve efficiency and liveness [2].

Of existing consensus algorithms, the one most similar to Raft is Viewstamped Replication [18] (VR). VR was published before Paxos, but it has received much less attention (this may be because VR's consensus mechanism was intertwined with an implementation of distributed transactions). The VR algorithm was recently updated to focus on the consensus mechanism and eliminate distributed transactions [15]. The new version of VR was developed at the same time we were independently designing Raft.

Raft is similar in many ways to the updated version of VR. Both use a distinguished leader and both have distinct mechanisms for leader election and replication. The VR notion of *view* is similar to that of a term in Raft; their logs have similar structure and the systems use similar mechanisms for basic replication.

However, VR and Raft use different approaches for elections, restoring log consistency, and configuration changes. Changing leaders in VR is more complex: instead of electing a leader with a sufficiently up-to-date log as in Raft, VR must transfer the most up-to-date log to the new leader during view changes; it applies optimizations to reduce the transfer overhead. The largest difference is in the area of configuration changes. VR implements configuration changes with an additional protocol that re-

quires three new message types, additional state, and several modifications to the mechanisms for replication and leader election. Furthermore, the VR approach requires the system to stop processing client requests during configuration changes. Raft's mechanism is simpler because it piggybacks on the existing consensus mechanism; the only addition is the notion of joint consensus (checking two majorities instead of one). Raft can continue processing client requests during configuration changes.

VR contains several performance optimizations not present in Raft, such as the ability to operate without synchronous writes to disk, and a *witness* mechanism to reduce the load on followers. Raft does not currently include similar optimizations, but the VR mechanisms could be incorporated into Raft.

The two best-known implementations of replicated state machines are Chubby [1, 2] and ZooKeeper [7, 8]. Chubby's consensus algorithm is based on Paxos but appears to have some similarities to Raft, including a master replica (similar to a leader) and epochs (similar to terms); the details of its algorithm have not been published. ZooKeeper's consensus mechanism, Zab, has been published in more detail [8]. It uses epochs in a fashion similar to Raft terms, and it employs a leader. Zab is designed to operate across a broader state space than Raft (e.g., a new leader can be elected without storing all of the committed entries), but these generalizations add to its complexity. Zab does not include a mechanism for cluster reconfiguration.

10 Conclusion

Algorithms are often designed with correctness, efficiency, and/or conciseness as the primary goals. Although these are all worthy goals, we believe that understandability is just as important. None of the other goals can be achieved until developers render the algorithm into a practical implementation, which will inevitably deviate from and expand upon the published form. Unless developers have a deep understanding of the algorithm and can create intuitions about it, it will be difficult for them to retain its desirable properties in their implementation.

In this paper we addressed the issue of distributed consensus, where a widely accepted but impenetrable algorithm, Paxos, has challenged students and developers for many years. We developed a new algorithm, Raft, which we have shown to be more understandable than Paxos. We also believe that Raft provides a better foundation for system building. Furthermore, it achieves these benefits without sacrificing efficiency or correctness. Using understandability as the primary design goal changed the way we approached the design of Raft; as the design progressed we found ourselves reusing a few techniques repeatedly, such as decomposing the problem and simplifying the state space. We believe that these techniques not only improved the understandability of Raft but also made it easier to convince ourselves of its correctness.

References

[1] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[2] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[3] COUSINEAU, D., DOLIGEZ, D., LAMPORT, L., MERZ, S., RICKETTS, D., AND VANZETTO, H. TLA$^+$ proofs. In *FM* (2012), D. Giannakopoulou and D. Méry, Eds., vol. 7436 of *Lecture Notes in Computer Science*, Springer, pp. 147–154.

[4] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[5] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Ssymposium on Operating Systems Principles* (1989), pp. 202–210.

[6] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12* (July 1990), 463–492.

[7] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIX ATC '10, USENIX Association, pp. 11–11.

[8] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 245–256.

[9] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[10] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[11] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[12] LAMPORT, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] LAMPSON, B. W. How to build a highly available system using consensus. In *Distributed Algorithms*, O. Baboaglu and K. Marzullo, Eds. Springer-Verlag, 1996, pp. 1–17.

[14] LAMPSON, B. W. The abcd's of paxos. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2001), PODC 2001, ACM, pp. 13–13.

[15] LISKOV, B., AND COWLING, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

[16] LogCabin source code. `http://github.com/logcabin/logcabin`.

[17] Raft user study. `http://raftuserstudy.s3-website-us-west-1.amazonaws.com/study/`.

[18] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (New York, NY, USA, 1988), PODC '88, ACM, pp. 8–17.

[19] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Commun. ACM 54* (July 2011), 121–130.

[20] Safety proof and formal specification for Raft. `http://raftuserstudy.s3-website-us-west-1.amazonaws.com/proof.pdf`.

[21] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[22] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.