In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University
(Draft of October 7, 2013)

Abstract

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

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [14, 15] 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 this work is a consensus algorithm called Raft. In designing Raft we applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety) and state space reduction (Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, students were able to answer questions about Raft 23% better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [28, 21]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a novel *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. It is simpler and more understandable than other algorithms; it is described completely enough to meet the needs of a practical system; it has several open-source implementations; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

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 5-8), evaluates Raft (Section 9), and discusses related work (Section 10).

2 Achieving fault-tolerance with replicated state machines

Consensus algorithms typically arise in the context of *replicated state machines* [34]. 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 dis-

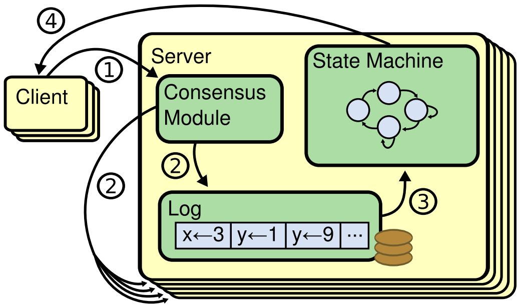


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

tributed systems. For example, large-scale systems that have a single cluster leader, such as GFS [7], HDFS [35], and RAMCloud [29], typically use a separate replicated state machine to manage leader election and store configuration information that must survive leader crashes. Examples of replicated state machines include Chubby [2] and ZooKeeper [10].

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. Since 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. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly-reliable state machine.

Consensus algorithms for practical systems typically have the following properties:

- They ensure *safety* (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering.
- 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.
- They do not depend on timing to ensure the consistency of the logs: faulty clocks and extreme message delays can, at worst, cause availability problems.
- 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.

3 What's wrong with Paxos?

Over the last ten years, Leslie Lamport's Paxos protocol [14] 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 [14] 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 [15, 19, 20]. These explanations focus on the single-decree subset, yet they are still challenging. 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 complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year.

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 it does not provide a good foundation for building practical implementations. One reason is that there is no widely agreed-upon algorithm for multi-Paxos. Lamport's descriptions are mostly about single-decree Paxos; he sketched possible approaches to multi-Paxos, but many details are missing. There have been several attempts to flesh out and optimize Paxos, such as [25], [36], and [12], but these differ from each other and from Lamport's sketches. Systems such as Chubby [4] have implemented Paxos-like algorithms, but in most cases their details have not been published.

Furthermore, the Paxos architecture is a poor one for building practical systems; this is another consequence of the single-decree decomposition. For example, there is little 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. Another problem is that Paxos 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 implementers 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 [4].

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 significantly reduces 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 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, safety, and membership changes.

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 [19]." 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, and Figure 3 lists key properties of the algorithm; the elements of these figures 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 management of the replicated log. For example, the leader can decide where to place new entries in the log without consulting other servers, and data flows in a simple fashion from the leader to other servers. A leader can fail or become disconnected from the other servers, in which case

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 timeout
  - 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).

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)

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)
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, last log index)

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 [33] describes the algorithm more precisely.

a new leader is elected.

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 (Section 5.2).
- **Log replication:** the leader must accept log entries from clients and replicate them across the cluster,

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

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

forcing the other logs to agree with its own (Section 5.3).

- **Safety:** the key safety property for Raft is the State Machine Safety Property in Figure 3: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index. Section 5.4 describes how Raft ensures this property; the solution involves slight extensions to the election and replication mechanisms described in Sections 5.2 and 5.3.

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 4 shows the states and their transitions; the transitions are discussed below.

Raft divides time into *terms* of arbitrary length, as shown in Figure 5. Terms are numbered with consecutive integers. Each term begins with an *election*, in which 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.

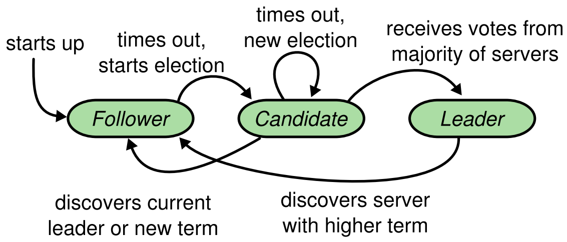


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

Different servers may observe the transitions between terms at different times, and in some situations a server may not observe an election or even entire terms. Terms act as a logical clock [13] 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 candidate or leader discovers that its term is out of date, it immediately reverts to follower state. If a server receives a request with a stale term number, it rejects the request.

Raft uses only two types of RPCs between servers for the basic consensus algorithm. 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). A third RPC is introduced in Section 7 for transferring snapshots between servers.

5.2 Leader election

Raft uses a heartbeat mechanism to trigger leader election. 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. 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 majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a

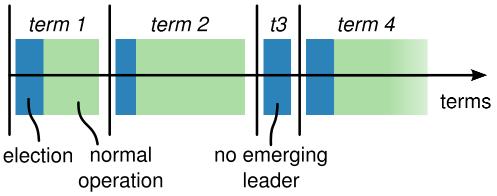


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The exact transitions may be observed at different times on different servers.

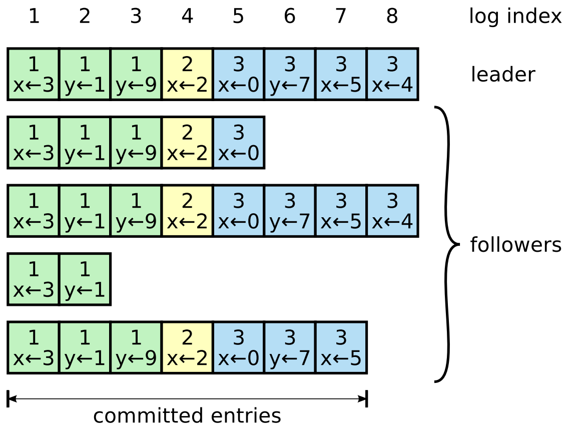
given term, on a first-come-first-served basis (note: Section 5.4 adds an additional restriction on votes). The majority rule ensures that at most one candidate can win the election for a particular term (the Election Safety Property in Figure 3). Once a candidate wins an election, it becomes leader. 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 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 time out and start a new election by incrementing its term and initiating another round of RequestVote RPCs. However, without extra measures split votes could repeat indefinitely.

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 a fixed interval (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 9.3 shows that this approach elects a leader rapidly.

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 extensions 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.



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

Figure 6: 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.

5.3 Log replication

Once a leader has been elected, it begins servicing client requests. Each client request contains a command to 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 entry has been safely replicated (as described below), the leader 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.

Logs are organized as shown in Figure 6. 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 some of the properties in Figure 3. Each log entry also has an integer index identifying its position in the log.

The leader decides when it is safe to apply a log entry to the state machines; such an entry is called *committed*. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines. 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 6). Section 5.4 will extend this rule to handle other situations. The leader keeps track of the highest index it knows 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 (in log order).

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, which together constitute the Log Matching Property in Figure 3:

- 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 are 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. If the follower does not find an entry in its log with the same index and term, then it refuses the new entries. The consistency check acts as an induction step: the initial empty state of the logs satisfies the Log Matching Property, and the consistency check preserves the Log Matching Property 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 7 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.

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

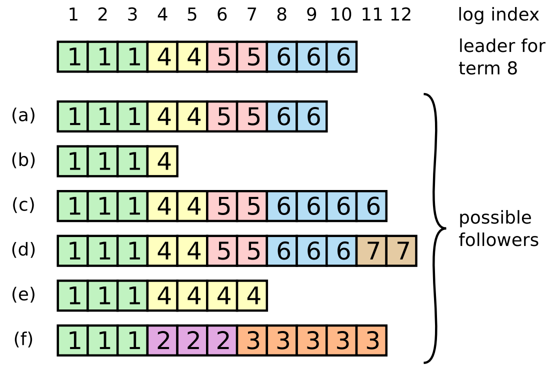


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a-f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a-b), may have extra uncommitted entries (c-d), or both (e-f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then 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.

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 7). 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 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 number and index of the first log entry 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 it is unlikely that there will be many inconsistent entries.

With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the AppendEntries consistency check. A leader never overwrites or deletes entries in its log (the Leader Append-Only Property in Figure 3).

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

The previous sections described how Raft elects leaders and replicates log entries. However, the mechanisms described so far are not quite sufficient to ensure that each state machine executes exactly the same commands in the same order. For example, a follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences.

This section completes the Raft algorithm with two extensions: it restricts which servers may be elected leader, and it restricts which entries are considered committed. Together, these restrictions ensure that the leader for any given term contains all of the entries committed in previous terms (the Leader Completeness Property from Figure 3). We then show how the Leader Completeness Property leads to correct behavior of the replicated state machine.

In any leader-based consensus algorithm, the leader must eventually store all of the committed log entries. In some consensus algorithms, such as Viewstamped Replication [21], a leader can be elected even if it doesn't initially contain all of the committed entries. These algorithms contain additional mechanisms to identify the missing entries and transmit them to the new leader, either during the election process or shortly afterwards. Unfortunately, this results in considerable additional mechanism and complexity. Raft uses a simpler approach where it guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader. This means that log entries only flow in one direction, from leaders to followers, and leaders never overwrite existing entries in their logs.

5.4.1 Election restriction

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries. A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate's log is at least as up-to-date as any other log in that majority (where "up-to-date" is defined precisely below), then it will hold all the committed entries. The RequestVote RPC implements this restriction: the RPC includes information about the candidate's log, and the voter denies its vote if its own log is more up-to-date than that of the candidate.

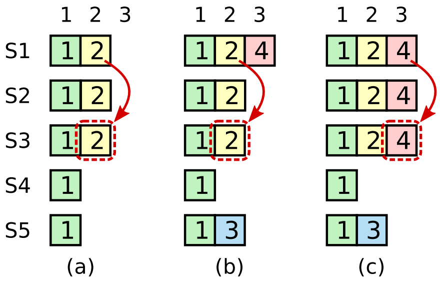


Figure 8: Scenarios for commitment. In each scenario S1 is leader and has just finished replicating a log entry to S3. In (a) the entry is from the leader's current term (2), so it is now committed. In (b) the leader for term 4 is replicating an entry from term 2; index 2 is not safely committed because 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.

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 the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

5.4.2 Restriction on commitment

We now explore whether the election restriction is sufficient to ensure the Leader Completeness Property. Consider the situations where a leader decides that a log entry is committed. There are two such situations, which are diagrammed in Figure 8. The most common case is where the leader replicates an entry from its current term (Figure 8(a)). In this case the entry is committed as soon as the leader confirms that it is stored on a majority of the full cluster. At this point only servers storing the entry can be elected as leader.

The second case for commitment is when a leader is committing an entry from an earlier term. This situation is illustrated in Figure 8(b). The leader for term 2 created an entry at log index 2 but replicated it only on S1 and S2 before crashing. S5 was elected leader for term 3 but was unaware of this entry (it received votes from itself, S3, and S4). Thus it created its own entry in log slot 2; then it crashed before replicating that entry. S1 was elected leader for term 4 (with votes from itself, S2 and S3). It then replicated its log index 2 on S3. In this situation, S1 cannot consider log index 2 committed even though it is stored on majority of the servers: S5 could still be elected

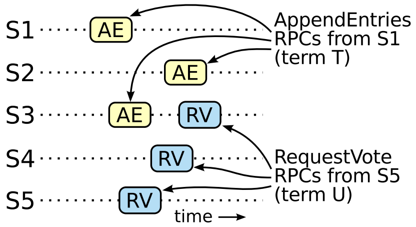

S1
S2
S3
S4
S5
AE
AE
AE
RV
RV
RV
AppendEntries RPCs from S1 (term T)
RequestVote RPCs from S5 (term U)
time →

Figure 9: Suppose that S1 (leader for term T) commits a new log entry from its term, but that entry is not stored by the leader for a later term U (S5). Then there must be at least one server (S3) that accepted the log entry and also voted for S5.

leader (since its log is more up-to-date than the logs of S2, S3, and S4) and propagate its own value for index 2.

Raft handles this situation with an additional restriction on committing log entries. A new leader may not conclude that any log entries are committed until it has committed an entry from its current term. Once this happens, all of the preceding entries in its log are also committed. Figure 8(c) shows how this preserves the Leader Completeness Property: once the leader has replicated an entry from term 4 on a majority of the cluster, the election rules prevent S5 from being elected leader.

5.4.3 Safety argument

Given the complete rules for commitment and election, we can now argue more precisely that the Leader Completeness Property holds (this argument is based on our safety proof; see Section 9.2). We assume that the Leader Completeness Property does not hold, then we prove a contradiction. Suppose the leader for term T (leader$_{\mathrm{T}}$) commits a log entry from its term, but that log entry is not stored by the leader of some future term. Consider the smallest term $\mathrm{U} > \mathrm{T}$ whose leader (leader$_{\mathrm{U}}$) does not store the entry.

1. The committed entry must have been absent from leader$_{\mathrm{U}}$'s log at the time of its election (leaders never delete or overwrite entries).
2. leader$_{\mathrm{T}}$ replicated the entry on a majority of the cluster, and leader$_{\mathrm{U}}$ received votes from a majority of the cluster. Thus, at least one server ("the voter") both accepted the entry from leader$_{\mathrm{T}}$ and voted for leader$_{\mathrm{U}}$, as shown in Figure 9. The voter is key to reaching a contradiction.
3. The voter must have accepted the committed entry from leader$_{\mathrm{T}}$ *before* voting for leader$_{\mathrm{U}}$; otherwise it would have rejected the AppendEntries request from leader$_{\mathrm{T}}$ (its current term would have been higher than T).
4. The voter still stored the entry when it voted for leader$_{\mathrm{U}}$, since every intervening leader contained the entry (by assumption), leaders never remove entries, and followers only remove entries if they conflict with the leader.
5. The voter granted its vote to leader$_{\mathrm{U}}$, so leader$_{\mathrm{U}}$'s log must have been as up-to-date as the voter's. This leads to one of two contradictions.
6. First, if the voter and leader$_{\mathrm{U}}$ shared the same last log term, then leader$_{\mathrm{U}}$'s log must have been at least as long as the voter's, so its log contained every entry in the voter's log. This is a contradiction, since the voter contained the committed entry and leader$_{\mathrm{U}}$ was assumed not to.
7. Otherwise, leader$_{\mathrm{U}}$'s last log term must have been larger than the voter's. Moreover, it was larger than T, since the voter's last log term was at least T (it contains the committed entry from term T). The earlier leader that created leader$_{\mathrm{U}}$'s last log entry must have contained the committed entry in its log (by assumption). Then, by the Log Matching Property, leader$_{\mathrm{U}}$'s log must also contain the committed entry, which is a contradiction.
8. This completes the contradiction. Thus, the leaders of all terms greater than T must contain all entries from term T that are committed in term T.
9. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly, such as index 2 in Figure 8(c).

Given the Leader Completeness Property, we can prove the State Machine Safety Property from Figure 3, which states that 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. At the time a server applies a log entry to its state machine, its log must be identical to the leader's log up through that entry and the leader must have decided the entry is committed. Now consider the lowest term in which any server applies a given log index; the Log Completeness Property guarantees that the leaders for all higher terms will store that same log entry, so servers that apply the index in later terms will apply the same value. Thus, the State Machine Safety Property holds.

Finally, Raft requires servers to apply entries in log index order. Combined with the State Machine Safety Property, this means that all servers will apply exactly the same set of log entries to their state machines, in the same order.

5.5 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.

5.6 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 should 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 should 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 persist 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

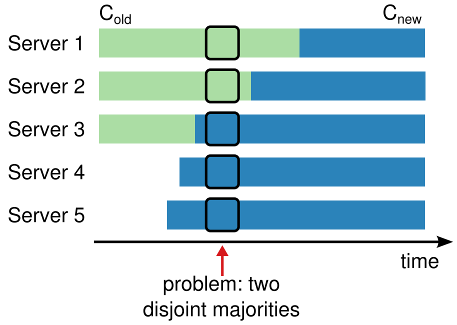


Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

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 would 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 for the same term. 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. As shown in Figure 10, 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 implement the two phases. For example, some systems (e.g., [21]) 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 configu-

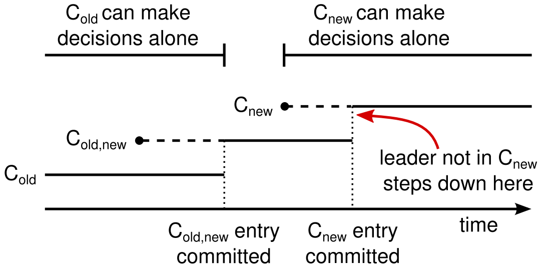

C_{old} can make decisions alone
C_{new} can make decisions alone
C_{new}
$C_{old,new}$
C_{old}
leader not in C_{new} steps down here
time
$C_{old,new}$ entry committed
C_{new} entry committed

Figure 11: 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.

rations.

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 11 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 (a server always uses the latest configuration in its log, regardless of whether the entry is 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 Completeness 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 11, there is no time when C_{old} and C_{new} can both make unilateral decisions; this guarantees safety.

There are three 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* (return to follower state). 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 have caught up with the rest of the cluster, the reconfiguration can proceed as described above.

The third issue is that servers that are removed from the cluster may still disrupt the cluster's availability. If these servers do not know that they have been removed, they can still start new elections. These elections cannot succeed, but they may cause servers in the new cluster to adopt larger term numbers, causing valid cluster leaders to step down. We are currently working on a solution to this problem.

7 Log compaction

In a practical system, the Raft log cannot grow without bound. As clients issue requests, the log grows longer, occupying more space and taking more time to replay. This will eventually cause availability problems without some mechanism to discard obsolete information that has accumulated in the log.

There are two basic approaches to compaction: log cleaning and snapshotting. Log cleaning [32] inspects log entries to determine whether they are *live*—whether they contribute to the current system state. Live entries are rewritten to the head of the log, then large consecutive regions of the log are freed. This process is incremental and efficient, but choosing which regions of the log to clean and determining which entries are live can be complex.

The second approach, snapshotting, operates on the current system state rather than on the log. In snapshotting, the entire current system state is written to a *snapshot* on stable storage, then the entire log up to that point is discarded. Compared to log cleaning, it is not incremental and less efficient (even information that has not changed since the last snapshot is rewritten). However, it is much simpler (for example, state machines need not track which log entries are live). Snapshotting is used in Chubby and ZooKeeper and is assumed for the remainder of this section.

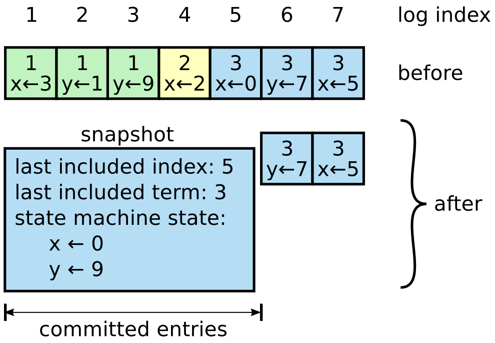


Figure 12: A server replaces the committed entries in its log (indexes 1 through 5) with a new snapshot, which stores just the current state (variables x and y in this example). The snapshot's last included index and term position the snapshot in the log preceding entry 6.

Figure 12 shows the basic idea of snapshotting in Raft. Each server takes snapshots independently, covering just the committed entries in its log. Most of the work consists of the state machine writing its current state to the snapshot. Raft also includes a small amount of metadata in the snapshot: the *last included index* is the index of the last entry in the log that the snapshot replaces (the last entry the state machine had applied), and the *last included term* is the term of this entry. These are preserved to support the AppendEntries consistency check for the first log entry following the snapshot, since that entry needs a previous log index and term. To enable cluster membership changes (Section 6), the snapshot also includes the latest configuration in the log as of last included index. Once a server completes writing a snapshot, it may delete all log entries up through the last included index, as well as any prior snapshot.

Although servers normally take snapshots independently, the leader must occasionally send snapshots to followers that lag behind. This happens when the leader has already discarded the next log entry that it needs to send to a follower. Fortunately, this situation is unlikely in normal operation: a follower that has kept up with the leader would already have this entry. However, an exceptionally slow follower or a new server joining the cluster (Section 6) would not. The way to bring such a follower up-to-date is for the leader to send it a snapshot over the network.

Our implementation uses a new RPC called InstallSnapshot for leaders to send snapshots to followers that are too far behind. Upon receiving a snapshot with this RPC, a follower must decide what to do with its existing log entries. It must remove any log entries that conflict with the snapshot (this is similar to the AppendEntries RPC). If the follower has an entry that matches the snapshot's last included index and term, then there is no conflict: it removes only the prefix of its log that the snapshot replaces. Otherwise, the follower removes its entire log; it is all superseded by the snapshot.

This snapshotting approach departs from Raft's strong leader principle, since followers can take snapshots without the knowledge of the leader. We considered an alternative leader-based approach in which only the leader would create a snapshot, then it would send this snapshot to each of its followers. However, this has two disadvantages. First, sending the snapshot to each follower would waste network bandwidth and slow the snapshotting process. Each follower already has the information needed to produce its own snapshots, and it is typically much cheaper for a server to produce a snapshot from its local state than it is to send and receive one over the network. Second, the leader's implementation would be more complex. For example, the leader would need to send snapshots to followers in parallel with replicating new log entries to them, so as not to block new client requests.

There are two more issues that impact snapshotting performance. First, servers must decide when to snapshot. If a server snapshots too often, it wastes disk bandwidth and energy; if it snapshots too infrequently, it risks exhausting its storage capacity, and it increases the time required to replay the log during restarts. One simple strategy is to take a snapshot when the log reaches a fixed size in bytes. If this size is set to be significantly larger than the expected size of a snapshot, then the disk bandwidth overhead for snapshotting will be small.

The second performance issue is that writing a snapshot can take a significant amount of time, and we do not want this to delay normal operations. The solution is to use copy-on-write techniques so that new updates can be accepted without impacting the snapshot being written. For example, state machines built with functional data structures naturally support this. Alternatively, the operating system's copy-on-write support (e.g., fork on Linux) can be used to create an in-memory snapshot of the entire state machine (our implementation uses this approach).

8 Client interaction

This section describes how clients interact with Raft, including finding the cluster leader and supporting linearizable semantics [9]. These issues apply to all consensus-based systems, and solutions are typically handled in similar ways.

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.

Our goal for Raft is to implement linearizable semantics (each operation appears to execute instantaneously, exactly once, at some point between its invocation and its response). However, as described so far Raft can execute a command multiple times: for example, if the leader crashes after committing the log entry but before responding to the client, the client will retry the command with a new leader, causing it to be executed a second time. The solution is for clients to assign unique serial numbers to every command. Then, the state machine tracks the latest serial number processed for each client, along with the associated response. If it receives a command whose serial number has already been executed, it responds immediately without re-executing the request.

Read-only operations can be made linearizable in several ways. One approach is to serialize them into the log just like other client requests, but this is relatively inefficient and not strictly necessary. Raft handles read-only requests without involving the log, but it must take two extra precautions to avoid returning stale information. First, a leader must have the latest information on which entries are committed. The Leader Completeness Property guarantees that a leader has all committed entries, but at the start of its term, it may not know which those are. To find out, it needs to commit an entry from its term. Raft handles this by having each leader commit a blank *no-op* entry into the log at the start of its term. Second, a leader must check whether it has been deposed before processing a read-only request (its information may be stale if a more recent leader has been elected). Raft handles this by having the leader exchange heartbeat messages with 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 [8], but this would rely on timing for safety (it assumes bounded clock skew).

9 Implementation and evaluation

We have implemented Raft as part of a replicated state machine that stores configuration information for RAMCloud [29] and assists in failover of the RAMCloud coordinator. The Raft implementation contains roughly 2000 lines of C++ code, not including tests, comments, or blank lines. The source code is freely available [22]. There are also about 25 other open source implementations [30] of Raft in various stages of development, based on drafts of this paper.

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

9.1 Understandability

To measure Raft's understandability relative to Paxos, we conducted an experimental study using upper-level undergraduate and graduate students in an Advanced Operating Systems course at Stanford University and a Distributed Computing course at U.C. Berkeley. We recorded a video lecture of Raft and another of Paxos, and created corresponding quizzes. The Raft lecture covered the content of this paper except for log compaction; the Paxos lecture covered enough material to create an equivalent replicated state machine, including single-decree Paxos, multi-decree Paxos, reconfiguration, and a few optimizations needed in practice (such as leader election). The quizzes tested basic understanding of the algorithms and also required students to reason about corner cases. Each student watched one video, took the corresponding quiz, watched the second video, and took the second quiz. About half of the participants did the Paxos portion first and the other half did the Raft portion first in order to account for both individual differences in performance and experience gained from the first portion of the study. We compared participants' scores on each quiz to determine whether participants showed a better understanding of Raft.

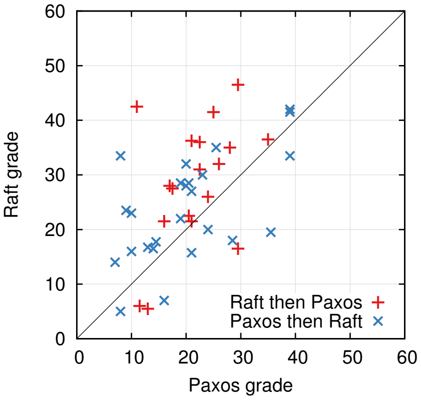


Figure 13: A scatter plot of 43 participants' grades comparing their performance on each exam. Points above the diagonal (33) represent participants who scored higher on the Raft exam.

We tried to make the comparison between Paxos and Raft as fair as possible. The experiment favored Paxos in 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. All of our materials are available for review [27].

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 13 shows their individual scores. A paired *t*-test states that, with 95% confidence, the true

Concern	Steps taken to mitigate bias	Materials for review [27]
Equal lecture quality	Same lecturer for both. Paxos lecture based on and improved from existing materials used in several universities. Paxos lecture is 14% longer.	videos
Equal quiz difficulty	Questions grouped in difficulty and paired across exams.	quizzes
Fair grading	Used rubric. Graded in random order, alternating between quizzes.	rubric

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

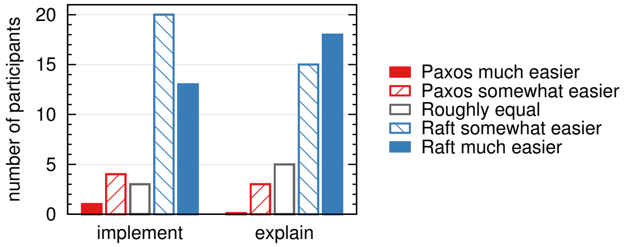


Figure 14: Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

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 14. 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.

9.2 Correctness

We have developed a formal specification and a proof of safety for the consensus mechanism described in Section 5. The formal specification [33] makes the information summarized in Figure 2 completely precise using the TLA+ specification language [16]. It is about 400 lines long and serves as the subject of the proof. It is also useful on its own for anyone implementing Raft. We have mechanically proven the Log Completeness Property using the TLA proof system [6]. However, this proof relies on invariants that have not been mechanically checked (for example, we have not proven the type safety of the specification).

Furthermore, we have written an informal proof [33] of the State Machine Safety property which is complete (it relies on the specification alone) and relatively precise (it is about 9 pages or 3500 words long).

9.3 Performance

Raft's performance is similar to other consensus algorithms such as Paxos. The most important case for performance is when an established leader is replicating new log entries. Raft achieves this using the minimal number of messages (a single round-trip from the leader to half the cluster). It is also possible to further improve Raft's performance. For example, it easily supports batching and pipelining requests for higher throughput and lower latency. Various optimizations have been proposed in the literature for other algorithms; many of these could be applied to Raft, but we leave this to future work.

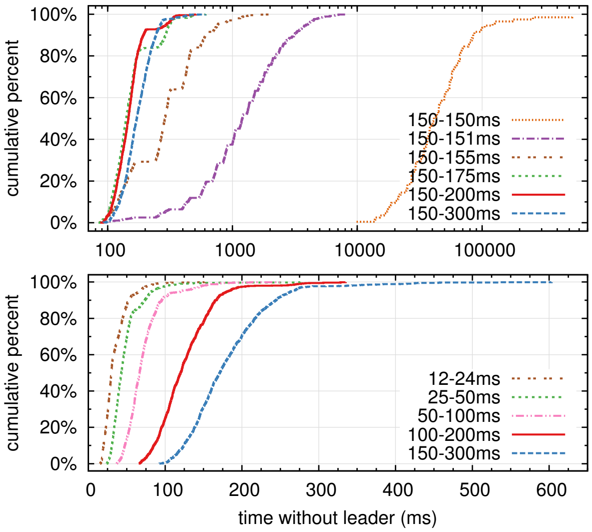


Figure 15: The time to detect and replace a crashed leader. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials (except for 100 trials for "150-150ms") and corresponds to a particular choice of 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.

We used our 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 15). 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 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 15 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 due to 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 15 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 beyond this point 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.

10 Related work

There have been numerous publications related to consensus algorithms, many of which fall into one of the following categories:

- Lamport's original description of Paxos [14], and attempts to explain it more clearly [15, 19, 20].
- Elaborations of Paxos, which fill in missing details and modify the algorithm to provide a better foundation for implementation [25, 36, 12].
- Systems that implement consensus algorithms, such as Chubby [2, 4], ZooKeeper [10, 11], and Spanner [5]. The algorithms for Chubby and Spanner have not been published in detail, though both claim to be based on Paxos. ZooKeeper's algorithm has been published in more detail, but it is quite different from Paxos.
- Performance optimizations that can be applied to Paxos [17, 18, 3, 24, 1, 26].
- Oki and Liskov's Viewstamped Replication (VR), an alternative approach to consensus developed around the same time as Paxos. The original description [28] was intertwined with a protocol for distributed transactions, but the core consensus protocol has been separated in a recent update [21]. VR uses a leader-based approach with many similarities to Raft.

The greatest difference between Raft and other consensus algorithms is Raft's strong leadership: Raft uses leader election as an essential part of the consensus protocol, and it concentrates as much functionality as possible in the leader. This approach results in a simpler algorithm that is easier to understand. For example, in Paxos, leader election is orthogonal to the basic consensus protocol: it serves only as a performance optimization and is not required for achieving consensus. However, this results in additional mechanism: Paxos includes both a two-phase protocol for basic consensus and a separate mechanism for leader election. In contrast, Raft incorporates leader election directly into the consensus algorithm and uses it as the first of the two phases of consensus. This results in less mechanism than in Paxos.

Raft also has less mechanism than VR or ZooKeeper, even though both of those systems are also leader-based. The reason for this is that Raft minimizes the functionality in non-leaders. For example, in Raft, log entries flow in only one direction: outward from the leader in AppendEntries RPCs. In VR log entries flow in both directions (leaders can receive log entries during the election process); this results in additional mechanism and complexity. The published description of ZooKeeper also transfers log entries both to and from the leader, but the implementation is apparently more like Raft [31]. Raft has fewer message types than any other algorithm for consensus-based log replication that we are aware of.

Several different approaches for cluster membership changes have been proposed or implemented in other work, including Lamport's original proposal [14], VR [21], and SMART [23]. We chose the joint consensus approach for Raft because it leverages the rest of the consensus protocol, so that very little additional mechanism is required for membership changes. Lamport's α-based approach was not an option for Raft because it assumes consensus can be reached without a leader. In comparison to VR and SMART, Raft's reconfiguration algorithm has the advantage that membership changes can occur without limiting the processing of normal requests; in contrast, VR must stop all normal processing during configuration changes, and SMART imposes an α-like limit on the number of outstanding requests. Raft's approach also adds less mechanism than either VR or SMART.

11 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. These techniques not only improved the understandability of Raft but also made it easier to convince ourselves of its correctness.

12 Acknowledgments

The user study would not have been possible without the support of Ali Ghodsi, David Mazières, and the students of CS 294-91 at Berkeley and CS 240 at Stanford. Scott Klemmer helped us design the user study, and Nelson Ray advised us on statistical analysis. The Paxos slides for the user study borrowed heavily from a slide deck originally created by Lorenzo Alvisi. A special thanks goes to David Mazières for finding the last (we hope!) and most subtle bug in Raft. Many people provided helpful feedback on the paper and user study materials, including Ed Bugnion, Michael Chan, Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, Robbert van Renesse, Mendel Rosenblum, Nicolas Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia, and anonymous conference reviewers. Werner Vogels tweeted a link to an earlier draft, which gave Raft significant exposure. This work was supported by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the National Science Foundation under Grant No. 0963859, and by grants from Facebook, Google, Mellanox, NEC, NetApp, SAP, and Samsung. Diego Ongaro is supported by The Junglee Corporation Stanford Graduate Fellowship.

References

[1] Bolosky, W. J., Bradshaw, D., Haagens, R. B., Kusters, N. P., and Li, P. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 11–11.

[2] 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.

[3] Camargos, L. J., Schmidt, R. M., and Pedone, F. Multicoordinated paxos. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 316–317.

[4] 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.

[5] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[6] 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.

[7] 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.

[8] 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.

[9] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12* (July 1990), 463–492.

[10] 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.

[11] 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.

[12] Kirsch, J., and Amir, Y. Paxos for system builders, 2008.

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

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

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

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

[17] Lamport, L. Generalized consensus and paxos. `http://research.microsoft.com/apps/pubs/default.aspx?id=64631`, 2005.

[18] Lamport, L. Fast paxos. `http://research.microsoft.com/apps/pubs/default.aspx?id=64624`, 2006.

[19] 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.

[20] 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.

[21] Liskov, B., and Cowling, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

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

[23] Lorch, J. R., Adya, A., Bolosky, W. J., Chaiken, R., Douceur, J. R., and Howell, J. The smart way to migrate replicated stateful services. In *Proceedings of the 1st*

ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (New York, NY, USA, 2006), EuroSys '06, ACM, pp. 103–115.

[24] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 369–384.

[25] MAZIÈRES, D. Paxos made practical. Jan. 2007.

[26] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating System Principles* (New York, NY, USA, 2013), SOSP 2013, ACM.

[27] Raft user study. `http://ramcloud.stanford.edu/~ongaro/userstudy/`.

[28] 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.

[29] 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.

[30] Raft implementations. `https://ramcloud.stanford.edu/wiki/display/logcabin/LogCabin`.

[31] REED, B. Personal communications, May 17, 2013.

[32] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10* (February 1992), 26–52.

[33] Safety proof and formal specification for Raft. `http://ramcloud.stanford.edu/~ongaro/raftproof.pdf`.

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

[35] 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.

[36] VAN RENESSE, R. Paxos made moderately complex. Tech. rep., Cornell University, 2012.