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

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

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

Abstract

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

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

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

CCS Concepts: • **Computer systems organization → Reliability**; • **Software and its engineering → Cloud computing**; • **Theory of computation → Distributed algorithms**.

Keywords: State machine replication, Distributed consensus, Paxos, Raft

PaPoC '20, April 27, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-7524-5/20/04…$15.00
https://doi.org/10.1145/3380787.3393681

ACM Reference Format:
Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: Have we reached consensus on distributed consensus? . In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3380787.3393681

1 Introduction

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

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

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

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

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

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

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

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

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

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

We conclude that there is no significant difference in understandability between the algorithms, and that Raft's leader election is surprisingly efficient given its simplicity.

2 Background

This paper examines distributed consensus in the context of state machine replication. State machine replication requires that an application's deterministic state machine is replicated across n servers with each applying the same set of operations in the same order. This is achieved using a replication log, managed by a distributed consensus algorithm, typically Paxos or Raft.

We assume that the system is *non-Byzantine* [21] but we do not assume that the system is synchronous. Messages may be arbitrarily delayed and participating servers may operate at any speed, but we assume message exchange is reliable and in-order (e.g., through use of TCP/IP). We do not depend upon clock synchronisation for safety, though we must for liveness [11]. We assume each of the n servers has an unique id s where $s \in \{0..(n-1)\}$. We assume that operations are unique, easily achieved by adding a pair of sequence number and server id to each operation.

3 Approach of Paxos & Raft

Many consensus algorithms, including Paxos and Raft, use a leader-based approach to solve distributed consensus. At a high-level, these algorithms operate as follows:

One of the n servers is designated the *leader*. All operations for the state machine are sent to the leader. The leader appends the operation to their log and asks the other servers to do the same. Once the leader has received acknowledgements from a majority of servers that this has taken place, it applies the operation to its state machine. This process repeats until the leader fails. When the leader fails, another server takes over as leader. This process of electing a new leader involves at least a majority of servers, ensuring that the new leader will not overwrite any previously applied operations.

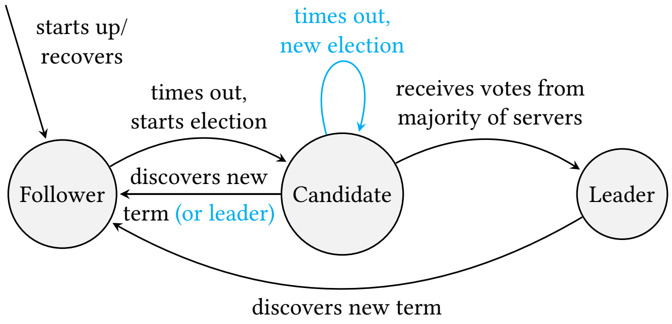


Figure 1. State transitions between the server states for Paxos & Raft. The transitions in blue are specific to Raft.

We now examine Paxos and Raft in more detail. Readers may find it helpful to refer to the summaries of Paxos and Raft provided in Appendices A & B. We focus here on the core elements of Paxos and Raft and, due to space constraints, do not compare garbage collection, log compaction, read operations or reconfiguration algorithms.

3.1 Basics

As shown in Figure 1, at any time a server can be in one of three states:

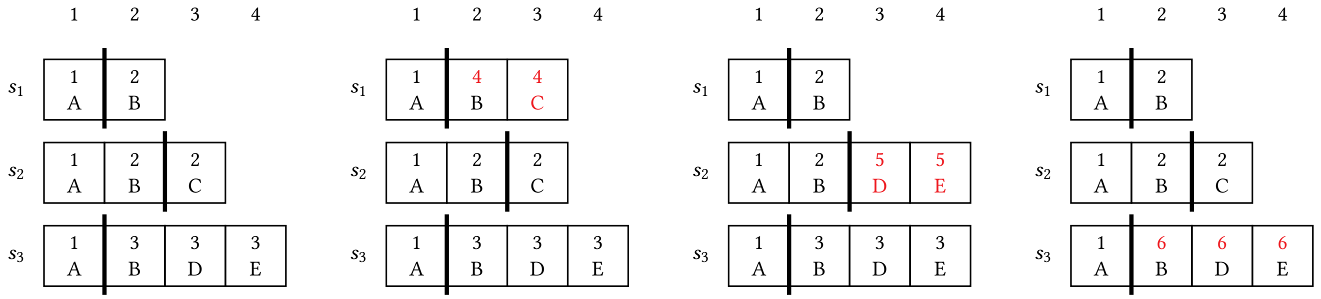
Follower A passive state where it is responsible only for replying to RPCs.

Candidate An active state where it is trying to become a leader using the *RequestVotes* RPC.

Leader An active state where it is responsible for adding operations to the replicated log using the AppendEntries RPC.

Initially, servers are in the *follower* state. Each server continues as a follower until it believes that the leader has failed. The follower then becomes a *candidate* and tries to be elected leader using RequestVote RPCs. If successful, the candidate becomes a leader. The new leader must regularly send AppendEntries RPCs as keepalives to prevent followers from timing out and becoming candidates.

Each server stores a natural number, the *term*, which increases monotonically over time. Initially, each server has a current term of zero. The sending server's (hereafter, the sender) current term is included in each RPC. When a server receives an RPC, it (hereafter, the server) first checks the included term. If the sender's term is greater than the server's, then the server will update its term before responding to the RPC and, if the server was either a candidate or a leader, step



(a) Initial state of logs before electing a new leader.

(b) s_1 is leader in term 4 after a vote from s_2.

(c) s_2 is leader in term 5 after a vote from s_3.

(d) s_3 is leader in term 6 after a vote from s_1 or s_2.

Figure 2. Logs of three servers running Paxos. Figure (a) shows the logs when a leader election was triggered. Figures (b—d) show the logs after a leader has been elected but before it has sent its first AppendEntries RPC. The black line shows the commit index and red text highlights the log changes.

down to become a follower. If the sender's term is equal to that of the server, then the server will respond to the RPC as usual. If the sender's term is less than that of the server, then the server will respond negatively to the sender, including its term in the response. When the sender receives such a response, it will step down to follower and update its term.

3.2 Normal operation

When a leader receives an operation, it appends it to the end of its log with the current term. The pair of operation and term are known as a *log entry*. The leader then sends AppendEntries RPCs to all other servers with the new log entry. Each server maintains a commit index to record which log entries are safe to apply to its state machine, and responds to the leader acknowledging successful receipt of the new log entry. Once the leader receives positive responses from a majority of servers, the leader updates its commit index and applies the operation to its state machine. The leader then includes the updated commit index in subsequent AppendEntries RPCs.

A follower will only append a log entry (or set of log entries) if its log prior to that entry (or entries) is identical to the leader's log. This ensures that log entries are added in-order, preventing gaps in the log, and ensuring followers apply the correct log entries to their state machines.

3.3 Handling leader failures

This process continues until the leader fails, requiring a new leader to be established. Paxos and Raft take different approaches to this process so we describe each in turn.

Paxos. A follower will timeout after failing to receive a recent AppendEntries RPC from the leader. It then becomes a candidate and updates its term to the next term such that $t \bmod n = s$ where t is the next term, n is the number of servers and s is the candidate's server id. The candidate will send RequestVote RPCs to the other servers. This RPC includes the candidate's new term and commit index. When a server receives the RequestVote RPC, it will respond positively provided the candidate's term is greater than its own. This response also includes any log entries that the server has in its log subsequent to the candidate's commit index.

Once the candidate has received positive RequestVote responses from a majority of servers, the candidate must ensure its log includes all committed entries before becoming a leader. It does so as follows. For each index after the commit index, the leader reviews the log entries it has received alongside its own log. If the candidate has seen a log entry for the index then it will update its own log with the entry and the new term. If the leader has seen multiple log entries for the same index then it will update its own log with the entry from the greatest term and the new term. An example of this is given in Figure 2. The candidate can now become a leader and begin replicating its log to the other servers.

Raft. At least one of the followers will timeout after not receiving a recent AppendEntries RPC for the leader. It will become a candidate and increment its term. The candidate will send RequestVote RPCs to the other servers. Each includes the candidate's term as well as the candidate's last log term and index. When a server receives the RequestVote request it will respond positively provided the candidate's term is greater than or equal to its own, it has not yet voted for a candidate in this term, and the candidate's log is at least as up-to-date as its own. This last criterion can be checked by ensuring that the candidate's last log term is greater than the server's or, if they are the same, that the candidate's last index is greater than the server's.

Once the candidate has received positive RequestVote responses from a majority of servers, the candidate can become a leader and start replicating its log. However, for safety Raft requires that the leader does not update its commit index until at least one log entry from the new term has been committed.

As there may be multiple candidates in a given term, votes may be split such that no candidate has a majority. In this case, the candidate times out and starts a new election with the next term.

3.4 Safety

Both algorithms guarantee the following property:

Theorem 3.1 (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.*

There is at most one leader per term and a leader will not overwrite its own log so we can prove this by proving the following:

Theorem 3.2 (Leader Completeness)**.** *If an operation op is committed at index i by a leader in term t then all leaders of terms $> t$ will also have operation op at index i.*

Proof sketch for Paxos. Assume operation *op* is committed at index i with term t. We will use proof by induction over the terms greater than t.

Base case: If there is a leader of term $t + 1$, then it will have the operation op at index i.

As any two majority quorums intersect and messages are ordered by term, then at least one server with operation *op* at index i and with term t must have positively replied to the RequestVote RPC from the leader of $t + 1$. This server cannot have deleted or overwritten this operation as it cannot have positively responded to an AppendEntries RPC from any other leader since the leader of term t. The leader will choose the operation *op* as it will not receive any log entries with a term $> t$.

Inductive case: Assume that any leaders of terms $t + 1$ to $t + k$ have the operation op at index i. If there is a leader of term $t + k + 1$ then it will also have operation op at index i.

As any two majority quorums intersect and messages are ordered by term, then at least one server with operation *op* at index i and with term t to $t + k$ must have positively replied to the RequestVote RPC from the leader of term $t+k+1$. This is because the server cannot have deleted or overwritten this operation as it has not positively responded to an AppendEntries RPC from any leader except those with terms t to $t + k$. From our induction hypothesis, all these leaders will also have operation *op* at index i and thus will not have overwritten it. The leader may choose another operation only if it receives a log entry with that different operation at index i and with a greater term. From our induction hypothesis, all leaders of terms t to $t + k$ will also have operation *op* at index i and thus will not write another operation at index i. □

The proof for Raft uses the same induction but the details differ due to Raft's different approach to leader election.

4 Discussion

Raft and Paxos take different approaches to leader election, summarised in Table 1. We compare two dimensions, understandability and efficiency, to determine which is best.

Understandability. Raft guarantees that if two logs contain the same operation then it will have the same index and term in both. In other words, each operation is assigned a unique index and term pair. However, this is not the case in Paxos, where an operation may be assigned a higher term by a future leader, as demonstrated by operations B and C in Figure 2b. In Paxos, a log entry before the commit index may be overwritten. This is safe because the log entry will only be overwritten by an entry with the same operation, but it not as intuitive as Raft's approach.

The flip side of this is that Paxos makes it safe to commit a log entry if it is present on a majority of servers; but this is not the case for Raft, which requires that a leader only commits a log entry from a previous term if it is present on the majority of servers and the leader has committed a subsequent log entry from the current term.

In Paxos, the log entries replicated by the leader are either from the current term or they are already committed. We can see this in Figure 2, where all log entries after the commit index on the leader have the current term. This is not the case in Raft where a leader may be replicating uncommitted entries from previous terms.

Overall, we feel that Raft's approach is slightly more understandable than Paxos' but not significantly so.

Efficiency. In Paxos, if multiple servers become candidates simultaneously, the candidate with the higher term will win the election. In Raft, if multiple servers become candidates simultaneously, they may split the votes as they will have the same term, and so neither will win the election. Raft mitigates this by having followers wait an additional period, drawn from a uniform random distribution, after the election timeout. We thus expect that Raft will be both slower and have higher variance in the time taken to elect a leader.

However, Raft's leader election phase is more lightweight than Paxos'. Raft only allows a candidate with an up-to-date log to become a leader and thus need not send log entries during leader election. This is not true of Paxos, where every positive RequestVote response includes the follower's log entries after the candidate's commit index. There are various options to reduce the number of log entries sent but ultimately, it will always be necessary for some log entries to be sent if the leader's log is not already up to date.

It is not just with the RequestVote responses that Paxos sends more log entries than Raft. In both algorithms, once a candidate becomes a leader it will copy its log to all other servers. In Paxos, a log entry may have been given a new term by the leader and thus the leader may send another copy of the log entry to a server which already has a copy.

	Paxos	**Raft**
How does it ensure that each term has at most one leader?	A server *s* can only be a candidate in a term *t* if $t \bmod n = s$. There will only be one candidate per term so only one leader per term.	A follower can become a candidate in any term. Each follower will only vote for one candidate per term, so only one candidate can get a majority of votes and become the leader.
How does it ensure that a new leader's log contains all committed log entries?	Each RequestVote reply includes the follower's log entries. Once a candidate has received RequestVote responses from a majority of followers, it adds the entries with the highest term to its log.	A vote is granted only if the candidate's log is at least as up-to-date as the followers'. This ensures that a candidate only becomes a leader if its log is at least as up-to-date as a majority of followers.
How does it ensure that leaders safely commit log entries from previous terms?	Log entries from previous terms are added to the leader's log with the leader's term. The leader then replicates the log entries as if they were from the leader's term.	The leader replicates the log entries to the other servers without changing the term. The leader cannot consider these entries committed until it has replicated a subsequent log entry from its own term.

Table 1. Summary of the differences between Paxos and Raft

This is not the case with Raft, where each log entry keeps the same term throughout its time in the log. Again, there are various options for mitigating this issue in Paxos but they are beyond the scope of our simplified Paxos algorithm.

Overall, compared to Paxos, Raft's approach to leader election is surprisingly efficient for such a simple approach.

5 Relation to classical Paxos

Readers who are familiar with Paxos may feel that our description of Paxos differs from those previously published, and so we now outline how our Paxos algorithm relates to those found elsewhere in the literature.

Roles. Some descriptions of Paxos divide the responsibility of Paxos into three roles: proposer, acceptor and learner [17] or leader, acceptor and replica [35]. Our presentation uses just one role, server, which incorporates all roles. This presentation using a single role has also used the name *replica* [7],

Terminology. Terms are also referred to as views, ballot numbers [35], proposal numbers [17], round numbers, sequence numbers [7] or epochs. Our leader is also referred to as a master [7], primary, coordinator or distinguished proposer [17]. Typically, the period during which a server is a candidate is known as phase-1 and the period during which a server is a leader is known as phase-2. The RequestVote RPCs are often referred to as phase1a and phase1b messages [35], prepare request and response [17] or prepare and promise messages. The AppendEntries RPCs are often referred to as phase2a and phase2b messages [35], accept request and response [17] or propose and accept messages.

Terms. Paxos requires only that terms are totally ordered and that each server is allocated a disjoint set of terms (for safety) and that each server can use a term greater than any other term (for liveness). Whilst some descriptions of Paxos use round-robin natural numbers like us [7], others use lexicographically ordered pairs, consisting of an integer and the server ID, where each server only uses terms containing its own ID [35].

Ordering. Our log entries are replicated and decided in-order. This is not necessary but it does avoid the complexities of filling log gaps [17]. Similarly, some descriptions of Paxos bound the number of concurrent decisions, often necessary for reconfiguration [17, 35].

6 Summary

The Raft algorithm was proposed to address the longstanding issues with understandability of the widely studied Paxos algorithm.

In this paper, we have demonstrated that much of the understandability of Raft comes from its pragmatic abstraction and excellent presentation. By describing a simplified Paxos algorithm using the same approach as Raft, we find that the two algorithms differ only in their approach to leader election. Specifically:

(i) Paxos divides terms between servers, whereas Raft allows a follower to become a candidate in any term but followers will vote for only one candidate per term.
(ii) Paxos followers will vote for any candidate, whereas Raft followers will only vote for a candidate if the candidate's log is at-least-as up-to-date.
(iii) If a leader has uncommitted log entries from a previous term, Paxos will replicate them in the current term whereas Raft will replicate them in their original term.

The Raft paper claims that Raft is significantly more understandable than Paxos, and as efficient. On the contrary, we find that the two algorithms are not significantly different in understandability but Raft's leader election is surprisingly lightweight when compared to Paxos'. Both algorithms we have presented are naïve by design and could certainly be optimised to improve performance, though often at the cost of increased complexity.

References

[1] Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Zhiyanan Zhiyanan, and Zhujianfeng Zhujianfeng. 2017. Leader or Majority: Why Have One When You Can Have Both? Improving Read Scalability in Raft-like Consensus Protocols. In *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing* (Santa Clara, CA) *(HotCloud'17)*. USENIX Association, USA, 14.

[2] atomix [n.d.]. Atomix. https://atomix.io.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[4] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. 2003. Deconstructing Paxos. *SIGACT News* 34, 1 (March 2003), 47–67. https://doi.org/10.1145/637437.637447

[5] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 335–350.

[6] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. 2007. Multicoordinated Paxos. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 316–317. https://doi.org/10.1145/1281100.1281150

[7] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 398–407. https://doi.org/10.1145/1281100.1281103

[8] cockroachdb [n.d.]. CockroachDB. https://www.cockroachlabs.com.

[9] consul [n.d.]. Consul by HashiCorp. https://www.consul.io.

[10] etcd [n.d.]. etcd. https://coreos.com/etcd/.

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[12] Eli Gafni and Leslie Lamport. 2000. Disk Paxos. In *Proceedings of the 14th International Conference on Distributed Computing (DISC '00)*. Springer-Verlag, Berlin, Heidelberg, 330–344.

[13] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[14] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[15] Kubernetes [n.d.]. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io.

[16] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[17] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[18] Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report MSR-TR-2005-33. Microsoft. 60 pages. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

[19] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (October 2006), 79–103. https://www.microsoft.com/en-us/research/publication/fast-paxos/

[20] Leslie Lamport and Mike Massa. 2004. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN '04)*. IEEE Computer Society, USA, 307.

[21] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[22] Butler Lampson. 2001. The ABCD's of Paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, USA) *(PODC '01)*. Association for Computing Machinery, New York, NY, USA, 13. https://doi.org/10.1145/383962.383969

[23] Butler W. Lampson. 1996. How to Build a Highly Available System Using Consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG '96)*. Springer-Verlag, Berlin, Heidelberg, 1–17.

[24] m3 [n.d.]. M3: Uber's Open Source, Large-scale Metrics Platform for Prometheus. https://eng.uber.com/m3/.

[25] Hein Meling and Leander Jehl. 2013. Tutorial Summary: Paxos Explained from Scratch. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304* (Nice, France) *(OPODIS 2013)*. Springer-Verlag, Berlin, Heidelberg, 1–10. https://doi.org/10.1007/978-3-319-03850-6_1

[26] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[27] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2671001

[28] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[29] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. 1997. Revisiting the Paxos Algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG '97)*. Springer-Verlag, Berlin, Heidelberg, 111–125.

[30] raft [n.d.]. The Raft Consensus Algorithm. https://raft.github.io.

[31] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, and et al. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 51–63. https://doi.org/10.1145/3035918.3056100

[32] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[33] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 351–364.

https://doi.org/10.1145/2465351.2465386

[34] trillian [n.d.]. Trillian: General Transparency. https://github.com/google/trillian/.

[35] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (Feb. 2015), 36 pages. https://doi.org/10.1145/2673577

[36] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[37] Yang Zhang, Eman Ramadan, Hesham Mekky, and Zhi-Li Zhang. 2017. When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network. In *Proceedings of the First Asia-Pacific Workshop on Networking* (Hong Kong, China) *(APNet'17)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3106989.3106999

[38] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: High-Availability Storage Made Practical in WeChat. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1730–1741. https://doi.org/10.14778/3137765.3137778

A Paxos Algorithm

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

State

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

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

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

Volatile state on all servers:

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

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

Volatile state on candidates: (Reinitialized after election)

entries[] Log entries received with votes

Volatile state on leaders: (Reinitialized after election)

nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader commit index + 1)

matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat

Arguments:

term leader's term

prevLogIndex index of log entry immediately preceding new ones

prevLogTerm term of prevLogIndex entry

entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)

leaderCommit leader's commitIndex

Results:

term currentTerm, for leader to update itself

success true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes

Arguments:

term candidate's term

leaderCommit candidate's commit index

Results:

term currentTerm, for candidate to update itself

voteGranted true indicates candidate received vote

entries[] follower's log entries after leaderCommit

Receiver implementation:

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

Rules for Servers

All Servers:

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

Followers:

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

Candidates:

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

Leaders:

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

B Raft Algorithm

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

State

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

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

votedFor candidateId that received vote in current term (or null if none)

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

Volatile state on all servers:

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

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

Volatile state on leaders: (Reinitialized after election)

nextIndex[ ] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)

matchIndex[ ] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries; also used as heartbeat

Arguments:

term leader's term

prevLogIndex index of log entry immediately preceding new ones

prevLogTerm term of prevLogIndex entry

entries[ ] log entries to store (empty for heartbeat; may send more than one for efficiency)

leaderCommit leader's commitIndex

Results:

term currentTerm, for leader to update itself

success true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex: set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes

Arguments:

term candidate's term

candidateId candidate requesting vote

lastLogIndex index of candidate's last log entry

lastLogTerm term of candidate's last log entry

Results:

term currentTerm, for candidate to update itself

voteGranted true indicates candidate received vote

Receiver implementation:

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

Rules for Servers

All Servers:

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

Followers:

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

Candidates:

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

Leaders:

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