

CONSENSUS: BRIDGING THEORY AND PRACTICE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Diego Ongaro
August 2014

© 2014 by Diego Andres Ongaro. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-3.0 United States License.

<http://creativecommons.org/licenses/by/3.0/us/>

This dissertation is online at: <http://purl.stanford.edu/qr033xr6097>

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Mazieres

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumpert, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Distributed consensus is fundamental to building fault-tolerant systems. It allows a collection of machines to work as a coherent group that can survive the failures of some of its members. Unfortunately, the most common consensus algorithm, Paxos, is widely regarded as difficult to understand and implement correctly.

This dissertation presents a new consensus algorithm called Raft, which was designed for understandability. Raft first elects a server as leader, then concentrates all decision-making onto the leader. These two basic steps are relatively independent and form a better structure than Paxos, whose components are hard to separate. Raft elects a leader using voting and randomized timeouts. The election guarantees that the leader already stores all the information it needs, so data only flows outwards from the leader to other servers. Compared to other leader-based algorithms, this reduces mechanism and simplifies the behavior. Once a leader is elected, it manages a replicated log. Raft leverages a simple invariant on how logs grow to reduce the algorithm's state space and accomplish this task with minimal mechanism.

Raft is also more suitable than previous algorithms for real-world implementations. It performs well enough for practical deployments, and it addresses all aspects of building a complete system, including how to manage client interactions, how to change the cluster membership, and how to compact the log when it grows too large. To change the cluster membership, Raft allows adding or removing one server at a time (complex changes can be composed from these basic steps), and the cluster continues servicing requests throughout the change.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. The algorithm has been formally specified and proven, its leader election algorithm works well in a variety of environments, and its performance is equivalent to Multi-Paxos. Many implementations of Raft are now available, and several companies are deploying Raft.

Preface

This dissertation expands on a paper written by Diego Ongaro and John Ousterhout entitled *In Search of an Understandable Consensus Algorithm* [89]. Most of the paper’s content is included in some form in this dissertation. It is reproduced in this dissertation and licensed under the Creative Commons Attribution license with permission from John Ousterhout.

Readers may want to refer to the Raft website [92] for videos about Raft and an interactive visualization of Raft.

Acknowledgments

Thanks to my family and friends for supporting me throughout the ups and downs of grad school. Mom, thanks for continuously pushing me to do well academically, even when I didn't see the point. I still don't know how you got me out of bed at 6 a.m. all those mornings. Dad, thanks for helping us earn these six (seven?) degrees, and I hope we've made you proud. Zeide, I wish I could give you a copy of this small book for your collection. Ernesto, thanks for sparking my interest in computers; I still think they're pretty cool. Laura, I'll let you know if and when I discover a RAMCloud. Thanks for listening to hours of my drama, even when you didn't understand the nouns. Jenny, thanks for helping me get through the drudgery of writing this dissertation and for making me smile the whole way through. You're crazy for having wanted to read this, and you're weird for having enjoyed it.

I learned a ton from my many labmates, both in RAMCloud and in SCS. Deian, I don't know why you always cared about my work; I never understood your passion for that IFC nonsense, but keep simplifying it until us mortals can use it. Ankita, you've single-handedly increased the lab's average self-esteem and optimism by at least 20%. I've watched you learn so much already; keep absorbing it all, and I hope you're able to see how far you've come. Good luck with your role as the new Senior Student. Thanks especially to Ryan and Steve, with whom I formed the first generation of RAMCloud students. Ryan, believe it or not, your optimism helped. You were always excited about wacky ideas, and I always looked forward to swapping CSBs ("cool story, bro") with you. You'll make a great advisor. Steve, I miss your intolerance for bullshit, and I strive to match your standards for your own engineering work. You continuously shocked the rest of us with those silent bursts of productivity, where you'd get quarter-long projects done over a single weekend. You guys also figured out all the program requirements before I did and told me all the tricks. I continue to follow your lead even after you've moved on. (Ryan, you incorrectly used the British spelling "acknowledgements" rather than the American "acknowledgments". Steve, you did too, but you're just Canadian, not wrong.)

Thanks to the many professors who have advised me along the way. John Ousterhout, my Ph.D.

advisor, should be a coauthor on this dissertation (but I don't think they would give me a degree that way). I have never learned as much professionally from any other person. John teaches by setting a great example of how to code, to evaluate, to design, to think, and to write *well*. I have never quite been on David Mazières's same wavelength; he's usually 10–30 minutes ahead in conversation. As soon as I could almost keep up with him regarding consensus, he moved on to harder Byzantine consensus problems. Nevertheless, David has looked out for me throughout my years in grad school, and I've picked up some of his passion for building useful systems and, more importantly, having fun doing so. Mendel Rosenblum carries intimate knowledge of low level details like x86 instruction set, yet also manages to keep track of the big picture. He's helped me with both over the years, surprising me with how quickly he can solve my technical problems and how clear my predicaments are when put into his own words. Thanks to Christos Kozyrakis and Stephen Weitzman for serving on my defense committee, and thanks to Alan Cox and Scott Rixner for introducing me to research during my undergraduate studies at Rice.

Many people contributed directly to this dissertation work. A special thanks goes to David Mazières and Ezra Hoch for each finding a bug in earlier versions of Raft. David emailed us one night at 2:45 a.m. as he was reading through the Raft lecture slides for the user study. He wrote that he found “one thing quite hard to follow in the slides,” which turned out to be a major issue in Raft's safety. Ezra found a liveness bug in membership changes. He posted to the Raft mailing list, “What if the following happens?” [35], and described an unfortunate series of events that could leave a cluster unable to elect a leader. Thanks also to Hugues Evrard for finding a small omission in the formal specification.

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.

Many people provided feedback on other content in this dissertation. In addition to my reading committee, Jennifer Wolochow provided helpful comments on the entire dissertation. Blake Mizerany, Xiang Li, and Yicheng Qin at CoreOS pushed me to simplify the membership change algorithm towards single-server changes. Anirban Rahut from Splunk pointed out that membership changes may be needlessly slow when a server joins with an empty log. Laura Ongaro offered helpful feedback on the user study chapter. Asaf Cidon helped direct me in finding the probability of split votes during elections. Eddie Kohler helped clarify the trade-offs in Raft's commitment rule, and Maciej Smoleński pointed out that because of it, if a leader were to restart an unbounded number of

times before it could mark entries committed, its log could grow without bound (see Chapter 11). Alexander Shraer helped clarify how membership changes work in Zab.

Many people provided helpful feedback on the Raft paper and user study materials, including Ed Bugnion, Michael Chan, Hugues Evrard, 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, 24 anonymous conference reviewers (with duplicates), and especially Eddie Kohler for shepherding the Raft paper.

Werner Vogels tweeted a link to an early draft of the Raft paper, which gave Raft significant exposure. Ben Johnson and Patrick Van Stee both gave early talks on Raft at major industry conferences.

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 was supported by The Jungle Corporation Stanford Graduate Fellowship. James Myers at Intel donated several SSDs used in benchmarking.

Contents

Abstract	iv
Preface	v
Acknowledgments	vi
Contents	ix
List of tables	xiv
List of figures	xv
1 Introduction	1
2 Motivation	5
2.1 Achieving fault tolerance with replicated state machines	5
2.2 Common use cases for replicated state machines	7
2.3 What's wrong with Paxos?	8
3 Basic Raft algorithm	11
3.1 Designing for understandability	11
3.2 Raft overview	12
3.3 Raft basics	14
3.4 Leader election	16
3.5 Log replication	17
3.6 Safety	22
3.6.1 Election restriction	22

3.6.2	Committing entries from previous terms	24
3.6.3	Safety argument	24
3.7	Follower and candidate crashes	26
3.8	Persisted state and server restarts	27
3.9	Timing and availability	27
3.10	Leadership transfer extension	28
3.11	Conclusion	30
4	Cluster membership changes	32
4.1	Safety	33
4.2	Availability	36
4.2.1	Catching up new servers	37
4.2.2	Removing the current leader	39
4.2.3	Disruptive servers	40
4.2.4	Availability argument	42
4.3	Arbitrary configuration changes using joint consensus	43
4.4	System integration	45
4.5	Conclusion	46
5	Log compaction	48
5.1	Snapshotting memory-based state machines	51
5.1.1	Snapshotting concurrently	52
5.1.2	When to snapshot	54
5.1.3	Implementation concerns	55
5.2	Snapshotting disk-based state machines	57
5.3	Incremental cleaning approaches	58
5.3.1	Basics of log cleaning	59
5.3.2	Basics of log-structured merge trees	60
5.3.3	Log cleaning and log-structured merge trees in Raft	60
5.4	Alternative: leader-based approaches	62
5.4.1	Storing snapshots in the log	62
5.4.2	Leader-based approach for very small state machines	63
5.5	Conclusion	64

6	Client interaction	66
6.1	Finding the cluster	66
6.2	Routing requests to the leader	68
6.3	Implementing linearizable semantics	69
6.4	Processing read-only queries more efficiently	72
6.4.1	Using clocks to reduce messaging for read-only queries	73
6.5	Conclusion	75
7	Raft user study	76
7.1	Study questions and hypotheses	77
7.2	Discussion about the methods	78
7.2.1	Participants	79
7.2.2	Teaching	81
7.2.3	Testing understanding	83
7.2.4	Grading	85
7.2.5	Survey	86
7.2.6	Pilots	86
7.3	Methods	87
7.3.1	Study design	87
7.3.2	Participants	87
7.3.3	Materials	88
7.3.4	Dependent measures	91
7.3.5	Procedure	91
7.4	Results	91
7.4.1	Quizzes	93
7.4.2	Survey	101
7.5	Discussion about the experimental approach	106
7.6	Conclusion	110
8	Correctness	111
8.1	Formal specification and proof for basic Raft algorithm	112
8.2	Discussion of prior verification attempts	114
8.3	Building correct implementations	115
8.4	Conclusion	116

9	Leader election evaluation	117
9.1	How fast will Raft elect a leader with no split votes?	118
9.2	How common are split votes?	122
9.3	How fast will Raft elect a leader when split votes are possible?	129
9.4	How fast will the complete Raft algorithm elect a leader in real networks?	132
9.5	What happens when logs differ?	136
9.6	Preventing disruptions when a server rejoins the cluster	136
9.7	Conclusion	137
10	Implementation and performance	139
10.1	Implementation	139
10.1.1	Threaded architecture	139
10.2	Performance considerations	141
10.2.1	Writing to the leader's disk in parallel	141
10.2.2	Batching and pipelining	142
10.3	Preliminary performance results	144
10.4	Conclusion	146
11	Related work	147
11.1	Overview of consensus algorithms	147
11.1.1	Paxos	147
11.1.2	Leader-based algorithms	149
11.2	Leader election	150
11.2.1	Detecting and neutralizing a failed leader	150
11.2.2	Selecting a new leader and ensuring it has all committed entries	151
11.3	Log replication and commitment	153
11.4	Cluster membership changes	155
11.4.1	α -based approaches	156
11.4.2	Changing membership during leader election	157
11.4.3	Zab	159
11.5	Log compaction	160
11.6	Replicated state machines vs. primary copy approach	161
11.7	Performance	162
11.7.1	Reducing leader bottleneck	164

11.7.2 Reducing number of servers (witnesses)	166
11.7.3 Avoiding persistent storage writes	168
11.8 Correctness	168
11.9 Understandability	169
12 Conclusion	171
12.1 Lessons learned	173
12.1.1 On complexity	173
12.1.2 On bridging theory and practice	173
12.1.3 On finding research problems	174
12.2 Final comments	175
A User study materials	176
A.1 Raft quiz	176
A.2 Paxos quiz	184
A.3 Survey	191
A.4 Supporting materials	195
B Safety proof and formal specification	201
B.1 Conventions	201
B.2 Specification	202
B.3 Proof	215
Bibliography	230

List of tables

7.1	Raft user study: study participation	80
7.2	Raft user study: lecture lengths	89
7.3	Raft user study: linear model of quiz grades, including school factor	99
7.4	Raft user study: linear model of quiz grades, excluding school factor	99
9.1	Leader election evaluation: summary of variables	120
9.2	Leader election evaluation: experimental setup for benchmark	132
10.1	Implementation and performance: experimental setup	144
11.1	Related work: summary of how different algorithms select a new leader	152
11.2	Related work: approaches to reduce the servers involved in each decision	167

List of figures

2.1	Motivation: replicated state machine architecture	6
2.2	Motivation: common patterns for using a single replicated state machine	7
2.3	Motivation: partitioned large-scale storage system using consensus	8
2.4	Motivation: summary of the single-decree Paxos protocol	9
3.1	Basic Raft algorithm: algorithm summary	13
3.2	Basic Raft algorithm: key properties	14
3.3	Basic Raft algorithm: server states	15
3.4	Basic Raft algorithm: terms	15
3.5	Basic Raft algorithm: log structure	18
3.6	Basic Raft algorithm: log inconsistencies	20
3.7	Basic Raft algorithm: commitment rule	23
3.8	Basic Raft algorithm: existence of voter in safety argument	25
4.1	Cluster membership changes: RPCs to change cluster membership	33
4.2	Cluster membership changes: safety challenge	34
4.3	Cluster membership changes: adding/removing one server maintains overlap	34
4.4	Cluster membership changes: how adding servers can put availability at risk	37
4.5	Cluster membership changes: rounds in server catchup algorithm	38
4.6	Cluster membership changes: example of progress depending on removed server	40
4.7	Cluster membership changes: example of disruptive server	41
4.8	Cluster membership changes: joint consensus timeline	44
5.1	Log compaction: summary of approaches	49
5.2	Log compaction: memory-based snapshotting approach	52
5.3	Log compaction: InstallSnapshot RPC	53

5.4	Log compaction: approaches to log cleaning in Raft	61
5.5	Log compaction: alternative: snapshot stored in log	63
6.1	Client interaction: summary of RPCs	67
6.2	Client interaction: example of incorrect results for duplicated command	70
6.3	Client interaction: lease mechanism for read-only queries	74
7.1	Raft user study: example lecture slide with stylus overlay	89
7.2	Raft user study: quiz score CDF	92
7.3	Raft user study: quiz score scatter plot (by school)	94
7.4	Raft user study: quiz score scatter plot (by prior Paxos exposure)	95
7.5	Raft user study: CDF of participants' quiz score difference	96
7.6	Raft user study: ordering effects	97
7.7	Raft user study: quiz score CDFs by question difficulty and ordering	100
7.8	Raft user study: quiz score CDFs by question	102
7.9	Raft user study: prior Paxos experience survey	103
7.10	Raft user study: fairness survey	104
7.11	Raft user study: preferences survey	105
9.1	Leader election evaluation: leader election timeline with no split votes	119
9.2	Leader election evaluation: earliest timeout example	119
9.3	Leader election evaluation: earliest timeout CDF	121
9.4	Leader election evaluation: split vote example with fixed latency	123
9.5	Leader election evaluation: split vote probability with fixed network latency	126
9.6	Leader election evaluation: split vote probability with variable network latency	128
9.7	Leader election evaluation: expected overall election time	131
9.8	Leader election evaluation: benchmark results on LAN cluster	133
9.9	Leader election evaluation: election performance on a simulated WAN cluster	135
9.10	Leader election evaluation: election performance with differing logs	135
10.1	Implementation and performance: threaded architecture	140
10.2	Implementation and performance: optimized request processing pipeline	142
10.3	Implementation and performance: preliminary measurements of LogCabin	145
11.1	Related work: differences in how new leaders replicate existing entries	154

11.2 Related work: primary copy architecture	161
A.1 User study materials: Raft summary	196
A.2 User study materials: Paxos summary, page 1 of 4	197
A.3 User study materials: Paxos summary, page 2 of 4	198
A.4 User study materials: Paxos summary, page 3 of 4	199
A.5 User study materials: Paxos summary, page 4 of 4	200

PREVIEW

Chapter 1

Introduction

Today’s datacenter systems and applications run in highly dynamic environments. They scale out by leveraging the resources of additional servers, and they grow and shrink according to demand. Server and network failures are also commonplace: about 2–4% of disk drives fail each year [103], servers crash about as often [22], and tens of network links fail every day in modern datacenters [31].

As a result, systems must deal with servers coming and going during normal operations. They must react to changes and adapt automatically within seconds; outages that are noticeable to humans are typically not acceptable. This is a major challenge in today’s systems; failure handling, coordination, service discovery, and configuration management are all difficult in such dynamic environments.

Fortunately, distributed consensus can help with these challenges. Consensus allows a collection of machines to work as a coherent group that can survive the failures of some of its members. Within a consensus group, failures are handled in a principled and proven way. Because consensus groups are highly available and reliable, other system components can use a consensus group as the foundation for their own fault tolerance. Thus, consensus plays a key role in building reliable large-scale software systems.

When we started this work, the need for consensus was becoming clear, but many systems still struggled with problems that consensus could solve. Some large-scale systems were still limited by a single coordination server as a single point of failure (e.g., HDFS [81, 2]). Many others included ad hoc replication algorithms that handled failures unsafely (e.g., MongoDB and Redis [44]). New systems had few options for readily available consensus implementations (ZooKeeper [38] was the most popular), forcing systems builders to conform to one or build their own.

Those choosing to implement consensus themselves usually turned to Paxos [48, 49]. Paxos had

dominated the discussion of consensus algorithms over the last two decades: most implementations of consensus were based on Paxos or influenced by it, and Paxos had 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 requires complex changes to support practical systems, and building a complete system based on Paxos requires developing several extensions for which the details have not been published or agreed upon. As a result, both system builders and students struggle with Paxos.

The two other well-known consensus algorithms are Viewstamped Replication [83, 82, 66] and Zab [42], the algorithm used in ZooKeeper. Although we believe both of these algorithms are incidentally better in structure than Paxos for building systems, neither has explicitly made this argument; they were not designed with simplicity or understandability as a primary goal. The burden of understanding and implementing these algorithms is still too high.

Each of these consensus options was difficult to understand and difficult to implement. Unfortunately, when the cost of implementing consensus with proven algorithms was too high, systems builders were left with a tough decision. They could avoid consensus altogether, sacrificing the fault tolerance or consistency of their systems, or they could develop their own ad hoc algorithm, often leading to unsafe behavior. Moreover, when the cost of explaining and understanding consensus was too high, not all instructors attempted to teach it, and not all students succeeded in learning it. Consensus is as fundamental as two-phase commit; ideally, as many students should learn it (even though consensus is fundamentally more difficult).

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 for practical systems and describe it in a way that is significantly easier to learn than Paxos? Furthermore, we wanted the algorithm to facilitate 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.

This algorithm also had to be complete enough to address all aspects of building a practical system, and it had to perform well enough for practical deployments. The core algorithm not only had to specify the effects of receiving a message but also describe what *should* happen and when; these are equally important for systems builders. Similarly, it had to guarantee consistency, and it also had to provide availability whenever possible. It also had to address the many aspects of a system that go beyond reaching consensus, such as changing the members of the consensus group.

These are necessary in practice, and leaving this burden to systems builders would risk ad hoc, suboptimal, or even incorrect solutions.

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 non-determinism and the ways servers can be inconsistent with each other). We also addressed all of the issues needed to build a complete consensus-based system. We considered each design choice carefully, not just for the benefit of our own implementation but also for the many others we hope to enable.

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 and is used by several companies; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

The primary contributions of this dissertation are as follows:

- The design, implementation, and evaluation of the Raft consensus algorithm. Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [83, 66]), but it is designed for understandability. This led to several novel features. For example, Raft uses a stronger form of leadership than other consensus algorithms. This simplifies the management of the replicated log and makes Raft easier to understand.
- The evaluation of Raft's understandability. A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos. We believe this is the first scientific study to evaluate consensus algorithms based on teaching and learning.
- The design, implementation, and evaluation of Raft's leader election mechanism. While many consensus algorithms do not prescribe a particular leader election algorithm, Raft includes a specific algorithm involving randomized timers. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly. The evaluation of leader election investigates its behavior and performance, concluding that this simple approach is sufficient in a wide variety of practical environments. It typically elects a leader in under 20 times the cluster's one-way network latency.

- The design and implementation of Raft’s cluster membership change mechanism. Raft allows adding or removing a single server at a time; these operations preserve safety simply, since at least one server overlaps any majority during the change. More complex changes in membership are implemented as a series of single-server changes. Raft allows the cluster to continue operating normally during changes, and membership changes can be implemented with only a few extensions to the basic consensus algorithm.
- A thorough discussion and implementation of the other components necessary for a complete consensus-based system, including client interaction and log compaction. Although we do not believe these aspects of Raft to be particularly novel, a complete description is important for understandability and to enable others to build real systems. We have implemented a complete consensus-based service to explore and address all of the design decisions involved.
- A proof of safety and formal specification for the Raft algorithm. The level of precision in the formal specification aids in reasoning carefully about the algorithm and clarifying details in the algorithm’s informal description. The proof of safety helps build confidence in Raft’s correctness. It also aids others who wish to extend Raft by clarifying the implications for safety of their extensions.

We have implemented many of the designs in this dissertation in an open-source implementation of Raft called LogCabin [86]. LogCabin served as our test platform for new ideas in Raft and as a way to verify that we understood the issues of building a complete and practical system. The implementation is described in more detail in Chapter 10.

The remainder of this dissertation introduces the replicated state machine problem and discusses the strengths and weaknesses of Paxos (Chapter 2); presents the Raft consensus algorithm, its extensions for cluster membership changes and log compaction, and how clients interact with Raft (Chapters 3–6); evaluates Raft for understandability, correctness, and leader election and log replication performance (Chapters 7–10); and discusses related work (Chapter 11).

Chapter 2

Motivation

Consensus is a fundamental problem in fault-tolerant systems: how can servers reach agreement on shared state, even in the face of failures? This problem arises in a wide variety of systems that need to provide high levels of availability and cannot compromise on consistency; thus, consensus is used in virtually all consistent large-scale storage systems. Section 2.1 describes how consensus is typically used to create replicated state machines, a general-purpose building block for fault-tolerant systems; Section 2.2 discusses various ways replicated state machines are used in larger systems; and Section 2.3 discusses the problems with the Paxos consensus protocol, which Raft aims to address.

2.1 Achieving fault tolerance with replicated state machines

Consensus algorithms typically arise in the context of *replicated state machines* [102]. In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines are used to solve a variety of fault tolerance problems in distributed systems, as described in Section 2.2. Examples of replicated state machines include Chubby [11] and ZooKeeper [38], which both provide hierarchical key-value stores for small amounts of configuration data. In addition to basic operations such as *get* and *put*, they also provide synchronization primitives like *compare-and-swap*, enabling concurrent clients to coordinate safely.

Replicated state machines are typically implemented using a replicated log, as shown in Figure 2.1. 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

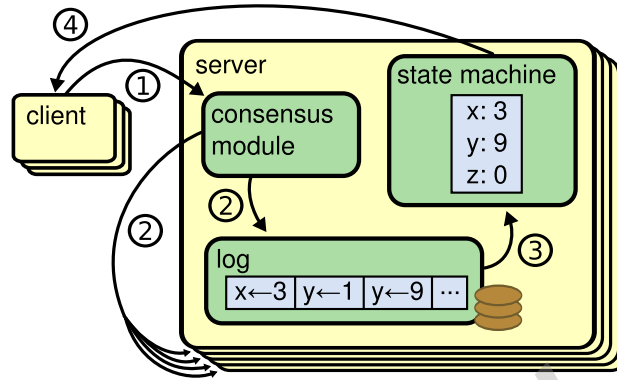


Figure 2.1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

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. 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, they are said to be *committed*. Each server's state machine processes committed commands 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. That is, they maintain safety under an *asynchronous* model [71], in which messages and processors proceed at arbitrary speeds.

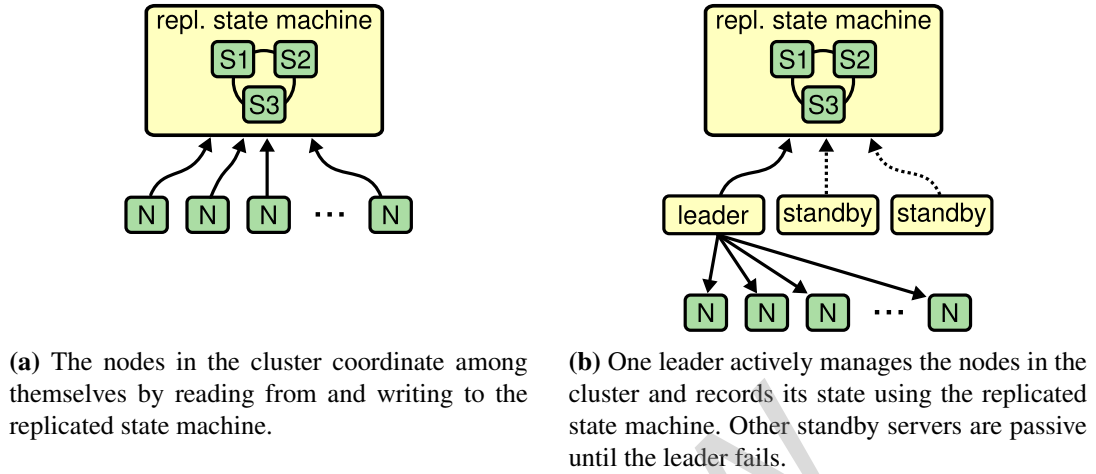


Figure 2.2: Common patterns for using a single replicated state machine.

- In the common case, a command can complete as soon as a 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.

2.2 Common use cases for replicated state machines

Replicated state machines are a general-purpose building block for making systems fault-tolerant. They can be used in a variety of ways, and this section discusses some typical usage patterns.

Most common deployments of consensus have just three or five servers forming one replicated state machine. Other servers can then use this state machine to coordinate their activities, as shown in Figure 2.2(a). These systems often use the replicated state machine to provide group membership, configuration management, or locks [38]. As a more specific example, the replicated state machine could provide a fault-tolerant work queue, and other servers could coordinate using the replicated state machine to assign work to themselves.

A common simplification to this usage is shown in Figure 2.2(b). In this pattern, one server acts as leader, managing the rest of the servers. The leader stores its critical data in the consensus system. In case it fails, other standby servers compete for the position of leader, and if they succeed, they use the data in the consensus system to continue operations. Many large-scale storage systems that have a single cluster leader, such as GFS [30], HDFS [105], and RAMCloud [90], use this approach.

Consensus is also sometimes used to replicate very large amounts of data, as shown in Figure 2.3. Large storage systems, such as Megastore [5], Spanner [20], and Scatter [32], store too